Ana Cláudia Leite Magalhães

Ana Cláudia Leite Magalhães OpenFOAM® simulation of the injection moulding filling stage UMinho | 2016

Universidade do Minho
Escola de Engenharia

Ana Cláudia Leite Magalhães

OpenFOAM® simulation of the injection moulding filling stage

setembro de 2016

Universidade do Minho
Escola de Engenharia

Ana Cláudia Leite Magalhães

OpenFOAM® simulation of the injection moulding filling stage

Dissertação de Mestrado
Ciclo de Estudos Integrados Conducentes ao
Grau de Mestre em Engenharia de Polímeros

Trabalho efetuado sob a orientação do
Doutor Luís Jorge Lima Ferrás
Doutor Célio Bruno Pinto Fernandes

setembro de 2016

Universidade do Minho

DECLARAÇÃO

Nome: Ana Cláudia Leite Magalhães

Endereço electrónico: aclaudialm@hotmail.com

Número do Bilhete de Identidade: 14373247

Título dissertação: OpenFOAM® simulation of the injection moulding filing stage

Orientadores: Doutor Luís Jorge Lima Ferrás e Doutor Célio Bruno Pinto Fernandes

Ano de conclusão: 2016

Designação do Mestrado: Mestrado Integrado em Engenharia de Polímeros

Nos exemplares das teses de doutoramento ou de mestrado ou de outros trabalhos entregues para prestação de provas públicas nas universidades ou outros estabelecimentos de ensino, e dos quais é obrigatoriamente enviado um exemplar para depósito legal na Biblioteca Nacional e, pelo menos outro para a biblioteca da universidade respectiva, deve constar uma das seguintes declarações:

1. É AUTORIZADA A REPRODUÇÃO INTEGRAL DESTA TESE/TRABALHO APENAS PARA EFEITOS DE INVESTIGAÇÃO, MEDIANTE DECLARAÇÃO ESCRITA DO INTERESSADO, QUE A TAL SE COMPROMETE;

2. É AUTORIZADA A REPRODUÇÃO PARCIAL DESTA TESE/TRABALHO (indicar, caso tal seja necessário, nº máximo de páginas, ilustrações, gráficos, etc.), APENAS PARA EFEITOS DE INVESTIGAÇÃO, , MEDIANTE DECLARAÇÃO ESCRITA DO INTERESSADO, QUE A TAL SE COMPROMETE;

3. DE ACORDO COM A LEGISLAÇÃO EM VIGOR, NÃO É PERMITIDA A REPRODUÇÃO DE QUALQUER PARTE DESTA TESE/TRABALHO

Universidade do Minho, ___/___/_____

Assinatura: _____

Acknowledgments

I am grateful to my advisor Luís Ferrás, for all his patience and support.

I am also grateful for the opportunity to participate in the 11th OpenFOAM® workshop.

My gratitude to Célio Pinto, without him I couldn't do this work, because no new boundary condition would exist and I also want to thank him for the viscoelasticInterFoam code.

Abstract

OpenFOAM® simulation of the injection moulding filling stage.

The main purpose of this work is the simulation of the injection moulding filling stage using the open source software OpenFOAM® (version 2.3.1). The program can already solve general problems, but, we would like to extend its capabilities, by creating and implementing a new boundary condition capable of letting the air out of the mould, but not the polymer.

We consider different locations for the air exit and study their influence on the quality of the final results.

In addition, since the material viscosity influences the evolution of the flow front and its behaviour inside the mould, we also study the filling of the mould cavity considering both a Newtonian and a generalized Newtonian model (Bird-Carreau) fluids.

In order to take into account non-isothermal flows, we have implemented the energy equation in the interFoam solver. We also added a temperature-dependent viscosity model (the Cross model) modified with the Arrhenius equation. The solver interFoam was also extended to viscoelastic flows, and simple tests were performed using an Oldroyd-B model.

Finally we have performed simulations of the filling process of a real injection part, a tensile specimen and its feeding system.

Resumo

Simulação em OpenFOAM® da fase de enchimento do processo de moldação por injeção.

