Escola de Camins
Escola Tècnica Superior d'Enginyeria de Camins, Canals i Ports
UPC BARCELONATECH

Study and development of a solidification model using CFD

MASTER FINAL THESIS

Final Thesis developed by:
Bazán Escoda, Aitor

Directed by:
Castilla, Robert

Master in:
Numerical methods in Engineering

Barcelona, date

Department of Fluid mechanics

UNIVERSITAT POLITÈCNICA DE CATALUNYA

MASTER THESIS

Study and development of a solidification model using CFD

Author:
Aitor BAZÁN ESCODA

Supervisor:
Dr. Robert CASTILLA

A thesis submitted in fulfillment of the requirements for the degree of Master Thesis

in the

Research Group Name
Escola Tècnica Superior d'Enginyeria de Camins, Canals i Ports de Barcelona

June 22, 2022

Declaration of Authorship

I, Aitor BAZÁN ESCODA, declare that this thesis titled, "Study and development of a solidification model using CFD" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

UNIVERSITAT POLITÈCNICA DE CATALUNYA

Abstract

Faculty Name

Escola Tècnica Superior d'Enginyeria de Camins, Canals i Ports de Barcelona

Master Thesis

Study and development of a solidification model using CFD

by Aitor BAZÁN ESCODA

Phase change materials (PCMs) are of great interest within the automotive industry field. Not only when used in thermal management applications but also in different areas where these materials are of vital importance for both a safe and comfortable driving. For such objective, the present project arises from the idea of understanding solidification processes in windshield washer tanks. In this context, this master's thesis produces a comprehensive state of the art of some of the current numerical methods to effectively represent water solidification.
An OpenFOAM 21.12. solver based on a multi-phase solver, multi-component incompressible solver based on a volume of fluid method is adapted to deal with diffusive-convective phase change. So as to reach this goal, an implementation of the enthalpy-porosity techinque is carried out. The work of Voller et al. is closely followed, and a detailed explanation of the used equations and the assumptions taken is given. Validation of the model is accomplished by comparing the results with the authors in Bourdillon and Kowaleski and Rebow.
On a second stage, a 2D semi-empirical model based on the work of Lee is adapted to account for the nucleation characteristics during the process of the water phase change. Validation of the model is done by comparing the obtained results to Neumann solutions for classical Stefan problem.
Finally, the current work is extended to couple a fluid region in which the liquid undergoes a phase-change and a solid region. This is done in the context of a conjugate heat transfer environment.

Keywords: PCMs, multip-phase, Enthalpy-porosity technique, Lee model, Stefan problem, conjugate heat transfer, OpenFOAM.

Acknowledgements

My first and biggest thanks goes to my supervisors, Dr. Robert Castilla and Dr. Gustavo Raush for their invaluable help throughout this work. I would like to thank my family and all my workmates in Barcelona Technical Center S.L. for giving me support in the darkest hours.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Thesis Statement. Background and motivation

During the last decade, the use of phase change materials has been growing in the automotive industry.

These substances release or absorb large amounts of latent heat when they go through a change in their physical state, as the material reaches its specific phase change temperature. Thus, in the process of latent heat release or absorption, the temperature of the PCM remains constant. Therefore, PCMs are considered to be efficient in terms of thermal storage.

However, some of these PCM's may present physical effects which most of times require special conditions for the containers in where they are placed. Thus, in such geometries, problems involving, i.e., solidification are of considerable relevance. And this is mainly due to a volumetric expansion originated by the thermal effects within the PCM which at its turn, generates stresses in the tank in which is bottled up.

Therefore, this master's thesis main objective aims to study different numerical techniques to represent solidification process and, specially, pure water phase change. This is accomplished by first stuying the physics of a pure convective solver.

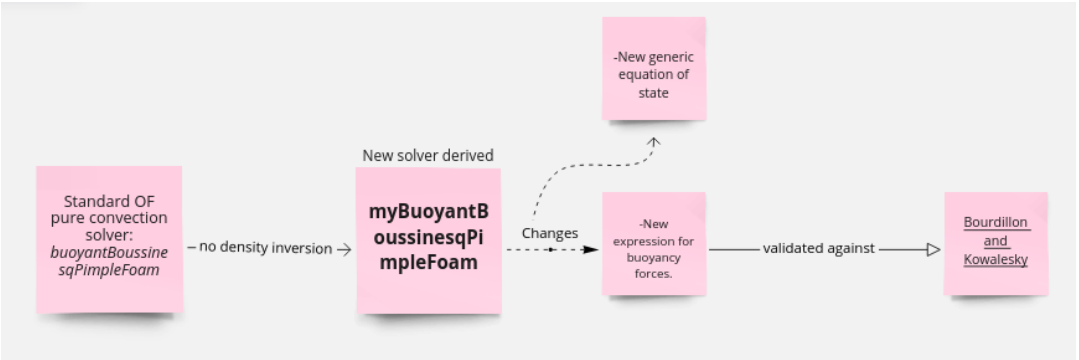


FIGURE 1.1: Workpath for buoyancy effects study.

And later, imlementing an enthalpy-porosity technique within the frame of a multi-phase incompressible solver based on volume-of-fluid (VOF) method for interface tracking. The objective of this first stage is to apply sensible and latent heat as source terms in the energy equation.

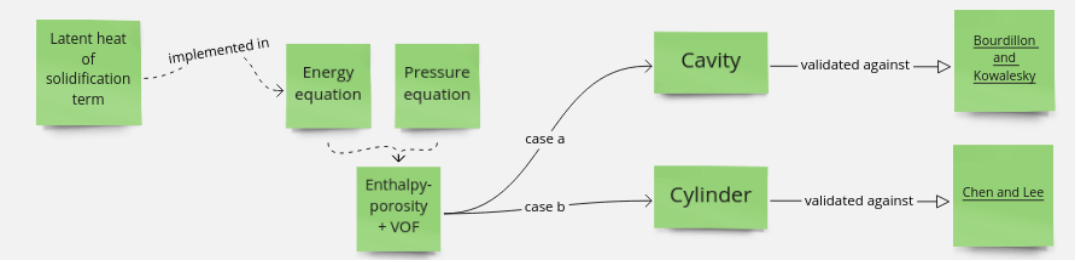


FIGURE 1.2: Workpath for latent heat implementation in Enthalpy-porosity model.

On a second stage of the thesis, a 2D semi-empirical model based on the work of Lee is adapted to account for the nucleation characteristics occured during the water phase transition. This is implemented within the same solver based on the technique of VOF as in the previous case.

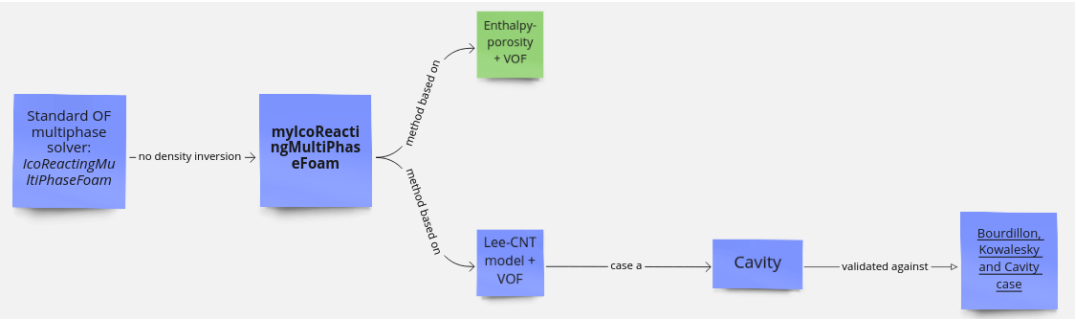


FIGURE 1.3: Workpath for nucleation theory implementation in Lee model.

The final stage of this thesis is devoted to an implementation of a multiregion solver to calculate conjugate heat transfer problems between solid and fluid zones with the singularity of being, the fluid zone, capable of handling phase change materials.

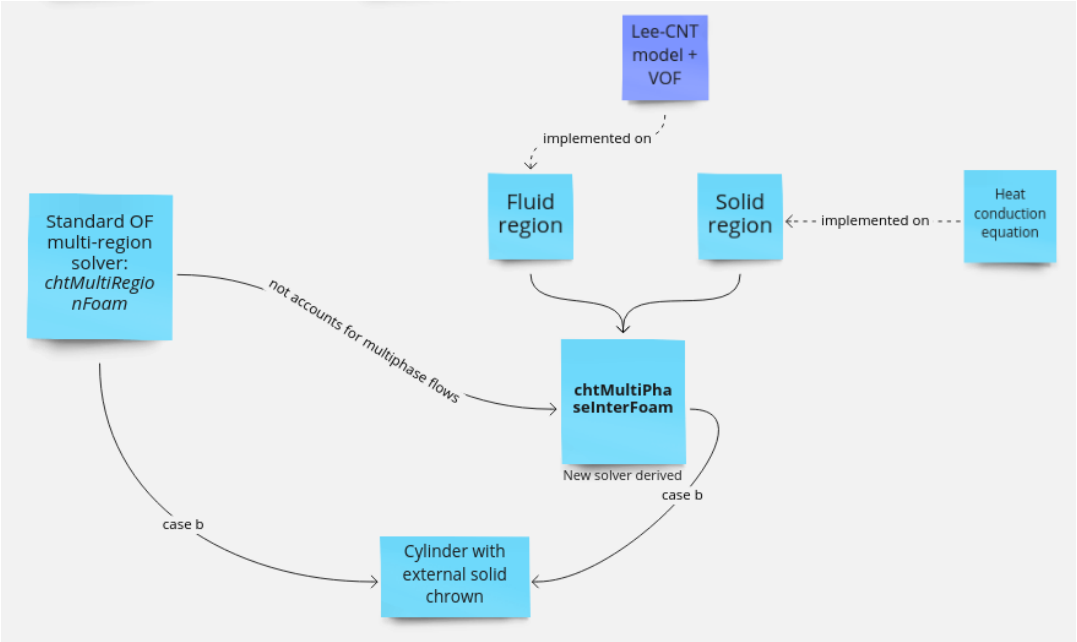
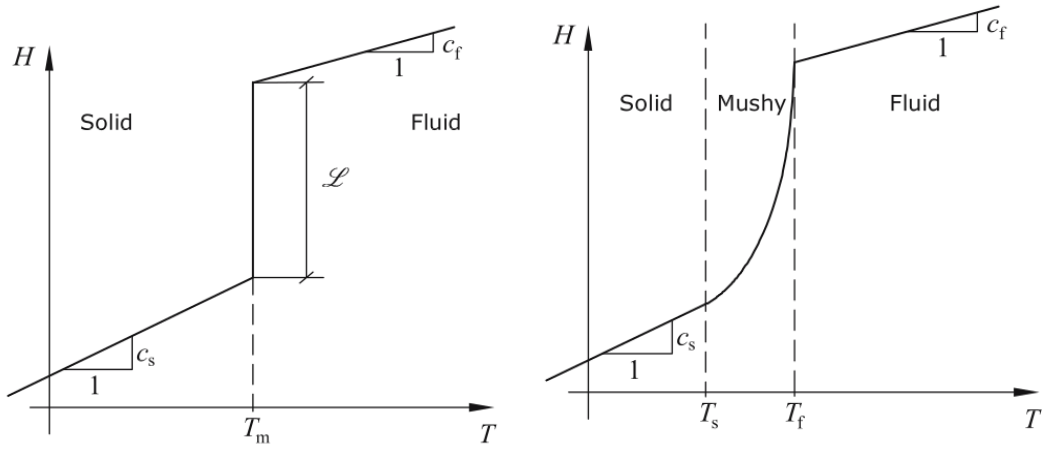


FIGURE 1.4: Workpath for conjugate heat transfer implementation.

The next chapters are mainly focused on describing phase change phenomena and heat transfer mechanisms used along the completion of this master's thesis.

1.2 Phase Change Process

The phase change is usually modelled by a sudden change in enthalpy per unit of temperature generated within a narrow temperature range near the freezing point. Often, this process is assumed to behave in a characteristic temperature, as shown in Fig. 1.5a, leading to a moving boundary problem. However, at a given critical temperature, both fluid and solid phases may coexist giving a state called mushy region as in Fig. 1.5b. In this case, one speaks of a non-linear diffusion problem rather than a moving boundary problem [13].



(A) Isothermal phase change.

(B) Non-isothermal phase change.

FIGURE 1.5: Phase change comparison.

As briefly introduced, two types of phase change are used to describe the way the latent heat is released or absorbed during freezing or melting processes:

- **Non-isothermal phase change:** the phase change takes place within a temperature range yielding a transition zone between a solid and a liquid phase called mushy zone. Typically, the thickness of this region is straightfully proportional to the temperature range in wich the phase change occurs.
- **Isothermal phase change:** the phase change is arisen instantaneously at the melting temperature. The release or absorption of the latent heat occurs at this point in which, consequently, there is no transition zone between solid and liquid phases. At this point, there is a narrow line, mainly derived from the discretization of the computational domain, characterizing the phase change phenomena.

As an important remark, when the mushy region is sufficiently narrow, the isothermal assumption is usually a good approximation. However, despite of the fact of being a convenient approach it may lead to significant complication when it comes to numerical solution techniques.

Along the next chapters, a more detailed description on existing techniques that deal with such non-trivial phenomena will be given.

1.2.1 Water phase change

In a similar manner, the water phase change takes place. A complex interaction of the molecular forces generate water to behave in a curious way when it gets frozen into ice. The vast majority of substances, when they are cooled down, become more dense in the frozen state than when liquid. However, when cooled under a specific temperature, water begins to expand and, once it starts freezing, it becomes less dense than water.

1.2.2 Phase diagram of ice

When water begins to become ice, these ice crystals might undergo different kinds of structures. Called ice Ih, in the form of hexagonal ice and, manifested in six cornered snow flakes, is the natural ice generally found in earth. However, at lower pressures below 2 kbar, many other ice structures may exist.

The ice phase diagram shown in Fig. 1.6, points out the conditions of stability for all ice phases. As it is cleared out, the line between the water and ice Ih is an equilibrium line with a negative slope, consequence of having, the solid, lower density than the liquid. These equilibrium lines extend in the form of metastable phase boundaries into the area of stability of other ice phases. Although there are at least 11 crystalline ice shapes, the only which is found in naturally on earth is the hexagonal form.

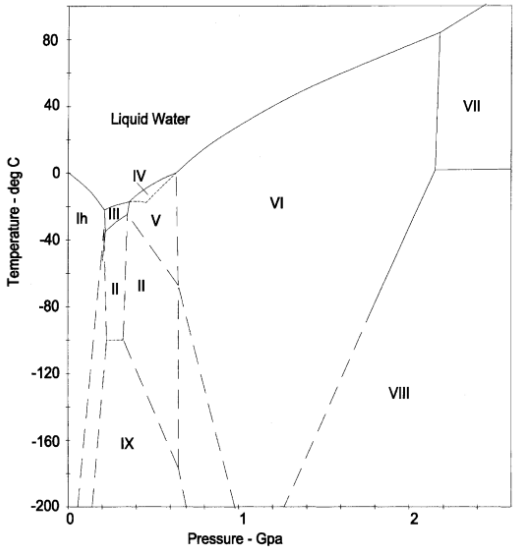


FIGURE 1.6: Phase diagram of ice.

As a remark, the implication that there is a rise on the pressure would not propitiate ice formation at 0°C, instead water would need to be cooled down.

1.2.3 Properties of ice

Ice, when subjected to visible light conditions, is transparent and has the lowest index of refraction for the sodium spectrum of any known crystalline material, as pointed out by Akyurt et al. [1].

Mechanically, ice behaves like a viscoelastic material with a non linear law. Pollycrystalline ice subjected to stress, deforms elastically, followed by a transient creep and finally, a secondary creep in the form of steady viscous flow is obtained.

As described in [1], the surface of ice Ih near the melting point has many dangling broken bounds that boost the presence of a liquid-like layer and as a consequence, low friction on such surface. Variation of density of ice with phase at 110 K is described in the table shown below.

Phase of ice	**Density (Mg/cm^3)**
Ih	0.93
II	1.18
III	1.15
IV	1.27
V	1.24
VI	1.33
VII	1.56
VII	1.56
IX	1.16
X	2.51

TABLE 1.1: Variation of ice density for every phase at 110K.

1.2.4 Freezing phenomena

Time-temperature diagram for freezing of pure water (ABCDE) and aqueous solutions (AB'C'D'E'), Fig. 1.7, shows the physical process that occurs during the solidification. The first stage, from A to B, belongs to undercooling, also called supercooling, and it is arisen below the freezing point T_f, which is equal to the melting point, T_m. This point is referred to a non equilibrium point and it is analogous to an activation energy necessary for the nucleation process. Before nucleation process starts, pure water may need to be cooled down several degrees. At point B, the system nucleates and releases its latent heat faster than the heat which is being removed from itself.

From C to D, the horizontal axis shows the evolution of the crystal growth in time. At C, there exists the nucleation point and, from there through D, latent heat gets removed out of the system at constant temperature. In this way, the mixture, which is in a partially frozen state, does not cool until all the potentially freezable water has crystallized.

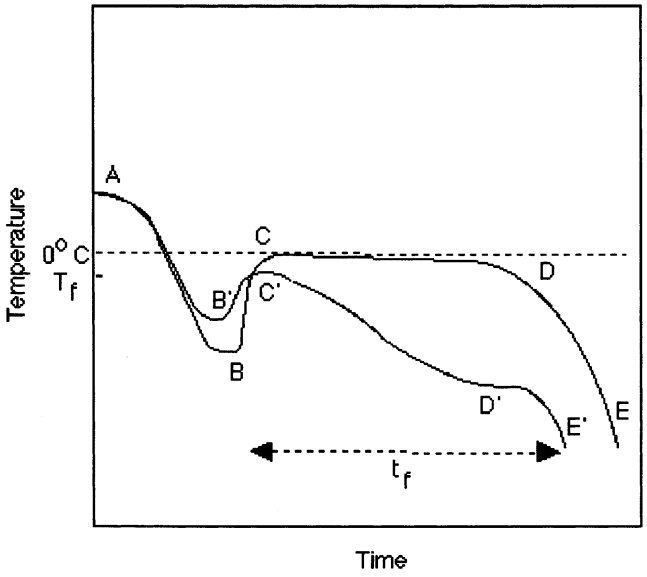


FIGURE 1.7: Process of crystallization of water.

1.3 Mechanisms of Heat Transfer. Heat convection

In a more generalistic point of view, phase changes occur due to the action of the heat transfer which may appear in the form of convection, conduction or radiation.

Convective heat transfer usually occurs in fluids due to the microscopic motion of the particles commonly understood as the bulk fluid motion.

Depending on the force originating such motion, one can distinguish two types of convection: natural convection and forced convection. Natural convection, the phenomena presented in this thesis, is originated due to differences of temperature gradients. These gradients, in the presence of the gravitational field, allow density to change within the fluid field. At the same time, the fact that density changes, enhances the rise of the colder liquid in contrast with the warmer which tends to sink thus, generating a motion in the fluid. On the other hand, the forced convection is driven by external sources which enforce the motion. In this case, buoyancy is not that relevant as it is for the first kind of convection.

In order to model the convection heat transfer in an object, the Newton's cooling law is typically used. As presented in Equation 1.1:

$$\frac{dQ}{dt} = hA\left(T_s(t) - T_\infty\right) \tag{1.1}$$

where Q is the heat source (thermal energy), T_s is the temperature on the surface of the object, T_∞ the temperature far away from the object, h is the heat transfer coefficient and A the heat transfer surface area.

The equation 1.1 states that the rate of heat loss is proportional to the temperature difference between the object itself and the medium by which is surrounded.

The process of water freezing in enclosures is common in engineering. When there exist temperature gradients within the liquid phase in the process of solidification, a natural buoyancy driven flow is initiated and such behavior is determined to affect the shape of the liquid/solid interface as well as the progress of solidification.

Indeed, these temperature differences in the liquid cause density variations so that the natural motion occurs. Boussinesq approximation can be validly used for fluids whose density varies linearly with temperature. However, pure water exhibits a maximum in its density when it ranges between 0ºC and 4ºC. Beyond the latter temperature, and known as density inversion point, density decreases in a nonlinear manner as the temperature passes through the freezing point. In convective heat transfer, surroundings of the temperature where the aforementioned maximum happens to be, behave in a complex manner leading to fully control the process of growth of the solid phase.

1.4 Conjugate Heat Transfer. Heat conduction

The other heat transfer mechanism appearing in the thesis is heat conduction which usually happens at molecular level in where the energy is transferred from particles with high energetic levels to lower energetic particles. The equation driving the conductive heat transfer is commonly known as the Fourier's law and it reads as:

$$q = -k\nabla T \tag{1.2}$$

where q is the heat transfer per unit of area, κ is the thermal conductivity of the material and ∇T is the gradient of temperatures within the studied bodies.

Alongside, conjugate heat transfer, the phenomena studied in the last part of this thesis, is referred to the heat transfer between solids and fluids. Therefore, a combination of conduction and convection are the main heat transfer mechanisms driving this part of the analysis.

Chapter 2

Numerical Methods for Phase Change Phenomena

2.1 State of Art. Numerical Methods

Considering the PCM density as constant in the model might be thought as a reasonable assumption in some cases, in others where thermo-mechanical coupling between the fluid and its container is intended, it makes impossible to account for some physical behaviors which may result from expansion or contraction during the phase change of the material. However, the main goal of this thesis is not to present a method that represents thermo-mechanical coupling but a technique that ensures volume expansion due to density changes through the fluid domain. To reach this point, it is important to summarize some of the numerous researches that have been conducted in order to investigate the problem of solidification.

At the present, the main used numerical methods representing the treatment of liquid-solid phase change are divided into these categories:

- **Front tracking method**,

  Volume-of-fluid method,

  Level set method,
- **Enthalpy method**,
- **Phase field method**.

2.1.1 Front tracking method

Several studies are carried out with this method. Juric et al. [11], presented a front-tracking method based on a finite difference approach of the heat equation and an explicit tracking of the fluid-solid interface to simulate time dependent two-dimensional dendritic solidification of pure substances. In similar fields, Al-Rawahi et al. [17] underwent also simulations of dendritic growth of pure substances by using front-tracking methods in which the fluid-solid interface was tracked explicitly and the release of latent heat during solidification was calculated with the normal temperature

gradient near the interface. Accordingly, Garimella et al. [14] proposed an explicit interface-tracking scheme involving reconstruction and advection of the moving interface in a fixed grid to solve moving-boundary problems associated with phase change phenomena. As they describe, the movement of the interface is tackled first by advection and tracking of the interface, later by the calculation of normal velocities near the interface region and finally, by solving the governing equations for the existing phases.

Volume-of-fluid method

Initially introduced by Harlow et al. [7], a technique called the marker and cell method tracked the interface by wightless particles which were transported convectively by the velocity of the fluid. Cells that were filled with marked particles were considered occupied by the fluid while, contrarily, these which were not filled with marked particles were not occupied by fluid. Later in time, the idea was extended to track the interface based on phase fractions in the volume-of-fluid method which is discussed in detail below.

Level set method

In the field of the current technique, Tan et al. [21] conducted a level set method combining properties of both fixed domain and front-tracking methods to model the microstructure evolution in multi-component alloy solidification. Phase interface is tracked by solving the multi-phase level set equations. From this tracked interface, a diffused one is constructed by means of the level set functions. Volume-averaging methods are latter used to solve energy, species and momentum equations. Rauschenberger et al. [16] pursued a comparative assessment between a Level set approach and a volume-of-fluid method to track interfaces in the context of dendritic ice growth in supercooled water. The Level Set method is used as an implicit tracking of the moving boundary.

2.1.2 Enthalpy method

In 2004, Esen et al. [6] worked out an enthalpy method based on finite difference approximations applied to the Stefan problem. An enthalpy function is defined representing the total heat content per unit of mass of the material. The need of tracking the interface between the fluid and the solid phase is thereby removed when using such formulation. El Ganaoui et al. [5] presented an enthalpy-porosity formulation on a fixed grid framework for liquid-to-solid phase transition. The method is extended to solve time-dependent solutal convection in the melt during directional solidification that undergo the majority of alloys. Within the alloy research field, Voller et al. [24] developed an enthalpy fixed grid method for dendritic growth modeling in under-cooled binary alloys. This method is devoted to couple explicit finite differences expressing the conservation of enthalpy and solute

to an iterative scheme which enforces node-to-node consistency between solute, liquid-fraction, enthalpy and under-cooling interface.

2.1.3 Phase field method

Emerged as an approach to model and predict mesoscale morphological and microstructure evolution in materials, Chen et al. [4] review some phase-field models used to describe various materials processes including solidification, crack propagation and dislocation microstructures among others. This paper describes the capability of phase-field methods to predict the evolution of arbirtary morphologies and complex microstructures without explicitly tracking the evolution of the interface.

2.2 Solidification methods

The challenge of a numerical investigation of a solidification process is to capture the free surface for the flow of the phase change material and, at the same time, account for the moving boundary induced by the phase change within the PCM. The free surface may be handled by the volume-of-fluid (VOF), originally introduced by Hirt and Nichols [8]. VOF relies on the definition of a transport indicator function within the finite volume method's framework.

Simultaneously, and in order to account for the phase changes, some of the used models are based on meso-scale. This is the phenomena occurring between microscopic and continuum length scales and, in the current context, the complex micro structure generated during the solidification is approximated as liquid, mushy (intermediate state), and solid regions. Mushy region is thereby described as an averaged value of the liquid and solid properties.

One of the most used methods is the enthalpy-porosity technique, originally developed by Voller and Prakash [23], which uses the typical conservation equations on a fixed Eulerian grid. The main concepts underlaying such method are: on the one side, an additional source term to the energy conservation equation is applied to describe the release of latent heat. On the other side, the solidification effects on the mass transport are modelled as a porosity variable and this is introduced as a Darcy-type source term to the momentum equation.

In the first aim of this thesis is the coupled use of the enthalpy-porosity technique with the VOF method. Some of the studies found on this topic, the coupling of both VOF and enthalpy-porosity methods, are mainly related to casting processes. Rösler and Brüggermann [19] introduced a numerical model for a solid-liquid phase change inside a latent heat thermal energy storage. Richter et al. [**richter_turnow_kornev_hassel_2016**], worked out a method for the simultaneous moult filling and solidification process which settles the developing of free surface flow and the liquid-solid phase transition under the volume-of-fluid and enthalpy-porosity methods.

However, no adaptation of these methods to purely solidification processes has been found. Therefore, and, the objectives of the first stage of the research are:

- To introduce a new solver based on the coupling of VOF and enthalpy-porosity techniques which covers the relevant physical effects during the process of solidification.
- To validate simulation results by using benchmark cases found in the literature.

Numerical methods commented here are deeply described next.

2.2.1 Volume-of-Fluid Method: General Aspects

The Volume-of-fluid method (VOF) is a numerical method based on an Eulerian approach to track the free surface in a two-phase flow. The VOF method, developed by Hirt and Nichols in 1981, [8], takes relevance when fluids coexist with other phases. An example could be the ice (solid phase) advancing front within the liquid phase. The surface in between both phases needs to be solved by means of the volume of fluid technique.

This is sometimes seen as the conservation of the mixture components along the path of a fluid region. The equation which allows that is described as:

$$\frac{\partial \alpha_{\text{phase}}}{\partial t} + \frac{\partial \left(\alpha_{\text{phase}}\, u_j\right)}{\partial x_j} = 0 \tag{2.1}$$

In which α_{phase} corresponds to the phase fraction and it applies:

$$\alpha_{phase} = \begin{cases} 0 & = \text{ solid PCM} \\ 0 < \alpha_{phase} < 1 & = \text{ cell contains the interface} \\ 1 & = \text{ liquid PCM} \end{cases} \tag{2.2}$$

As Eq. 2.1 exposes, the principle that lies behind the method is the definition of the phase field (α), Eq. 2.2, which has a value between '0' and '1'. The value of '1' corresponds to any point filled with fluid and zero otherwise. Thus, the average value of α in a cell indicates the fractional volume of that cell occupied by the fluid. Consequently, if a cell has an average value of $\alpha = 1$ implies a fully filled cell of fluid and oppositely, a value of $\alpha = 0$ means that the fluid is not present in the cell. However, a cell presenting an average value between 0 and 1 would lead the presence of an interface in that region as it is clearly seen in Fig. 2.1.

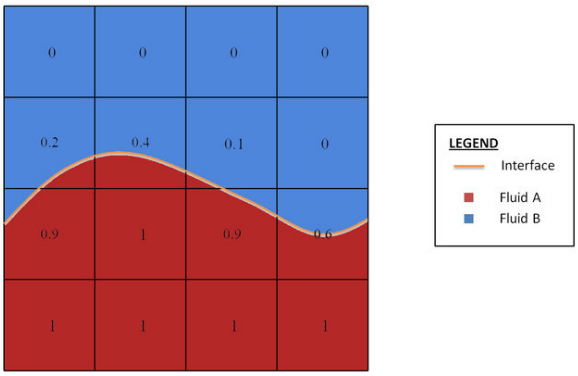


FIGURE 2.1: Volume-of-fluid approach.

Near the interface, it clearly exists a jump in the fluid properties that need to be corrected by properly averaging phase properties in that region.

In the present study, this method is used in conjunction with other techniques to carry out some cases undergoing phase-transition phenomena.

2.2.2 Enthalpy-Porosity Model. Governing Equations

The first technique implemented is the Enthalpy-porosity method. Here, the energy equation takes center stage.

The energy equation based on the enthalpy formulation for convective-diffusive heat transfer states that,

$$\frac{\partial \rho h}{\partial t} + \frac{\partial}{\partial x_j}\left(\rho u_j h\right) = \nabla \cdot \left(k_i \nabla T_i\right) \tag{2.3}$$

where u is the velocity component and κ_i is the thermal conductivity of the fluid. h can be expressed as a function of its latent heat and the specific sensible parts,

However, the enthalpy-porosity method describes the enthalpy h of the mixture by its sensible part and the latent heat of solidification. The release of the latent heat is dependent on the stage of the phase change, and must be

restricted to the phase change material.

$$h = \int_{T_r}^{T} c_p dT + \alpha_\ell L \tag{2.4}$$

where the latent heat is is driven by the evolution of the liquid α_l. The phase transition is modelled by expressing the liquid volume fraction as a function of the temperature,

$$\alpha_l = \begin{cases} 1 & T > T_{liq} \\ \frac{T - T_{\text{sol}}}{T_{\text{liq}} - T_{\text{sol}} + \varepsilon} & T_{\text{sol}} < T < T_{liq} \\ 0 & T < T_{\text{sol}} \end{cases} \tag{2.5}$$

For seek of brevity on the following expressions, it is adapted the term γ_{phase} to γ_l If expression 2.5 is replaced in 2.4,

$$\begin{aligned} &\frac{\partial \left(\rho c_p T + \gamma_l \alpha_l L\right)}{\partial t} + \frac{\partial \left(\rho u_j c_p T + u_j \gamma_l \alpha_l L\right)}{\partial x_j} \\ &= \nabla \cdot (k_i \nabla T_i) \end{aligned} \tag{2.6}$$

Rearranging terms, it yields the complete energy equation with the sensible and latent heat parts for the correct representation of solidification processes.

$$\begin{aligned} \frac{\partial(\rho C_p T)}{\partial t} + \nabla \cdot \left(u_j \rho C_p T\right) + L \left[\frac{\partial(\rho \alpha_l \gamma_l)}{\partial t} + \frac{\partial(u_j \rho \alpha_l \gamma_l)}{\partial x_j}\right] &= \nabla \cdot (k_i \nabla T_i) \\ S = -L \left[\frac{\partial(\rho \alpha_l \gamma_l)}{\partial t} + \frac{\partial(u_j \rho \alpha_l \gamma_l)}{\partial x_j}\right] & \end{aligned} \tag{2.7}$$

The momentum equation is discussed in detail in the sub-chapter *Interphase porosity models*.

2.2.3 Lee model

The second technique implemented in this thesis is based on the Lee model.

The Lee model is based in the liquid-vapour mass transfer. Governed by the vapour transport equation 2.8, this model is applicable during melting or solidification of a fluid.

$$\frac{\partial}{\partial t}\left(\alpha_i \rho_i\right) + \nabla \left(\alpha_i \rho_i u_i\right) = S_{m_i} \tag{2.8}$$

ρ_i and u_i are the fluid density and fluid velocity of the *ith* phase. Moreover, S_{m_i} is the mass source which takes on a zero value at the interface.