Este trabalho tem como principal objetivo simular a fase de enchimento do processo de moldação por injeção, usando o *software* de código aberto, OpenFOAM® (versão 2.3.1) O programa já consegue resolver problemas gerais, no entanto, para simular melhor a realidade é necessário adaptá-lo. Deste modo, a primeira fase deste estudo foi tentar resolver um dos problemas da abordagem computacional do processo de moldação por injeção, a saída do ar. Para isso, criámos e implementamos uma condição fronteira capaz de deixar o ar sair, mas o polímero não. Depois, estudámos a influência da localização desta condição na frente do fundido.

Adicionalmente, como a viscosidade do material tem influência no seu movimento e na evolução da sua frente de fluxo, também estudámos o enchimento de um molde usando um fluido Newtoniano e um modelo Newtoniano generalizado (modelo de Bird-Carreau).

Como o solver atualmente existente é isotérmico e a moldação por injeção é um processo não-isotérmico, a equação da energia foi implementada no solver interFoam. Adicionámos ainda um modelo para a viscosidade, que dependente da temperatura (modelo de Cross modificado com a equação de Arrhenius).

O solver interFoam foi ainda estendido para fluídos viscoelásticos, tendo sido feito algumas simulações simples com o modelo Oldroyd-B.

Por último, foram feitas simulações usando uma geometria real do processo de moldação por injeção. Essa geometria é um provete usado nos testes de tração, incluindo o seu sistema de alimentação.

Contents

Index of Figures

Index of tables

List of symbols

$\mathbf{u}$	Velocity vector
t	Time
p	Pressure
$\boldsymbol{\tau}$	Stress tensor
ρ	Density
g	Gravity acceleration vector
$\mathbf{D}$	Strain tensor
$\dot{\gamma}$	Shear rate
II_D	Second invariant of the rate of strain tensor
G	Modulus of elasticity
$\overset{\nabla}{\tau}$	Upper convective derivative
η_s	Solvent viscosity
λ	Relaxation time
$\boldsymbol{\alpha}$	Mobility factor
η_p	Polymer viscosity coefficient
C_p	Specific heat capacity
k	Thermal conductivity
T	Temperature
τ	Shear stress
α	Phase fraction
σ	Surface tension
k	Mean curvature of free surface
p_d	Modified pressure
ν	Cinematic viscosity
δt	Time step
δx	Dimension of the cell

Co	Courant number

1.Introduction

Nowadays, any project activities should be supported by suitable simulation tools, aiming the optimization of the process and the minimization of the used resources.

In the market there are numerous modelling codes able to simulate extremely complex processes, but, the dissemination of these codes in industry faces two huge difficulties, the cost of commercial simulation licenses and/or the nonexistence of human resources able to use these tools adequately. For the first problem, costs limitations, a possible solution is the use of open source modelling software. For the second problem, specialized human resources, the R&D section should be progressively added/improved in the companies to help in the products development.

Along the years, the number of open source code users increased, and these users became more organized. As a consequence, the quality of open source simulation codes was improved, leading to the development of codes with the same capabilities of the commercial ones. Also, teams that in the past developed their own simulation codes are now gathering their efforts to provide powerful and common simulation codes, to every member of the community.

A good example is the software OpenFOAM® (Open source Field Operation and Manipulation) [1] which is capable of simulating complex systems behaviour, involving fluids and/or solids, multiphase systems, being also able to make parallel calculations.

One of the main advantages of using OpenFOAM® is the possibility of own programming/adaptation, which allows the development of new applications using a symbolic language.

In this work we are interested in the simulation of the polymer injection process, more precisely, the filling stage, and accordingly to what was stated above, the OpenFOAM® software seems to be the adequate tool to perform such studies.

1.1. The injection moulding machine and process

The consumption of plastics has increased significantly in the last few decades, overcoming the consumption of any other raw material. Because of their versatility, polymers can be used in several sectors, such as, for example: automotive, packaging, agriculture, electrical, etc.

In order to obtain the final plastic product with the desire shape, we have to transform raw material, granules or powders of plastic/polymer, into a polymer melt that can easily be shaped, and, one of the most used techniques to achieve this, it is the injection moulding process. In this process the raw material is forced to move along a heated channel (where the material becomes a polymer melt), being then injected into a mould, that gives the final shape of the part being produced.

The equipment used in this processing technique has four units: control, power, clamping and injection units. In Figure 1.1 it is possible to observe this four units as well as the tool needed for part creation – the mould.

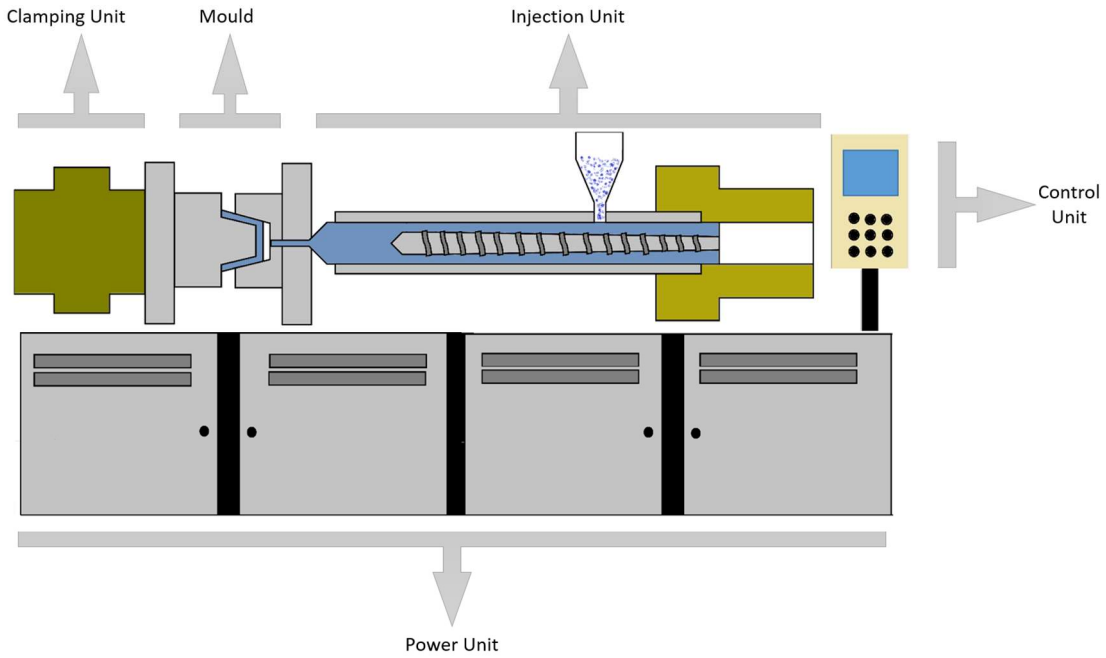

Figure 1.1: Injection moulding equipment scheme.

The power unit provides the energy to the system. The control unit is responsible for monitoring and control of the operative variables (temperature, pressure, velocity, time, stroke), being the interface that connects the operator with the injection moulding machine. Clamping and injection units and mould are the three elements that together create the stages of the injection process:

1- Mould closing: the injection moulding process is initiated when the mould closes (clamping unit is responsible for this stage).

2- Injection or filling phase: after the closing of the mould the screw moves forward and the molten material is injected into the mould cavity (the injection unit is responsible for this stage).
3- Holding: when the filling is completed, we may obtain a filling that goes from 90% to 99%, and, in order to be sure that the material is correctly packed, we keep the pressure – the holding phase begins. This stage is needed if we want to add more material to compensate the shrinkage of material as it cools.
4- Cooling and plasticization: afterwards there is a time for cooling so the part can be ejected without losing its form. At the same time, new molten material homogenization is being done (phases responsible by mould and injection unit, respectively).
5- Open and ejection: When the material is cold enough the mould is opened and the final part is ejected, and, another cycle starts again (the clamping unit is responsible for these phases).

Now that we know the basic concepts of the injection moulding process, we should not get the wrong idea that this is a simple and easy process. Injection moulding is a very complex process, and, a large number of important details need to be taken into account in order to have a robust and efficient process. For example, the nature of the injection moulding, the material properties, the complexity of the part and the mould, the process optimization/stability, are crucial for obtaining a good final product.

When we think about the optimization of the injection moulding process, the first idea that pops to our heads is to perform experimental tests, so that the process under optimization can be known in detail, and, the role of the different variables influencing the quality of the final result can be understood. This way of doing things can be an expensive choice, since tools for injection moulding are expensive. On the other hand, simulation can be comparatively cheaper in the initial steps of mould and part design, and, allows the evaluation of other design options of the mould, part and material. [2]

In this work we are going to focus our attention in the simulation of the filling phase, not considering any compressibility of the material.