During melting, $T_l > T_{sat}$,

$$\frac{dm_{sl}}{dt} = C_f \rho_s \alpha_s \left(\frac{T_s - T_{sat}}{T_{sat}} \right) \tag{2.9}$$

During solidification, $T_l < T_s at$

$$\frac{dm_{ls}}{dt} = C_f \rho_l \alpha_l \left(\frac{T_{sat} - T_l}{T_{sat}} \right) \tag{2.10}$$

The coefficient C_f might be interpreted as a time rate and must be empirically tunned. Its magnitude is expressed in $\frac{1}{s}$. α represents the phase volume fraction. $\frac{dm_i}{dt}$ are the mass transfer rates from one phase to another. The subscripts "s", "l", refer to solid and liquid phases respectively. T_{sat}, is the phase transition temperature which, in case of pure water would be 273.15 K. The source term of 2.8 is then calculated as,

$$S_{m_i} = \begin{cases} \frac{dm_{sl}}{dt} - \frac{dm_{ls}}{dt}, & \text{for water phase} \\ \frac{dm_{ls}}{dt} - \frac{dm_{sl}}{dt}, & \text{for ice phase} \end{cases} \tag{2.11}$$

2.2.3.1 Momentum Equation

In the momentum equation, the flow is modelled as,

$$\begin{aligned} &\frac{\partial \left(\rho u_i\right)}{\partial t} + \frac{\partial \left(\rho u_i u_j\right)}{\partial x_j} \\ &\quad = -\alpha_i \nabla p + \frac{\partial}{\partial x_j} \left(\mu \frac{\partial u_i}{\partial x_j} \right) + F_{\sigma i} + S_{u_i} \end{aligned} \tag{2.12}$$

The source term for the momentum equation can be written as,

$$S_{u_i} = \begin{cases} \frac{dm_{sl}}{dt} u_l - \frac{dm_{ls}}{dt} u_s, & \text{for water phase} \\ \frac{dm_{ls}}{dt} u_s - \frac{dm_{sl}}{dt} u_l, & \text{for ice phase} \end{cases} \tag{2.13}$$

where u_l and u_s are the liquid and solid velocity components accordingly.

The source terms related to interphase porosity (2.29) may be added to the momentum equation presented here for the Lee model 2.12.

2.2.3.2 Energy Equation

The energy equation for the Lee model can be described as,

$$\frac{\partial(\rho C_p T)}{\partial t} + \nabla \cdot \left(u_j \rho C_p T \right) = \nabla \cdot \left(k_i \nabla T_i \right) + S_{H_i} \tag{2.14}$$

where the heat source term due to mass transfer in the energy equation is calculated as,

$$S_{h_i} = \begin{cases} \frac{dm_{sl}}{dt} H_{L,} & \text{for water phase} \\ \frac{dm_{ls}}{dt} H_L & \text{for ice phase} \end{cases} \tag{2.15}$$

where H_l is the latent heat induced by the phase transition and k_i, the thermal conductivity.

2.2.3.3 Classical nucleation theory. The coefficient C_f.

The coefficient C_f that appears on Equations 2.9 and 2.10 is computed accordingly to the work of Huang et al. [9]. In these work, the Lee model is used and the nucleation rate is introduced for the calculation of mass transfer rate between phases.

The concept behind the *Classical Nucleation Theory*, CNT, as described in [10] resides in the idea of droplet freezing. This is initiated in the fluctuation of molecules of a supercooled liquid due to thermal vibration which lead, at its turn, to spontaneous formation of ordered solid molecule clusters (ice embryos). The size of these embryos oscillates as individual water molecules are crystallized or lost from the liquid phase. When the size of the embryo reaches a critical value, it leads a faster and auspicious thermodynamic joining of further water molecules to the crystal lattice. This means the critical embryo enhancing the "parent phase", supercooled liquid, to undergo a macroscopic phase transition: droplet freezing.

And this is what CNT aims to describe; the freezing process in terms of temperature-dependent nucleation rate by joining two components: thermodynamic and kinetic. These components, briefly described in the following chapters, are based on the theory found at Lai et al. [25] and Huang et al. [9].

As a remark, in this thesis a brief introduction of this theory is given. However, for further details on the assumptions used refer to the literature.

Thermodynamic component

This thermodynamic component seeks for the number of critical embryos formed per unit of volume at a specific temperature. A decrease in the enthalpy, and consequently a change in Gibbs free energy required to form an ice embryo containing water molecules generates an energy barrier to nucleation. However, for ice embryo formation, this barrier needs to be overcome.

$$\Delta G_c = \underbrace{\Delta G_V}_{\text{volume term}} + \underbrace{\Delta G_S}_{\text{surface term}} \tag{2.16}$$

where the volume and surface terms decomposed,

$$\Delta G_c = -\frac{4}{3} \cdot \frac{\pi r^3}{\Omega} \cdot \Delta g_v + 4\pi r^2 \gamma_{sf} \tag{2.17}$$

where r is the radius of a simplified spherical embryo, γ_{sf} the interfacial tension between phases Ω the volume of a single molecule $(\Omega = V_{m,w}/N_A)$, $V_{m,w}$ is the molar volume and Δg_v represents the decrease in volume of the Gibbs free energy of a molecule and is defined as:

$$\Delta g_v = \frac{\Delta_m H_1}{N_A} \frac{\Delta T}{T^*} \tag{2.18}$$

where $\Delta_m H_1$ is the molar latent heat of crystallization, N_A is the Avogadro's number, T^* is the freezing temperature and $\delta T = T^* - T$, the degree of supercooling. The radius has an influence on the change in Gibbs free energy. This is when:

- $r < r_{crit} \Rightarrow \Delta G_c > 0 \;||\; \Delta G_c \uparrow \;\Rightarrow\; r \uparrow \Leftarrow$ endothermic process
- $r > r_{crit} \Rightarrow \Delta G_c < 0 \;||\; \Delta G_c \downarrow \;\Rightarrow\; r \uparrow \Leftarrow$ exothermic process

The critical radius exists when the global enthalpy variation gets negative.

By differentiating Eq. **??** and setting $\frac{d(\Delta G_c)}{dr} = 0$, the critical radius is defined as:

$$r_{crit} = \frac{2\gamma_{sf} T^* V_{m,w}}{\Delta_m H_1 \Delta T} \tag{2.19}$$

Then, if subsituting Eq. 2.18 and 2.19 in Eq. 2.17, it is obtained the energy barrier:

$$\Delta G_{crit} = \frac{16\pi}{3} \cdot \frac{\gamma_{sf}^3 V_{m,w}^2 T^2}{\Delta_m H_1^2 \Delta T^2} = \frac{1}{3}\left(4\pi r_{crit}^2 \gamma_{sf}\right) \tag{2.20}$$

In Huang et al. [9], the expression concerning the variation of Gibbs function for the phase change does not include the molar volume of water but a shape coefficient of nucleation. It involves the influence of the contact angle when going from a uniform state to an inhomogeneous one. This shape factor is defined as:

$$\alpha_{ey} = \frac{2 - 3\cos\theta + \cos\theta^3}{4} \tag{2.21}$$

Temperature and saturation dependent number of ice embryos per unit volume of water may be expressed in a Boltzmann distribution form using ΔG:

$$N_{\text{embryo}}\left[\text{m}^{-3}\right] = N_1 \cdot \exp\left(-\frac{\Delta G}{k_B T}\right) \tag{2.22}$$

where N_1 is a volume-based number density of water molecules in the liquid phase.

Kinetic component

The kinetic part of the nucleation rate is introduced in the form of water molecules flux. This is expressed as a Boltzmann distribution such that:

$$\Phi = \frac{k_B T}{h} \cdot \exp\left(-\frac{\Delta g}{k_B T}\right) \tag{2.23}$$

where h is the Planck's physical constant, and Δg the activation energy for the transfer of a water molecule across the phase boundary.

The rate at which the water molecules are transferred into an ince embryo is defined as:

$$K = n_s \cdot 4\pi r^2_{\text{embryo}} \cdot Z \cdot \Phi \tag{2.24}$$

where n_s is the number of molecules and $4\pi r^2_{\text{embryo}}$ is the surface area of the critical embryo and Z a kinetic prefactor. For seek of simplification, the authors of the theory suggest that the product of these terms are close to unity. Thus, considering this change, the equation yields as:

$$K = \Phi \tag{2.25}$$

Nucleation rate

Combining the thermodynamic component Eq. 2.22 and the kinetic one 2.25, the formulation of the nucleation rate can be expressed as:

$$J_{\text{hom}}\left[\text{m}^{-3}\cdot\text{s}^{-1}\right] = \underbrace{K}_{\text{Kinetics}} \cdot \underbrace{N_1 \cdot \exp\left(-\frac{\Delta G}{k_B T}\right)}_{\text{Number of embryos}} \tag{2.26}$$

As a final step, inserting Eq. 2.25, which at its turn is equal to Eq. 2.23, into Eq. 2.26, the nucleation rate is expressed in the form of:

$$\begin{aligned} J_{\text{hom}}\left[\text{m}^{-3}\cdot\text{s}^{-1}\right] = &\frac{k_B T}{h} \cdot \underbrace{\exp\left(-\frac{\Delta g^{\#}}{k_B T}\right)}_{\text{diffusion of molecules effect}} \\ &\cdot N_1 \cdot \underbrace{\exp\left(-\frac{\Delta G}{k_B T}\right)}_{\text{nucleation effect}} \end{aligned} \tag{2.27}$$

Fig. 2.2 characterizes the variation of the crystallization rate in function of the temperature. In the image, the dotted line shows how the nucleation effect is 0 close to the cooling, and as the temperature values decrease, this lines tend to 1. Moreover, the dashed line shows how the effect of the diffusion of the molecules increases as the temperature does. This prompts out the ease of the embryo formation when at low temperatures since the molecules cannot overcome the energy barrier to enter the embryo. However, the crystal growth becomes harder. This may be seen as constant search of equilibrium among the nucleation and the crystal growth. Finally, the coefficient C_f that appears on Equations 2.9 and 2.10 as commented above, is defined for the Lee model as:

$$C_f = J_{\text{hom}} \cdot V_l \tag{2.28}$$

where V_l is the volume of water in each cell.

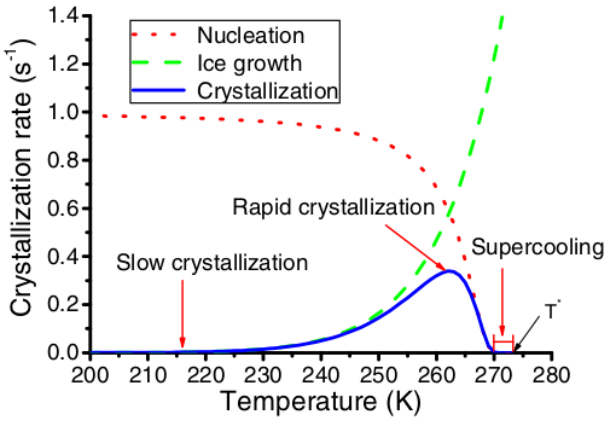


FIGURE 2.2: Crystallization rate versus temperature.

2.2.4 Interphase porosity models

Interphase porosity models add an aritificial momentum source over the interface between phases to compute the sink of velocity in the solidified region. Therefore, influencing the behavior of the physics during the process of solidification or melting.

The model implemented in OpenFOAM is *Voller Prakash method* [23], and it defines the source terms, S_y and S_z such that when along the fluid domain these terms take on a value of zero, the momentum equations are driven by the actual values of the velocities. On the other side, when it comes to treat the mushy region (i.e. porous region), the value of these source terms dominate convective, diffusive and transient terms and the momentum equation tends to approximate de Darcy law.

The two source terms as specified above,

$$\begin{cases} S_y = -Av \\ S_z = -Aw \end{cases} \tag{2.29}$$

Then, to specify a term for the function A, it is used the *Carman-Koseny equation*, which is derived from the Darcy law. The former expresses the gradient for the pressure as a combination of the velocity, $\mathbf{u}$, and the porosity, λ. The coefficient C depends on the morphology of the medium.

$$gradP = -\left(\frac{C(1-\lambda)^2}{\lambda^3}\right)\mathbf{u} \tag{2.30}$$

To avoid division by zero, q is added to the equations shown

$$A = -\left(\frac{C(1-\lambda)^2}{\lambda^3 + q}\right) \tag{2.31}$$

The source terms S_y and S_z in 2.29 are added in the Eq. 2.32 and 2.33. The

source term S_b corresponds to the body forces of the fluid and will be discused later on this thesis.

$$\frac{\partial(\rho v)}{\partial t} + \operatorname{div}(\rho \mathbf{u} v) = \operatorname{div}(\mu \operatorname{grad} v) - \frac{\partial P}{\partial y} + S_y \tag{2.32}$$

$$\frac{\partial(\rho w)}{\partial t} + \operatorname{div}(\rho \mathbf{u} w) = \operatorname{div}(\mu \operatorname{grad} w) - \frac{\partial P}{\partial z} + S_z + S_b \tag{2.33}$$

2.2.4.1 Surface tension model

The surface tension is only specified on a phase pair basis. In this version of OpenFOAM, it is present a constant model for a given σ.

Chapter 3

Numerical Simulation of Solidification Process

3.1 OpenFOAM. General Aspects

OpenFOAM is a free open-source software written in C++ and mainly conceived to perform computational fluid dynamics (CFD) simulations based on a finite volume discretization (FVM).

3.1.1 The finite volume method

Fluid equations usually take the form of non-linear partial differential equations and so, most of time, no analytical solution can be derived from them. In that context, different numerical techniques are employed to reach an approximation of the solution to these problems. These methods require a discretization of the domain in which the solution is going to be calculated. As aforementioned, OpenFOAM uses the finite volume method, which is, indeed, one of the most widely techniques used in computational fluid dynamics, and the one used in this thesis.

This technique turns the partial differential equations, which at their turn represent conservation laws over differential volumes, into discrete algebraic equations over finite volumes. Similarly to the finite element method, the FVM also needs a discretization of the geometric domain but in this numerical method, the elements used to integrate the algebraic equations representing the conservation partial differential equations are finite volumes or non-overlapping elements.

Some of the terms in the conservation equation are converted into face fluxes and evaluated in the discretized finite volumes. These face fluxes are strictly conservative. This is that the flux entering the volume is equal to the flux leaving the adjacent volume. This property makes the finite volume method the preferred technique for CFD [15].

Geometric domain discretization

The intrinsic properties of the finite volume method need the computational domain to be discretized in volume cells, known as control volumes (CV). Each one of these volumes has a centroid or computational point in which the solution is obtained.

Alongside with this idea, OpenFOAM follows a cell-centered approach in which the unknowns are defined at the center of these volumes or cells. The value of these are computed as an average value of the variable in that cell.

Moreover, the control volume is defined by the neighbours. This is, in the case the volume has an adjacent neighbour, an internal face is delimiting the separation of both. On the other hand, if the volume is not sharing a face with a neighour volume, the face is considered to be a boundary.

Fluid dynamic equations discretization

The continuity equation, the Navier-Stokes equations and, the heat equation stated in section 2 can be stated in a more general form under the formulation of the Reynolds transport theorem:

$$\underbrace{\int_{V_P} \frac{\partial \rho\phi}{\partial t} dV}_{\text{Temporal term}} + \underbrace{\int_{V_P} \nabla \cdot (\rho\vec{u}\phi) dV}_{\text{Convective term}} = \underbrace{\int_{V_P} \nabla \cdot \left(\rho\Gamma_\phi \nabla\phi\right) dV}_{\text{Diffusive term}} + \underbrace{\int_{V_P} S_\phi dV}_{\text{Source term}} \tag{3.1}$$

where V_P is the control volume cell, ϕ may be any scalar or vectorial variable of the continuum, Γ_ϕ is the diffusivity of the variable and S_ϕ is a source term.

In order to recover the continuity, momentum and energy equations, the parameters shown in table 3.1 need to be shaped in the transport equation.

Equation	ϕ	Γ_ϕ	S_ϕ
Continuity	1	0	0
Momentum	$\vec{u}u$	ν	-∇p
Energy	C_pT	κ	0

TABLE 3.1: Parameters to recover continuity, momentum and energy equations.

The fluid variable is defined as a ratio of itself integrated along the volume cell. Thus, it yelds the following form,

$$\phi = \phi_P = \frac{1}{V_p} \int_{V_P} \phi(x) dV \tag{3.2}$$

Therefore, a complete discretization of the previous terms is needed to solve the physics regarding a general fluid dynamics problem.

3.1.2 OpenFOAM functioning

In this first section, a brief introduction on the structure and functioning of the OpenFOAM software is given.

In the folder structure tree shown in Fig. 3.1, it is shown a typical case setup for a phase change problem using *icoReactingMultiphaseFoam* solver.

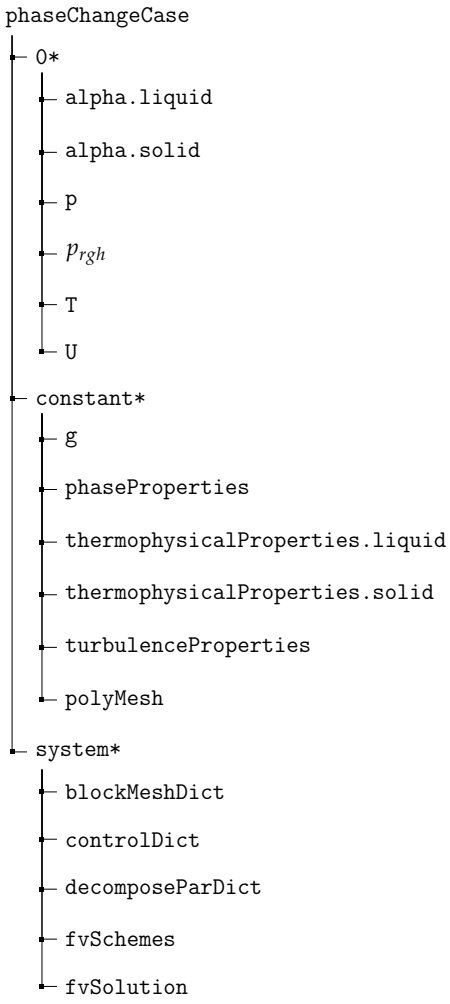


FIGURE 3.1: General structure of an OpenFOAM case.

3.1.2.1 Boundary Conditions Directory

The "0" directory gathers all the boundary conditions at time zero and the initial conditions to set up the case. As the simulation starts running, the information of these fields is saved in folders at every timestep.

3.1.2.2 Constant Properties Directory

The "constant" directory contains all the information typically regarding the physical properties which are kept constant through the simulation. Moreover, once the dictionary *blockMeshDict* is run, OpenFOAM creates a folder

called *polyMesh* containing all the information relevant to the mesh (points, faces,...).

3.1.2.3 System Directory

This folder contains the files required by the control of the solver and the solution itself. The most common files are:

- **blockMeshDict:** in this file the parameters required to build up the computational domain, the mesh and the boundaries are found. The command **blockMesh** executes this dictionary creating the *polyMesh* folder commented above.
- **controlDict:** Time parameters associated to the computation are set in this file.
- **decomposeParDict:** In the realization of this thesis, the help of parallel computing is required. Thus, in this file, parameters regarding the decomposition of the mesh are configured. It is executed by means of the **decomposePar** appliation implicit in OF. The mesh is afterwards reconstructed by using **reconstructPar**
- **fvSchemes:** Schemes selected for the discretization of the derivative terms are defined. Among others, time schemes, gradient schemes, laplacian schemes, divergent schemes, interpolation schemes can be declared here.
- **fvSolution:** contains sub-dictionaries used to control the solvers and the solution algorithms. It also allows the definition of the fields resolution.

3.2 Solidification process. Methodology

A convection solver is used to represent the flow behavior generated by the density difference due to existing temperature gradients whithin the volume of control. A polynomic water density is implemented in the native OpenFOAM solver and compared with the standard Boussinesq approximation. Besides, a proposed buoyancy term by Bourdillon [3] is added in the computation of the momentum equation. The current model is validated against numerical results from the literature. The solution of this convection solver is later used as initial conditions, before solidification phenomena plays a role.

For the solidification phenomena representation, the aforementioned Enthalpy-porosity technique and Lee model based on the *Classical Nucleation theoy* are implemented within a multi-phase native solver. These methods are compared against [3], Kowalesky et al. [12] and Chen et al. [**chen_lee_1998**]. A final remark on the *classic Stefan problem* is done. This part will be discussed later in this thesis.

3.3 OpenFOAM: BuoyantBoussinesqPimpleFOAM. Natural Convection solver

In a natural convection environment, the motion of the fluid is mainly driven by the density difference within the fluid volume of control. At its turn, the differences in the density, responsible for buoyancy forces, are generated by the existing temperature gradients. Within a physical context, the fluid near a hot heat source gets warmed up and, as a result, it becomes less dense moving up inside a domain. Consequently, the fluid in contact of the cold heat source is pushed from its zone to replace the hot fluid location. At this point, the cycle starts again repeating the phyisical phenomena.

3.3.1 Case Description

Within the context of natural convection, the current case aims to develop a comprehensive state of the capabilities that OpenFoam solvers bring to solve this phenomena. To reach the objective, and on purpose of controlling the physics generated on the simulation, a regular squared geometry of 0.038 *mm* side length is created:

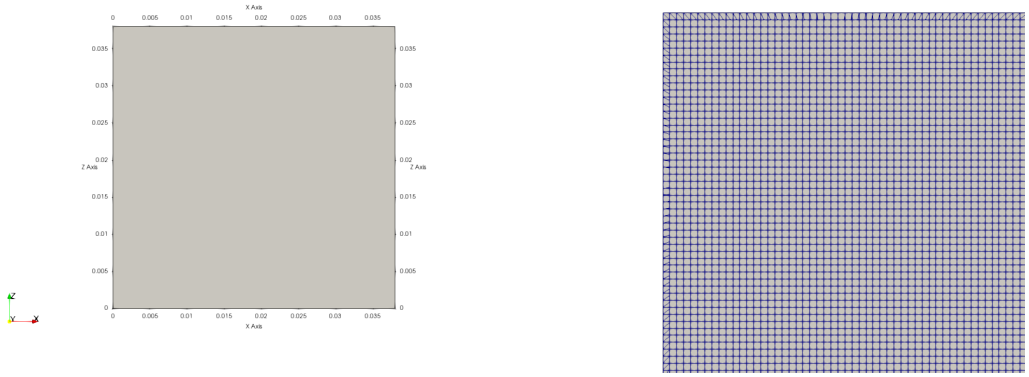


FIGURE 3.2: Geometric characteristics for the cavity.

The mesh is structured and consists of 1971212 nodes.

3.3.2 Hypotheses And Assumptions

To carry out the current problem, a series of assumptions are taken into account in order to simplify the solving of the fluid equations involved.

Laminar regime: The Reynolds number, computed from the maximum velocity is not high enough to consider turbulent effects.

Convective heat transfer: To determine whether the heat transfer is assumed to be convective, the Prandtl number and the Rayleigh number should be assessed.

The Prandtl number, as the relation between the viscosity and the thermal conductivity of a fluid or, in other words, the correlation between momentum transport and thermal transport capacity is calculated as:

$$\mathrm{Pr} = \frac{v}{\alpha} = \frac{\mu}{\rho\alpha} = \frac{\mu c_p}{\lambda} = \frac{\text{momentum transport}}{\text{heat transport}} \tag{3.3}$$

where μ is the dynamic viscosity, c_p is the specific heat and λ is the thermal conductivity.

Thus, a small Prandtl number are owned by free-flowing flows with high thermal conductivitiy.

On the other hand, the Rayleigh number is referred to the time scale relation between the diffusive and the convective thermal transports. It is thus used to determine wheter the buoyancy-driven natural convection plays an important role in the heat transfer. The dimensionless number is assessed in this context by this form:

$$\mathrm{Ra}_x = \frac{g \cdot \beta}{\nu \cdot \alpha} \cdot (T_s - T_{\mathrm{inf}}) \cdot x^3 \tag{3.4}$$

Being g, the gravity, β, the coefficient of thermal expansion, ν, the kinematic viscosity, α, the thermal diffusivity, and T_s and T_{inf}, the temperature on the wall surface and the temperature of the fluid far from the wall accordingly.

In the current case-scenario, a Prandtl close to 7 and a Rayleigh of 2517629 determine a convective heat transfer. The values used to estimate the Rayleigh number calculation are: $\beta = 6.734e-5K^{-1}$, $\nu = 1.003e-6m^2.s^{-1}$, $\alpha = 1.435e-7m^2.s^{-1}$, $T_s = 283K$, $T_{inf} = 273K$ and $x = 0.038m$. The values used for the laminar Prandtl number calculation are: $\mu = 0.001003Kg.m^{-1}.s^{-1}$, $\lambda = 0.6W.m^{-1}.K^{-1}$ and $C_p = 4182J.Kg.K^{-1}$.

Newtonian fluid: The viscosity of the fluid is assumed to be constant.

Thermophysical properties: specific heat, C_p, the thermal expansion coefficient, β, thermal conductivity, κ, kinematic viscosity, ν are assumed to be non-dependent of temperature. However, the density will be dependent of temperature so as it plays an important role in the buoyancy effects through the later explained in this section.

The conservative equations used to describe the motion of the fluid along time and space are described next.

3.3.3 Governing Equations

In this section, the governing equations for the used solver are described first.

The conservation of mass states that the mass flowing into the volume of control (CV) must be equal to the mass flowing out of such volume.

$$\frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = 0 \tag{3.5}$$

3.3.3.1 Momentum Equation

Throughout the CV the momentum of the fluid flow is preserved and here below it is expressed for the y-direction and z-direction.

$$\frac{\partial(\rho v)}{\partial t} + \operatorname{div}(\rho \mathbf{u} v) = \operatorname{div}(\mu \operatorname{grad} v) - \frac{\partial P}{\partial y} \tag{3.6}$$

$$\frac{\partial(\rho w)}{\partial t} + \operatorname{div}(\rho \mathbf{u} w) = \operatorname{div}(\mu \operatorname{grad} w) - \frac{\partial P}{\partial z} + S_b \tag{3.7}$$

where in the case of the *Boussinesq approximation* where the density variation is linear:

$$S_b = g \cdot \rho_r [1 - \beta(T - T_r)] \tag{3.8}$$

in the case of the implemented polynomial density which accounts for the inversion point as in [3]:

$$S_b = g \cdot [\rho_r - \rho(T)] \tag{3.9}$$

where the polynomial expression from ρ is:

$$\begin{aligned} \rho(T) = {} & 999.840281167108 + 0.0673268037314653 \times T \\ & - 0.00894484552601798 \times T^2 \\ & + 8.78462866500416.10^{-5} \times T^3 - 6.62139792627547.10^{-7} \times T^4 \end{aligned} \tag{3.10}$$

As it will be pointed out later, the native solver uses the Boussinesq approximation to account for the buoyancy effects. However, this linear assumption is only valid as the density variations meet:

$$\frac{\Delta\rho}{\rho_r} << 1 \tag{3.11}$$

Therefore, to account for the inversion points present during the freezing process, a density variation like the described in Eq. 3.10 is implemented in the solver.

3.3.3.2 Temperature Equation

The temperature equation representing the convection phenomena yields as:

$$\frac{\partial T}{\partial t} + \frac{\partial(u_j T)}{\partial x_j} = \frac{\partial}{\partial x_j}\left(\gamma \frac{\partial T}{\partial x_j}\right) \tag{3.12}$$

where the thermal diffusivity, γ, is defined as:

$$\gamma = \frac{\lambda}{\rho_r c_p} \tag{3.13}$$

All these equations are regarded by the solver *buoyantBoussinesqPimpleFoam*.

3.3.4 Solver descripton. Control Loop

The *buoyantBoussinesqPimpleFoam* is a solver used to solve non-steady buoyancy-driven fluids by using the Boussinesq approximation as a coupling between density and temperature fields. It considers the fluid as incompressible and uses the PIMPLE algorithm for the pressure-velocity coupling. The flowchart of the integration procedure for the presented solvers *buoyantBoussinesqPimpleFoam* and *icoReactingMultiphaseinterFoam* is presented below:

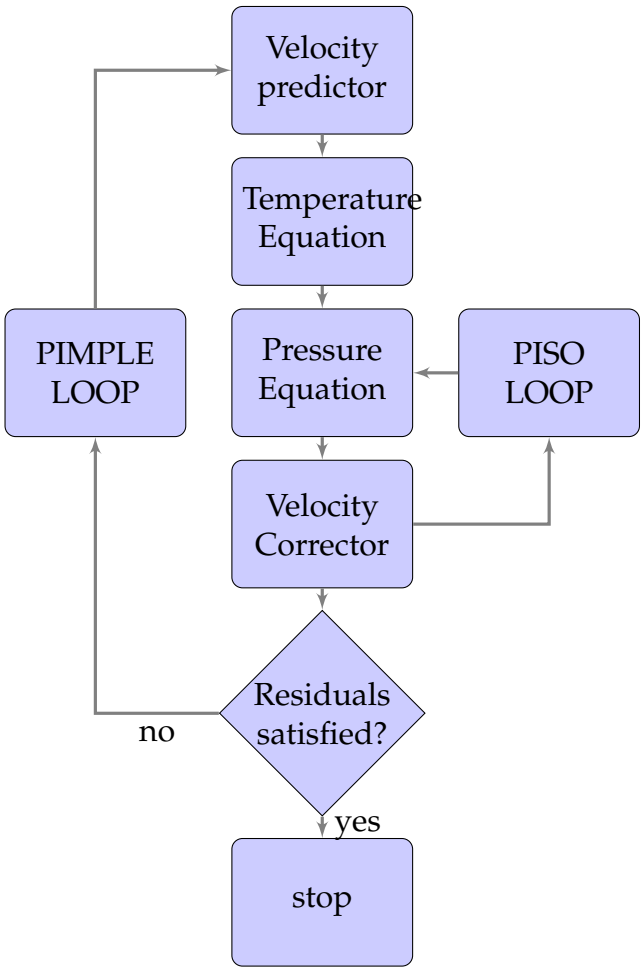


FIGURE 3.3: Flowchart of integration procedure. *buoyantBoussinesqPimpleFoam*

3.3.5 Code implementations

As described in the *Governing equations* section, the need for a polynomial density expression and a variation of the momentum source terms devoted to reflect the buoyancy effects is derived.

To do so, a new equation of state is implemented within the OpenFoam framework. Now and, in order to take into account this bouyancy forces, the pressure equation is studied. This is beacause in the context of a pressure-velocity corrector scheme, and in the case of ensuring stability and simplifying the boundary conditions definition, the modified pressure, p_{rgh}, within the pressure equation implementation, is the term that accounts for the gravity terms.

Here, it is presented a general form of a momentum equation with the continuity equation corresponding to a incompressible flow.

$$\begin{cases} \frac{\partial(\rho\mathbf{v})}{\partial t} + \nabla \cdot (\rho\mathbf{v} \otimes \mathbf{v}) = -\nabla p + \nabla \cdot (\mu(\nabla\mathbf{v} + \nabla\mathbf{v}^T)) \\ \nabla \cdot \mathbf{v} = 0 \end{cases} \tag{3.14}$$

From this general equation, it will be given the term $H(\mathbf{u})$, as later on will be needed for the pressure equation calculation.

Therefore, this term comes from considering the linearization of the advective term under the assumption of small Courant numbers (Co < 1). Leading the term $\mathbf{v}^0 \cong \mathbf{v}$.

$$\begin{aligned} \int_\Omega \nabla \cdot \left(\mathbf{v} \otimes \mathbf{v}^0\right) d\Omega &\cong \sum_f \mathbf{v}_f \mathbf{v}_f^0 \cdot \mathbf{S}_f \\ &= \sum_f F^0 \mathbf{v}_f \\ &= a_P \mathbf{v} + \sum_f a_N \mathbf{v}_N \end{aligned} \tag{3.15}$$

$$a_P \mathbf{v}_P = \mathbf{H}(\mathbf{v}) - \nabla p \tag{3.16}$$

$$\mathbf{H}(\mathbf{v}) = \underbrace{-\sum_f a_N \mathbf{v}_N}_{\text{Diagonal term}} + \underbrace{\frac{\mathbf{v}^0}{\Delta t}}_{\text{Off-diagonal term}} \tag{3.17}$$

where $\mathbf{v}^0$ is the velocity at previous time-step and F^0 is the face flux at the previous time-step.

In addition, by discretizing the continuity equation, it is possible to get the final form of the pressure equation.

So as to give stability to the solution and to simplify the boundary conditions definition as described in Berberovic et al. [2], a modified pressure is defined

as,

$$p_{rgh} = p - \rho_r \mathbf{g}\cdot\mathbf{x} + \rho(T)\mathbf{g}\cdot\mathbf{x} \tag{3.18}$$

being, the pressure gradient the next expression,

$$-\nabla p + \rho_r \mathbf{g} = -\nabla p_{rgh} - \mathbf{g}\cdot\mathbf{x}\nabla\rho_r + \mathbf{g}\cdot\mathbf{x}\nabla\rho(T) + \rho(T)\mathbf{g} \tag{3.19}$$

and rearranging terms,

$$-\nabla p + \rho_r \mathbf{g} + \mathbf{g}\cdot\mathbf{x}\nabla\rho_r - \mathbf{g}\cdot\mathbf{x}\nabla\rho(T) - \rho(T)\mathbf{g} = -\nabla p_{rgh} \tag{3.20}$$

If one tries to describe the discretized pressure equation in *buoyantBoussinesqPimpleFoam*, there is a first term called **phig**, which is,

$$\Phi_f^{\nu+1} = \Phi_u^{\nu+1} - \left[(\mathbf{g}\cdot\mathbf{x})_f \left(\nabla {\rho_r}^{n+1}\right)_f + (\mathbf{g}\cdot\mathbf{x})_f \left(\nabla\rho(T)^{n+1}\right)_f\right] \frac{|\mathbf{S}_f|}{(a_P)_f} \tag{3.21}$$

A face flux calculated by the term **H(v)**, appearing in equation 3.17

$$\Phi_u^{\nu+1} = \Phi_f^{\nu+1} + \left(\frac{H(\mathbf{v}^\nu)}{a_P}\right)_f \cdot \mathbf{S}_f + \left(\frac{1}{a_P}\right)_f \mathrm{ddt\,PhiCorr}\left(\mathbf{v}^\nu, \Phi^\nu\right) \tag{3.22}$$

where ddt PhiCorr is a flux adjustment due to the time-step. This is resolved by applying a *Rhie-Chow interpolation* [18], the next term in the pressure equation, **phiHbyA**, reads as,

$$\Phi_f^{\nu+1} = \Phi_f^{\nu+1} - \left[\left(\frac{1}{a_P}\right)_f \left(\nabla p_{rgh}\right)_f\right] \cdot \mathbf{S}_f \tag{3.23}$$

The p_{rgh} term is thus assembled as,

$$\sum_f \left[\left(\frac{1}{a_P}\right)_f \left(\nabla p_{rgh}^{\nu+1}\right)_f\right] \cdot \mathbf{S}_f = \sum_f \Phi^{\nu+1} \tag{3.24}$$

The flux, ϕ, is adjusted by the p_{rgh} term yielding the following expression,

$$\Phi_f^{\nu+1} = \Phi_f^{\nu+1} - \left[\left(\frac{1}{a_P}\right)_f \left(\nabla p_{rgh}\right)_f\right] \cdot \mathbf{S}_f \tag{3.25}$$

$$\begin{aligned}\Phi_f^{\nu+1} =& \Phi_f^{\nu+1} + \left[\left(\frac{1}{a_P}\right)_f \left[(-\nabla p)_f + (\mathbf{g}\cdot\mathbf{x})_f \left(\nabla {\rho_r}^{n+1}\right)_f - (\mathbf{g}\cdot\mathbf{x})_f \left(\nabla\rho(T)^{n+1}\right)_f \right.\right. \\ & \left.\left. + ({\rho_r}^{n+1}\mathbf{g})_f - (\rho(T)^{n+1}\mathbf{g})_f\right]\right] \cdot \mathbf{S}_f\end{aligned} \tag{3.26}$$

Finally, the velocity calculated at the center of the volume reads as,

$$\mathbf{v}^{\nu+1} = \mathbf{v}^{\nu+1} + \frac{1}{a_P}\mathcal{R}\left[\left(\Phi^{\nu+1} f - \Phi_u^{\nu+1} f\right)(a_P)_f\right] \tag{3.27}$$

where $\mathcal{R}$ is an operator used to recover cell-centered fields from fields given as fluxes at faces. Then, the static pressure, p, is reconstructed from p_{rgh}, leading the expression,

$$p = p_{rgh} + (\rho_r - \rho(T))\mathbf{g} \cdot \mathbf{x} \tag{3.28}$$

3.3.6 Case Setup

Once the implementation is done, a first case is studied with the existing solver, *buoyantBoussinesqPimpleFoam*. The boundary conditions, thermophysical properties and some other solver parameters are described along the following subsections.
As commented before, all studies are calculated on a computational domain of *38mm x 38mm*.

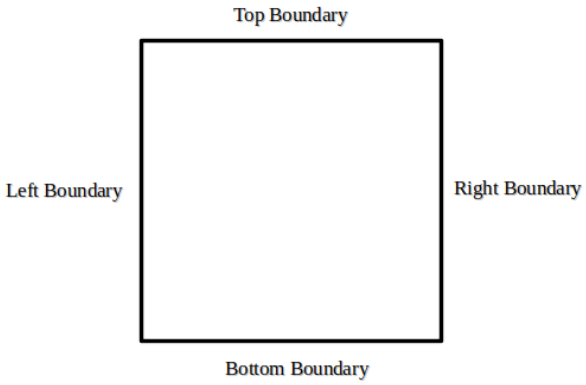


FIGURE 3.4: Setting of cavity computational domain.

Boundary conditions

Five boundaries are defined in the current case:
Left: is considered a wall with a fixed value of temperature. This is the hot wall. No velocity is prescribed.
Right: considered to be the cold wall with a fixed temperature. No velocity is prescribed.
Top: this is considered the top wall and it is adiabatic, thus, no heat transfer is assumed and zero gradient is applied. No velocity is applied.
Bottom: This shares similar conditions as the top wall.
frontAndBack: this uses a symmetry plane condition in the z direction since the problem is considered to be 2-dimensional. For such boundary type, no more conditions need to be prescribed.

Boundary	Conditions
Left	$T_l = 283, v_l = 0$
Right	$T_r = 273, v_r = 0$
Top	$\frac{\partial T_u}{\partial n} = 0, v_u = 0$
Bottom	$\frac{\partial T_b}{\partial n} = 0, v_b = 0$

TABLE 3.2: Boundary conditions for natural convection case.

Thermophysical properties

The thermophysical properties for the natural convection calculation are described in 3.3.

Water properties	Symbol	Values	Units
Density	ρ_r	999.8	$kg.m^{-3}$
Dynamic viscosity	μ	0.001003	$kg.m^{-1}.s^{-1}$
Thermal conductivity	λ	0.6	$W.m^{-1}.K^{-1}$
Heat capacity	C_p	4182	$J.kg.K^{-1}$
Gravitational acceleration	g	9.81	$m.s^{-2}$
Thermal diffusivity	γ	1.435e-7	$m^2.s^{-1}$
Thermal expansion coefficient	β	6.734e-5	K^{-1}
Laminar Prandtl number	P_r	6.99	-
Reference temperature	T_r	6.734e-5	K

TABLE 3.3: Water properties for natural convection.

Here below are presented the discretization schemes used for the terms appearing on the equations involved in the calculation. For more information on the used ones, refer to [**openfoamuserguide:cfddirect**].

Modeling Term	Keyword	Scheme	Remarks
Time derivatives	ddtSchemes	Euler	First order, bounded, implicit
Divergence term	divSchemes		Second order, unbounded
Gradient term	gradSchemes	Gauss linear	Second order, unbounded
Laplacian term	laplacianSchemes	Gauss linear orthogonal	Second order
Grad. normal to cell face	snGradSchemes	orthogonal	Second order
Point to point interpolation	interpolationSchemes	linear	Central differencing

TABLE 3.4: Discretization schemes.

The equation solvers are shown in 3.5.

Equation	Linear Solver	Smoother/Preconditioner	Tolerance
Pressure correction equation	PCG	DIC	1e-8
Momentum equation	PBiCGStab	DILU	1e-6
Temperature equation	PBiCGStab	DILU	1e-6

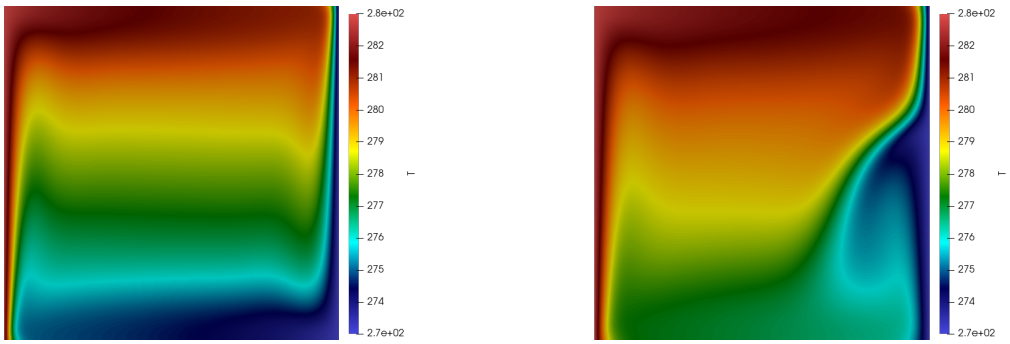
TABLE 3.5: Solvers for the discretised equations.

Table 3.6 presents the parameters used for the inner and outter loops performed within the calculation.

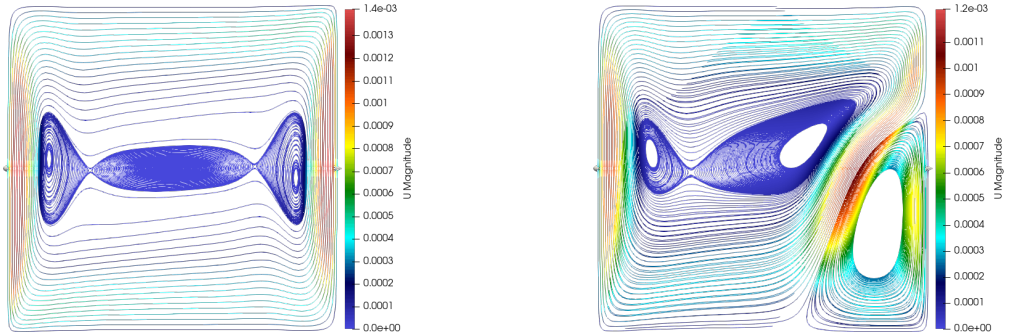
Parameter	Value
momentumPredictor	no
nOuterCorrectors	1
nNonOrthogonalCorrectors	0
nCorrectors	2

TABLE 3.6: Parameters for the discretised equations.

3.3.7 Validation of Results and Conclusions



(A) Temperature magnitude comparison at t = 1500s. Left: BuoyantBoussinesqPimpleFoam. Right: NCMF



(B) Velocity magnitude comparison at t = 1500s. Left: BuoyantBoussinesqPimpleFoam. Right: NCMF

FIGURE 3.5: Comparison between BuoyantBoussinesqPimpleFoam and NCMF*

BBPF*: BuoyantBoussinesqPimpleFoam solver. NCMF**: Natural convection modified solver.

In order to compare consistently the obtained results with those of the literature, the following dimensionless values are pointed out:

$$\tilde{T} = \frac{T - T_{\text{cold}}}{T_{\text{hot}} - T_{\text{cold}}} = \frac{T - 273}{10} \tag{3.29}$$

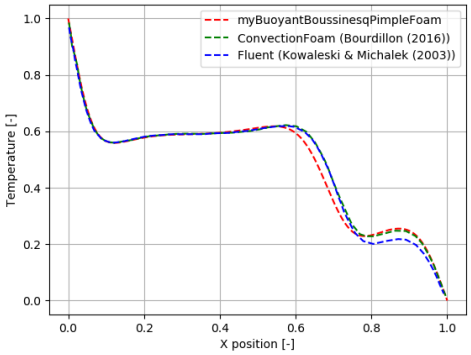
$$\tilde{x} = \frac{x}{\ell} = \frac{x}{38 \times 10^{-3}} \tag{3.30}$$

$$\tilde{v} = \frac{v\ell}{\gamma} = \frac{v38 \times 10^{-3}}{1.435 \times 10^{-7}} \tag{3.31}$$

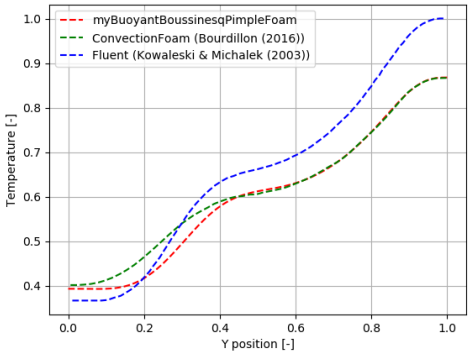
$$\tilde{u} = \frac{u\ell}{\gamma} = \frac{u38 \times 10^{-3}}{1.435 \times 10^{-7}} \tag{3.32}$$

$$\tilde{t} = \frac{t\gamma}{\ell^2} = \frac{t \times 1.435 \times 10^{-7}}{1.444 \times 10^{-6}} \tag{3.33}$$

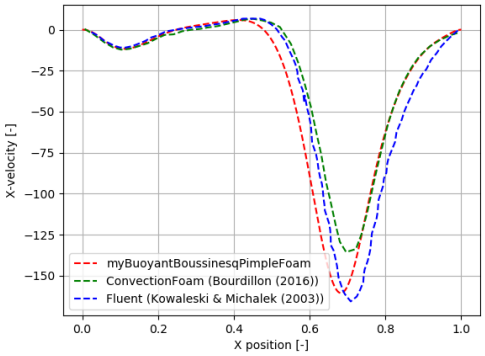
$$\tilde{y} = \frac{y}{\ell} = \frac{y}{38 \times 10^{-3}} \tag{3.34}$$



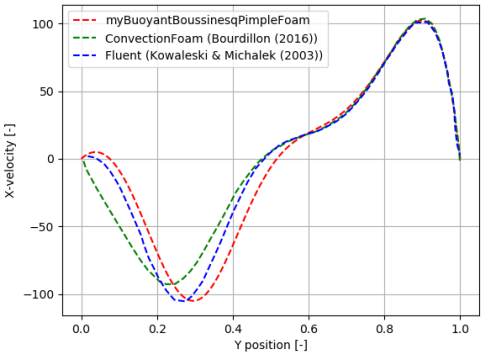
(A) Temperature along horizontal line.



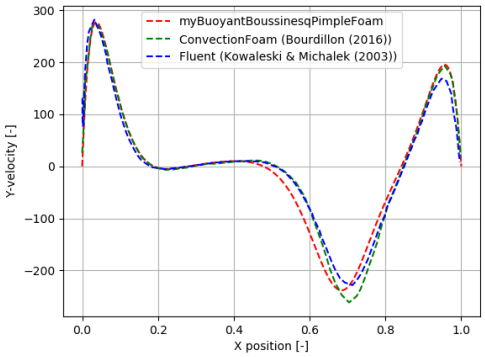
(B) Temperature along vertical line.



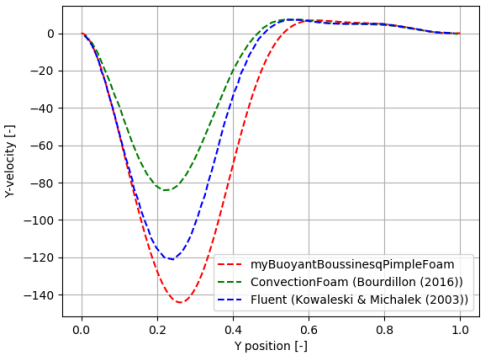
(C) U-velocity along horizontal line.



(D) U-velocity along vertical line.



(E) V-velocity along horizontal line.



(F) V-velocity along vertical line.

FIGURE 3.6: Adimensional magnitudes comparison.

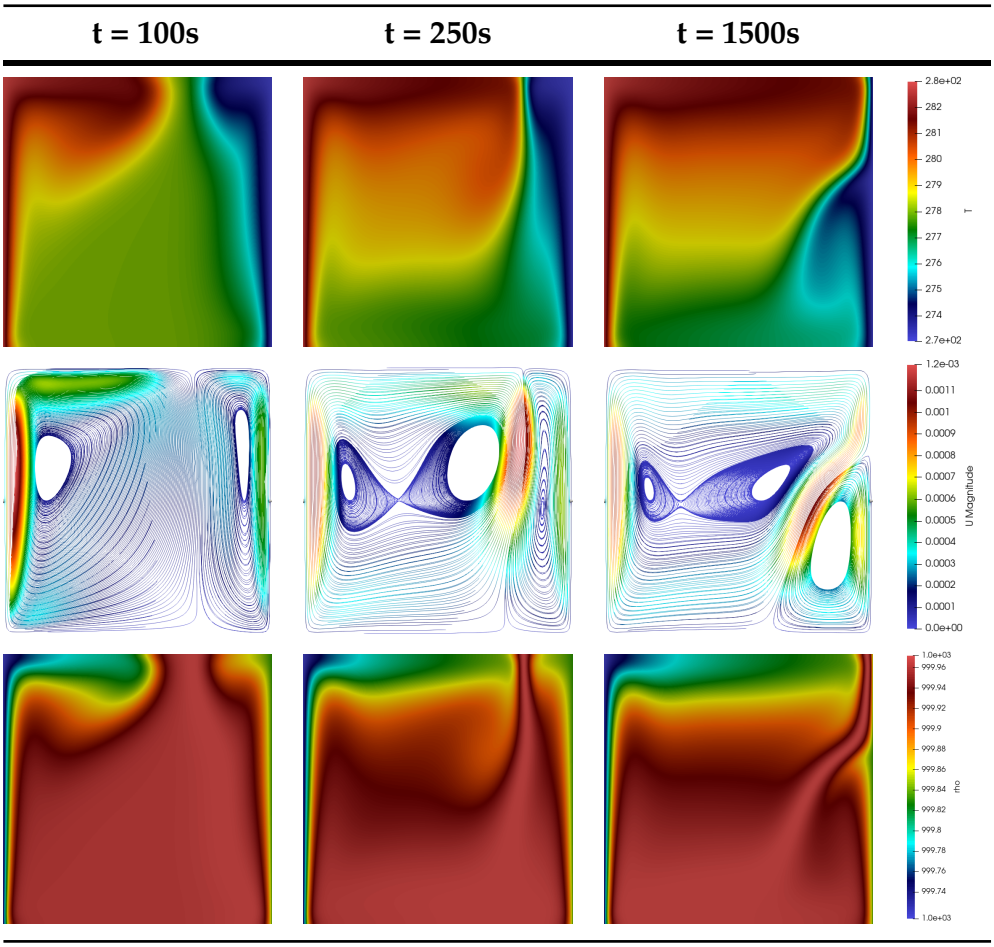


TABLE 3.7: Numerical results of Natural convection modified solver between $t = 100s$ and $1500s$.

3.4 OpenFOAM: IcoReactingMultiphaseInterFOAM. Phase-Change Process

The solidification process is assessed in this section with two elaborated models. Both of the models are implemented within a multi-phase solver based on the volume of fluid technique. This technique aims to capture interface and enhances contact angle and surface tension for each phase. Thus, the first model is based on the coupling of the VOF and the enthalpy-porosity method. To accomplish the inclusion of the enthalpy-porosity method, a library in which the latent heat is implemented as an explicit source term for the energy equation in the solver.

On the other hand, the second model uses the VOF method combined with a semi-empirical model based on the work of Lee. The empirical constant is adapted here to be used in conjunction with the use of the *Classical Nucleation Theory*.

3.5 Case Description.

Two regular geomtries are created: a squared cavity, used in the pure convection case and a cylindrical plane geometry. Both geometries test both solidification models.

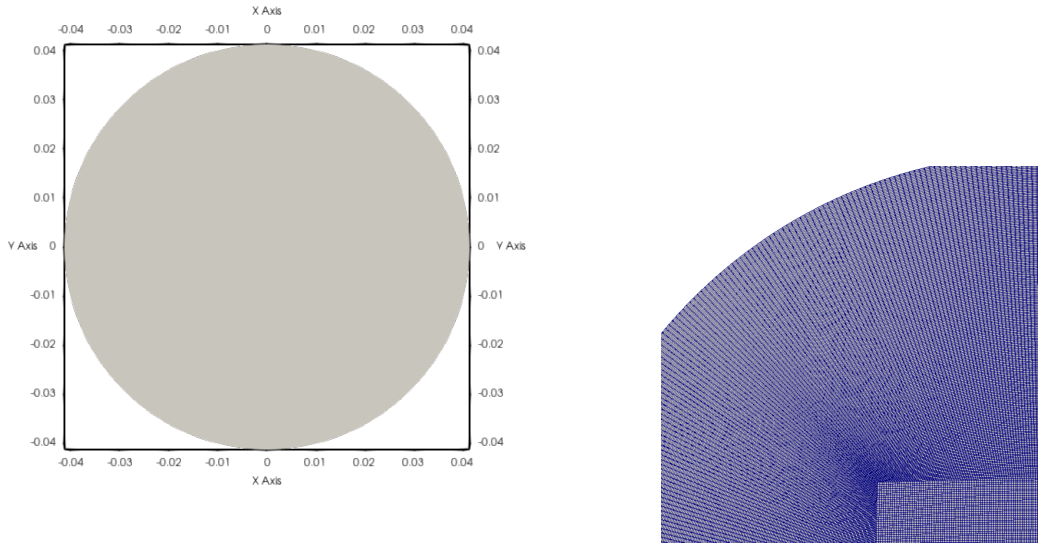


FIGURE 3.7: Geometric characteristics for cylinder.

The computed structured mesh consists of 572404 nodes.

3.5.1 Hypotheses And Assumptions

To carry out the phase-transition process, some assumptions are taken into account so as to simplify the multiphysics ocurring during such arising phenomena.

Laminar regime: The Reynolds number, computed from the maximum velocity is not high enough to consider turbulent effects.

In the current case-scenario, a Prandtl close to 7. The values used for the laminar Prandtl number calculation are: $\mu = 0.001003 Kg.m^{-1}.s^{-1}$, $\lambda = 0.56 W.m^{-1}.K^{-1}$ and $C_p = 4202 J.Kg.K^{-1}$.

Newtonian fluid: The viscosity of the fluid is assumed to be constant. The thermophysical properties treatment is described below.

Quasy-steady state: Bourdillon [3] used the hypothesis of quasy-steady solution of the natural convective solver as initial condition in the solidification process. As many researchers as Yan et al. **[yan_xu_qiu_gang_2017]** suggest, water presents a high latent heat of solidification when the heat released during freezing plays a greater role than the transient process of heat accumulation in the layer of the phase being developed. In other words, when the latent heat of the phase change material is larger than its sensible heat, the latter is having little influence on the temperature distribution of the PCM. In such case, the interface is moving slowly and the temperature distribution, at a given time step, keeps constant. Therefore, for comparison purposes against literature results, the solidification process in a cavity is tested using a quasy-steady solution obtained in the pure convection case.

Two phase properties

Within a multiphase framework, a model reflects a jump in properties through the interphase. Thus, a smooth transition between phase properties must be achieved.

$$\lambda = \lambda_\ell \alpha_\ell + \lambda_s f_s \tag{3.35}$$

$$C_p = C_{p_\ell} \alpha_\ell + C_{p_s} f_s \tag{3.36}$$

$$\mu = \mu_\ell \alpha_\ell + \mu_s f_s \tag{3.37}$$

In the current case-scenario, $C_{p_s} = C_{p_l}$.

In the case of polynomial density variation it is settled in a similar manner. The polynomial is not thought to suit negative temperatures, and when the problem is within this range, the density should take ice's density.

$$\rho(T)' = \rho(T)\alpha_\ell + \rho_s \alpha_s \tag{3.38}$$

where α_l and α_s are liquid and solid volume fractions, respectively.

3.5.2 Governing Equations

This section is devoted to describe the governing equations that the solidification process requires. Beside the presented conservation equation for the volume of fraction needed for the VOF method, Eq. 3.39,

$$\frac{\partial \alpha_{\text{phase}}}{\partial t} + \frac{\partial \left(\alpha_{\text{phase}}\, u_j\right)}{\partial x_j} = 0 \tag{3.39}$$

in the next sections, momentum and energy equations are revisited.

3.5.2.1 Momentum Equation

The momentum equation has the same terms as per each one of the models. Here it is the equation recalled from previous section:

$$\begin{aligned} &\frac{\partial \left(\rho u_i\right)}{\partial t} + \frac{\partial \left(\rho u_i u_j\right)}{\partial x_j} \\ &\quad = -\alpha_i \nabla p + \frac{\partial}{\partial x_j}\left(\mu \frac{\partial u_i}{\partial x_j}\right) + F_{\sigma i} + S_{u_i} \end{aligned} \tag{3.40}$$

3.5.2.2 Energy Equation

On the other side, the energy equation slightly differs from one model to the other. As pointed out before, here there are recalled both energy equations. The energy equation for the Enthalpy-porosity model:

$$\frac{\partial(\rho C_p T)}{\partial t} + \nabla \cdot \left(u_j \rho C_p T\right) + L\left[\frac{\partial(\rho \alpha_l \gamma_l)}{\partial t} + \frac{\partial(u_j \rho \alpha_l \gamma_l)}{\partial x_j}\right] = \nabla \cdot \left(k_i \nabla T_i\right) \tag{3.41}$$

The energy equation for the Lee model in conjunction with the nucleation theory:

$$\frac{\partial(\rho C_p T)}{\partial t} + \nabla \cdot \left(u_j \rho C_p T\right) = \nabla \cdot \left(k_i \nabla T_i\right) + S_{H_i} \tag{3.42}$$

3.5.3 Solver description. Control Loop

IcoReactingMultiphaseInterFoam solver is a multiphase, multicomponent incompressible solver based on volume of fluid method. The solver captures the interfaces and includes contact angle and surface tension effects for each phase. Moreover, this solver supports mass and heat transfer across phases.

3.5.4 Mass transfer models

For each pair of phases, two mass transfer models might be used:

- **Lee model:** Used for solid melting and liquid solidification.

- **KineticGasEvaporation:** Used for condensation and evaporation.

In this thesis, only the Lee model will be considered for further explanation.

3.5.5 Code implementations

Within the entrophy-porosity model, the source term belonging to the calculation of the latent heat is added within the OpenFOAM framework. The energy equation of the solver shown in Eq. 3.42 is thereby implemented in Fig. 3.8 In the term belonging to the RHS of the equation, the solver calls

```
fvScalarMatrix TEqn
(
    fvm::ddt(rhoCp, T)
  + fvm::div(rhoCpPhi, T, "div(phi,T)")
  - fvm::Sp(fvc::ddt(rhoCp) + fvc::div(rhoCpPhi), T)
  - fvm::laplacian(kappaEff, T,  "laplacian(kappa,T)")
  ==
    fvOptions(rhoCp, T)
);
```

FIGURE 3.8: Energy equation of IcoReactingMultiphaseInterFoam.

the library *mySolidificationMeltingSource* that calculates the latent heat source term as it appears in the figure 3.9: Here, the alpha variable showing up in the

```
const auto& CpVoF = mesh_.lookupObject<volScalarField>(CpName_);
const auto& rhoCpPhiVoF = mesh_.lookupObject<surfaceScalarField>(rhoCpPhiName_);
dimensionedScalar L("L", dimEnergy/dimMass, L_);

// contributions added to rhs of solver equation
if (eqn.psi().dimensions() == dimTemperature)
{

    eqn -= (L*alpha1_)/CpVoF*(fvc::ddt(rho, alphaC_) + fvc::div(rhoCpPhiVoF, alphaC_));

}
else
{
    //This option is not activated since fvOptions in TEqn does not enable this condition
    eqn -= L*alpha1_*(fvc::ddt(rho, alphaC_));
}
```

FIGURE 3.9: Latent heat source term present in mySolidificationMeltingSource library.

calculation is obtained through a linear expression that gives the amount of energy contained in the fluid cell above the melting point. This is divided by the latent heat to obtain the liquid fraction. Then, this fraction is constained between 0 and 1. Further details on the code can be found in A.1.

The latent heat source term is afterwards weighted by the volume fraction calculated by means of the transport equation in the context of the volume of fluid method as it appears in Richter et al. [**richter_turnow_kornev_hassel_2016**].

The **rhoCpPhiVoF** term, which is called in this library, is created in *createFields.H* as a variable field so that it can be called from everywhere within the code.

```
surfaceScalarField rhoCpPhi
(
    IOobject
    (
        "rhoCpPhi",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::NO_WRITE
    ),
    fvc::interpolate(fluid.Cp())*rhoPhi
);

rhoCpPhi.oldTime();
```

FIGURE 3.10: **rhoCpPhi** field in *createFields.H*

The implemented library can be found in the Appendix A, section A.1.

On the core of the other model, the basis of the Lee model is already implemented in OpenFOAM. However, there is a parameter, *C*, devoted to act as a condensation rate. This is referred as an empirical coefficient used to speed-up or slow down the mass and heat transfer. The physics behind are unknown for this solver, therefore, in favor of tunning this parameter in accordance with the characteristic behavior of the water when it freezes, the *Classical Nucleation theory* is followed.

```
template<class Thermo, class OtherThermo>
Foam::meltingEvaporationModels::LeeCNT<Thermo, OtherThermo>::LeeCNT
(
    const dictionary& dict,
    const phasePair& pair
)
:
    InterfaceCompositionModel<Thermo, OtherThermo>(dict, pair),
    C_("C", inv(dimTime), dict),
    Tactivate_("Tactivate", dimTemperature, dict),
    planck_("planck", dimEnergy*dimTime, dict),
    boltzmann_("boltzmann", dimEnergy/dimTemperature, dict),
    deltag_("deltag", dimEnergy, dict),
    nL_("nL", inv(dimVolume), dict),
    gammaYW_("gammaYW", dimEnergy/dimArea, dict),
    hLV_("hLV", dimEnergy/dimVolume, dict),
    alphaEY_("alphaEY", dict),
    alphaMin_(dict.getOrDefault<scalar>("alphaMin", 0)),
    interfaceVolume_
    (
        IOobject
        (
            "cellVolume",
            this->mesh_.time().timeName(),
            this->mesh_,
            IOobject::NO_READ,
            IOobject::NO_WRITE
        ),
        this->mesh_,
        dimensionedScalar(dimVolume, Zero)
    )
{}
```

FIGURE 3.11: Function used to ask for the required user inputs.

In the function shown above, all of the parameters concerning the formulas required to calculate the nucleation rate of the water are user-defined parameters except for the **cellVolume** which is an implicit function that calculates the volume per each cell. In the Appendix A, the code related with the Lee model using this theory can be found.

3.5.6 Case Setup

Boundary conditions

For the squared cavity: The initial conditions for the velocity and temperature fields are inherited from the last timestep of the natural convection case.

Boundary	Conditions
Left	$\frac{\partial \alpha_l}{\partial n} = 0, \frac{\partial \alpha_s}{\partial n} = 0$
Right	$\alpha_l = 1, \alpha_s = 0$
Upper	$\frac{\partial \alpha_l}{\partial n} = 0, \frac{\partial \alpha_s}{\partial n} = 0$
Bottom	$\frac{\partial \alpha_l}{\partial n} = 0, \frac{\partial \alpha_s}{\partial n} = 0$

TABLE 3.8: Boundary conditions for natural convection case.

frontAndBack boundary is set to *empty* to define a 2-dimensional case-scenario.

For the cylinder:

Boundary	Conditions
Walls	$T = 255, v = 0, \frac{\partial \alpha_l}{\partial n} = 0, \frac{\partial \alpha_s}{\partial n} = 0$
Internal field	$T = 294, v = 0, \alpha_l = 1, \alpha_s = 0$

TABLE 3.9: Boundary conditions for natural convection case.

As in the previous geometry, there is an empty boundary called frontAndBack to define a planar case.

The thermophysical properties and solver parameters defined below are setted up for both geometries.

Thermophysical properties

Water properties	Symbol	Values	Units
Water density	ρ_l	999.8	$kg.m^{-3}$
Ice density	ρ_s	916.8	$kg.m^{-3}$
Water kinematic viscosity	ν_l	1.79e-6	$m^2.s^{-1}$
Ice kinematic viscosity	ν_s	2.0e-6	$m^2.s^{-1}$
Water thermal conductivity	λ_l	0.56	$W.m^{-1}.K^{-1}$
Ice thermal conductivity	λ_s	2.26	$W.m^{-1}.K^{-1}$
Heat capacity	$C_{p_l} = C_{p_s}$	4202	$J.kg.K^{-1}$
Gravitational acceleration	g	9.81	$m.s^{-2}$
Thermal diffusivity	γ	1.435e-7	$m^2.s^{-1}$
Thermal expansion coefficient	β	6.734e-5	K^{-1}
Latent heat	L	335000	$J.K^{-1}$
Laminar Prandtl number	P_r	6.99	-
Reference temperature	T_r	6.734e-5	K
Darcy's constant	D_c	10e8	-

TABLE 3.10: Water properties for natural convection.

In table 3.11 are detailed the parameters used in the implemented nucleation library for the Lee model. The solver parameters used for the discretiza-

Water nucleation properties	Symbol	Values	Units
Planck constant	h	6.63e-34	$J.s$
Boltzmann constant	k_B	1.38e-23	$J.K^{-1}$
Gibbs free energy	Δ_{gv}	4e-20	J
Interfacial tension	γ_{yw}	2.91e-2	$J.m^{-2}$
Latent heat per volume	H_{lv}	3.10e8	$J.m^{-3}$
Shape coefficient of nucleation	α_{ey}	0.0001	-
Water molecule per volume	n_L	5.5e4	m^3

TABLE 3.11: Water properties for solidification.

tion of the different terms in the equations are pointed out next.

Solver parameters

So as to obtain a minimum expected accuracy during the calculation, in tables 3.12 are the chosen parameters for the equation solvers.

Equation	Linear Solver	Smoother/Preconditioner	Tolerance
Pressure correction equation (P)	PCG	DIC	1e-5
Momentum equation (U)	smoothSolver	symGaussSeidel	1e-06
Volume fraction equation (alpha)	smoothSolver	symGaussSeidel	1e-8
Species equation (Y)	smoothSolver	symGaussSeidel	1e-09
Energy equation (T)	PBiCG	DILU	
	1e-08		

TABLE 3.12: Solvers for the discretised equations.

3.5.7 Validation of Results and Conclusions

The validation of the phase change problem is achieved by different methodologies. First, the enthalpy-porosity method is compared with available data found in the doctoral thesis of Borudillon [3] and the experimental data of Kowalewski et al. [12].