1.2. The OpenFOAM® computational library

The development of the originally called FOAM started in eighties with Henry Weller and Hrvoje Jasak. They founded Nabla, Ltd, but the company failed to lead FOAM successfully in the market. In 2004, Henry Weller and Hrvoje Jasak created their own company, Hrvoje Jasak founded Wikki, Lda to release foam-extended and Henry Weller together with Chris Greenshields and Mattijs Janssens founded OpenCFD, Ltd to develop and release OpenFOAM. [3]

OpenFOAM (**Open** **F**ield **O**peration and **M**anipulation) [4] is an open source software capable of solving anything from complex fluid flows involving turbulence, heat transfer, chemical reactions and much more.

This software can also simulate multiphase flows, and, to track the fluid front or interface between two different phases, OpenFOAM® uses a modified volume of fluid method (VOF), that will be explained later.

To solve the differential equations, OpenFOAM® uses a finite volume method (FVM) based methodology [5].

This software includes tools for meshing and pre and post-processing, and, it also allows parallel computations. The program also provides a great flexibility, allowing the user to modify or add existing functionalities, being this a huge advantage over the commercial software's.

1.3. State of the art

1.3.1. One-Dimensional injection simulation

The first developments in the mathematical modelling of the injection moulding filling phase were limited to one-dimensional cases. These works started in the late fifties [2] but only in the seventies numerical simulations became a reality.

In the early seventies Kamal and Kenig [6] proposed a theoretical model for the injection moulding behaviour. The study took into account the non-Newtonian behaviour of molten polymers and the effect of temperature on density and viscosity. They obtained numerical solutions for radial flow in a semi-circular cavity. A year later, in 1973, Berger and Gogos [7] presented a numerical simulation of mould filling in a disk. They were able to predict filling

times considering both isothermal and non-isothermal flows. In 1975 Williams and Lord [8] developed a finite difference analysis to predict temperature, pressure and velocity for plastic flow in circular channels. A year later, Wu, Huang and Gogos [9] solved the equations governing the filling of a disk considering a transient nature and a non-isothermal flow. They also predicted gate pressures. In a subsequent work [10] by the same authors, their model was tested in more complex geometries.

1.3.2. 2.5-Dimensional injection simulation

In the early seventies we witnessed an increased interest in the Hele-Shaw flow modelling approach [2]. Models that use Hele-Shaw approximations neglect pressure and velocity variation in the thickness direction which results in a two-dimensional problem [11]. Because of this, models using Hele-Shaw approximations are called 2.5-dimensional models.

The following articles used this Hele-Shaw simplification. In 1972 Richardson [12] studied the injection filling of a narrow channel. They placed a drop of a Newtonian fluid inside the channel, and they injected further fluid with the same material properties, to determine the way the drop expands. In the next year Broyer, Tadmor and Gutfinger [13] proposed a method for solving the filling of a rectangular channel with a Newtonian fluid under isothermal conditions. To solve the problem they used a flow analysis network (FAN). In 1975, White [14] analysed the filling of a rectangular cavity considering isothermal and non-isothermal flow. In the same year Kamal, Kuo and Doan [15] presented two models for simulating the injection moulding filling of thin rectangular cavities. One of the models used Hele-Shaw approximation. Their studies also showed comparisons between experimental and computational results. They concluded that the employment of the numerical model, describing the injection process was a success, but it had a deficiency in the prediction of the melt front shape. In 1980 Hieber and Shen [16] presented a model for non-Newtonian fluids under non-isothermal conditions. To solve their model they implemented a finite-element/finite-difference scheme.

1.3.3. Three-Dimensional injection simulation

Parts are becoming more and more complex, meaning that thickness plays an important role that cannot be overlooked. Therefore, the validity of the Hele-Shaw approximation became limited to simple geometries/conditions. To overcome this problem, the use of three-dimensional modelling to simulate injection moulding is a demand.

Nowadays, most of studies use three-dimensional numerical models. In 1998 Hétu *et al.* [17] presented a three-dimensional finite element method capable of predicting velocity, pressure, temperature and the position of the flow fronts. The fluid behaviour was modelled using Carreau and Arrhenius constitutive models. In the same year Pichelin and Coupez [18] simulated the filling process considering a viscous incompressible flow under isothermal conditions. To solve the fluid motion equation they introduced a Taylor-Galerkin scheme. Three years later Chang and Yang [19] presented a finite volume approach to simulate the injection moulding filling stage. They simulated the isothermal flow of an incompressible Newtonian fluid. They also compared their study with the results obtained by assuming the Hele-Shaw simplification. The two models produce identical results for the filling of thin cavities, but the three-dimensional model showed a better accuracy in the filling of thicker cavities.

More recently Yan, Zhou and Li [20] solved the Navier-Stokes equations using streamline–upwind/Petrov–Galerkin (SUPG) and pressure-stabilizing/ Petrov–Galerkin (PSPG) formulations. Their results were compared with the commercial software Moldflow, revealing that the applied model achieved identical results for the filling process.

More and more works are being published everyday on the subject of numerical simulation of the injection moulding process, considering new and different phenomena that occur along the process, and also taking into account the characteristics of the different fluids. For instance, we have the study of Azaman *et al.* [21] on residual stress distribution in the injection moulding process using wood polymer composites and the study of Kim and Isayev [22] in birefringence using co-injection moulding.

1.4. Motivation

Nowadays, simulation is a crucial tool in the injection moulding process. By performing simulations, it is possible to predict potential defects on the filling stage, obtain a better understanding of the injection moulding process, and comprehend the different variables involved in the filling stage.

Since commercial simulation software's are very expensive, the use of the open-source OpenFOAM® software seems to be a good alternative (we will be using version 2.3.1). The software does not come with boundary conditions that allows the air out and keep the fluid inside the mould, and therefore, the main motivation of this work is to upgrade the solver interFOAM, in order to model correctly the air vents present in the real moulds.

1.5. Objectives

The main objective of this work is to simulate the injection moulding filling phase, and for that we need to accomplish the following:

- Create a computational approach to accurately model the air vents.
- Study the filling with a Newtonian fluid, inelastic and viscoelastic models.
- Study the filling of the mould considering a non-isothermal flow, so that the influence of temperature can be taken into account.

1.6. Thesis structure

This thesis is divided into eight main chapters. In Chapter 2 we present the governing equations for the filling stage of the injection moulding process. In Chapter 3 we describe the basics of the numerical method used to solve the governing equations.

From Chapters 4 to 8 we show the studies we have performed for simulating the filling stage in simple geometries. In Chapter 4, we use a computational approach for reproducing the real air vents used in the mould. In Chapter 5, we focus our attention on a problem arising in most of the simulations. In Chapter 6, we simulate a non-isothermal flow with a temperature-dependent viscosity model, and we compare our results with the experimental and numerical study of Wang, Li and Han [23]. In Chapter 8, we

show some simple results obtained for the filling simulation using a viscoelastic model.

In Chapter 9 we simulate the injection moulding filling stage considering a more complex part, a tensile specimen. In the last chapters we present a critical and brief discussion on the major achievements of this work and possible future work.

2. Governing equations

In the next subsections we present a brief description of the models used to model the fluid flow. We start by presenting the bulk governing equations, considering inelastic and viscoelastic fluids, and, we then present the extra equations needed to calculate free surface flows.

2.1. Newtonian and non-Newtonian fluids

During the filling phase of the injection moulding process the melt is assumed to be incompressible. The flow of non-Newtonian inelastic, incompressible fluids is governed by continuity,

$$\nabla.\mathbf{u} = 0 \tag{1}$$

and the momentum equations,

$$\rho\frac{\partial \mathbf{u}}{\partial t} + \rho\nabla.\mathbf{uu} = -\nabla p + \nabla.\boldsymbol{\tau} - \rho g \tag{2}$$

In Equation (2), t is the time, $\mathbf{u}$ is the velocity vector, p is the pressure, $\boldsymbol{\tau}$ is the deviatoric stress tensor, ρ is the fluid density and g is the gravity acceleration vector. All equations are written in a coordinate free form. The stress tensor obeys the following law for generalized Newtonian fluids,

$$\boldsymbol{\tau} = 2\eta(\dot{\gamma})\mathbf{D} \tag{3}$$

with the rate of strain tensor $\mathbf{D}$ given by,

$$\mathbf{D} = \frac{1}{2}\left(\left[\nabla\mathbf{u}\right] + \left[\nabla\mathbf{u}\right]^{\mathrm{T}}\right) \tag{4}$$

and $\eta(\dot{\gamma})$ representing the fluid viscosity function and $\dot{\gamma}$ the shear rate.