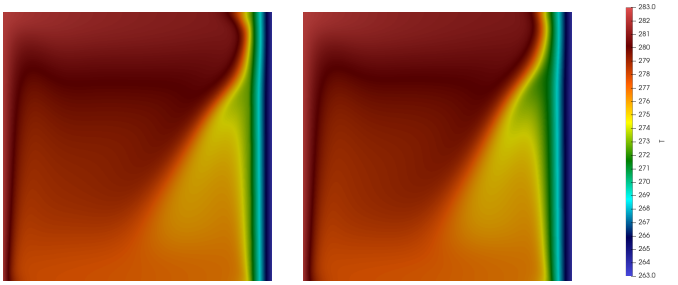
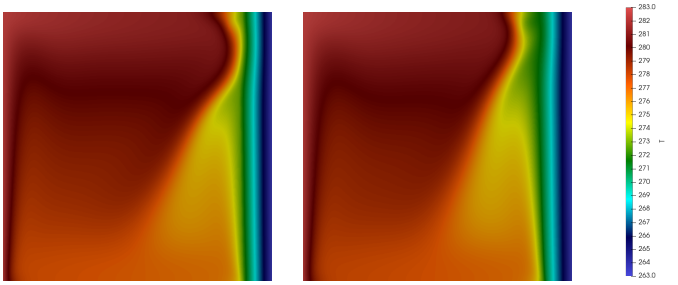
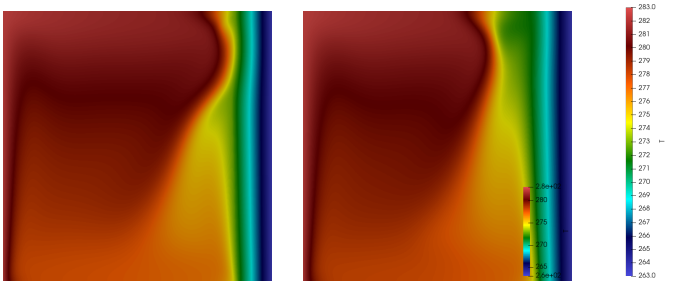
	Enthalpy-porosity	Lee model-CNT
T=100s		
T=200s		
T=300s		

TABLE 3.13: Numerical results of temperature distributions for Enthalpy-porosity and Lee-CNT models at t = *100, 200, 300s*.

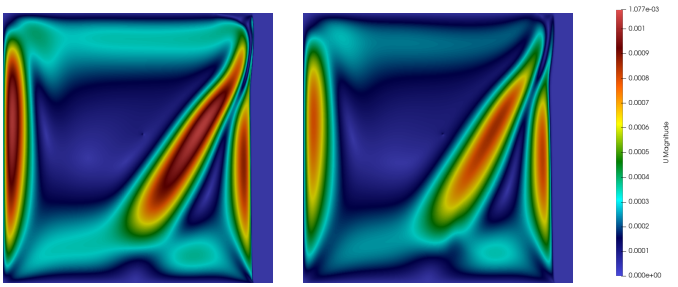
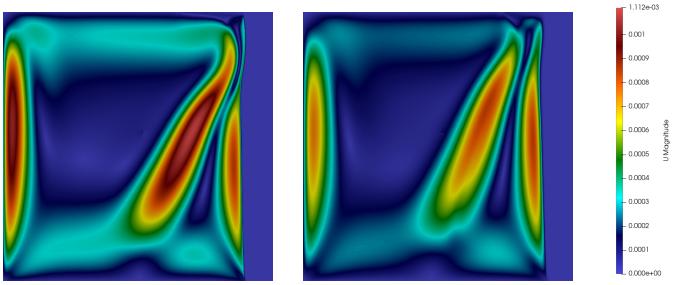
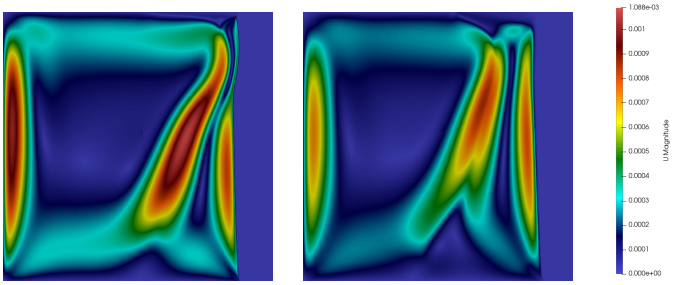
	Enthalpy-porosity	Lee model-CNT
T=100s		
T=200s		
T=300s		

TABLE 3.14: Numerical results of velocity distributions for Enthalpy-porosity and Lee-CNT models at $t = 100, 200, 300s$.

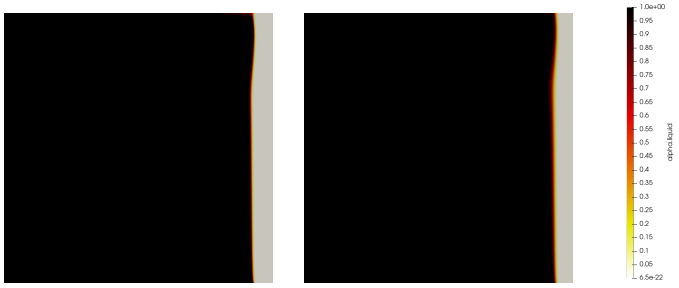
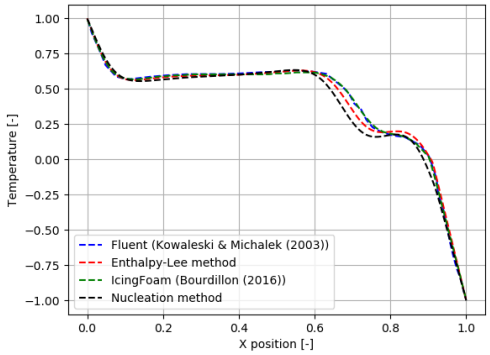
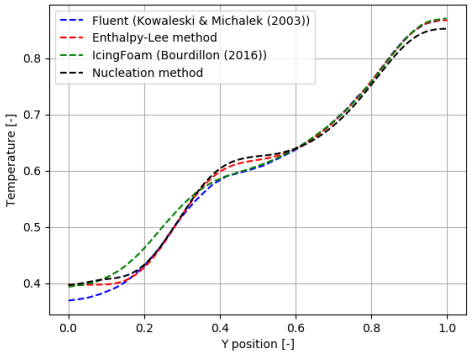
	Enthalpy-porosity	Lee model-CNT
T=100s		
T=200s		
T=300s		

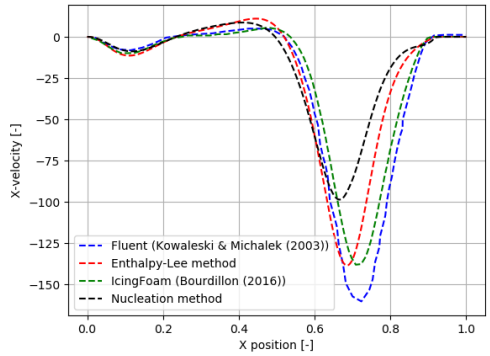
TABLE 3.15: Numerical results of fluid fraction distributions for Enthalpy-porosity and Lee-CNT models at $t = 100, 200, 300s$.



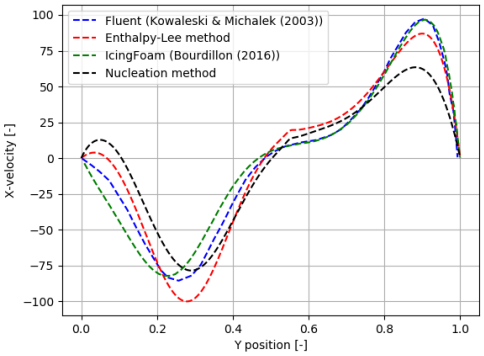
(A) Temperature along horizontal line.



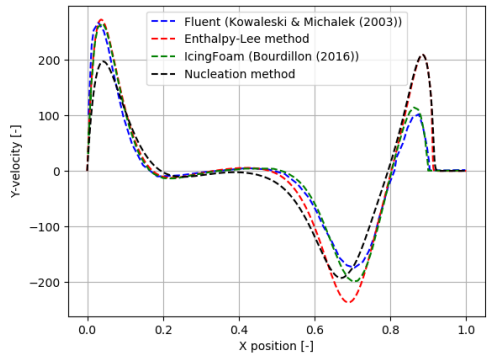
(B) Temperature along vertical line.



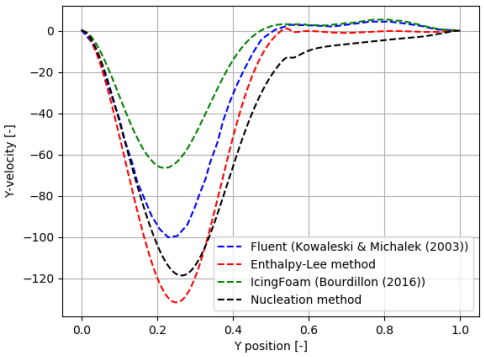
(C) U-velocity along horizontal line.



(D) U-velocity along vertical line.



(E) V-velocity along horizontal line.



(F) V-velocity along vertical line.

FIGURE 3.12: Adimensional magnitudes comparison.

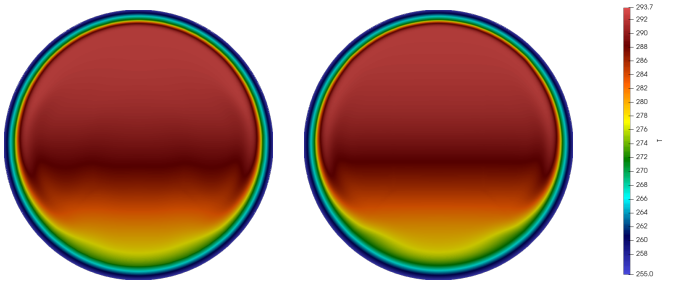
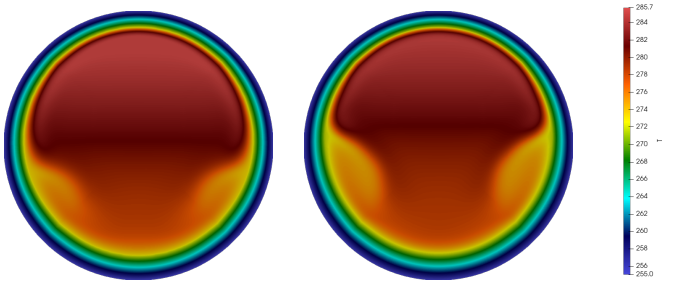
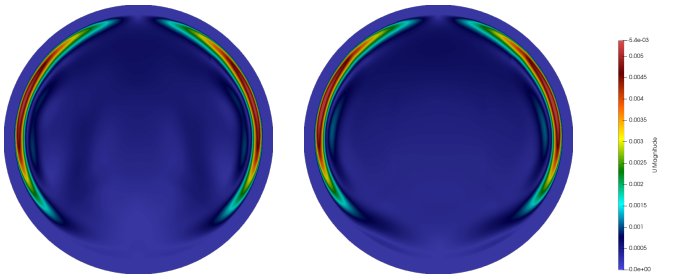
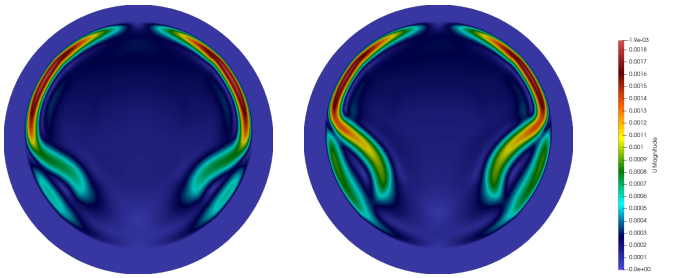
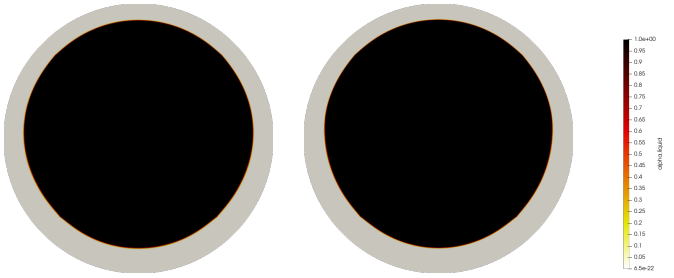
	Enthalpy-porosity	Lee model-CNT
T=100s		
T=300s		
T=100s		
T=300s		
T=100s		
T=300s		

TABLE 3.16: Numerical results of Enthalpy-porosity and Lee-CNT models at $t = 100s$ and $300s$ in a cylinder.

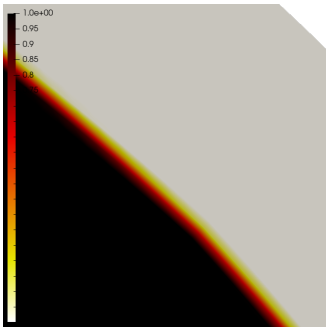


FIGURE 3.13: Gradient of the interface between liquid and solid phases for Lee-CNT model.

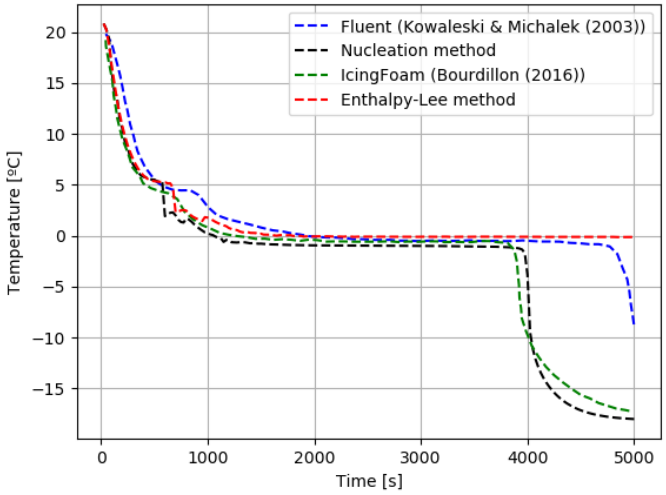


FIGURE 3.14: Numerical results of temperature profiles in center position of cylindrical geometry.

For the Lee model based on the *Classical Nucleation Theory*, the results are compared against the analytical solution given by the Neumann solutions of the Stefan problem.

3.5.7.1 Stefan Problem

The Stefan problem, is an initial boundary value problem of a parabolic differential equation with discontinuous coefficients on the phase transitions interfaces. The analytical solution to the classical Stefan problem exists in a limited range of idealized situations.

The governing equations for a general solid-liquid phase change problem are:

The heat equation for the solid phase,

$$\rho_s c_s \frac{\partial T_s}{\partial t} = \nabla \cdot (k_s \nabla T_s) \quad \text{on } \Omega_s \tag{3.43}$$

for the liquid phase, advective term is also considered:

$$\rho_l c_l \left(\frac{\partial T_l}{\partial t} + \mathbf{u} \cdot \nabla T_l \right) = \nabla \cdot (k_l \nabla T_l) \quad \text{on } \Omega_l \tag{3.44}$$

At the interface, the Stefan condition is satisfied and then,

$$\rho_s L(t) V_n = k_s \nabla T|_\Gamma - k_l \nabla T|_\Gamma \text{ on } \Gamma \tag{3.45}$$

where V_n is the normal velocity at the interface.

$$T = T_\mathrm{m} \quad \text{on } \Gamma \tag{3.46}$$

One-dimensional problem

In seek of simplification, and recalling the 1D problem as shown in the figure:

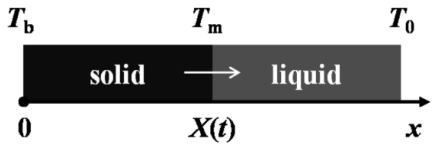


FIGURE 3.15: Schematic diagram of Stefan problem

the initial conditions are expressed as [26]:

$$u_0(x) = u_0, \quad t = 0, \quad x \in [0, L], \tag{3.47}$$

while the boundary conditions are the shown below:

$$u(0,t) = -20^\circ C, \quad \frac{\partial u}{\partial x}(L,t) = 0, \quad t > 0 \tag{3.48}$$

In table 3.17, there are summarized the boundary conditions applied in the cavity geometry of previous cases. The internal field is initiallized at 283.15 K. The used thermophysical properties as well as the solver parameters are

Boundary	Conditions
Left	$T_l = 253.15, \alpha_l = 1, \alpha_s = 0$
Right	$\frac{\partial T_r}{\partial n} = 0, \frac{\partial \alpha_l}{\partial n} = 0, \frac{\partial \alpha_s}{\partial n} = 0$
Upper	$\frac{\partial T_t}{\partial n} = 0, \frac{\partial \alpha_l}{\partial n} = 0, \frac{\partial \alpha_s}{\partial n} = 0$
Bottom	$\frac{\partial T_b}{\partial n} = 0, \frac{\partial \alpha_l}{\partial n} = 0, \frac{\partial \alpha_s}{\partial n} = 0$

TABLE 3.17: Boundary conditions for Stefan problem.

similar to the previous solidification cases.
The discontinuous exact solutions for the Stefan problem are:

$$\begin{cases} T_l(x,t) = \dfrac{\operatorname{erfc}\left(\frac{x}{2\sqrt{a_1 t}}\right)}{\operatorname{erfc}\left(\lambda\sqrt{\frac{a_s}{a_1}}\right)}(T_m - T_0) + T_0, & x > \xi(t), \\ T_s(x,t) = \dfrac{\operatorname{erf}\left(\frac{x}{2\sqrt{a_s t}}\right)}{\operatorname{erf}\left(\lambda\sqrt{\frac{a_s}{a_1}}\right)}(T_m - T_b) + T_b, & x \leq \xi(t). \end{cases} \tag{3.49}$$

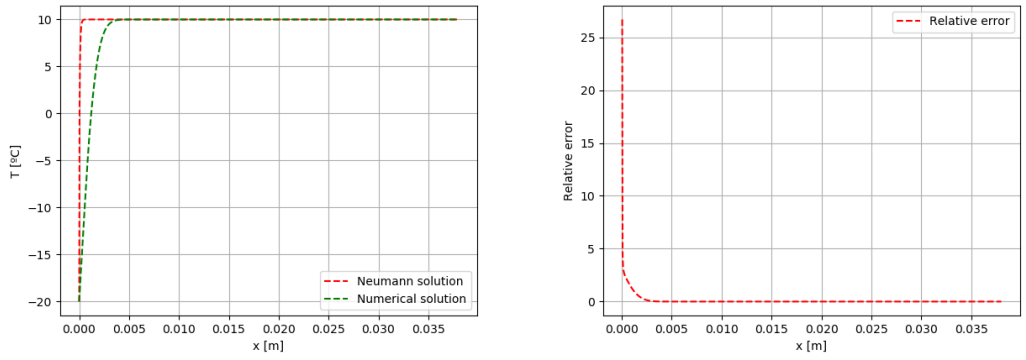
By using a phase change interface condition, a solution to the trascendental equation may be found:

$$\frac{e^{-\lambda^2}}{\operatorname{erf}(\lambda)} + \frac{k_l}{k_s}\sqrt{\frac{a_s}{a_1}}\frac{T_m - T_0}{T_m - T_b}\frac{e^{-\frac{a_s}{a_1}\lambda^2}}{\operatorname{erfc}\left(\lambda\sqrt{\frac{a_s}{a_1}}\right)} = \frac{\lambda L\sqrt{\pi}}{c_{ps}(T_m - T_b)} \tag{3.50}$$

where $\operatorname{erf}(x)$ is the complementary error function expressed as $1 - \operatorname{erf}(x)$.

The secant method is used as the iterative scheme to find the root of the given function with $tol < 1e - 12$. The root of λ is 0.2299545377262345.

In the following figures, the method is tested against the exact solutions of the Stefan problem.



(A) Neumann solution vs numerical solution. (B) Relative error of the numerical solution.

FIGURE 3.16: Numerical solutions of the Lee model-CNT vs Neumann analytical solutions.

3.5.7.2 Interface height

The theoretical solution for the evolution of the interface is:

$$X(t) = 2\lambda\sqrt{a_s t} \tag{3.51}$$

Alongside, a post-process function to calculate the tracking of position of the interface is done. To do so, the values of the liquid fraction per timestep are first obtained. Thus, these values for each time are read to find the position in which alpha is 0.5. The python code is attached in Appendix A, section A.4.

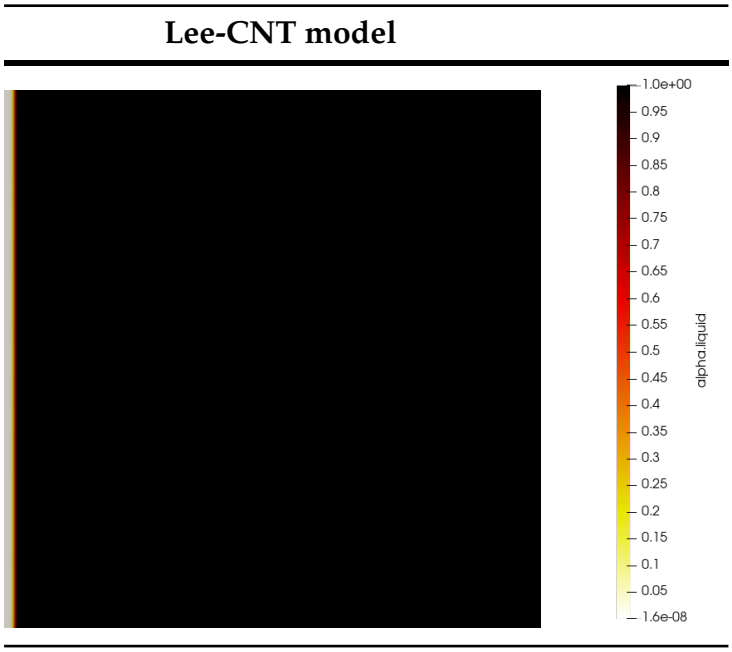
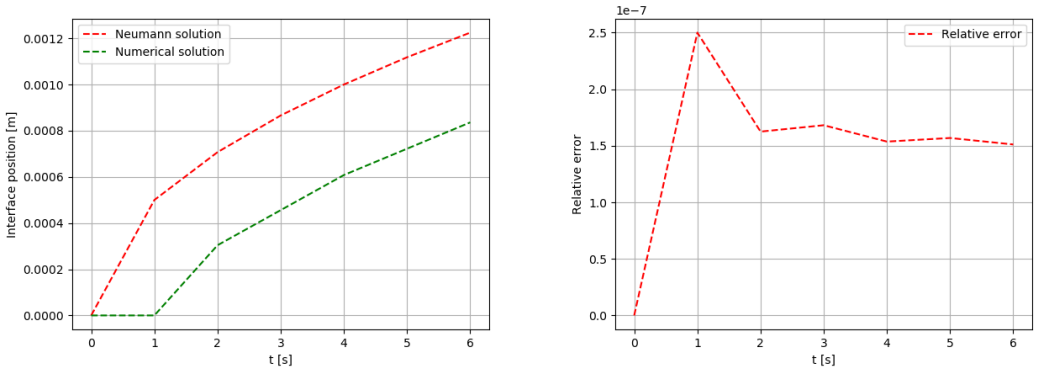


TABLE 3.18: Numerical results of interface evolution for Lee-CNT model at $t = 6s$.



(A) Interface position in time. (B) Relative error of interface position in time.

FIGURE 3.17: Numerical solutions of the Lee model-CNT vs Neumann analytical solutions for interface position.

3.5.7.3 Conclusions on the Stefan problem

From the figures 3.17a and 3.16a it is clearly visible that the Neumann solutions of the Stefan problem for the temperature distribution and the interface position do not match the numerical solution obtained with the Lee model undergoing nucleation characteristics. As depicted, the behavior of the analytical solution is faster than the numerical one.

The explanation resides behind the theory that macroscale models for phase-change generally take the assumptions of constant thermophysical properties, constant latent heat, $L(t) = L_m$ and constant melting/solidification temperatures within phases.

Alternatively, in this thesis it is proposed a model (Lee model) in which the $L(t)$ is not constant but dependent of the product of the net mass transfer and the difference of enthalpy of fusion between phases as indicated in Eq. 3.52. It is also remarkable that density is not constant for the fluid phase due to the implementation of the new equation of state in the solver. Thus, it adds more variability to the latent heat calculation.

Therefore, as the evolution of the latent heat balances the temperature distribution (it is implicit in the energy equation), so it does the development of the interface position.

$$L = \frac{dm_{ls}}{dt}\Delta T \Delta H_f = C_f \rho_l \alpha_l \left(\frac{T_{sat} - T_l}{T_{sat}}\right)(T_{sat} - T_l)(H_l - H_s) \tag{3.52}$$

In the nanoscale, the surface tension in the interface would affect the solution as well. The melting temperature could not be constant anymore since at the interface the condition that $T = T_m$ is not achieved. Leaving, at the interface Γ:

$$s\,(T - T_{\mathrm{m}}) = -\sigma\,(\kappa + \alpha V_{\mathrm{n}}) \qquad \text{on } \Gamma \tag{3.53}$$

as pointed out by Zhao et al. [26]. Where κ is the interface curvature, α is a kinetic coefficient, a proportional constant of the velocity of the interface and the kinetic undercooling, S, the entropy density difference between phases and σ the surface tension. Being, the kinetic undercooling, the state of equilibrium of the liquid submitted to temperatures under melting point without undergoing phase transition.

This means that as the undercooling velocity (velocity of nuclei formation) increases, so it does this coefficient. Therefore, the kinetic coefficient jointly with the surface tensions are also the parameters to account for when modeling phase-changes in a nanoscale.

Chapter 4

Numerical Simulation of Heat Transfer

4.1 OpenFOAM: chtMultiphaseInterFOAM. Conjugate Heat Transfer

The last objective of this thesis is to extend the multiphase solver of the previous section so it can account for multiregion purposes. To do so, a new solver derived from the concept of an existing multiregion solver is implemented.

The existing solver, *chtMultiRegionFoam* is developed on the basis that the fluid it solves undergoes the compressible Navier-Stokes equations with buoyancy forces and the energy equation per unit mass with gravity terms as follows:

Continuity Equation

The continuity equation reads as:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \boldsymbol{u}) = 0 \tag{4.1}$$

Momentum Equation

The momentum conservation equation yields as:

$$\begin{aligned} &\frac{\partial \rho \boldsymbol{u}}{\partial t} + \nabla \cdot (\rho \boldsymbol{u} \otimes \boldsymbol{u}) = \\ &-\nabla p_{\text{rgh}} + \nabla \cdot \left[\mu \left\{\nabla \otimes \boldsymbol{u} + (\nabla \otimes \boldsymbol{u})^{\text{T}}\right\}\right] - \nabla \left(\frac{2}{3}\mu \nabla \cdot \boldsymbol{u}\right) - \boldsymbol{g} \cdot \boldsymbol{x} \nabla \rho \end{aligned} \tag{4.2}$$

Energy Equation

The energy equation as listed in the native solver is:

$$\frac{\partial \rho h}{\partial t} + \nabla \cdot (\rho \boldsymbol{u} h) + \nabla \cdot (\rho \boldsymbol{u} K) = \nabla \cdot \left(\frac{\lambda}{c_p} \nabla h \right) + \rho \boldsymbol{u} \cdot \boldsymbol{g} \tag{4.3}$$

where $\boldsymbol{u}$ is the velocity vector, h is the enthalpy, $K = 0.5^*\boldsymbol{u} \cdot \boldsymbol{u}$ is the kinetic energy per unit mass, $p_{rgh} = p - \rho \boldsymbol{g} \cdot \boldsymbol{x}$ the modified pressure so that the momentum equation accounts for the buoyancy terms, and the remaining thermophysical properties, μ, λ, C_p being the kinematic viscosity, the thermal conductivity and the specific heat accordingly. The energy equation does not include radiation, heat generation term and chemical reaction.

Therefore, the challenge of this part is to couple the multiphase solver (*IcoReactingMultiPhaseInterFoam*) that allows for the solving of a fluid undergoing phase-change with a solid region.

4.1.1 Case description

Within this new case, a mesh for the solid region is required. To do so, a second structured mesh region is implemented within the framework of a provided script which builts cylindrical computational meshes in OpenFOAM format. The script is attached in Appendix B.

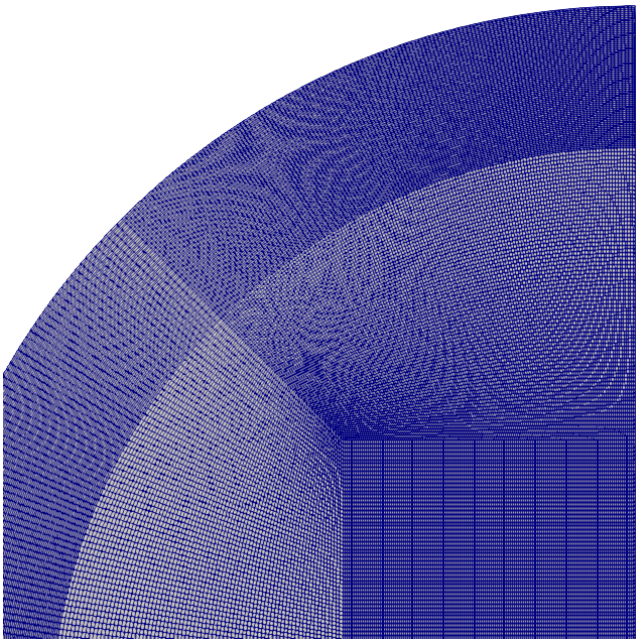


FIGURE 4.1: Computational mesh for the conjugate heat transfer case.

In the image shown above, a structured mesh of 731734 nodes is generated.

4.1.2 Hypotheses And Assumptions

Heat transfer: Conductive heat transfer is transferred throughout an isotropic material and convective heat transfer is arisen within the fluid region.

Laminar regime: The Reynolds number, computed from the maximum velocity is not high enough to consider turbulent effects.

In the current case-scenario, a Prandtl close to 7. The values used for the laminar Prandtl number calculation are: $\mu = 0.001003 Kg.m^{-1}.s^{-1}$, $\lambda = 0.6 W.m^{-1}.K^{-1}$ and $C_p = 4182 J.Kg.K^{-1}$.

Newtonian fluid: The viscosity of the fluid is assumed to be constant. As per the solidification cases, the treatment of the thermophysical properties is performed in a similar manner.

4.1.3 Governing Equations of the Fluid Region

Momentum equation

The momentum equation is recalled here in terms of viscous stress tensor. The surface tension forces as the body forces and the Darcy term is added in the momentum equation as shown below.

$$\begin{aligned} &\frac{\partial\left(\rho u_i\right)}{\partial t}+\frac{\partial\left(\rho u_i u_j\right)}{\partial x_j} \\ &\quad=-\alpha_i \nabla p+\nabla \cdot \tau+F_{\sigma i}+S_{u_i}+S_b \end{aligned} \tag{4.4}$$

Energy equation

The energy equation is also described here in terms of temperature and specific heat. Moreover, as for in the energy equation in the multiphase solver, the latent term here is also added, S_{H_i} but also the aforementioned terms concerning the buoyancy, the pressure and the viscous dissipation.

$$\frac{\partial(\rho C_p T)}{\partial t}+\nabla \cdot\left(u_j \rho C_p T\right)+\nabla \cdot(\mathbf{u} p)=\nabla \cdot\left(k_i \nabla T_i\right)+\nabla \cdot(\boldsymbol{\tau} \cdot \mathbf{u})+\rho g \cdot \mathbf{u}+S_{H_i} \tag{4.5}$$

4.1.4 Governing Equations of the Solid Region

4.1.4.1 Energy Equation

The heat transfer in solids is mainly governed by the heat conduction equation:

$$\frac{\partial(\rho h)}{\partial t}-\nabla \cdot\left(\frac{\lambda}{\rho c_p} \nabla h\right)=0 \tag{4.6}$$

4.1.5 Solver description. Control Loop

chtMultiphaseInterFoam is a new solver derived from the existing solver chtMultiRegionFoam. It is implemented to cope with transient fluid flow and solid heat conduction with conjugate heat transfer between regions.

The solution follows a sequential strategy: equations of the fluid are first solved using the temperatures of the solid of the preceding loop to set the boundary conditions for the fluid part.Then, the equation for the solid is solved with the temperatures of the fluid to define lately the boundary conditions of the solid. This process is iteratively executed until convergence is reached.