For the case where viscosity is constant, $\eta(\dot{\gamma}) = \mu$, we are in the presence of a Newtonian fluid, otherwise when the viscosity varies (for example, with the shear rate) we say the fluid is non-Newtonian.

Depending on how viscosity changes with time, the flow behaviour is characterized as thixotropic (viscosity decreases with time) or rheopetic (viscosity increases with time)

Fluids can also be catalogued depending on how viscosity changes with shear rate: shear thinning (the viscosity decreases with increased shear rate) as for example polymer melts; shear thickening (viscosity increases with increased

shear rate); plastic (a certain stress must be applied before flow occurs). Examples of shear thinning fluids are for example, polymer melts and shampoo.

Based on these differences between fluids, empirical models were proposed in the literature for the viscosity dependence on the second invariant of the stress tensor (which coincides with the shear rate for a simple shear deformation). Some examples are:

The power law model, that is given by

$$\eta(\dot{\gamma}) = a\dot{\gamma}^{n-1} \tag{5}$$

with a and n (n=1: Newtonian fluid, 0<n<1: shear thinning, n>1: shear thickening) empirical parameters. This model presents some limitations, such as a zero viscosity for high shear rates, and inexistence of a Newtonian plateau.

The Carreau model (more complex) already accounts for the features lacking the power-law model,

$$\eta(\dot{\gamma}) = \mu_\infty + \left(\mu_0 - \mu_\infty\right)\left[1 + \left(\lambda\dot{\gamma}\right)^2\right]^{\frac{n-1}{2}} \tag{6}$$

Here, μ_∞, μ_0, λ, and n are constant parameters which are determined by experimental investigations. For both models, $\dot{\gamma} = \sqrt{-4II_D}$, with II_D the second invariant of the rate of strain tensor.

The fluids just described cannot take into account the effects of viscoelasticity, and therefore, more complex models were derived. These models either come from molecular theories, network theories or are empirical.

2.2. Viscoelastic fluids

The key feature of viscoelastic fluids is the existence of relaxation and retardation times. When we apply a stress to a Newtonian fluid the response is instantaneous (the relaxation time is zero, or almost zero). On the other hand, if we have a viscoelastic fluid, the response will have a delay (the relaxation time is different from zero).

Maxwell proposed a viscoelastic model (originally it was derived for gases) that couples two properties of the viscoelastic fluids, elasticity and viscosity. To represent the mechanical equivalent of this model we can assume a spring (elasticity) connected to a dashpot (viscosity), with both objects subject to the same stress (Figure 2.1).

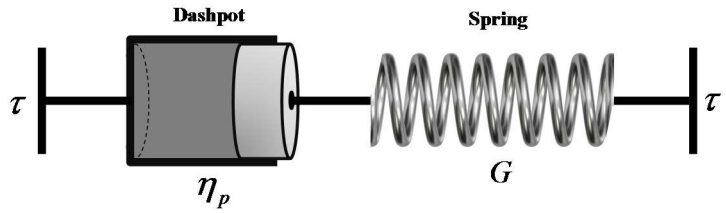

Figure 2.1: Representation of the viscoelastic behaviour with a spring and dashpot.

For quick deformations the fluid behaves as a Hookean elastic solid with modulus of elasticity *G*, for small deformations the fluid behaves as a Newtonian fluid. For solids, the stress is given by a constant (*G*) times the deformation (strain) $\tau = G\gamma_e$, while for a liquid, the deformation can be infinite so the measure "deformation" is of no use. The rate of deformation ($\dot{\gamma}_v$) is used instead, $\tau = \eta_p \dot{\gamma}_v$ · The total rate of deformation is given by $\dot{\gamma} = \dot{\gamma}_e + \dot{\gamma}_v$, meaning that,

$$\frac{1}{G}\frac{d\tau}{dt} + \frac{\tau}{\eta_p} = \dot{\gamma} \tag{7}$$

With some algebra and using tensor variables, we arrive at the Maxwell model,

$$\boldsymbol{\tau} + \lambda \frac{\partial \boldsymbol{\tau}}{\partial t} = 2\eta D \tag{8}$$

This model can be generalized in order to become frame invariant, resulting in the Upper Convected Maxwell model,

$$\boldsymbol{\tau} + \lambda \overset{\nabla}{\boldsymbol{\tau}} = 2\eta D \quad with \quad \overset{\nabla}{\boldsymbol{\tau}} = \frac{\partial \boldsymbol{\tau}}{\partial t} + \mathbf{u}.\nabla\boldsymbol{\tau} - \left[\left(\nabla \mathbf{u}\right)^{\mathrm{T}}.\boldsymbol{\tau}\right] - \left[\boldsymbol{\tau}.\nabla\mathbf{u}\right] \tag{9}$$

Where $\overset{\nabla}{\tau}$ stands for the upper convective derivative.

If in Equation (2) $\boldsymbol{\tau} = 2\eta_s D + \boldsymbol{\tau}_{UCM}$, where η_s is the solvent viscosity and $\boldsymbol{\tau}_{UCM}$ is the stress obtained with the UCM model (Equation (9)), then the Oldroyd-B model is obtained.

Several other models exist in the literature, and a possible way to construct new models could be the use of different combination of springs and dashpots (Figure 2.2)

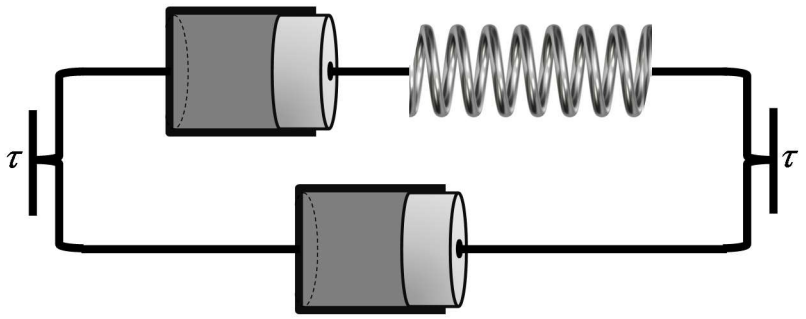

Figure 2.2: Representation of the viscoelastic behaviour with springs and dashpots (different combinations).

The combination of springs and dashpots is up the imagination of the person creating the model, but the model should be physically acceptable, providing a physical behaviour when deformed. The different parameters used for the springs and the dashpots allow the fit of the model to the fluid under study. Note that as said before, not all models are derived based on springs and dashpots.

We now present some viscoelastic models popular in the literature.

2.2.1. PTT model

The **PTT** model, according to [24] is:

$$f\left(tr\left[\boldsymbol{\tau}\right]\right)\boldsymbol{\tau}+\lambda\left(\frac{\partial\boldsymbol{\tau}}{\partial t}+\mathbf{u}.\nabla\boldsymbol{\tau}-\left[\left(\nabla\mathbf{u}\right)^{\mathrm{T}}.\boldsymbol{\tau}+\boldsymbol{\tau}.\nabla\mathbf{u}\right]\right)=\eta\left(\nabla\mathbf{u}+\left(\nabla\mathbf{u}\right)^{\mathrm{T}}\right) \tag{10}$$

Where $f\left(tr\left[\boldsymbol{\tau}\right]\right)$ is a function depending on the trace of the stress tensor $\boldsymbol{\tau}$, λ is the relaxation time and η is the viscosity.

In the literature there are two functions $f\left(tr\left[\boldsymbol{\tau}\right]\right)$. The first one is the linearized function, presented by,

$$f\left(tr\left[\boldsymbol{\tau}\right]\right)=1+\frac{\varepsilon\lambda}{\eta}tr\left[\boldsymbol{\tau}\right] \tag{11}$$

which is acceptable for low Reynolds numbers, where small molecular deformation occurs.

The second function, is exponential and is given by,

$$f\left(tr\left[\boldsymbol{\tau}\right]\right)=\exp\left(\frac{\varepsilon\lambda}{\eta}tr\left[\boldsymbol{\tau}\right]\right) \tag{12}$$

The parameter ε is related to the elongational behaviour that the model predicts.