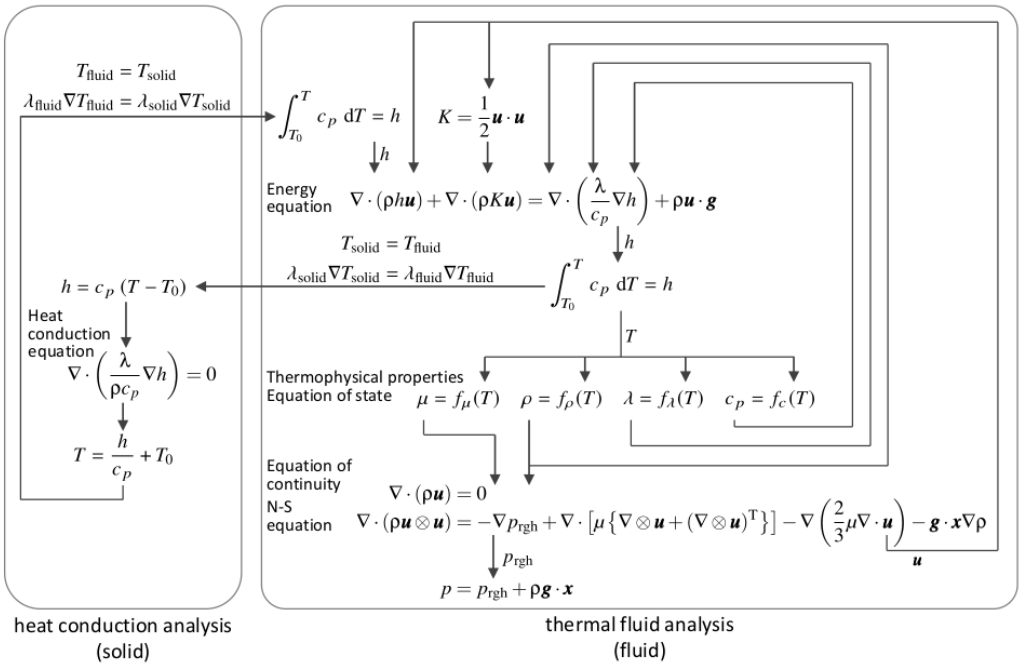


FIGURE 4.2: Flowchart of the conjugate heat transfer solver [20].

4.1.6 Code implementations

As remarked in the previous section, in the context of multiregion solvers, OpenFOAM offers the possibility of solving a fluid representing the compressible Navier-Stokes equations. However, the purpose of this final part is to enhance the capability of this solver so it can handle multiphase fluids submitted to conjugate heat transfer conditions.

To do so, and in favour of using the majority of possibilities that the solver brings, the energy equation in the solid part is kept without change. On the other side, the fluid part of the solver is implemented by integrating the multiphase solver used in the solidification section of this thesis.

The implemented solver can be found in Appendix B but here are presented the main changes.

The first change is the way in which the fluid is solved. In Fig. 4.3 the loop in the fluid is corrected in such a way it incorporates the *IcoReactingMultiphaseInterFoam* solver.

```
if (finalIter)
{
    mesh.data::add("finalIteration", true);
}

#include "initCorrectPhi.H"

if (firstIter)
{
    thermol.correctMassSources(T);
    thermol.solve();
    rho = thermol.rho();

}

if (frozenFlow)
{
    #include "TEqnFluidPhase.H"
}
else
{

    #include "UEqnFluidPhase.H"
    #include "YEqnFluidPhase.H"
    #include "TEqnFluidPhase.H"
        // --- PISO loop

    for (int corr=0; corr<nCorr; corr++)
    {
        #include "pEqnFluidPhase.H"
    }
    if (pimple.turbCorr())
    {
        turbulence.correct();
    }

}

if (finalIter)
{
    rho = thermol.rho();
    mesh.data::remove("finalIteration");
}
```

FIGURE 4.3: Control loop for the fluid region in CHT.

On the other side, the energy equation is adapted so it can cope with multiphase fluids in the scope of multi region solvers. In the way of building up the equation, some of the terms that are newly introduced are the buoyancy energy, $\rho(U\&g)$, and the pressure terms $\nabla \cdot (\mathbf{u}p)$ and the viscous dissipation term, $\nabla \cdot (\boldsymbol{\tau} \cdot \mathbf{u})$, where τ, the viscous stress tensor is calculated in the upper part of the code as the product of 2 by both the dynamic viscosity and the strain rate tensor.

$$\boldsymbol{\tau} = 2\mu\mathbf{D} \tag{4.7}$$

Where $\mathbf{D}$ is the strain rate tensor:

$$\mathbf{D} = -\frac{1}{2}\left[\nabla\mathbf{u} + \nabla\mathbf{u}^{\top}\right] = -\frac{1}{2}\begin{pmatrix} 2\frac{\partial u_x}{\partial x} & \frac{\partial u_y}{\partial x} + \frac{\partial u_x}{\partial y} & \frac{\partial u_z}{\partial x} + \frac{\partial u_x}{\partial z} \\ \frac{\partial u_x}{\partial y} + \frac{\partial u_y}{\partial x} & 2\frac{\partial u_y}{\partial y} & \frac{\partial u_z}{\partial y} + \frac{\partial u_y}{\partial z} \\ \frac{\partial u_x}{\partial z} + \frac{\partial u_z}{\partial x} & \frac{\partial u_y}{\partial z} + \frac{\partial u_z}{\partial y} & 2\frac{\partial u_z}{\partial z} \end{pmatrix} \tag{4.8}$$

If substituting terms, it yields:

$$\boldsymbol{\tau} = -\mu \left[\nabla \mathbf{u} + \nabla \mathbf{u}^{\top} \right] \tag{4.9}$$

```
{
    rhoCp =  rho*thermol.Cp();

    rhok = rho - rhoR;
    const surfaceScalarField rhoCpPhi(fvc::interpolate(thermol.Cp())*rhoPhi);
    volTensorField gradU = fvc::grad(U);
    volTensorField tau = turbulence.muEff() * (gradU + gradU.T());
    const volScalarField kappaEff
    (
        "kappaEff",
        thermol.kappa() + thermol.Cp()*turbulence.mut()/thermol.Prt()
    );


    fvScalarMatrix TEqn
    (
        fvm::ddt(rhoCp, T)
      + fvm::div(rhoCpPhi, T, "div(phi,T)")
      + fvc::div(rhoPhi/fvc::interpolate(rho),p,"div(phiv,p)")
      + fvc::ddt(rho, K) + fvc::div(rhoPhi, K)
      - fvm::laplacian(kappaEff, T,  "laplacian(kappa,T)")
      ==
        thermol.heatTransfer(T)
        + fvc::div(tau & U, "div(tau,U)")
        + rho*(U&g)
        + fvOptions(rhoCp, T)
    );
    TEqn.relax();

    fvOptions.constrain(TEqn);

    TEqn.solve(mesh.solver(T.select(finalIter)));

    fvOptions.correct(T);
    thermol.correct();

    Info<< "min/max(T) = " << min(T).value() << ", " << max(T).value() << endl;
}
```

FIGURE 4.4: Energy equation for the fluid in CHT.

4.1.7 Case Setup

The case geometry is a plane cylinder, taking profit form the fluid mesh generated for the solidification case presented in the previous section. However, in this section, symmetry conditions are applied on the vertical axis (axis Y) so as to reduce the computational cost.

4.1.7.1 Boundary conditions

The boundary conditions are, in this case, setted up for both solid and fluid regions. Here it is shown a table summarizing the used ones.

For the fluid region: For the solid region:

Boundary	**Conditions**
Internal field	$T = 298, u = 0, \alpha_l = 1, \alpha_s = 0$
fluidFrontAndBack	empty
fluidSymmetryBC	symmetryPlane

TABLE 4.1: Boundary conditions for the fluid region in CHT problem.

Boundary	**Conditions**
Internal field	$T = 298$
solidWalls	$T = 258$
solidSymmetryBC	symmetryPlane
solidFrontAndBack	empty

TABLE 4.2: Boundary conditions for the solid region in CHT problem.

At the interface between solid and liquid regions, it is required to set an appropriate boundary condition which couples the energy equations in these areas.

Considering two cells at each side of an interface in where T_c and T_p is the temperature at the cell center and on the patch (2D boundary) accordingly. q_1 is the heat flux going out of the $cell_1$ and q_2 the heat flux entering the $cell_2$. The energy conservation in this zone constrains the temperature and heat fluxes to be equat at both sides of the interface. Then, temperature, in magnitude yields as

$$T_{p,1} = T_{p,2} = T_p, \tag{4.10}$$

and as well, for the fluxes

$$q_1'' = q_2'' = q'' \tag{4.11}$$

while the magnitude for the heat fluxes is derived from the one-dimensional expression for the Fourier's law and it gives

$$-k_1 \left.\frac{\partial T}{\partial n}\right|_{\text{side 1}} = -k_2 \left.\frac{\partial T}{\partial n}\right|_{\text{side 2}} \tag{4.12}$$

where κ is the termal conductivity and n the direction normal to the patch.

Discretizing linearly the temperature gradient of the previous equation, and with respect of the scheme of the Figure [], the differential equation that yields

$$k_1 \Delta_1 \left(T_{c,1} - T_p\right) = k_2 \Delta_2 \left(T_p - T_{c,2}\right) \tag{4.13}$$

where the temperatures and fluxes at the center of the patches are described as

$$T_p = \frac{k_1 \Delta_1 T_{c,1} + k_2 \Delta_2 T_{c,2}}{k_1 \Delta_1 + k_2 \Delta_2} \qquad (4.14)$$
$$q'' = k_1 \Delta_1 \left(T_{c,1} - T_p \right) = k_2 \Delta_2 \left(T_p - T_{c,2} \right).$$

This boundary condition is given in OpenFOAM under the name *turbulentTemperatureCoupledBaffleMixed*. The required input is the temperature at the patch. Therefore, that temperature for the interface between the liquid and the solid regions is initially setted at 298ºC.

4.1.7.2 Thermophysical properties

The thermophysical properties for the fluid are similarly applied as in previous solidification cases. For the solid region, the thermophysical properties are chosen as for the polyethylene.

Polyethylene properties	**Symbol**	**Values**	**Units**
Density	ρ	940	$kg.m^{-3}$
Thermal conductivity	λ	0.56	$W.m^{-1}.K^{-1}$
Heat capacity	C_p	1330	$J.kg.K^{-1}$
Latent heat	L	178600	$J.K^{-1}$

TABLE 4.3: Polyethylene properties for solind region definition.

4.1.8 Validation of Results and Conclusions

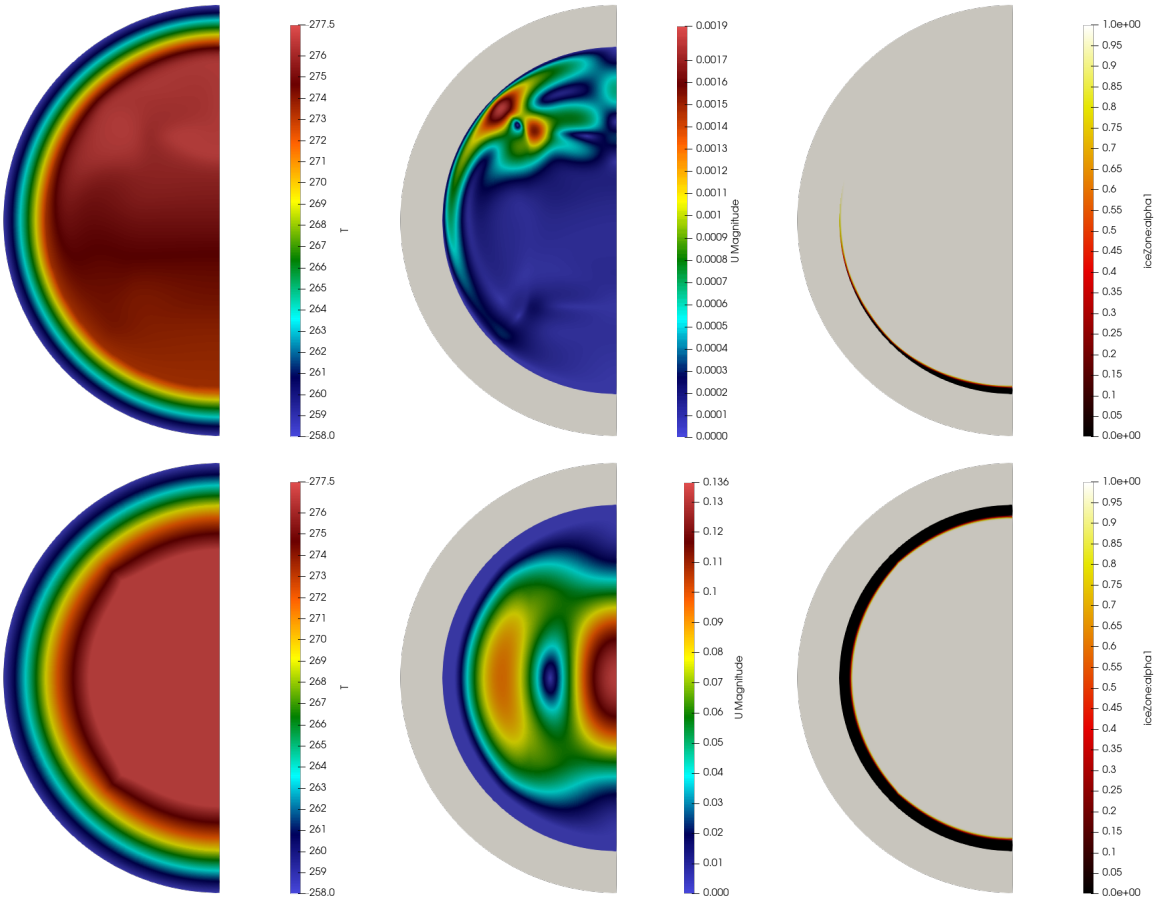
T	**U**	α_l
		

TABLE 4.4: Numerical results of chtMultiRegionFoam (first row) and chtMultiPhaseInterFoam (second row) at $t = 2100s$ in a cylinder.

Chapter 5

Conclusions

Chapter 6

Future Works

Bibliography

[1] M. Akyurt, G. Zaki, and B. Habeebullah. "Freezing phenomena in ice-water systems". In: *Energy Conversion and Management* 43.14 (2002), pp. 1773–1789. DOI: 10.1016/s0196-8904(01)00129-7.

[2] Edin Berberovic et al. "Drop impact onto a liquid layer of finite thickness: Dynamics of the cavity evolution". In: *Physical Review E* 79.3 (2009). DOI: 10.1103/physreve.79.036306.

[3] Arnaud Bourdillon. "Investigation towards a coupling between population balance and solidification models". PhD thesis. Cranfield University, 2016.

[4] Long-Qing Chen. "Phase-Field Models for Microstructure Evolution". In: *Annual Review of Materials Research* 32.1 (2002), pp. 113–140. DOI: 10.1146/annurev.matsci.32.112001.132041.

[5] M El Ganaoui et al. "Computational solution for fluid flow under solid/liquid phase change conditions". In: *Computers Fluids* 31.4-7 (2002), pp. 539–556. DOI: 10.1016/s0045-7930(01)00067-6.

[6] A Esen and S Kutluay. "A numerical solution of the Stefan problem with a Neumann-type boundary condition by enthalpy method". In: *Applied Mathematics and Computation* 148.2 (2004), pp. 321–329. DOI: 10.1016/s0096-3003(02)00846-9.

[7] H. Harlow and J. Welch. "Numerical Calculation of Time-Dependent Viscous Incompressible Flow of Fluid with Free Surface". In: *Physics of Fluids* 8.12 (1965), p. 2182. DOI: 10.1063/1.1761178.

[8] C.W Hirt and B.D Nichols. "Volume of fluid (VOF) method for the dynamics of free boundaries". In: *Journal of Computational Physics* 39.1 (1981), pp. 201–225. DOI: 10.1016/0021-9991(81)90145-5.

[9] Chengyu Huang, Wenhua Wang, and Weizhong Li. "A Novel 2D Model for Freezing Phase Change Simulation during Cryogenic Fracturing Considering Nucleation Characteristics". In: *Applied Sciences* 10.9 (2020), p. 3308. DOI: 10.3390/app10093308.

[10] Luisa Ickes et al. "Classical nucleation theory of homogeneous freezing of water: thermodynamic and kinetic parameters". In: *Physical Chemistry Chemical Physics* 17.8 (2015), pp. 5514–5537. DOI: 10.1039/c4cp04184d.

[11] Damir Juric and Gretar Tryggvason. "A Front-Tracking Method for Dendritic Solidification". In: *Journal of Computational Physics* 123.1 (1996), pp. 127–148. DOI: 10.1006/jcph.1996.0011.

[12] T. A. KOWALEWSKI and M. REBOW. "Freezing of Water in a Differentially Heated Cubic Cavity". In: *International Journal of Computational Fluid Dynamics* 11.3-4 (1999), pp. 193–210. DOI: 10.1080/10618569908940874.

[13] K. Krabbenhoft, L. Damkilde, and M. Nazem. "An implicit mixed enthalpy-temperature method for phase-change problems". In: *Heat and Mass Transfer* 43.3 (2006), pp. 233–241. DOI: 10.1007/s00231-006-0090-1.

[14] Chin Li, Suresh Garimella, and James Simpson. "Fixed-grid front-tracking algorithm for solidification problems, part I: method and validation". In: *Numerical Heat Transfer, Part B: Fundamentals* 43.2 (2003), pp. 117–141. DOI: 10.1080/713836172.

[15] Fadl Hassan Moukalled, L Mangani, and M Darwish. *The finite volume method in computational fluid dynamics*. Springer, 2016, pp. 4–5.

[16] P. Rauschenberger et al. "Comparative assessment of Volume-of-Fluid and Level-Set methods by relevance to dendritic ice growth in supercooled water". In: *Computers Fluids* 79 (2013), pp. 44–52. DOI: 10.1016/j.compfluid.2013.03.010.

[17] Nabeel Al-Rawahi and Gretar Tryggvason. "Numerical Simulation of Dendritic Solidification with Convection: Two-Dimensional Geometry". In: *Journal of Computational Physics* 180.2 (2002), pp. 471–496. DOI: 10.1006/jcph.2002.7092.

[18] C. M. Rhie and W. L. Chow. "Numerical study of the turbulent flow past an airfoil with trailing edge separation". In: *AIAA Journal* 21.11 (1983), pp. 1525–1532. DOI: 10.2514/3.8284.

[19] Fabian Rösler and Dieter Brüggemann. "Shell-and-tube type latent heat thermal energy storage: numerical analysis and comparison with experiments". In: *Heat and Mass Transfer* 47.8 (2011), pp. 1027–1033. DOI: 10.1007/s00231-011-0866-9.

[20] T. Sugimoto et al. "Thermal Fluid Coupled Analysis of Hydrothermal Destruction Reactor". In: *14th WCCM-ECCOMAS Congress* (2021). DOI: 10.23967/wccm-eccomas.2020.342.

[21] Lijian Tan and Nicholas Zabaras. "A level set simulation of dendritic solidification of multi-component alloys". In: *Journal of Computational Physics* 221.1 (2007), pp. 9–40. DOI: 10.1016/j.jcp.2006.06.003.

[22] Vasily Vasilyev and Maria Vasilyeva. "An Accurate Approximation of the Two-Phase Stefan Problem with Coefficient Smoothing". In: *Mathematics* 8.11 (2020), p. 1924. DOI: 10.3390/math8111924.

[23] V. R. Voller and C. Prakash. "A fixed grid numerical modelling methodology for convection-diffusion mushy region phase-change problems". In: *International Journal of Heat and Mass Transfer* 30.8 (1987), pp. 1709–1719. DOI: 10.1016/0017-9310(87)90317-6.

[24] V.R. Voller. "An enthalpy method for modeling dendritic growth in a binary alloy". In: *International Journal of Heat and Mass Transfer* 51.3-4 (2008), pp. 823–834. DOI: 10.1016/j.ijheatmasstransfer.2007.04.025.

[25] Daoyong Wu, Yuanming Lai, and Mingyi Zhang. "Heat and mass transfer effects of ice growth mechanisms in a fully saturated soil". In: *International Journal of Heat and Mass Transfer* 86 (2015), pp. 699–709. DOI: 10.1016/j.ijheatmasstransfer.2015.03.044.

[26] Y. Zhao, C.Y. Zhao, and Z.G. Xu. "Numerical study of solid-liquid phase change by phase field method". In: *Computers Fluids* 164 (2018), pp. 94–101. DOI: 10.1016/j.compfluid.2017.05.032.

Appendix A

Appendix A: Solidification models

A.1 Enthalpy-porosity library

A.1.1 mySolidificationMeltingSource.H

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | www.openfoam.com
     \\/     M anipulation  |
-------------------------------------------------------------------------------
    Copyright (C) 2014-2017 OpenFOAM Foundation
    Copyright (C) 2018-2020 OpenCFD Ltd.
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

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

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

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

```
Class
    Foam::fv::mySolidificationMeltingSource

Group
    grpFvOptionsSources

Description
    This source is designed to model the effect of
       solidification and
    melting processes, e.g. windhield defrosting, within a
       specified region.
    The phase change occurs at the melting temperature, \c Tmelt
       .

    The presence of the solid phase in the flow field is
       incorporated into the
    model as a momentum porosity contribution; the energy
       associated with the
    phase change is added as an enthalpy contribution.

    References:
    \verbatim
        Voller, V. R., & Prakash, C. (1987).
        A fixed grid numerical modelling methodology for
        convection-diffusion mushy region phase-change problems.
        International Journal of Heat and Mass Transfer, 30(8),
           1709-1719.
        DOI:10.1016/0017-9310(87)90317-6

        Swaminathan, C. R., & Voller, V. R. (1992).
        A general enthalpy method for modeling solidification
           processes.
        Metallurgical transactions B, 23(5), 651-664.
        DOI:10.1007/BF02649725
    \endverbatim

    The model generates a field \c \<name\>:alpha1 which can be
       visualised to
    to show the melt distribution as a fraction [0-1].

Usage
    Minimal example by using \c constant/fvOptions:
    \verbatim
    mySolidificationMeltingSource1
    {
        // Mandatory entries (unmodifiable)
        type            mySolidificationMeltingSource;

        // Mandatory entries (runtime modifiable)
        Tmelt           273;
        L               334000;
        thermoMode      <thermoModeName>;
        rhoRef          800;
        beta            5e-6;

        // Optional entries (runtime modifiable)
        relax           0.9;
```

```
        T               <Tname>;
        rho             <rhoName>;
        U               <Uname>;
        phi             <phiName>;
        Cu              1e5;
        q               1e-2;

        // Conditional optional entries (runtime modifiable)

            // when thermoMode=lookup
            Cp          Cp;

        // Conditional mandatory entries (runtime modifiable)

            // when Cp=CpRef
            CpRef   1000;

        // Mandatory/Optional (inherited) entries
        ...
    }
    \endverbatim

    where the entries mean:
    \table
      Property  | Description                          | Type    |
         Reqd | Dflt
      type      | Type name: mySolidificationMeltingSource  |
         word   | yes   | -
      Tmelt     | Melting temperature [K]              | scalar |
         yes  | -
      L         | Latent heat of fusion [J/kg]         | scalar |
         yes  | -
      thermoMode  | Thermo mode                        | word    |
         yes  | -
      rhoRef    | Reference (solid) density            | scalar |
         yes  | -
      beta      | Thermal expansion coefficient [1/K] | scalar |
          yes | -
      relax     | Relaxation factor [0-1]              | scakar |
         no   | 0.9
      T         | Name of operand temperature field | word    |
         no   | T
      rho       | Name of operand density field      | word    |
         no   | rho
      U         | Name of operand velocity field     | word    |
         no   | U
      phi       | Name of operand flux field         | word    |
         no   | phi
      Cu        | Mushy region momentum sink coefficient [1/s]
         <!--
                --->                                 | scalar |
                    no   | 1e5
      q         | Coefficient used in porosity calc | scalar |
         no   | 1e-2
      Cp        | Name of specific heat capacity field | word |
         cndtnl | Cp
```

```
      CpRef      | Specific heat capacity value       | scalar |
         cndtnl  | -
    \endtable

    The inherited entries are elaborated in:
     - \link fvOption.H \endlink
     - \link cellSetOption.H \endlink

    Options for the \c thermoMode entry:
    \verbatim
      thermo     | Access Cp information from database
      lookup     | Access Cp information by looking up from
         dictionary
    \endverbatim

SourceFiles
    mySolidificationMeltingSource.C
    mySolidificationMeltingSourceTemplates.C

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

#ifndef mySolidificationMeltingSource_H
#define mySolidificationMeltingSource_H

#include "fvMesh.H"
#include "volFields.H"
#include "cellSetOption.H"
#include "Enum.H"

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

namespace Foam
{
namespace fv
{

/*---------------------------------------------------------------------
   -------------*\
                Class mySolidificationMeltingSource Declaration
\*---------------------------------------------------------------------
   -------------*/

class mySolidificationMeltingSource
:
    public cellSetOption
{

    // Private Data

        //- Temperature at which melting occurs [K]
        scalar Tmelt_;

        //- Latent heat of fusion [J/kg]
        scalar L_;
```

```
        //- Phase fraction under-relaxation coefficient
        scalar relax_;

        //- Name of operand temperature field
        word TName_;

        //- Name of specific heat capacity field
        word CpName_;

        //- Name of operand velocity field
        word UName_;

        //- Name of operand flux field
        word rhoCpPhiName_;

        //- Calculated Phase fraction indicator field for PCM
        volScalarField alphaC_;

        //- Phase fraction indicator field for VOF
        volScalarField alpha1_;

        //- Current time index (used for updating)
        label curTimeIndex_;

        void update();

        //- Helper function to apply to the energy equation
        template<class RhoFieldType>
        void apply(const RhoFieldType& rho, fvMatrix<scalar>&
            eqn);


public:

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


    // Constructors

        //- Construct from explicit source name and mesh
        mySolidificationMeltingSource
        (
            const word& sourceName,
            const word& modelType,
            const dictionary& dict,
            const fvMesh& mesh
        );

        //- No copy construct
        mySolidificationMeltingSource
        (
            const mySolidificationMeltingSource&
        ) = delete;

        //- No copy assignment
```

```
        void operator=(const mySolidificationMeltingSource&) =
            delete;


    //- Destructor
    ~mySolidificationMeltingSource() = default;


    // Member Functions

        //- Add explicit contribution to enthalpy equation
        virtual void addSup(fvMatrix<scalar>& eqn, const label
            fieldi);

        //- Add explicit contribution to compressible enthalpy
            equation
        virtual void addSup
        (
            const volScalarField& rho,
            fvMatrix<scalar>& eqn,
            const label fieldi
        );

        //- Read source dictionary
        virtual bool read(const dictionary& dict);
};


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

} // End namespace fv
} // End namespace Foam

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

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

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

#endif

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

A.1.2 mySolidificationMeltingSource.C

```
/*---------------------------------------------------------------
    ------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD
     Toolbox
   \\    /   O peration     |
```

```
    \\  /    A nd           | www.openfoam.com
     \\/     M anipulation  |
-------------------------------------------------------------------------
    ---------------
    Copyright (C) 2014-2017 OpenFOAM Foundation
    Copyright (C) 2018-2020 OpenCFD Ltd.
-------------------------------------------------------------------------
    ---------------
License
    This file is part of OpenFOAM.

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

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

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

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

#include "mySolidificationMeltingSource.H"
#include "fvMatrices.H"
#include "basicThermo.H"
#include "gravityMeshObject.H"
#include "zeroGradientFvPatchFields.H"
#include "extrapolatedCalculatedFvPatchFields.H"
#include "addToRunTimeSelectionTable.H"
#include "geometricOneField.H"
#include <cmath>
// * * * * * * * * * * * * * * Static Member Functions * * * * * *
    * * * * * * //

namespace Foam
{
namespace fv
{
    defineTypeNameAndDebug(mySolidificationMeltingSource, 0);
    addToRunTimeSelectionTable(option,
       mySolidificationMeltingSource, dictionary);
}
}
```

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

void Foam::fv::mySolidificationMeltingSource::update()
{
    if (curTimeIndex_ == mesh_.time().timeIndex())
    {
        return;
    }

    if (debug)
    {
        Info<< type() << ": " << "alpha.liquid" << " - updating
            phase indicator" << endl;
    }

    // update old time alpha1 field
    alphaC_.oldTime();
    const auto& CpVoF = mesh_.lookupObject<volScalarField>(
        CpName_);

    const auto& T = mesh_.lookupObject<volScalarField>(TName_);
    scalar Tsol = Tmelt_-0.25;
    scalar Tliq = Tmelt_+0.75;
    scalar eps = 0.0001;
    forAll(cells_, i)
    {
        label celli = cells_[i];

        scalar Tc = T[celli];
        scalar Cpc = CpVoF[celli];
        scalar alpha1New = alphaC_[celli] + relax_*Cpc*(Tc -
            Tmelt_)/L_;
        // scalar alpha1New = 0.5 + 0.5*std::erf(4 * ((Tc - (
            Tliq + Tsol)/2)/(Tliq - Tsol + eps)));
        alphaC_[celli] = max(0, min(alpha1New, 1));
    }

    alpha1_.correctBoundaryConditions();

    curTimeIndex_ = mesh_.time().timeIndex();
}


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

Foam::fv::mySolidificationMeltingSource::
    mySolidificationMeltingSource
(
    const word& sourceName,
    const word& modelType,
    const dictionary& dict,
    const fvMesh& mesh
)
:
```

```
    cellSetOption(sourceName, modelType, dict, mesh),
    Tmelt_(coeffs_.get<scalar>("Tmelt")),
    L_(coeffs_.get<scalar>("L")),
    relax_(coeffs_.getOrDefault<scalar>("relax", 0.9)),
    TName_(coeffs_.getOrDefault<word>("T", "T")),
    CpName_(coeffs_.getOrDefault<word>("Cp", "Cp")),
    UName_(coeffs_.getOrDefault<word>("U", "U")),
    rhoCpPhiName_(coeffs_.getOrDefault<word>("rhoCpPhi", "
        rhoCpPhi")),
    alphaC_
    (
        IOobject
        (
            "alphaPCM",
            mesh.time().timeName(),
            mesh,
            IOobject::READ_IF_PRESENT,
            IOobject::NO_WRITE
        ),
        mesh,
        dimensionedScalar(dimless, Zero),
        zeroGradientFvPatchScalarField::typeName
    ),
    alpha1_
    (
        IOobject
        (
            "alpha.liquid",
            mesh.time().timeName(),
            mesh,
            IOobject::READ_IF_PRESENT,
            IOobject::AUTO_WRITE
        ),
        mesh,
        dimensionedScalar(dimless, Zero),
        zeroGradientFvPatchScalarField::typeName
    ),
    curTimeIndex_(-1)
{
    fieldNames_.resize(2);
    fieldNames_[0] = UName_;
    fieldNames_[1] = TName_;

    fv::option::resetApplied();
}


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

void Foam::fv::mySolidificationMeltingSource::addSup
(
    fvMatrix<scalar>& eqn,
    const label fieldi
)
{
    apply(geometricOneField(), eqn);
```

```
}


void Foam::fv::mySolidificationMeltingSource::addSup
(
    const volScalarField& rho,
    fvMatrix<scalar>& eqn,
    const label fieldi
)
{
    apply(rho, eqn);
}


bool Foam::fv::mySolidificationMeltingSource::read(const
   dictionary& dict)
{
    if (cellSetOption::read(dict))
    {
        coeffs_.readEntry("Tmelt", Tmelt_);
        coeffs_.readEntry("L", L_);

        coeffs_.readIfPresent("relax", relax_);
        coeffs_.readIfPresent("T", TName_);
        coeffs_.readIfPresent("U", UName_);
        coeffs_.readIfPresent("Cp", CpName_);
        coeffs_.readIfPresent("rhoCpPhi", rhoCpPhiName_);

        return true;
    }

    return false;
}


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

A.1.3 mySolidificationMeltingSourceTemplates.C

```
/*---------------------------------------------------------------
   ------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD
     Toolbox
   \\    /   O peration     |
    \\  /    A nd           | www.openfoam.com
     \\/     M anipulation  |
-----------------------------------------------------------------
   ---------------
    Copyright (C) 2014-2015 OpenFOAM Foundation
-----------------------------------------------------------------
   ---------------
License
    This file is part of OpenFOAM.
```

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

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

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

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

#include "fvMatrices.H"
#include "fvcDdt.H"
#include "fvcDiv.H"
#include "basicThermo.H"

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

template<class RhoFieldType>
void Foam::fv::mySolidificationMeltingSource::apply
(
    const RhoFieldType& rho,
    fvMatrix<scalar>& eqn
)
{
    if (debug)
    {
        Info<< type() << ":␣applying␣source␣to␣" << eqn.psi().
            name() << endl;
    }

    update();
    const auto& CpVoF = mesh_.lookupObject<volScalarField>(
        CpName_);
    const auto& rhoCpPhiVoF = mesh_.lookupObject<
        surfaceScalarField>(rhoCpPhiName_);
    dimensionedScalar L("L", dimEnergy/dimMass, L_);

    // contributions added to rhs of solver equation
    if (eqn.psi().dimensions() == dimTemperature)
    {
```

```
        eqn -= (L*alpha1_)/CpVoF*(fvc::ddt(rho, alphaC_) + fvc::
            div(rhoCpPhiVoF, alphaC_));

    }
    else
    {
        //This option is not activated since fvOptions in TEqn
            does not enable this condition
        eqn -= L*alpha1_*(fvc::ddt(rho, alphaC_));
    }
}


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

A.2 Lee-Nucleation library

A.2.1 LeeCNT.H

```
/*---------------------------------------------------------------
    ------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD
     Toolbox
   \\    /   O peration     |
    \\  /    A nd           | www.openfoam.com
     \\/     M anipulation  |
-----------------------------------------------------------------
    ---------------
    Copyright (C) 2017-2020 OpenCFD Ltd.
-----------------------------------------------------------------
    ---------------
License
    This file is part of OpenFOAM.

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

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

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

```
Class
    Foam::meltingEvaporationModels::LeeCNT

Description
    Mass transfer LeeCNT model. Simple model driven by field
       value difference as:

    \f[
        \dot{m} = C \rho \alpha (T - T_{activate})/T_{activate}
    \f]

    where C is a model constant.

    if C > 0:
    \f[
        \dot{m} = C \rho \alpha (T - T_{activate})/T_{activate}
    \f]
        for \f[ T > T_{activate} \f]

    and

        \f[ mDot = 0.0 \f]  for \f[  T < T_{activate} \f]


    if C < 0:
    \f[
        \dot{m} = -C \rho \alpha (T_{activate} - T)/T_{activate}
    \f]
        for \f[ T < T_{activate} \f]

    and
        \f[ \dot{m} = 0.0 \f] for \f[ T > T_{activate} \f]

    Based on the reference:
    -# W. H. LeeCNT. "A Pressure Iteration Scheme for Two-Phase
       Modeling".
    Technical Report LA-UR 79-975. Los Alamos Scientific
       Laboratory,
    Los Alamos, New Mexico. 1979.

Usage
    Example usage:
    \verbatim
        massTransferModel
        (
            (solid to liquid)
            {
                type            LeeCNT;
                C               40;
                Tactivate       302.78;
            }
        );
    \endverbatim

    Where:
```

```
    \table
        Property     | Description             | Required    |
           Default value
        Tactivate    | Activation temperature | yes
        C            | Model constant          | yes
        includeVolChange    | Volumen change  | no          |
           yes
        species      | Specie name on the other phase | no   |
           none
    \endtable

SourceFiles
    LeeCNT.C

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

#ifndef meltingEvaporationModels_LeeCNT_H
#define meltingEvaporationModels_LeeCNT_H

#include "InterfaceCompositionModel.H"

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

namespace Foam
{
namespace meltingEvaporationModels
{

/*---------------------------------------------------------------
   ------------*\
                            Class LeeCNT Declaration
\*---------------------------------------------------------------
   -------------*/

template<class Thermo, class OtherThermo>
class LeeCNT
:
    public InterfaceCompositionModel<Thermo, OtherThermo>
{
    // Private Data

        //- Condensation coefficient [1/s]
        dimensionedScalar C_;

        volScalarField interfaceVolume_;
        //- Phase transition temperature
        const dimensionedScalar Tactivate_;

        //- Phase minimum value for activation
        scalar alphaMin_;

        //- Planck constant [J.s]
        const dimensionedScalar planck_;

        //- Boltzmann constant [J/K]
```

```
        const dimensionedScalar boltzmann_;

        //- Activation energy of water molecules passing through
            water-ice interface [J]
        const dimensionedScalar deltag_;

        //- Number of water molecule in a water volume [m3]
        const dimensionedScalar nL_;

        //- Superficial free energy of the water-ice interface [
           J/m2]
        const dimensionedScalar gammaYW_;

        //- Latent heat per volume [J/m3]
        const dimensionedScalar hLV_;

        //- Shape coefficient of nucleation
        const dimensionedScalar alphaEY_;


public:

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


    // Constructors

        //- Construct from components
        LeeCNT
        (
            const dictionary& dict,
            const phasePair& pair
        );


    //- Destructor
    virtual ~LeeCNT() = default;


    // Member Functions

        //- Explicit total mass transfer coefficient
        virtual tmp<volScalarField> Kexp
        (
            const volScalarField& field
        );

        //- Implicit mass transfer coefficient
        virtual tmp<volScalarField> KSp
        (
            label modelVariable,
            const volScalarField& field
        );

        //- Explicit mass transfer coefficient
        virtual tmp<volScalarField> KSu
```

```
        (
            label modelVariable,
            const volScalarField& field
        );

        //- Return T transition between phases
        virtual const dimensionedScalar& Tactivate() const;

        //- Add/subtract alpha*div(U) as a source term
        //- for alpha, substituting div(U) = mDot(1/rho1 - 1/
           rho2)
        virtual bool includeDivU();

        virtual const dimensionedScalar& planck() const;

        virtual const dimensionedScalar& boltzmann() const;

        virtual const dimensionedScalar& deltag() const;

        virtual const dimensionedScalar& nL() const;

        virtual const dimensionedScalar& gammaYW() const;

        virtual const dimensionedScalar& hLV() const;

        virtual const dimensionedScalar& alphaEY() const;


};


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

} // End namespace meltingEvaporationModels
} // End namespace Foam

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

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

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

#endif

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

A.2.2 LeeCNT.C

```
/*---------------------------------------------------------------
   ------------*\
  =========                 |
```

```
  \\      /  F ield          | OpenFOAM: The Open Source CFD
     Toolbox
   \\    /   O peration      |
    \\  /    A nd            | www.openfoam.com
     \\/     M anipulation   |
-------------------------------------------------------------------
    ---------------
    Copyright (C) 2017-2020 OpenCFD Ltd.
-------------------------------------------------------------------
    ---------------
License
    This file is part of OpenFOAM.

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

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

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

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

#include "LeeCNT.H"
#include "addToRunTimeSelectionTable.H"
#include "mathematicalConstants.H"


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

template<class Thermo, class OtherThermo>
Foam::meltingEvaporationModels::LeeCNT<Thermo, OtherThermo>::
    LeeCNT
(
    const dictionary& dict,
    const phasePair& pair
)
:
    InterfaceCompositionModel<Thermo, OtherThermo>(dict, pair),
    C_("C", inv(dimTime), dict),
    Tactivate_("Tactivate", dimTemperature, dict),
    planck_("planck", dimEnergy*dimTime, dict),
```

```
    boltzmann_("boltzmann", dimEnergy/dimTemperature, dict),
    deltag_("deltag", dimEnergy, dict),
    nL_("nL", inv(dimVolume), dict),
    gammaYW_("gammaYW", dimEnergy/dimArea, dict),
    hLV_("hLV", dimEnergy/dimVolume, dict),
    alphaEY_("alphaEY", dict),
    alphaMin_(dict.getOrDefault<scalar>("alphaMin", 0)),
    interfaceVolume_
    (
        IOobject
        (
            "cellVolume",
            this->mesh_.time().timeName(),
            this->mesh_,
            IOobject::NO_READ,
            IOobject::NO_WRITE
        ),
        this->mesh_,
        dimensionedScalar(dimVolume, Zero)
    )
{}


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


template<class Thermo, class OtherThermo>
Foam::tmp<Foam::volScalarField>
Foam::meltingEvaporationModels::LeeCNT<Thermo, OtherThermo>::
   Kexp
(
    const volScalarField& refValue
)
{

    {
        const fvMesh& mesh = this->mesh_;
        const volScalarField deltaG
        (
            (16.0 * (constant::mathematical::pi) * pow(gammaYW_
               ,3.0) * pow(Tactivate_,2.0) * alphaEY_)/(3.0 *
               pow(hLV_,2.0) * pow((Tactivate_ - refValue),2.0)
               )
        );

        const volScalarField J
        (
            (boltzmann_*refValue)/(planck_) * (exp(-deltag_/(
               boltzmann_*refValue))  * nL_ * exp(-deltaG/(
               boltzmann_*refValue)))
        );

        forAll(interfaceVolume_, celli)
        {
            interfaceVolume_[celli] = mesh.V()[celli];
        }
```

```
        const volScalarField lambda
        (
            J*interfaceVolume_
        );

        const volScalarField from
        (
            min(max(this->pair().from(), scalar(0)), scalar(1))
        );

        const volScalarField coeff
        (
            C_*from*this->pair().from().rho()*pos(from -
               alphaMin_)
           *(refValue - Tactivate_)
           /Tactivate_
        );

        const volScalarField coeff1
        (
            -lambda*from*this->pair().from().rho()*pos(from -
               alphaMin_)
           *(refValue - Tactivate_)
           /Tactivate_
        );

        if (sign(C_.value()) > 0)
        {
            return
            (
                coeff*pos(refValue - Tactivate_)
            );
        }
        else
        {
            return
            (
                coeff1*pos(Tactivate_ - refValue)
            );
        }
    }
}


template<class Thermo, class OtherThermo>
Foam::tmp<Foam::volScalarField>
Foam::meltingEvaporationModels::LeeCNT<Thermo, OtherThermo>::KSp
(
    label variable,
    const volScalarField& refValue
)
{

    if (this->modelVariable_ == variable)
    {
        const fvMesh& mesh = this->mesh_;
```

```
        const volScalarField deltaG
        (
            (16.0 * (constant::mathematical::pi) * pow(gammaYW_
               ,3.0) * pow(Tactivate_,2.0) * alphaEY_)
            /(3.0 * pow(hLV_,2.0) * pow((Tactivate_ - refValue)
               ,2.0))
        );

        const volScalarField J
        (
            (boltzmann_*refValue)/(planck_) * (exp(-deltag_/(
               boltzmann_*refValue)) * nL_ * exp(-deltaG/(
               boltzmann_*refValue)))
        );

        forAll(interfaceVolume_, celli)
        {
            interfaceVolume_[celli] = mesh.V()[celli];
        }
        const volScalarField lambda
        (
            J*interfaceVolume_
        );

        volScalarField from
        (
            min(max(this->pair().from(), scalar(0)), scalar(1))
        );

        const volScalarField coeff
        (
            C_*from*this->pair().from().rho()*pos(from -
               alphaMin_)
            /Tactivate_
        );

        const volScalarField coeff1
        (
            -lambda*from*this->pair().from().rho()*pos(from -
               alphaMin_)
            /Tactivate_
        );

        if (sign(C_.value()) > 0)
        {
            return
            (
                coeff*pos(refValue - Tactivate_)
            );
        }
        else
        {
            return
            (
                coeff1*pos(Tactivate_ - refValue)
            );
```

```
        }
    }
    else
    {
        return tmp<volScalarField> ();
    }
}


template<class Thermo, class OtherThermo>
Foam::tmp<Foam::volScalarField>
Foam::meltingEvaporationModels::LeeCNT<Thermo, OtherThermo>::KSu
(
    label variable,
    const volScalarField& refValue
)
{

    if (this->modelVariable_ == variable)
    {
        const fvMesh& mesh = this->mesh_;

        const volScalarField deltaG
        (
            (16.0 * (constant::mathematical::pi) * pow(gammaYW_
                ,3.0) * pow(Tactivate_,2.0) * alphaEY_)
            /(3.0 * pow(hLV_,2.0) * pow((Tactivate_ - refValue)
                ,2.0))
        );

        const volScalarField J
        (
            (boltzmann_*refValue)/(planck_) * (exp(-deltag_/(
                boltzmann_*refValue)) * nL_ * exp(-deltaG/(
                boltzmann_*refValue)))
        );

        forAll(interfaceVolume_, celli)
        {
            interfaceVolume_[celli] = mesh.V()[celli];
        }
        const volScalarField lambda
        (
            J*interfaceVolume_
        );

        volScalarField from
        (
            min(max(this->pair().from(), scalar(0)), scalar(1))
        );

        const volScalarField coeff
        (
            C_*from*this->pair().from().rho()*pos(from -
                alphaMin_)
        );
```

```
        const volScalarField coeff1
        (
            -lambda*from*this->pair().from().rho()*pos(from -
               alphaMin_)
        );

        if (sign(C_.value()) > 0)
        {
            return
            (
                -coeff*pos(refValue - Tactivate_)
            );
        }
        else
        {
            return
            (
                -coeff1*pos(Tactivate_ - refValue)
            );
        }
    }
    else
    {
        return tmp<volScalarField> ();
    }
}


template<class Thermo, class OtherThermo>
const Foam::dimensionedScalar&
Foam::meltingEvaporationModels::LeeCNT<Thermo, OtherThermo>::
   Tactivate() const
{
    return Tactivate_;
}

template<class Thermo, class OtherThermo>
const Foam::dimensionedScalar&
Foam::meltingEvaporationModels::LeeCNT<Thermo, OtherThermo>::
   planck() const
{
    return planck_;
}

template<class Thermo, class OtherThermo>
const Foam::dimensionedScalar&
Foam::meltingEvaporationModels::LeeCNT<Thermo, OtherThermo>::
   boltzmann() const
{
    return boltzmann_;
}

template<class Thermo, class OtherThermo>
const Foam::dimensionedScalar&
Foam::meltingEvaporationModels::LeeCNT<Thermo, OtherThermo>::
   deltag() const
{
```

```
    return deltag_;
}

template<class Thermo, class OtherThermo>
const Foam::dimensionedScalar&
Foam::meltingEvaporationModels::LeeCNT<Thermo, OtherThermo>::nL
    () const
{
    return nL_;
}

template<class Thermo, class OtherThermo>
const Foam::dimensionedScalar&
Foam::meltingEvaporationModels::LeeCNT<Thermo, OtherThermo>::
    gammaYW() const
{
    return gammaYW_;
}

template<class Thermo, class OtherThermo>
const Foam::dimensionedScalar&
Foam::meltingEvaporationModels::LeeCNT<Thermo, OtherThermo>::hLV
    () const
{
    return hLV_;
}

template<class Thermo, class OtherThermo>
const Foam::dimensionedScalar&
Foam::meltingEvaporationModels::LeeCNT<Thermo, OtherThermo>::
    alphaEY() const
{
    return alphaEY_;
}

template<class Thermo, class OtherThermo>
bool
Foam::meltingEvaporationModels::LeeCNT<Thermo, OtherThermo>::
    includeDivU()
{
    return true;
}


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

A.2.3 Library header files

```
/*---------------------------------------------------------------
    ------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD
     Toolbox
   \\    /   O peration     |
    \\  /    A nd           | www.openfoam.com
     \\/     M anipulation  |
```

```
-------------------------------------------------------------------------
    ---------------
     Copyright (C) 2017 OpenCFD Ltd.
-------------------------------------------------------------------------
    ---------------
License
     This file is part of OpenFOAM.

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

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

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

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

#include "addToRunTimeSelectionTable.H"

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

#include "interfaceCompositionModel.H"
#include "InterfaceCompositionModel.H"
//#include "thermoPhysicsTypes.H"
#include "myThermoPhysicsTypes.H"

#include "bPolynomial.H"

#include "pureMixture.H"

#include "rhoThermo.H"

#include "heRhoThermo.H"

#include "solidThermo.H"
#include "heSolidThermo.H"
#include "solidThermoPhysicsTypes.H"

#include "Lee.H"
#include "LeeCNT.H"
```

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

#define makeInterfacePureType(Type, Thermo, Comp, Mix, Phys,
    OtherThermo, OtherComp, OtherMix, OtherPhys)\
                                                                          \

    typedef Thermo<Comp, Mix<Phys>>
                                                                \
        Type##Thermo##Comp##Mix##Phys;
                                                                 \
                                                                          \

    typedef OtherThermo<OtherComp, OtherMix<OtherPhys>>
                                    \
        Type##Other##OtherThermo##OtherComp##OtherMix##OtherPhys
           ;                \
                                                                          \

    addInterfaceCompositionToRunTimeSelectionTable
                                       \
    (

        \
        Type,

            \
        Type##Thermo##Comp##Mix##Phys,
                                                                \
        Type##Other##OtherThermo##OtherComp##OtherMix##OtherPhys
                             \
    )

// Addition to the run-time selection table
#define addInterfaceCompositionToRunTimeSelectionTable(Type,
    Thermo, OtherThermo)\
                                                                          \

    typedef Type<Thermo, OtherThermo>
                                                          \
        Type##Thermo##OtherThermo;
                                                                  \
                                                                          \

    defineTemplateTypeNameAndDebugWithName
                                                 \
    (

        \
        Type##Thermo##OtherThermo,
                                                                  \
        (

            \
            word(Type##Thermo##OtherThermo::typeName_()) + "<"
                                    \
```

```
            + word(Thermo::typeName) + ","                              \
            + word(OtherThermo::typeName) + ">"                         \
        ).c_str(),                                                      \
        0                                                               \
    );                                                                  \

    addToRunTimeSelectionTable                                          \
    (                                                                   \
        interfaceCompositionModel,                                      \
        Type##Thermo##OtherThermo,                                      \
        dictionary                                                      \
    )

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
namespace Foam
{
    using namespace meltingEvaporationModels;

    //NOTE: First thermo (from) and second otherThermo (to)


    // Lee model definitions

        // From pure phase (poly) to phase (solidThermo)
        makeInterfacePureType
        (
            Lee,
            heRhoThermo,
            rhoThermo,
            pureMixture,
            constbPolFluidHThermoPhysics,
            heSolidThermo,
            solidThermo,
            pureMixture,
            hConstSolidThermoPhysics
        );

        makeInterfacePureType
        (
            LeeCNT,
```

```
            heRhoThermo,
            rhoThermo,
            pureMixture,
            constbPolFluidHThermoPhysics,
            heSolidThermo,
            solidThermo,
            pureMixture,
            hConstSolidThermoPhysics
        );

        makeInterfacePureType
        (
            Lee,
            heRhoThermo,
            rhoThermo,
            pureMixture,
            polybPolFluidHThermoPhysics,
            heSolidThermo,
            solidThermo,
            pureMixture,
            hConstSolidThermoPhysics
        );

        makeInterfacePureType
        (
            LeeCNT,
            heRhoThermo,
            rhoThermo,
            pureMixture,
            polybPolFluidHThermoPhysics,
            heSolidThermo,
            solidThermo,
            pureMixture,
            hConstSolidThermoPhysics
        );
        // interfaceHeatResistance model definitions

        // From pure phase (poly) to phase (solidThermo)
        // makeInterfacePureType
        // (
        //     interfaceHeatResistance,
        //     heRhoThermo,
        //     rhoThermo,
        //     pureMixture,
        //     constbPolFluidHThermoPhysics,
        //     heSolidThermo,
        //     solidThermo,
        //     pureMixture,
        //     hConstSolidThermoPhysics
        // );

}


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

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | www.openfoam.com
     \\/     M anipulation  |
-------------------------------------------------------------------------------
    Copyright (C) 2011-2017 OpenFOAM Foundation
    Copyright (C) 2020-2021 OpenCFD Ltd.
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

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

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

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

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

#include "rhoThermo.H"
#include "makeThermo.H"

#include "specie.H"
#include "bPolynomial.H"

#include "hConstThermo.H"
#include "sensibleEnthalpy.H"
#include "thermo.H"

#include "constTransport.H"
#include "polynomialTransport.H"
#include "heRhoThermo.H"
#include "pureMixture.H"
```

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

namespace Foam
{

/* * * * * * * * * * * * * * * * Private Static Data * * * * * * * * * * *
    * * * * * * */


makeThermos
(
    rhoThermo,
    heRhoThermo,
    pureMixture,
    constTransport,
    sensibleEnthalpy,
    hConstThermo,
    bPolynomial,
    specie
);
makeThermos
(
    rhoThermo,
    heRhoThermo,
    pureMixture,
    polynomialTransport,
    sensibleEnthalpy,
    hConstThermo,
    bPolynomial,
    specie
);

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

} // End namespace Foam

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

```
/*---------------------------------------------------------------------
    ------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD
     Toolbox
   \\    /   O peration     |
    \\  /    A nd           | www.openfoam.com
     \\/     M anipulation  |
-----------------------------------------------------------------------
    ---------------
    Copyright (C) 2012-2017 OpenFOAM Foundation
    Copyright (C) 2018 OpenCFD Ltd.
-----------------------------------------------------------------------
    ---------------
License
    This file is part of OpenFOAM.
```

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

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

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

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

#include "makeReactionThermo.H"

#include "rhoReactionThermo.H"
#include "heRhoThermo.H"

#include "specie.H"
// #include "perfectGas.H"
// #include "incompressiblePerfectGas.H"
#include "hConstThermo.H"
// #include "janafThermo.H"
#include "sensibleEnthalpy.H"
#include "thermo.H"
// #include "rhoConst.H"
// #include "rPolynomial.H"
// #include "perfectFluid.H"
// #include "adiabaticPerfectFluid.H"
// #include "Boussinesq.H"

#include "constTransport.H"
#include "polynomialTransport.H"
// #include "homogeneousMixture.H"
// #include "inhomogeneousMixture.H"
// #include "veryInhomogeneousMixture.H"
// #include "multiComponentMixture.H"
// #include "reactingMixture.H"
// #include "singleStepReactingMixture.H"
#include "singleComponentMixture.H"

#include "myThermoPhysicsTypes.H"

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

```
namespace Foam
{

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

makeReactionThermo
(
    rhoReactionThermo,
    heRhoThermo,
    singleComponentMixture,
    constTransport,
    sensibleEnthalpy,
    hConstThermo,
    bPolynomial,
    specie
);

makeReactionThermo
(
    rhoReactionThermo,
    heRhoThermo,
    singleComponentMixture,
    polynomialTransport,
    sensibleEnthalpy,
    hConstThermo,
    bPolynomial,
    specie
);


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

} // End namespace Foam

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

A.3 equationOfState

equationOfState

```
/*---------------------------------------------------------------------------
    ------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD
     Toolbox
   \\    /   O peration     |
    \\  /    A nd           | www.openfoam.com
     \\/     M anipulation  |
-------------------------------------------------------------------------------
    ---------------
    Copyright (C) 2011-2017 OpenFOAM Foundation
```

```
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

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

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

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

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

#include "bPolynomial.H"
#include "IOstreams.H"

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

namespace Foam
{

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

template<class Specie, int PolySize>
bPolynomial<Specie, PolySize>::bPolynomial(const dictionary& dict)
:
    Specie(dict),
    rhoRef_(dict.subDict("equationOfState").get<scalar>("rhoRef")),
    Tref_(dict.subDict("equationOfState").get<scalar>("Tref")),
    rhoCoeffs_(dict.subDict("equationOfState").lookup(coeffsName("rho")))
{}


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

```
template<class Specie, int PolySize>
void bPolynomial<Specie, PolySize>::write(Ostream& os) const
{
    Specie::write(os);

    // Entries in dictionary format
    {
        os.beginBlock("equationOfState");
        os.writeEntry("rhoRef", rhoRef_);
        os.writeEntry("Tref", Tref_);
        os.writeEntry(coeffsName("rho"), rhoCoeffs_);
        os.endBlock();
    }
}


// * * * * * * * * * * * * * * * * * Ostream Operator  * * * * * * *
    * * * * * * //

template<class Specie, int PolySize>
Ostream& operator<<(Ostream& os, const bPolynomial<Specie,
   PolySize>& ip)
{
    ip.write(os);
    return os;
}


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

} // End namespace Foam

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

```
/*---------------------------------------------------------------
   ------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD
     Toolbox
   \\    /   O peration     |
    \\  /    A nd           | www.openfoam.com
     \\/     M anipulation  |
-----------------------------------------------------------------
   ---------------
    Copyright (C) 2011-2017 OpenFOAM Foundation
    Copyright (C) 2020 OpenCFD Ltd.
-----------------------------------------------------------------
   ---------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software: you can redistribute it and/or
       modify it
    under the terms of the GNU General Public License as
       published by
```

```
    the Free Software Foundation, either version 3 of the
       License, or
    (at your option) any later version.

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

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

Class
    Foam::bPolynomial

Group
    grpSpecieEquationOfState

Description
    Incompressible, polynomial form of equation of state, using
       a polynomial
    function for density.

Usage
    \table
        Property     | Description
        rhoCoeffs<8> | Density polynomial coefficients
    \endtable

    Example of the specification of the equation of state:
    \verbatim
    equationOfState
    {
        rhoCoeffs<8>    ( 1000 -0.05 0.003 0 0 0 0 0 );
    }
    \endverbatim

    The polynomial expression is evaluated as so:

        \f[
            \rho = 1000 - 0.05 T + 0.003 T^2
        \f]

Note
    Input in [kg/m3], but internally uses [kg/m3/kmol].

SourceFiles
    bPolynomialI.H
    bPolynomial.C

See also
    Foam::Polynomial
```

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

#ifndef bPolynomial_H
#define bPolynomial_H

#include "autoPtr.H"
#include "Polynomial.H"

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

namespace Foam
{

// Forward Declarations

template<class Specie, int PolySize>
class bPolynomial;

template<class Specie, int PolySize>
bPolynomial<Specie, PolySize> operator+
(
    const bPolynomial<Specie, PolySize>&,
    const bPolynomial<Specie, PolySize>&
);

template<class Specie, int PolySize>
bPolynomial<Specie, PolySize> operator*
(
    const scalar,
    const bPolynomial<Specie, PolySize>&
);

template<class Specie, int PolySize>
bPolynomial<Specie, PolySize> operator==
(
    const bPolynomial<Specie, PolySize>&,
    const bPolynomial<Specie, PolySize>&
);

template<class Specie, int PolySize>
Ostream& operator<<
(
    Ostream&,
    const bPolynomial<Specie, PolySize>&
);


/*---------------------------------------------------------------
    -------------*\
                        Class bPolynomial Declaration
\*---------------------------------------------------------------
    -------------*/

template<class Specie, int PolySize=8>
```

```
class bPolynomial
:
    public Specie
{
    // Private Data

        //- Reference density
        scalar rhoRef_;

        //- Reference temperature
        scalar Tref_;

        //- Density polynomial coefficients
        Polynomial<PolySize> rhoCoeffs_;

    // Private Member Functions

        //- Coeffs name. Eg, "rhoCoeffs<10>"
        inline static word coeffsName(const char* name)
        {
            return word(name + ("Coeffs<" + std::to_string(
                PolySize) + '>'));
        }


public:

    // Generated Methods: copy construct, copy assignment

    // Constructors

        //- Construct from components
        inline bPolynomial
        (
            const Specie& sp,
            const scalar rhoRef,
            const scalar Tref,
            const Polynomial<PolySize>& rhoPoly
        );

        //- Construct from dictionary
        explicit bPolynomial(const dictionary& dict);

        //- Construct as named copy
        inline bPolynomial(const word& name, const bPolynomial&)
            ;

        //- Construct and return a clone
        inline autoPtr<bPolynomial> clone() const;

        // Selector from dictionary
        inline static autoPtr<bPolynomial> New(const dictionary&
            dict);


    // Member Functions
```

```
        //- Return the instantiated type name
        static word typeName()
        {
            return "bPolynomial<" + word(Specie::typeName_()) +
                '>';
        }


        // Fundamental properties

            //- Is the equation of state is incompressible i.e.
                rho != f(p)
            static const bool incompressible = true;

            //- Is the equation of state is isochoric i.e. rho =
                 const
            static const bool isochoric = false;

            //- Return density [kg/m^3]
            inline scalar rho(scalar p, scalar T) const;

            //- Return enthalpy departure [J/kg]
            inline scalar H(const scalar p, const scalar T)
                const;

            //- Return Cp departure [J/(kg K]
            inline scalar Cp(scalar p, scalar T) const;

            //- Return internal energy departure [J/kg]
            inline scalar E(const scalar p, const scalar T)
                const;

            //- Return Cv departure [J/(kg K]
            inline scalar Cv(scalar p, scalar T) const;

            //- Return entropy [J/(kg K)]
            inline scalar S(const scalar p, const scalar T)
                const;

            //- Return compressibility rho/p [s^2/m^2]
            inline scalar psi(scalar p, scalar T) const;

            //- Return compression factor []
            inline scalar Z(scalar p, scalar T) const;

            //- Return (Cp - Cv) [J/(kg K]
            inline scalar CpMCv(scalar p, scalar T) const;


        // IO

            //- Write to Ostream
            void write(Ostream& os) const;


    // Member Operators
```

```
        inline void operator+=(const bPolynomial&);
        inline void operator*=(const scalar);


    // Friend Operators

        friend bPolynomial operator+ <Specie, PolySize>
        (
            const bPolynomial&,
            const bPolynomial&
        );

        friend bPolynomial operator* <Specie, PolySize>
        (
            const scalar s,
            const bPolynomial&
        );

        friend bPolynomial operator== <Specie, PolySize>
        (
            const bPolynomial&,
            const bPolynomial&
        );


    // IOstream Operators

        friend Ostream& operator<< <Specie, PolySize>
        (
            Ostream&,
            const bPolynomial&
        );
};


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

} // End namespace Foam

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

#define makebPolynomial(PolySize)
                                                    \


defineTemplateTypeNameAndDebugWithName
                                                \
(

    \
    bPolynomial<Specie, PolySize>,
                                                        \
    "bPolynomial<"#PolySize">",
                                                            \
```

```
    0

        \
);

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

#include "bPolynomialI.H"

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

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

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

#endif

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

```
/*---------------------------------------------------------------
   -------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD
     Toolbox
   \\    /   O peration     |
    \\  /    A nd           | www.openfoam.com
     \\/     M anipulation  |
-----------------------------------------------------------------
   ---------------
    Copyright (C) 2011-2017 OpenFOAM Foundation
-----------------------------------------------------------------
   ---------------
License
    This file is part of OpenFOAM.

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

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

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

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

#include "bPolynomial.H"

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

template<class Specie, int PolySize>
inline Foam::bPolynomial<Specie, PolySize>::bPolynomial
(
    const Specie& sp,
    const scalar rhoRef,
    const scalar Tref,
    const Polynomial<PolySize>& rhoCoeffs
)
:
    Specie(sp),
    rhoRef_(rhoRef),
    Tref_(Tref),
    rhoCoeffs_(rhoCoeffs)
{}


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

template<class Specie, int PolySize>
inline Foam::bPolynomial<Specie, PolySize>::bPolynomial
(
    const word& name,
    const bPolynomial<Specie, PolySize>& ip
)
:
    Specie(name, ip),
    rhoRef_(ip.rhoRef_),
    Tref_(ip.Tref_),
    rhoCoeffs_(ip.rhoCoeffs_)
{}


template<class Specie, int PolySize>
inline Foam::autoPtr<Foam::bPolynomial<Specie, PolySize>>
Foam::bPolynomial<Specie, PolySize>::clone() const
{
    return autoPtr<bPolynomial<Specie, PolySize>>::New(*this);
}


template<class Specie, int PolySize>
inline Foam::autoPtr<Foam::bPolynomial<Specie, PolySize>>
Foam::bPolynomial<Specie, PolySize>::New(const dictionary& dict)
```

```
{
    return autoPtr<bPolynomial<Specie, PolySize>>::New(dict);
}


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

template<class Specie, int PolySize>
inline Foam::scalar Foam::bPolynomial<Specie, PolySize>::rho
(
    scalar p,
    scalar T
) const
{
    return rhoCoeffs_.value(T-Tref_);
}


template<class Specie, int PolySize>
inline Foam::scalar Foam::bPolynomial<Specie, PolySize>::H
(
    scalar p,
    scalar T
) const
{
    return 0;
}


template<class Specie, int PolySize>
inline Foam::scalar Foam::bPolynomial<Specie, PolySize>::Cp
(
    scalar p,
    scalar T
) const
{
    return 0;
}


template<class Specie, int PolySize>
inline Foam::scalar Foam::bPolynomial<Specie, PolySize>::E
(
    scalar p,
    scalar T
) const
{
    return 0;
}


template<class Specie, int PolySize>
inline Foam::scalar Foam::bPolynomial<Specie, PolySize>::Cv
(
    scalar p,
    scalar T
```

```
) const
{
    return 0;
}


template<class Specie, int PolySize>
inline Foam::scalar Foam::bPolynomial<Specie, PolySize>::S
(
    scalar p,
    scalar T
) const
{
    return 0;
}


template<class Specie, int PolySize>
inline Foam::scalar Foam::bPolynomial<Specie, PolySize>::psi
(
    scalar p,
    scalar T
) const
{
    return 0;
}


template<class Specie, int PolySize>
inline Foam::scalar Foam::bPolynomial<Specie, PolySize>::Z
(
    scalar p,
    scalar T
) const
{
    return 0;
}


template<class Specie, int PolySize>
inline Foam::scalar Foam::bPolynomial<Specie, PolySize>::CpMCv
(
    scalar p,
    scalar T
) const
{
    return 0;
}


// * * * * * * * * * * * * * * * * * * Member Operators  * * * * * * *
    * * * * * * * //

template<class Specie, int PolySize>
inline void Foam::bPolynomial<Specie, PolySize>::operator+=
(
    const bPolynomial<Specie, PolySize>& ip
```

```
)
{
    scalar Y1 = this->Y();
    Specie::operator+=(ip);

    if (mag(this->Y()) > SMALL)
    {
        Y1 /= this->Y();
        const scalar Y2 = ip.Y()/this->Y();
        rhoRef_ = Y1*rhoRef_ + Y2*ip.rhoRef_;
        Tref_ = Y1*Tref_ + Y2*ip.Tref_;
        rhoCoeffs_ = Y1*rhoCoeffs_ + Y2*ip.rhoCoeffs_;
    }
}


template<class Specie, int PolySize>
inline void Foam::bPolynomial<Specie, PolySize>::operator*=(
    const scalar s)
{
    Specie::operator*=(s);
}


// * * * * * * * * * * * * * * * * Friend Operators  * * * * * * *
    * * * * * * //

template<class Specie, int PolySize>
Foam::bPolynomial<Specie, PolySize> Foam::operator+
(
    const bPolynomial<Specie, PolySize>& ip1,
    const bPolynomial<Specie, PolySize>& ip2
)
{
    Specie sp
    (
        static_cast<const Specie&>(ip1)
      + static_cast<const Specie&>(ip2)
    );

    if (mag(sp.Y()) < SMALL)
    {
        return bPolynomial<Specie, PolySize>
        (
            sp,
            ip1.rhoRef_,
            ip1.Tref_,
            ip1.rhoCoeffs_
        );
    }
    else
    {
        const scalar Y1 = ip1.Y()/sp.Y();
        const scalar Y2 = ip2.Y()/sp.Y();

        return bPolynomial<Specie, PolySize>
        (
```

```
            sp,
            Y1*ip1.rhoRef_ + Y2*ip2.rhoRef_,
            Y1*ip1.Tref_ + Y2*ip2.Tref_,
            Y1*ip1.rhoCoeffs_ + Y2*ip2.rhoCoeffs_
        );
    }
}


template<class Specie, int PolySize>
Foam::bPolynomial<Specie, PolySize> Foam::operator*
(
    const scalar s,
    const bPolynomial<Specie, PolySize>& ip
)
{
    return bPolynomial<Specie, PolySize>
    (
        s*static_cast<const Specie&>(ip),
        ip.rhoRef_,
        ip.Tref_,
        ip.rhoCoeffs_
    );
}


template<class Specie, int PolySize>
Foam::bPolynomial<Specie, PolySize> Foam::operator==
(
    const bPolynomial<Specie, PolySize>& ip1,
    const bPolynomial<Specie, PolySize>& ip2
)
{
    Specie sp
    (
        static_cast<const Specie&>(ip1)
     == static_cast<const Specie&>(ip2)
    );

    const scalar Y1 = ip1.Y()/sp.Y();
    const scalar Y2 = ip2.Y()/sp.Y();

    return bPolynomial<Specie, PolySize>
    (
        sp,
        Y2*ip2.rhoRef_ - Y1*ip1.rhoRef_,
        Y2*ip2.Tref_   - Y1*ip1.Tref_,
        Y2*ip2.rhoCoeffs_ - Y1*ip1.rhoCoeffs_
    );
}


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

A.4 Python code for Stefan Problem

```
import os
import re
from tkinter import Tk
from tkinter.filedialog import askdirectory
import csv
import matplotlib.pyplot as plt
import numpy as np
from glob import iglob
from cmath import pi, sqrt, exp

path = askdirectory(title='Select Folder')
for root, dirs, files in os.walk(path):
    numL2 = []
    denL2 = []
    L2 = []
    numL2_sol2 = []
    denL2_sol2 = []
    L2_sol2 = []
    dir = []
    n = 0
    for i in files:
        with os.scandir(root) as it:
            alphaField = []
            posField = []
            t = []

            j = 0
            for entry in it:
                if entry.name.endswith(".csv") and entry.is_file():

                    t = np.append(t,int(re.search(r'\d+', entry.
    name).group()))
                    with open(entry.path, 'r') as f:
                        fields = []
                        alpha = []
                        x = []
                        csvreader = csv.reader(f)
                        fields = next(csvreader)
                        for row in csvreader:
                            alpha = np.append(alpha, float(row[0]))
                            x = np.append(x,float(row[1]))
                        idx = (np.abs(alpha - 0.5)).argmin()
                        alp = alpha[idx]
                        loc = x[idx]
                        # if alp == 1:
                        #     loc = 0.0
                        #     alp = 1

                posField = np.append(posField, loc)
                alphaField = np.append(alphaField, alp)

                j = j + 1
            m = t.argsort()
            alphaField = alphaField[m]
            posField = posField[m]
```

```
            # Analytical Interface position
            cps = 2050
            k1 = 2.22
            rho1 = 916.2
            a1 = k1/(rho1*cps)
            lambd = 0.00032622525325939834
            lambd1 = 0.2299545377262345
            psi = []
            psi2 = []
            t = np.sort(t)
            for x in range(0,len(t)):
                psi = np.append(psi, lambd*sqrt(t[x]))
                psi2 = np.append(psi2, 2*lambd1*sqrt(a1*t[x]))

        numL2 = []
        denL2 = []
        L2 = []
        for k in range(0,len(t)):
            numL2 = np.append(numL2, np.sum(np.power((posField[k]-
    psi2[k]),2)))
            L2 = numL2

    f1 = plt.figure()
    f2 = plt.figure()

    ax1 = f1.add_subplot(111)
    ax1.plot(t, psi2.real, 'r--', label='Neumann solution')
    ax1.plot(t, posField, 'g--', label='Numerical solution')
    ax1.set(xlabel='t [s]', ylabel= 'Interface position [m]')
    ax1.grid(True)
    ax1.legend()
    L2 = L2.real
    ax2 = f2.add_subplot(111)
    ax2.plot(t, L2, 'r--', label='Relative error')
    ax2.set(xlabel='t [s]', ylabel= 'Relative error')
    ax2.grid(True)

    ax2.legend()
    plt.show()
```

```
from scipy import optimize
from scipy.special import erfc
from scipy.special import erf
from cmath import pi, sqrt, exp

# Roots of "An Accurate Approximation of the Two-Phase Stefan
    Problem with Coefficient Smoothing"
g = -20
u0 = 10
k1 = 2.26
k2 = 0.59
c1 = 4.182E6
c2 = 4.182E6
D = 3.35E8
a1 = sqrt(k1/c1)
a2 = sqrt(k2/c2)

```

```
def func(x):
    return ((((k1/a1) * g * exp(-(x/(2*a1))**2)) / erf(x/(2*a1))) +
     ((k2/a2) * u0 * (exp(-(x/(2*a2))**2)) / (1.0-erf(x/(2*a2)))) +
    ((x * D * sqrt(pi)) / 2))

sol = optimize.root_scalar(func, rtol=1E-12, method='secant', x0
    =-0.1, x1=0.0005)
lambd = sol.root.real
print("An Accurate Approximation of the Two-Phase Stefan Problem
    with Coefficient Smoothing: ", lambd)


tm = 0.15
t0 = 10
tb = -20
L = 335000
cps = 4182
k1 = 2.26
k2 = 0.59
rho1 = 916.8
rho2 = 999.8
a1 = k1/(rho1*cps)
a2 = k2/(rho2*cps)

def f(x):
    return ((exp(-(x**2))/erf(x)) + (k2/k1) * sqrt(a1/a2) * ((tm-t0
    )/(tm-tb)) * (exp(-(a1/a2)*(x**2))/erfc(x*sqrt(a1/a2))) - (x*L*
    sqrt(pi))/(cps*(tm-tb)))

sol1 = optimize.root_scalar(f, rtol=1E-12, method='secant', x0=0.1,
     x1=0.5)
lambd1 = sol1.root.real

import os
import re
from tkinter import Tk
from tkinter.filedialog import askdirectory
import csv
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
path = askdirectory(title='Select Folder')

for root, dirs, files in os.walk(path):
    for i in files:
        if i == 'mesh1_TX_950s.xlsx':
            dfs1 = pd.read_excel(root+'/'+i)
            T = []
            x = []
            idx1 = np.where(dfs1.columns == "T")[0][0]
            T = dfs1.values[:,idx1]
            T = T - 273.15
            idx2 = np.where(dfs1.columns == "X")[0][0]
            x = dfs1.values[:,idx2]

            t = np.linspace(0,50,len(x))
            g = -20
```

```
            u0 = 10
            k1 = 2.26
            k2 = 0.59
            c1 = 4.182E6
            c2 = 4.182E6
            D = 3.33E8
            a1 = (sqrt(k1/c1))
            a2 = (sqrt(k2/c2))
            fxt = []
            den = []
            psi = []
            fyt = []

            j = 0
            for i in x:
                psi = np.append(psi, lambd*sqrt(t[j]))
                if i <= psi[j]:
                    den = np.append(den, 2*a1*sqrt(t[j]))
                    if (t[j]==0.0):
                        fxt = g
                    else:
                        fxt = np.append(fxt, (g * (erf(psi[j]/den[j
])-erf(x[j]/den[j])))/(erf(psi[j]/den[j])))

                else:
                    den = np.append(den, 2*a2*sqrt(t[j]))
                    if (t[j]==0.0):
                        fxt = 0.0
                    else:
                        fxt = np.append(fxt, (u0 * (erf(x[j]/den[j
])-erf(psi[j]/den[j])))/(1-erf(psi[j]/den[j])))

                j = j + 1


            tm = 0.15
            t0 = 10
            tb = -20
            L = 335000
            cps = 4182
            k1 = 2.26
            k2 = 0.59
            rho1 = 916.8
            rho2 = 999.8
            a1 = k1/(rho1*cps)
            a2 = k2/(rho2*cps)

            j = 0
            for i in x:
                psi = np.append(psi, 2*lambd1*sqrt(a1*x[j]))
                if i <= psi[j]:
                    if (t[j]==0.0):
                        fyt = tb
                    else:
                        fyt = np.append(fyt, (erf(x[j]/(2*sqrt(a1*t
[j])))/erf(lambd1)) * (tm-tb) + tb)
                else:
```

```
                    if (t[j]==0.0):
                        fyt = 0.0
                    else:
                        fyt = np.append(fyt, (erfc(x[j]/(2*sqrt(a2*t[j])))/erfc(lambd1*sqrt(a1/a2))) * (tm-t0) + t0)

                j = j + 1

        else:
            continue

        num_solution = T
        ana_solution1 = fxt
        ana_solution2 = fyt
        numL2 = []
        denL2 = []
        L2 = []
        for k in range(0,len(T)):
            numL2 = np.append(numL2, abs(num_solution[k]-ana_solution2[k])/ana_solution2[k])
            L2 = numL2

dydx = np.gradient(fyt.real, x)
dydxT = np.gradient(T,x)

f1 = plt.figure()
f2 = plt.figure()
f3 = plt.figure()

ax1 = f1.add_subplot(111)
ax1.plot(x, fyt.real, 'r--', label='Neumann solution')
ax1.plot(x, T, 'g--', label='Numerical solution')
ax1.set(xlabel='x [m]', ylabel= 'T [C]')
ax1.grid(True)
ax1.legend()
L2 = L2.real/len(T)
ax2 = f2.add_subplot(111)
ax2.plot(x[1:], L2[1:], 'r--', label='Relative error')
ax2.set(xlabel='x [m]', ylabel= 'Relative error')
ax2.grid(True)
ax2.legend()

ax3 = f3.add_subplot(111)
ax3.plot(x,dydx,'r--', label='Neumann solution')
ax3.plot(x,dydxT,'g--', label='Numerical solution')
ax3.set(xlabel='x [m]', ylabel= 'dy/dx')
ax3.grid(True)
ax3.legend()

plt.show()
```

Appendix B

Appendix B: Solver implementations

B.1 Computational Mesh script

```
from ofblockmeshdicthelper import BlockMeshDict, Vertex, Point,
    SimpleGrading
import numpy as np
import os

from tkinter import Tk
from tkinter.filedialog import askdirectory
path = askdirectory(title='Select Folder')

bmd = BlockMeshDict()
r = 41.4
Hp = 1
Nx_pipe = 300
Ny_pipe = 300
Nz_pipe = 1
r2 = r + 10
rati = r/r2


bmd.add_vertex(-r/2,-r/2,-Hp,'v0')
bmd.add_vertex(0,-r/2,-Hp,'v1')
bmd.add_vertex(0,r/2,-Hp,'v2')
bmd.add_vertex(-r/2,r/2,-Hp,'v3')
bmd.add_vertex(-r/2,-r/2,0,'v4')
bmd.add_vertex(0,-r/2,0,'v5')
bmd.add_vertex(0,r/2,0,'v6')
bmd.add_vertex(-r/2,r/2,0,'v7')
bmd.add_vertex(-r*np.sin(np.pi/4),-r*np.cos(np.pi/4),-Hp,'v8')
bmd.add_vertex(0,-r,-Hp,'v9')
bmd.add_vertex(-r*np.sin(np.pi/4),r*np.cos(np.pi/4),-Hp,'v10')
bmd.add_vertex(0,r,-Hp,'v11')
bmd.add_vertex(-r*np.sin(np.pi/4),-r*np.cos(np.pi/4),0,'v12')
bmd.add_vertex(0,-r,0,'v13')
bmd.add_vertex(-r*np.sin(np.pi/4),r*np.cos(np.pi/4),0,'v14')
bmd.add_vertex(0,r,0,'v15')


```

```
bmd.add_vertex(-r2*np.sin(np.pi/4),-r2*np.cos(np.pi/4),-Hp,'v16')
bmd.add_vertex(0,-r2,-Hp,'v17')
bmd.add_vertex(-r2*np.sin(np.pi/4),r2*np.cos(np.pi/4),-Hp,'v18')
bmd.add_vertex(0,r2,-Hp,'v19')
bmd.add_vertex(-r2*np.sin(np.pi/4),-r2*np.cos(np.pi/4),0,'v20')
bmd.add_vertex(0,-r2,0,'v21')
bmd.add_vertex(-r2*np.sin(np.pi/4),r2*np.cos(np.pi/4),0,'v22')
bmd.add_vertex(0,r2,0,'v23')
bmd.add_arcedge(('v8','v9'),'arc1',Vertex(-r*np.sin(np.pi/8),-r*np.
    cos(np.pi/8),-Hp,'v_arc1'))
bmd.add_arcedge(('v12','v13'),'arc2',Vertex(-r*np.sin(np.pi/8),-r*
    np.cos(np.pi/8),0,'v_arc2'))
bmd.add_arcedge(('v8','v10'),'arc3',Vertex(-r,0,-Hp,'v_arc3'))
bmd.add_arcedge(('v12','v14'),'arc4',Vertex(-r,0,0,'v_arc4'))
bmd.add_arcedge(('v10','v11'),'arc5',Vertex(-r*np.sin(np.pi/8),r*np
    .cos(np.pi/8),-Hp,'v_arc5'))
bmd.add_arcedge(('v14','v15'),'arc6',Vertex(-r*np.sin(np.pi/8),r*np
    .cos(np.pi/8),0,'v_arc6'))
bmd.add_arcedge(('v16','v17'),'arc7',Vertex(-r2*np.sin(np.pi/8),-r2
    *np.cos(np.pi/8),-Hp,'v_arc7'))
bmd.add_arcedge(('v20','v21'),'arc8',Vertex(-r2*np.sin(np.pi/8),-r2
    *np.cos(np.pi/8),0,'v_arc8'))
bmd.add_arcedge(('v16','v18'),'arc9',Vertex(-r2,0,-Hp,'v_arc9'))
bmd.add_arcedge(('v20','v22'),'arc10',Vertex(-r2,0,0,'v_arc10'))
bmd.add_arcedge(('v18','v19'),'arc11',Vertex(-r2*np.sin(np.pi/8),r2
    *np.cos(np.pi/8),-Hp,'v_arc11'))
bmd.add_arcedge(('v22','v23'),'arc12',Vertex(-r2*np.sin(np.pi/8),r2
    *np.cos(np.pi/8),0,'v_arc12'))


prism_pipe = bmd.add_hexblock(('v0','v1','v2','v3','v4','v5','v6','
    v7'), (int(Nx_pipe/2),int(Ny_pipe/2),Nz_pipe),'prims_pipe',
                  grading=SimpleGrading(1,1,1))
south_pipe = bmd.add_hexblock(('v8','v9','v1','v0','v12','v13','v5'
    ,'v4'),(int(Nx_pipe/2),int(Ny_pipe/4),Nz_pipe),'south_pipe',
                  grading=SimpleGrading(1,1,1))
east_pipe = bmd.add_hexblock(('v8','v0','v3','v10','v12','v4','v7',
    'v14'),(int(Ny_pipe/4),int(Ny_pipe/2),Nz_pipe),'east_pipe',
                  grading=SimpleGrading(1,1,1))
north_pipe = bmd.add_hexblock(('v3','v2','v11','v10','v7','v6','v15
    ','v14'),(int(Nx_pipe/2),int(Ny_pipe/4),Nz_pipe),'north_pipe',
                  grading=SimpleGrading(1,1,1))

#Region #2
south_pipe_region2 = bmd.add_hexblock(('v16','v17','v9','v8','v20',
    'v21','v13','v12'),(int(Nx_pipe/2),int(Ny_pipe/4),Nz_pipe),'
    south_pipe_region2',
                  grading=SimpleGrading(1,1,1))
east_pipe_region2 = bmd.add_hexblock(('v16','v8','v10','v18','v20',
    'v12','v14','v22'),(int(Ny_pipe/4),int(Ny_pipe/2),Nz_pipe),'
    east_pipe_region2',
                  grading=SimpleGrading(1,1,1))
north_pipe_region2 = bmd.add_hexblock(('v10','v11','v19','v18','v14
    ','v15','v23','v22'),(int(Nx_pipe/2),int(Ny_pipe/4),Nz_pipe),'
    north_pipe_region2',
                  grading=SimpleGrading(1,1,1))

```

```
77 bmd.add_boundary('wall','solidWall',[south_pipe_region2.face('s'),
       east_pipe_region2.face('w'),north_pipe_region2.face('n')])
78 bmd.add_boundary('empty','fluidFrontAndBack',[prism_pipe.face('b'),
       south_pipe.face('b'),east_pipe.face('b'),north_pipe.face('b'),
       prism_pipe.face('t'),south_pipe.face('t'),east_pipe.face('t'),
       north_pipe.face('t')])
79 bmd.add_boundary('empty','solidFrontAndBack',[south_pipe_region2.
       face('b'),east_pipe_region2.face('b'),north_pipe_region2.face('b
       '),south_pipe_region2.face('t'),east_pipe_region2.face('t'),
       north_pipe_region2.face('t')])
80 bmd.add_boundary('symmetryPlane','fluidSymmetryBC',[prism_pipe.face
       ('e'),south_pipe.face('e'),north_pipe.face('e')])
81 bmd.add_boundary('symmetryPlane','solidSymmetryBC',[
       south_pipe_region2.face('e'),north_pipe_region2.face('e')])
82
83
84 bmd.set_metric('mm')
85 bmd.assign_vertexid()
86 print(bmd.format())
87
88 print(bmd.format())
89
90 filetxt = bmd.format()
91 nameOfFile = 'blockMeshDict'
92 completeName = os.path.join(path, nameOfFile)
93 with open(completeName,'w') as f:
94     f.writelines(filetxt)
95
96 f.close()
```

B.2 chtMultiphaseInterFoam solver

```
/*---------------------------------------------------------------------
   ------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD
     Toolbox
   \\    /   O peration     |
    \\  /    A nd           | www.openfoam.com
     \\/     M anipulation  |
-----------------------------------------------------------------------
   --------------
    Copyright (C) 2011-2016 OpenFOAM Foundation
    Copyright (C) 2017 OpenCFD Ltd.
-----------------------------------------------------------------------
   --------------
License
    This file is part of OpenFOAM.

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

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

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

Application
    chtMultiPhaseInterFoam

Group
    grpHeatTransferSolvers


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

#include "fvCFD.H"
#include "subCycle.H"
#include "multiphaseSystem.H"
#include "turbulentFluidThermoModel.H"
#include "pimpleControl.H"
#include "CombustionModel.H"
#include "fixedGradientFvPatchFields.H"
#include "regionProperties.H"
#include "alphaCourantNo.H"
#include "ddtAlphaNo.H"
#include "compressibleCourantNo.H"
#include "solidRegionDiffNo.H"
#include "solidThermo.H"
#include "radiationModel.H"
#include "fvOptions.H"
#include "coordinateSystem.H"
#include "loopControl.H"
#include "pressureControl.H"
#include "CorrectPhi.H"

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

int main(int argc, char *argv[])
{
    argList::addNote
    (
        "Transient␣solver␣for␣buoyant,␣turbulent␣fluid␣flow␣and␣
            solid␣heat"
        "␣conduction␣with␣conjugate␣heat␣transfer"
        "␣between␣solid␣and␣fluid␣regions."
    );
```

```
    #define NO_CONTROL
    #define CREATE_MESH createMeshesPostProcess.H
    #include "postProcess.H"
    #include "setRootCaseLists.H"
    #include "createTime.H"
    #include "createMeshes.H"
    #include "createFields.H"
    #include "createFieldRefs.H"
    #include "initContinuityErrs.H"
    #include "createTimeControls.H"
    #include "readFluidTimeControls.H"
    #include "readSolidTimeControls.H"
    #include "ddtAlphaMultiRegionNo.H"
    #include "compressibleMultiRegionCourantNo.H"
    #include "solidRegionDiffusionNo.H"
    #include "setInitialMultiRegionDeltaT.H"
    #include "validateTurbulenceModel.H"

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

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

        #include "alphaCourantMultiRegionNo.H"
        #include "ddtAlphaMultiRegionNo.H"
        #include "compressibleMultiRegionCourantNo.H"
        #include "solidRegionDiffusionNo.H"
        #include "setMultiRegionDeltaT.H"


        ++runTime;

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

        // --- PIMPLE loop
        for (int oCorr=0; oCorr<nOuterCorr; ++oCorr)
        {
            const bool firstIter = (oCorr == 0);
            const bool finalIter = (oCorr == nOuterCorr-1);

            forAll(fluidRegions, i)
            {
                Info<< "\nSolving for fluid region "
                    << fluidRegions[i].name() << endl;
                #include "setRegionFluidFields.H"
                #include "readFluidMultiRegionPIMPLEControls.H"
                #include "solveFluid.H"
            }

            forAll(solidRegions, i)
            {
                Info<< "\nSolving for solid region "
                    << solidRegions[i].name() << endl;
```

```
                #include "setRegionSolidFields.H"
                #include "readSolidMultiRegionPIMPLEControls.H"
                #include "solveSolid.H"
            }

                    // Additional loops for energy solution
                       only
            if (!oCorr && nOuterCorr > 1)
            {
                loopControl looping(runTime, pimpleCHT, "
                   energyCoupling");

                while (looping.loop())
                {
                    Info<< nl << looping << nl;

                    forAll(fluidRegions, i)
                    {
                        Info<< "\nSolving for fluid region "
                            << fluidRegions[i].name() << endl;
                       #include "setRegionFluidFields.H"
                       #include "
                          readFluidMultiRegionPIMPLEControls.H"
                       frozenFlow = true;
                       #include "solveFluid.H"
                    }

                    forAll(solidRegions, i)
                    {
                        Info<< "\nSolving for solid region "
                            << solidRegions[i].name() << endl;
                        #include "setRegionSolidFields.H"
                        #include "
                           readSolidMultiRegionPIMPLEControls.H
                           "
                        #include "solveSolid.H"
                    }
                }
            }
        }

        runTime.write();

        runTime.printExecutionTime(Info);
    }

    Info<< "End\n" << endl;

    return 0;
}


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

createFields

```
#include "createFluidFields.H"
#include "createSolidFields.H"
```

createMeshes

```
regionProperties rp(runTime);

#include "createFluidMeshes.H"
#include "createSolidMeshes.H"
```

createMeshesPostProcess

```
#include "createMeshes.H"

if (!fluidRegions.size() && !solidRegions.size())
{
    FatalErrorIn(args.executable())
        << "No region meshes present" << exit(FatalError);
}

fvMesh& mesh = fluidRegions.size() ? fluidRegions[0] :
   solidRegions[0];
```

readPIMPLEControls

```
    // We do not have a top-level mesh. Construct the fvSolution
        for
    // the runTime instead.
    fvSolution solutionDict(runTime);

    const dictionary& pimpleCHT = solutionDict.subDict("PIMPLE")
        ;

    const int nOuterCorr =
        pimpleCHT.getOrDefault<int>("nOuterCorrectors", 1);
```

B.2.1 Fluid region

createFluidMeshes

```
    const wordList fluidNames(rp["fluid"]);

    PtrList<fvMesh> fluidRegions(fluidNames.size());
    // PtrList<dynamicFvMesh> fluidRegions(fluidNames.size());
    forAll(fluidNames, i)
    {
        Info<< "Create fluid mesh for region " << fluidNames[i]
            << " for time = " << runTime.timeName() << nl <<
                endl;

        fluidRegions.set
        (
            i,
            new fvMesh
            // dynamicFvMesh::New
            (
                IOobject
                (
```

```
                        fluidNames[i],
                        runTime.timeName(),
                        runTime,
                        IOobject::MUST_READ
                    )
                )
            );
    }
```

alphaCourantNo

```
/*---------------------------------------------------------------------------*\
    ------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | www.openfoam.com
     \\/     M anipulation  |
-------------------------------------------------------------------------------
    ---------------
    Copyright (C) 2011-2014 OpenFOAM Foundation
    Copyright (C) 2020 OpenCFD Ltd.
-------------------------------------------------------------------------------
    ---------------
License
    This file is part of OpenFOAM.

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

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

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

Global
    alphaCourantNo

Description
    Calculates and outputs the mean and maximum alpha Courant Numbers.
```

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

#ifndef alphaCourantNo_H
#define alphaCourantNo_H

#include "fvMesh.H"
#include "multiphaseSystem.H"

namespace Foam
{
    scalar alphaCourantNo
    (
        const fvMesh& mesh,
        const Time& runTime,
        const multiphaseSystem& thermol,
        const surfaceScalarField& phi
    );
}

#endif

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

ddtAlphaNo.H

```
/*---------------------------------------------------------------------------
    ------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD
     Toolbox
   \\    /   O peration     |
    \\  /    A nd           | www.openfoam.com
     \\/     M anipulation  |
-------------------------------------------------------------------------------
    --------------
    Copyright (C) 2011-2014 OpenFOAM Foundation
    Copyright (C) 2020 OpenCFD Ltd.
-------------------------------------------------------------------------------
    --------------
License
    This file is part of OpenFOAM.

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

    OpenFOAM is distributed in the hope that it will be useful,
       but WITHOUT
    ANY WARRANTY; without even the implied warranty of
       MERCHANTABILITY or
```

```
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General
       Public License
    for more details.

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

Global
    alphaCourantNo

Description
    Calculates and outputs the mean and maximum alpha Courant
       Numbers.

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

#ifndef ddtAlphaNo_H
#define ddtAlphaNo_H

#include "fvMesh.H"
#include "multiphaseSystem.H"

namespace Foam
{
    scalar ddtAlphaNo
    (
        const fvMesh& mesh,
        const Time& runTime,
        const multiphaseSystem& thermol,
        const surfaceScalarField& phi
    );
}

#endif
```

ddtAlphaNo.C

```
/*---------------------------------------------------------------
   -------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD
     Toolbox
   \\    /   O peration     |
    \\  /    A nd           | www.openfoam.com
     \\/     M anipulation  |
-----------------------------------------------------------------
   ---------------
    Copyright (C) 2011-2016 OpenFOAM Foundation
-----------------------------------------------------------------
   ---------------
License
    This file is part of OpenFOAM.
```

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

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

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

\*---------------------------------------------------------------
    -------------*/
#include "ddtAlphaNo.H"
#include "fvc.H"

Foam::scalar Foam::ddtAlphaNo
(
    const fvMesh& mesh,
    const Time& runTime,
    const multiphaseSystem& thermol,
    const surfaceScalarField& phi
)
{
    scalar maxAlphaDdt
    (
        runTime.controlDict().getOrDefault("maxAlphaDdt", GREAT)
    );
    scalar ddtAlphaNum = 0.0;
    if (mesh.nInternalFaces())
    {
        ddtAlphaNum = thermol.ddtAlphaMax().value()*runTime.
            deltaTValue();
    }
    return ddtAlphaNum;
}
```

compressibleCourantNo.H

```
/*---------------------------------------------------------------
    -------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD
      Toolbox
   \\    /   O peration     |
    \\  /    A nd           | www.openfoam.com
     \\/     M anipulation  |
```

```
-------------------------------------------------------------------
    ---------------
     Copyright (C) 2011 OpenFOAM Foundation
-------------------------------------------------------------------
    ---------------
License
     This file is part of OpenFOAM.

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

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

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

Description
     Calculates and outputs the mean and maximum Courant Numbers
        for the fluid
     regions

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

#ifndef compressibleCourantNo_H
#define compressibleCourantNo_H

#include "fvMesh.H"

namespace Foam
{
    scalar compressibleCourantNo
    (
        const fvMesh& mesh,
        const Time& runTime,
        const volScalarField& rho,
        const surfaceScalarField& phi
    );
}

#endif

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

compressibleCourantNo.C

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | www.openfoam.com
     \\/     M anipulation  |
-------------------------------------------------------------------------------
    Copyright (C) 2011-2016 OpenFOAM Foundation
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

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

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

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

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

#include "compressibleCourantNo.H"
#include "fvc.H"

Foam::scalar Foam::compressibleCourantNo
(
    const fvMesh& mesh,
    const Time& runTime,
    const volScalarField& rho,
    const surfaceScalarField& phi
)
{

    scalar CoNum = 0.0;
    scalar meanCoNum = 0.0;
```

```
    {
        scalarField sumPhi
        (
            fvc::surfaceSum(mag(phi))().primitiveField()
        );

        CoNum = 0.5*gMax(sumPhi/mesh.V().field())*runTime.
           deltaTValue();

        meanCoNum =
            0.5*(gSum(sumPhi)/gSum(mesh.V().field()))*runTime.
               deltaTValue();
    }

    Info<< "Region: " << mesh.name() << " Courant Number mean: "
        << meanCoNum
        << " max: " << CoNum << endl;

    return CoNum;
}


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

correctPhi

```
CorrectPhi
(
    U,
    phi,
    p_rgh,
    surfaceScalarField("rAUf", fvc::interpolate(rAU())),
    geometricZeroField(),
    pimple
);

// #include "continuityErrs.H"
```

createFluidFields

```
// Initialise fluid field pointer lists

PtrList<multiphaseSystem> thermoFluid(fluidRegions.size());
PtrList<volScalarField> rhoFluid(fluidRegions.size());
PtrList<volScalarField> TFluid(fluidRegions.size());
PtrList<volVectorField> UFluid(fluidRegions.size());
PtrList<uniformDimensionedScalarField> hRefFluid(fluidRegions.
   size());
PtrList<volScalarField> ghFluid(fluidRegions.size());
PtrList<surfaceScalarField> ghfFluid(fluidRegions.size());
PtrList<volScalarField> kappaLK(fluidRegions.size());
PtrList<CompressibleTurbulenceModel<multiphaseSystem>>
   turbulenceFluid(fluidRegions.size());
PtrList<volScalarField> p_rghFluid(fluidRegions.size());
PtrList<volScalarField> KFluid(fluidRegions.size());
PtrList<volScalarField> dpdtFluid(fluidRegions.size());
```

```
PtrList<multivariateSurfaceInterpolationScheme<scalar>::
    fieldTable>
     fieldsFluid(fluidRegions.size());
List<scalar> initialMassFluid(fluidRegions.size());
List<bool> frozenFlowFluid(fluidRegions.size(), false);
List<bool> correctPhiFluid(fluidRegions.size(), true);
List<bool> ddtCorrFluid(fluidRegions.size(), true);
PtrList<fv::options> fluidFvOptions(fluidRegions.size());
List<label> pRefCellFluid(fluidRegions.size());
List<scalar> pRefValueFluid(fluidRegions.size());
PtrList<dimensionedScalar> rhoMinFluid(fluidRegions.size());
PtrList<dimensionedScalar> rhoMaxFluid(fluidRegions.size());
PtrList<dimensionedScalar> rhoRFluid(fluidRegions.size());
PtrList<volScalarField> rhokFluid(fluidRegions.size());
PtrList<volScalarField> CpFluid(fluidRegions.size());
PtrList<volScalarField> rhoCpFluid(fluidRegions.size());
PtrList<volScalarField> pFluid(fluidRegions.size());
PtrList<surfaceScalarField> rhoPhiFluid(fluidRegions.size());
PtrList<pimpleControl> pimpleFluid(fluidRegions.size());

PtrList<pressureControl> pressureControls(fluidRegions.size());

const uniformDimensionedVectorField& g = meshObjects::gravity::
    New(runTime);

// Populate fluid field pointer lists
forAll(fluidRegions, i)
{
    Info<< "***␣Reading␣fluid␣mesh␣thermophysical␣properties␣for
        ␣region␣"
        << fluidRegions[i].name() << nl << endl;
    pimpleFluid.set
    (
        i,
        new pimpleControl(fluidRegions[i])
    );

    p_rghFluid.set
    (
        i,
        new volScalarField
        (
            IOobject
            (
                "p_rgh",
                runTime.timeName(),
                fluidRegions[i],
                IOobject::MUST_READ,
                IOobject::AUTO_WRITE
            ),
            fluidRegions[i]
        )
    );
    Info<< "␣␣␣␣Adding␣to␣UFluid\n" << endl;
    UFluid.set
    (
        i,
```

```
        new volVectorField
        (
            IOobject
            (
                "U",
                runTime.timeName(),
                fluidRegions[i],
                IOobject::MUST_READ,
                IOobject::AUTO_WRITE
            ),
            fluidRegions[i]
        )
    );
    Info<< "    Adding to TFluid\n" << endl;
    TFluid.set
    (
        i,
        new volScalarField
        (
            IOobject
            (
                "T",
                runTime.timeName(),
                fluidRegions[i],
                IOobject::MUST_READ,
                IOobject::AUTO_WRITE
            ),
            fluidRegions[i]
        )
    );


    Info<< "    Adding to hRefFluid\n" << endl;
    hRefFluid.set
    (
        i,
        new uniformDimensionedScalarField
        (
            IOobject
            (
                "hRef",
                runTime.constant(),
                fluidRegions[i],
                IOobject::READ_IF_PRESENT,
                IOobject::NO_WRITE
            ),
            dimensionedScalar("hRef", dimLength, Zero)
        )
    );

    Info<< "Calculating field g.h\n" << endl;
    #include "readGravitationalAcceleration.H"

    dimensionedScalar ghRef
    (
        mag(g.value()) > SMALL
      ? g & (cmptMag(g.value())/mag(g.value()))*hRefFluid[i]
```

```
        : dimensionedScalar("ghRef", g.dimensions()*dimLength, 0)
    );

    Info<< "    Adding to ghFluid\n" << endl;
    ghFluid.set
    (
        i,
        new volScalarField
        (
            "gh",
            (g & fluidRegions[i].C()) - ghRef
        )
    );

    Info<< "    Adding to ghfFluid\n" << endl;
    ghfFluid.set
    (
        i,
        new surfaceScalarField
        (
            "ghf",
            (g & fluidRegions[i].Cf()) - ghRef
        )
    );

    pFluid.set
    (
        i,
        new volScalarField
        (
            IOobject
            (
                "p",
                runTime.timeName(),
                fluidRegions[i],
                IOobject::NO_READ,
                IOobject::AUTO_WRITE
            ),
            p_rghFluid[i]
        )
    );
    Info<< "    Adding to thermoFluid\n" << endl;
    thermoFluid.set(i, multiphaseSystem::New(fluidRegions[i]).
       ptr());
    Info<< "    Adding to rhoFluid\n" << endl;
    rhoFluid.set
    (
        i,
        new volScalarField
        (
            IOobject
            (
                "rho",
                runTime.timeName(),
                fluidRegions[i],
                IOobject::NO_READ,
                IOobject::AUTO_WRITE
```

```
            ),
            thermoFluid[i].rho()
        )
    );
    rhoFluid[i].oldTime();

    const dictionary& thermophysicalPropertiesFluid =
        fluidRegions[i].lookupObject<IOdictionary>("
            thermophysicalProperties.liquid");


    rhoRFluid.set
    (
        i,
        new dimensionedScalar
        (
        "rhoRef",
        dimDensity,
        thermophysicalPropertiesFluid.subDict("mixture").subDict
            ("equationOfState")
        )
    );

    Info<< "    Calculating rhok\n" << endl;
    rhokFluid.set
    (
        i,
        new volScalarField
        (
            IOobject
            (
                "rhok",
                runTime.timeName(),
                fluidRegions[i],
                IOobject::NO_READ,
                IOobject::AUTO_WRITE
            ),
            thermoFluid[i].rho() - rhoRFluid[i]
        )
    );
    rhokFluid[i].oldTime();

    Info<< "    Adding to rhoPhiFluid\n" << endl;
    rhoPhiFluid.set
    (
        i,
        new surfaceScalarField
        (
            IOobject
            (
                "rhoPhi",
                runTime.timeName(),
                fluidRegions[i],
                IOobject::NO_READ,
                IOobject::NO_WRITE
            ),
            thermoFluid[i].rhoPhi()
```

```
        )
    );

    Info<< "    Adding to CpFluid\n" << endl;
    CpFluid.set
    (
        i,
        new volScalarField
        (
            IOobject
            (
                "Cp",
                runTime.timeName(),
                fluidRegions[i],
                IOobject::NO_READ,
                IOobject::AUTO_WRITE
            ),
            thermoFluid[i].Cp()
        )
    );

    Info<< "    Adding to Kappa Lookup\n" << endl;
    kappaLK.set
    (
        i,
        new volScalarField
        (
            IOobject
            (
                "kappa",
                runTime.timeName(),
                fluidRegions[i],
                IOobject::NO_READ,
                IOobject::AUTO_WRITE
            ),
            thermoFluid[i].kappa()
        )
    );

    Info<< "    Adding to rhoCpFluid\n" << endl;
    rhoCpFluid.set
    (
        i,
        new volScalarField
        (
            IOobject
            (
                "rhoCp",
                runTime.timeName(),
                fluidRegions[i],
                IOobject::NO_READ,
                IOobject::NO_WRITE
            ),
            thermoFluid[i].rho()*thermoFluid[i].Cp()
        )
    );
    rhoCpFluid[i].oldTime();
```

```
    Info<< "␣␣␣␣Adding␣to␣turbulenceFluid\n" << endl;
    turbulenceFluid.set
    (
        i,
        CompressibleTurbulenceModel<multiphaseSystem>::New
        (
            rhoFluid[i],
            UFluid[i],
            rhoPhiFluid[i],
            thermoFluid[i]
        )
    );

    pFluid[i] = p_rghFluid[i] + rhokFluid[i]*ghFluid[i];

    Info<< "␣␣␣␣Adding␣to␣KFluid\n" << endl;
    KFluid.set
    (
        i,
        new volScalarField
        (
            "K",
            0.5*magSqr(UFluid[i])
        )
    );

    Info<< "␣␣␣␣Adding␣to␣dpdtFluid\n" << endl;
    dpdtFluid.set
    (
        i,
        new volScalarField
        (
            IOobject
            (
                "dpdt",
                runTime.timeName(),
                fluidRegions[i]
            ),
            fluidRegions[i],
            dimensionedScalar(thermoFluid[i].p().dimensions()/
               dimTime, Zero)
        )
    );

    pimpleFluid[i].dict().readIfPresent("correctPhi",
       correctPhiFluid[i]);
    pimpleFluid[i].dict().readIfPresent("ddtCorr", ddtCorrFluid[
       i]);
    const dictionary& pimpleDict =
        fluidRegions[i].solutionDict().subDict("PIMPLE");

    pimpleDict.readIfPresent("frozenFlow", frozenFlowFluid[i]);

    rhoMaxFluid.set
    (
        i,
```

```
        new dimensionedScalar("rhoMax", dimDensity, GREAT,
            pimpleDict)
    );

    rhoMinFluid.set
    (
        i,
        new dimensionedScalar("rhoMin", dimDensity, Zero,
            pimpleDict)
    );

    pressureControls.set
    (
        i,
        new pressureControl(thermoFluid[i].p(), rhoFluid[i],
            pimpleDict, false)
    );

    Info<< "    Adding fvOptions\n" << endl;
    fluidFvOptions.set
    (
        i,
        new fv::options(fluidRegions[i])
    );

    pRefCellFluid[i] = 0;
    pRefValueFluid[i] = 0.0;

    setRefCell
    (
        pFluid[i],
        p_rghFluid[i],
        pimpleDict,
        pRefCellFluid[i],
        pRefValueFluid[i]
    );

    if (p_rghFluid[i].needReference())
    {
        pFluid[i] += dimensionedScalar
            (
            "p",
            pFluid[i].dimensions(),
            pRefValueFluid[i] - getRefCellValue(pFluid[i],
                pRefCellFluid[i])
        );
        p_rghFluid[i] = pFluid[i] - rhokFluid[i]*ghFluid[i];
    }

}
```

createFieldRefs

```
PtrList<surfaceScalarField> phiFluid(fluidRegions.size());

forAll(fluidRegions, i)
{
```

```
    Info<< "␣␣␣␣Adding␣to␣phiFluid\n" << endl;
    phiFluid.set
    (
        i,
        new surfaceScalarField
        (
            IOobject
            (
                "phi",
                runTime.timeName(),
                fluidRegions[i],
                IOobject::READ_IF_PRESENT,
                IOobject::AUTO_WRITE
            ),
            thermoFluid[i].phi()
        )
    );
}
```

initContinuityErrs

```
PtrList<uniformDimensionedScalarField> cumulativeContErrIO(
    fluidRegions.size());
forAll(cumulativeContErrIO, i)
{
    const fvMesh& mesh = fluidRegions[i];

    cumulativeContErrIO.set
    (
        i,
        new uniformDimensionedScalarField
        (
            IOobject
            (
                "cumulativeContErr",
                runTime.timeName(),
                "uniform",
                mesh,
                IOobject::READ_IF_PRESENT,
                IOobject::AUTO_WRITE
            ),
            dimensionedScalar(dimless, Zero)
        )
    );
}

UPtrList<scalar> cumulativeContErr(cumulativeContErrIO.size());
forAll(cumulativeContErrIO, i)
{
    cumulativeContErr.set(i, &cumulativeContErrIO[i].value());
}
```

ddtAlphaMultiRegionNo

```
    scalar ddtAlphaNum = -GREAT;
    forAll(fluidRegions, regioni)
    {
        ddtAlphaNum = max
```

```
        (
            ddtAlphaNo
            (
                fluidRegions[regioni],
                runTime,
                thermoFluid[regioni],
                phiFluid[regioni]
            ),
            ddtAlphaNum
        );
    }
```

compressibleMultiRegionCourantNo

```
    scalar CoNum = -GREAT;

    forAll(fluidRegions, regioni)
    {
        CoNum = max
        (
            compressibleCourantNo
            (
                fluidRegions[regioni],
                runTime,
                rhoFluid[regioni],
                phiFluid[regioni]
            ),
            CoNum
        );
    }
```

validateTurbulenceModel

```
forAll(fluidRegions, i)
{
    turbulenceFluid[i].validate();
}
```

readFluidTimeControls

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | www.openfoam.com
     \\/     M anipulation  |
-------------------------------------------------------------------------------
    Copyright (C) 2011-2015 OpenFOAM Foundation
    Copyright (C) 2020 OpenCFD Ltd.
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.
```

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

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

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

Global
    readTimeControls

Description
    Read the control parameters used by setDeltaT

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

scalar maxAlphaCo =
    runTime.controlDict().get<scalar>("maxAlphaCo");

scalar maxAlphaDdt =
    runTime.controlDict().getOrDefault<scalar>("maxAlphaDdt",
        GREAT);

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

setRegionFluidFields

```
    fvMesh& mesh = fluidRegions[i];

    multiphaseSystem& thermol = thermoFluid[i];
    volScalarField& kappa = kappaLK[i];
    volScalarField& rho = rhoFluid[i];
    volVectorField& U = UFluid[i];
    volScalarField& T = TFluid[i];
    surfaceScalarField& phi = phiFluid[i];
    volScalarField& Cp = CpFluid[i];
    volScalarField& rhoCp = rhoCpFluid[i];
    surfaceScalarField& rhoPhi = rhoPhiFluid[i];
    volScalarField& rhok = rhokFluid[i];
    dimensionedScalar& rhoR = rhoRFluid[i];
    CompressibleTurbulenceModel<multiphaseSystem>& turbulence =
        turbulenceFluid[i];
```

```
    volScalarField& K = KFluid[i];
    volScalarField& dpdt = dpdtFluid[i];
    volScalarField& p = pFluid[i];
    volScalarField& p_rgh = p_rghFluid[i];

    const volScalarField& gh = ghFluid[i];
    const surfaceScalarField& ghf = ghfFluid[i];

    fv::options& fvOptions = fluidFvOptions[i];

    bool frozenFlow = frozenFlowFluid[i];

    const label pRefCell = pRefCellFluid[i];
    const scalar pRefValue = pRefValueFluid[i];

    const dimensionedScalar rhoMax = rhoMaxFluid[i];
    const dimensionedScalar rhoMin = rhoMinFluid[i];

    const pressureControl& pressureControl = pressureControls[i
        ];

    pimpleControl& pimple = pimpleFluid[i];

    bool correctPhi = correctPhiFluid[i];
    bool ddtCorr = ddtCorrFluid[i];
```

readFluidMultiRegionPIMPLEControls

```
    const dictionary& pimpleCHT = mesh.solutionDict().subDict("
        PIMPLE");

    const int nCorr =
        pimpleCHT.getOrDefault<int>("nCorrectors", 1);

    const int nNonOrthCorr =
        pimpleCHT.getOrDefault<int>("nNonOrthogonalCorrectors",
            0);

    const bool momentumPredictor =
        pimpleCHT.getOrDefault("momentumPredictor", false);

    // correctPhi = pimple.getOrDefault("correctPhi", true);

    // ddtCorr = pimple.getOrDefault("ddtCorr", true);
```

solveFluid

```
if (finalIter)
{
    mesh.data::add("finalIteration", true);
}

#include "initCorrectPhi.H"

if (firstIter)
{
    thermol.correctMassSources(T);
    thermol.solve();
```

```
    rho = thermol.rho();

}

if (frozenFlow)
{
    #include "TEqnFluidPhase.H"
}
else
{

    #include "UEqnFluidPhase.H"
    #include "YEqnFluidPhase.H"
    #include "TEqnFluidPhase.H"
        // --- PISO loop

    for (int corr=0; corr<nCorr; corr++)
    {
        #include "pEqnFluidPhase.H"
    }
    if (pimple.turbCorr())
    {
        turbulence.correct();
    }

}

if (finalIter)
{
    rho = thermol.rho();
    mesh.data::remove("finalIteration");
}
```

initCorrectPhi

```
tmp<volScalarField> rAU;

if (correctPhi)
{
    rAU = new volScalarField
    (
        IOobject
        (
            "rAU",
            runTime.timeName(),
            mesh,
            IOobject::READ_IF_PRESENT,
            IOobject::AUTO_WRITE
        ),
        mesh,
        dimensionedScalar("rAU", dimTime/dimDensity, 1)
    );

    #include "correctPhi.H"
}
else
{
```

```
    CorrectPhi
    (
        U,
        phi,
        p_rgh,
        dimensionedScalar("rAUf", dimTime/rho.dimensions(), 1),
        geometricZeroField(),
        pimple
    );

}
```

createFluidMeshes

```
{
    rhoCp =  rho*thermol.Cp();

    rhok = rho - rhoR;
    const surfaceScalarField rhoCpPhi(fvc::interpolate(thermol.
       Cp())*rhoPhi);
    volTensorField gradU = fvc::grad(U);
    volTensorField tau = turbulence.muEff() * (gradU + gradU.T()
       );
    const volScalarField kappaEff
    (
        "kappaEff",
        thermol.kappa() + thermol.Cp()*turbulence.mut()/thermol.
           Prt()
    );


    fvScalarMatrix TEqn
    (
        fvm::ddt(rhoCp, T)
      + fvm::div(rhoCpPhi, T, "div(phi,T)")
      + fvc::div(rhoPhi/fvc::interpolate(rho),p,"div(phiv,p)")
      + fvc::ddt(rho, K) + fvc::div(rhoPhi, K)
      - fvm::laplacian(kappaEff, T,  "laplacian(kappa,T)")
      ==
        thermol.heatTransfer(T)
        + fvc::div(tau & U, "div(tau,U)")
        + rho*(U&g)
        + fvOptions(rhoCp, T)
    );
    TEqn.relax();

    fvOptions.constrain(TEqn);

    TEqn.solve(mesh.solver(T.select(finalIter)));

    fvOptions.correct(T);
    thermol.correct();

    Info<< "min/max(T)=" << min(T).value() << ", " << max(T).
       value() << endl;

}
```

UEqnFluidPhase

```
rhok = rho - rhoR;

fvVectorMatrix UEqn
(
    fvm::ddt(rho, U)
  + fvm::div(rhoPhi, U)
  + turbulence.divDevRhoReff(U)
  ==
    fvOptions(rho, U)
);

UEqn.relax();

thermol.addInterfacePorosity(UEqn);

if (momentumPredictor)
{
    solve
    (
        UEqn
      ==
        fvc::reconstruct
        (
            (
                thermol.surfaceTensionForce()
                - ghf*fvc::snGrad(rhok)
                - fvc::snGrad(p_rgh)
            ) * mesh.magSf()
        ),
        mesh.solver(U.select(finalIter))
    );

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

YEqnFluidPhase

```
{
    for (phaseModel& phase : thermol.phases())
    {
        PtrList<volScalarField>& Y = phase.Y();

        if (!Y.empty())
        {
            //- Su phase source terms
            PtrList<volScalarField::Internal> Sus(Y.size());
            //- Sp phase source terms
            PtrList<volScalarField::Internal> Sps(Y.size());

            forAll(Sus, i)
            {
                Sus.set
                (
                    i,
```

```
                    new volScalarField::Internal
                    (
                        IOobject
                        (
                            "Su" + phase.name(),
                            mesh.time().timeName(),
                            mesh
                        ),
                        mesh,
                        dimensionedScalar(dimless/dimTime, Zero)
                    )
                );
                Sps.set
                (
                    i,
                    new volScalarField::Internal
                    (
                        IOobject
                        (
                            "Sp" + phase.name(),
                            mesh.time().timeName(),
                            mesh
                        ),
                        mesh,
                        dimensionedScalar(dimless/dimTime, Zero)
                    )
                );
            }

            forAll(Y, i)
            {
                // Calculate mass exchange for species
                    consistent with
                // alpha's source terms.
                thermol.massSpeciesTransfer(phase, Sus[i], Sps[i
                    ], Y[i].name());
            }
            phase.solveYi(Sus, Sps);
        }
    }
}
```

TEqnFluidPhase

```
{
    rhoCp =  rho*thermol.Cp();

    rhok = rho - rhoR;
    const surfaceScalarField rhoCpPhi(fvc::interpolate(thermol.
        Cp())*rhoPhi);
    volTensorField gradU = fvc::grad(U);
    volTensorField tau = turbulence.muEff() * (gradU + gradU.T()
        );
    const volScalarField kappaEff
    (
        "kappaEff",
```

```
        thermol.kappa() + thermol.Cp()*turbulence.mut()/thermol.
            Prt()
    );


    fvScalarMatrix TEqn
    (
        fvm::ddt(rhoCp, T)
      + fvm::div(rhoCpPhi, T, "div(phi,T)")
      + fvc::div(rhoPhi/fvc::interpolate(rho),p,"div(phiv,p)")
      + fvc::ddt(rho, K) + fvc::div(rhoPhi, K)
      - fvm::laplacian(kappaEff, T,  "laplacian(kappa,T)")
      ==
        thermol.heatTransfer(T)
        + fvc::div(tau & U, "div(tau,U)")
        + rho*(U&g)
        + fvOptions(rhoCp, T)
    );
    TEqn.relax();

    fvOptions.constrain(TEqn);

    TEqn.solve(mesh.solver(T.select(finalIter)));

    fvOptions.correct(T);
    thermol.correct();

    Info<< "min/max(T)=" << min(T).value() << ", " << max(T).
        value() << endl;

}
```

pEqnFluidPhase

```
{
    bool closedVolume = p_rgh.needReference();
    rho = thermol.rho();

    rhok = rho - rhoR;

    if (correctPhi)
    {
        rAU.ref() = 1.0/UEqn.A();
    }
    else
    {
        rAU = 1.0/UEqn.A();
    }

    surfaceScalarField rAUf("rAUf", fvc::interpolate(rAU()));

    volVectorField HbyA("HbyA", U);

    HbyA = rAU()*UEqn.H();

    surfaceScalarField phiHbyA
    (
```

```
        "phiHbyA",
        (fvc::interpolate(HbyA) & mesh.Sf())
        + fvc::interpolate(rho*rAU())*fvc::ddtCorr(U, phi)
    );


    if (p_rgh.needReference())
    {
        fvc::makeRelative(phiHbyA, U);
        adjustPhi(phiHbyA, U, p_rgh);
        fvc::makeAbsolute(phiHbyA, U);
    }

    surfaceScalarField phig
    (
        (
            thermol.surfaceTensionForce()
            - ghf*fvc::snGrad(rhok)
        )*rAUf*mesh.magSf()
    );

    phiHbyA += phig;

    // Update the fixedFluxPressure BCs to ensure flux
       consistency
    constrainPressure(p_rgh, U, phiHbyA, rAUf);

    for (int nonOrth=0; nonOrth<=nNonOrthCorr; nonOrth++)
    {

        fvScalarMatrix p_rghEqn
        (
            fvc::div(phiHbyA)
            - fvm::laplacian(rAUf, p_rgh)
        );


        if (thermol.includeVolChange())
        {
            p_rghEqn += thermol.volTransfer(p_rgh);
        }

        p_rghEqn.setReference(pRefCell, pRefValue);

        p_rghEqn.solve
        (
            mesh.solver
            (
                p_rgh.select
                (
                    (
                        oCorr == nOuterCorr-1
                        && corr == nCorr-1
                        && nonOrth == nNonOrthCorr
                    )
                )
            )
```

```
        );

        if (nonOrth == nNonOrthCorr)
        {

            phi = phiHbyA + p_rghEqn.flux();

            p_rgh.relax();

            U = HbyA + rAU()*fvc::reconstruct((phig + p_rghEqn.
                flux())/rAUf);

            U.correctBoundaryConditions();

            fvOptions.correct(U);

            K = 0.5*magSqr(U);

        }

    }

    #include "incompressibleContinuityErrors.H"

    fvc::makeRelative(phi, U);
    p == p_rgh + rhok*gh;


    // For closed-volume cases adjust the pressure and density
        levels
    // to obey overall mass continuity
    if (p_rgh.needReference())
    {
        p += dimensionedScalar
        (
            "p",
            p.dimensions(),
            pRefValue - getRefCellValue(p, pRefCell)
        );
        p_rgh = p - rhok*gh;
    }

    if (!correctPhi)
    {
        rAU.clear();
    }
    rho = thermol.rho();

// Update pressure time derivative if needed
    if (thermol.dpdt())
    {
        dpdt = fvc::ddt(p);
    }
}
```

B.2.2 Solid region

createSolidMeshes

```
    const wordList solidNames(rp["solid"]);

    PtrList<fvMesh> solidRegions(solidNames.size());

    forAll(solidNames, i)
    {
        Info<< "Create␣solid␣mesh␣for␣region␣" << solidNames[i]
            << "␣for␣time␣=␣" << runTime.timeName() << nl <<
               endl;

        solidRegions.set
        (
            i,
            new fvMesh
            (
                IOobject
                (
                    solidNames[i],
                    runTime.timeName(),
                    runTime,
                    IOobject::MUST_READ
                )
            )
        );

        // Force calculation of geometric properties to prevent
           it being done
        // later in e.g. some boundary evaluation
        //(void)solidRegions[i].weights();
        //(void)solidRegions[i].deltaCoeffs();
    }
```

createSolidFields

```
    // Initialise solid field pointer lists
    PtrList<coordinateSystem> coordinates(solidRegions.size());
    PtrList<solidThermo> thermos(solidRegions.size());
    PtrList<radiation::radiationModel> radiations(solidRegions.
       size());
    PtrList<fv::options> solidHeatSources(solidRegions.size());
    // PtrList<volScalarField> rhos(solidRegions.size());
    // PtrList<volScalarField> cps(solidRegions.size());
    // PtrList<volScalarField> rhoCpSolid(solidRegions.size());
    // PtrList<volScalarField> Ks(solidRegions.size());
    // PtrList<volScalarField> Ts(solidRegions.size());
    PtrList<volScalarField> betavSolid(solidRegions.size());
    PtrList<volSymmTensorField> aniAlphas(solidRegions.size());

    // Populate solid field pointer lists
    forAll(solidRegions, i)
    {
        Info<< "***␣Reading␣solid␣mesh␣thermophysical␣properties
           ␣for␣region␣"
            << solidRegions[i].name() << nl << endl;
```

```
        // Info<< "    Adding to Ts\n" << endl;
        // Ts.set
        // (
        //     i,
        //     new volScalarField
        //     (
        //         IOobject
        //         (
        //             "T",
        //             runTime.timeName(),
        //             solidRegions[i],
        //             IOobject::MUST_READ,
        //             IOobject::AUTO_WRITE
        //         ),
        //         solidRegions[i]
        //     )
        // );
        Info<< "    Adding to thermos\n" << endl;
        thermos.set(i, solidThermo::New(solidRegions[i]));

        Info<< "    Adding to radiations\n" << endl;
        radiations.set(i, radiation::radiationModel::New(thermos
            [i].T()));

        Info<< "    Adding fvOptions\n" << endl;
        solidHeatSources.set
        (
            i,
            new fv::options(solidRegions[i])
        );

        if (!thermos[i].isotropic())
        {
            Info<< "    Adding coordinateSystems\n" << endl;
            coordinates.set
            (
                i,
                coordinateSystem::New
                (
                    solidRegions[i],
                    thermos[i],
                    coordinateSystem::typeName_()
                )
            );

            tmp<volVectorField> tkappaByCp =
                thermos[i].Kappa()/thermos[i].Cp();

            aniAlphas.set
            (
                i,
                new volSymmTensorField
                (
                    IOobject
                    (
                        "Anialpha",
```

```
                        runTime.timeName(),
                        solidRegions[i],
                        IOobject::NO_READ,
                        IOobject::NO_WRITE
                    ),
                    solidRegions[i],
                    dimensionedSymmTensor(tkappaByCp().
                        dimensions(), Zero),
                    zeroGradientFvPatchSymmTensorField::typeName
                )
            );

            aniAlphas[i].primitiveFieldRef() =
                coordinates[i].transformPrincipal
                (
                    solidRegions[i].cellCentres(),
                    tkappaByCp()
                );
            aniAlphas[i].correctBoundaryConditions();

        }

        IOobject betavSolidIO
        (
            "betavSolid",
            runTime.timeName(),
            solidRegions[i],
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        );

        if (betavSolidIO.typeHeaderOk<volScalarField>(true))
        {
            betavSolid.set
            (
                i,
                new volScalarField(betavSolidIO, solidRegions[i
                    ])
            );
        }
        else
        {
            betavSolid.set
            (
                i,
                new volScalarField
                (
                    IOobject
                    (
                        "betavSolid",
                        runTime.timeName(),
                        solidRegions[i],
                        IOobject::NO_READ,
                        IOobject::NO_WRITE
                    ),
                    solidRegions[i],
                    dimensionedScalar("1", dimless, scalar(1))
```

```
                )
            );
        }
        // Info<< "    Adding to rhos\n" << endl;
        // rhos.set
        // (
        //     i,
        //     new volScalarField
        //     (
        //         IOobject
        //         (
        //             "rho",
        //             runTime.timeName(),
        //             solidRegions[i],
        //             IOobject::NO_READ,
        //             IOobject::AUTO_WRITE
        //         ),
        //         thermos[i].rho()
        //     )
        // );

        // Info<< "    Adding to Cps\n" << endl;
        // cps.set
        // (
        //     i,
        //     new volScalarField
        //     (
        //         IOobject
        //         (
        //             "cp",
        //             runTime.timeName(),
        //             solidRegions[i],
        //             IOobject::NO_READ,
        //             IOobject::AUTO_WRITE
        //         ),
        //         thermos[i].Cp()
        //     )
        // );

        // Info<< "Calculating field rhoCps\n" << endl;
        // rhoCpSolid.set
        // (
        //     i,
        //     new volScalarField
        //     (
        //         IOobject
        //         (
        //             "rhoCp",
        //             runTime.timeName(),
        //             solidRegions[i],
        //             IOobject::NO_READ,
        //             IOobject::NO_WRITE
        //         ),
        //         thermos[i].rho()*thermos[i].Cp()
        //     )
        // );
```

```
        // Info<< "    Adding to Ks\n" << endl;
        // Ks.set
        // (
        //     i,
        //     new volScalarField
        //     (
        //         IOobject
        //         (
        //             "K",
        //             runTime.timeName(),
        //             solidRegions[i],
        //             IOobject::NO_READ,
        //             IOobject::AUTO_WRITE
        //         ),
        //         thermos[i].kappa()
        //     )
        // );



    }
```

solidRegionDiffusionNo

```
scalar DiNum = -GREAT;

forAll(solidRegions, i)
{
    //- Note: do not use setRegionSolidFields.H to avoid double
       registering Cp
    //#include "setRegionSolidFields.H"
    const solidThermo& therms = thermos[i];

    tmp<volScalarField> magKappa;
    if (therms.isotropic())
    {
        magKappa = therms.kappa();
    }
    else
    {
        magKappa = mag(therms.Kappa());
    }

    tmp<volScalarField> tcp = therms.Cp();
    const volScalarField& cp = tcp();

    tmp<volScalarField> trho = therms.rho();
    const volScalarField& rho = trho();

    DiNum = max
    (
        solidRegionDiffNo
        (
            solidRegions[i],
            runTime,
            rho*cp,
```

```
            magKappa()
        ),
        DiNum
    );

}
```

solidRegionDiffNo.H

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

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

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

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

Description
    Calculates and outputs the mean and maximum Diffusion Numbers for the solid
    regions

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

#ifndef solidRegionDiffNo_H
#define solidRegionDiffNo_H
```

```
#include "fvMesh.H"

namespace Foam
{
    scalar solidRegionDiffNo
    (
        const fvMesh& mesh,
        const Time& runTime,
        const volScalarField& Cprho,
        const volScalarField& kappa
    );
}

#endif

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

solidRegionDiffNo.C

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | www.openfoam.com
     \\/     M anipulation  |
-------------------------------------------------------------------------------
    Copyright (C) 2011-2017 OpenFOAM Foundation
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

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

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

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

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

#include "solidRegionDiffNo.H"
#include "surfaceInterpolate.H"

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

Foam::scalar Foam::solidRegionDiffNo
(
    const fvMesh& mesh,
    const Time& runTime,
    const volScalarField& Cprho,
    const volScalarField& kappa
)
{
    surfaceScalarField kapparhoCpbyDelta
    (
        sqr(mesh.surfaceInterpolation::deltaCoeffs())
       *fvc::interpolate(kappa)
       /fvc::interpolate(Cprho)
    );

    const scalar DiNum = max(kapparhoCpbyDelta).value()*runTime.
       deltaTValue();
    const scalar meanDiNum =
        average(kapparhoCpbyDelta).value()*runTime.deltaTValue()
           ;

    Info<< "Region:␣" << mesh.name() << "␣Diffusion␣Number␣mean:
       ␣" << meanDiNum
        << "␣max:␣" << DiNum << endl;

    return DiNum;
}

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

setRegionSolidFields

```
fvMesh& mesh = solidRegions[i];
solidThermo& therms = thermos[i];

// volScalarField& rho = rhos[i];
// volScalarField& cp = cps[i];
// volScalarField& rhoCps = rhoCpSolid[i];
// volScalarField& K = Ks[i];
// volScalarField& T = Ts[i];

tmp<volScalarField> trho = therms.rho();
const volScalarField& rhosol = trho();

tmp<volScalarField> tcp = therms.Cp();
const volScalarField& cpsol = tcp();
```

```
// tmp<volScalarField> tkappa = thermo.kappa();
// const volScalarField& kappa = tkappa();

// volScalarField& Tsol = Ts[i];

tmp<volSymmTensorField> taniAlpha;
if (!therms.isotropic())
{
    volSymmTensorField& aniAlpha = aniAlphas[i];
    tmp<volVectorField> tkappaByCp = therms.Kappa()/cpsol;
    const coordinateSystem& coodSys = coordinates[i];

    aniAlpha.primitiveFieldRef() =
        coodSys.transformPrincipal
        (
            mesh.cellCentres(),
            tkappaByCp()
        );

    aniAlpha.correctBoundaryConditions();

    taniAlpha = tmp<volSymmTensorField>
    (
        new volSymmTensorField(aniAlpha)
    );
}

const volScalarField& betav = betavSolid[i];

fv::options& fvOptions = solidHeatSources[i];
volScalarField& h = therms.he();
```

readSolidMultiRegionPIMPLEControls

```
const dictionary& pimpleCHT = mesh.solutionDict().subDict("
   PIMPLE");

int nNonOrthCorr =
    pimpleCHT.getOrDefault<int>("nNonOrthogonalCorrectors", 0);
```

solveSolid

```
if (finalIter)
{
    mesh.data::add("finalIteration", true);
}

{
    for (int nonOrth=0; nonOrth<=nNonOrthCorr; ++nonOrth)
    {
        fvScalarMatrix hEqn
        (
            fvm::ddt(betav*rhosol, h)
          - (
               therms.isotropic()
             ? fvm::laplacian(betav*therms.alpha(), h, "
                laplacian(alpha,h)")
```

```
                : fvm::laplacian(betav*taniAlpha(), h, "laplacian(
                    alpha,h)")
            )
          ==
            fvOptions(rhosol, h)
        );

        hEqn.relax();

        fvOptions.constrain(hEqn);

        hEqn.solve(mesh.solver(h.select(finalIter)));

        fvOptions.correct(h);
    }

    therms.correct();

    Info<< "Min/max T:" << min(therms.T()).value() << ' '
        << max(therms.T()).value() << endl;
}

if (finalIter)
{
    mesh.data::remove("finalIteration");
}
```

readSolidTimeControls

```
/*---------------------------------------------------------------------
    ------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD
     Toolbox
   \\    /   O peration     |
    \\  /    A nd           | www.openfoam.com
     \\/     M anipulation  |
-----------------------------------------------------------------------
    ---------------
    Copyright (C) 2011 OpenFOAM Foundation
    Copyright (C) 2020 OpenCFD Ltd.
-----------------------------------------------------------------------
    ---------------
License
    This file is part of OpenFOAM.

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

    OpenFOAM is distributed in the hope that it will be useful,
       but WITHOUT
```

```
    ANY WARRANTY; without even the implied warranty of
        MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General
        Public License
    for more details.

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

Global
    readSolidTimeControls

Description
    Read the control parameters used in the solid

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

scalar maxDi = runTime.controlDict().getOrDefault<scalar>("maxDi
    ", 10);

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