2.2.2. Giesekus model

The **Giesekus** model, according to [25] is given by,

$$\boldsymbol{\tau}_p+\lambda_i\left[\frac{\partial\boldsymbol{\tau}_p}{\partial t}+\mathbf{u}.\nabla\boldsymbol{\tau}_p-\left(\left(\nabla\mathbf{u}\right)^{\mathrm{T}}.\boldsymbol{\tau}_p+\boldsymbol{\tau}_p.\nabla\mathbf{u}\right)\right]+\frac{\boldsymbol{\alpha}\lambda}{\eta_p}\left(\boldsymbol{\tau}_p.\boldsymbol{\tau}_p\right)=\eta_p\left(\nabla\mathbf{u}+\left(\nabla\mathbf{u}\right)^{\mathrm{T}}\right) \tag{13}$$

Where λ is the polymer relaxation time, $\boldsymbol{\alpha}$ is the mobility factor, associated with anisotropic Brownian motion and anisotropic hydrodynamic drag of the polymer molecules, and, η_p is the polymer viscosity coefficient.

2.2.3. Other models

Other models exists in the literature that show some improvements over the models previously described. For example, we have the FENE-P (finitely extensible nonlinear elastic) model [26] that takes into account the fact that a molecule should not be stretched infinitely. The constitutive equation is based on the evolution of the configuration tensor $\mathbf{A}$ that can be linked to the stress tensor $\boldsymbol{\tau}_p$:

$$\frac{\partial\mathbf{A}_p}{\partial t}+\mathbf{u}.\nabla\mathbf{A}_p-\left(\left(\nabla\mathbf{u}\right)^{\mathrm{T}}.\mathbf{A}_p+\mathbf{A}_p.\nabla\mathbf{u}\right)=-\frac{1}{\lambda}\left(f\left(tr\left[\mathbf{A}\right]\right)\mathbf{A}-a\mathbf{I}\right) \tag{14}$$

$$\boldsymbol{\tau}_p=\frac{\eta_p}{\lambda}\left(\frac{L^2}{L^2-tr\left[\mathbf{A}\right]}\mathbf{A}-a\mathbf{I}\right) \tag{15}$$

In these equations the constant model parameter L^2 is the extensibility parameter and $a=1/(1-3/L^2)$.

Another way to improve the modelling of the physical behaviour of viscoelastic fluids is to use integral models, which take into account all the history of deformation. Another way to improve the modelling is to sum several

constitutive equations so that the variety of relaxation times present in a real material is better represented.

$$\boldsymbol{\tau}_p = \sum_{i=1}^{n} \boldsymbol{\tau}_{pi} \tag{16}$$

2.3. Non-isothermal flow

The injection moulding filling stage is a non-isothermal process, that is, the temperature is not constant along the process. For that reason, we need to consider the energy equation, given by Equation (17)

$$\begin{aligned} \rho C_p \left(\frac{\partial T}{\partial t} + \mathbf{u} \cdot \nabla T \right) - \nabla \cdot (k \nabla T) = \rho (& \tau_{xx} \frac{\partial u}{\partial x} + \tau_{xy} \frac{\partial u}{\partial y} + \tau_{zx} \frac{\partial u}{\partial z} + \\ & + \tau_{xy} \frac{\partial v}{\partial x} + \tau_{yy} \frac{\partial v}{\partial y} + \tau_{zy} \frac{\partial v}{\partial z} + \tau_{xz} \frac{\partial w}{\partial x} + \tau_{yz} \frac{\partial w}{\partial y} + \tau_{zz} \frac{\partial w}{\partial z}) \end{aligned} \tag{17}$$

where C_p is the specific heat capacity, k the thermal conductivity, and T is the temperature. Both C_p and k , vary with temperature, but are considered constant in the injection filling stage [27].

The right-hand-side terms of Equation (17) that are being multiplied by density (ρ), represent the viscous heat dissipation (the stress terms are the usual Cartesian stress components).

2.4. Free surface

The equations presented until now are for confined flows, and only one phase. In this work we are interested in modelling free surface flows, and therefore, the previous equations need to be updated or upgraded.

The volume of fluid (VOF) method was proposed by Hirt and Nicholas [28]. In this method a species transport equation is used to determine the relative volume fraction of the two phases, or phase fraction α (for example of air and molten polymer) in each computational cell. The nature of the VOF method dictates that an interface between the two phases is not clearly calculated, instead it appears as a phase fraction field.

Since the phase fraction can have any value between zero and one, the interface is never clearly defined, but occupies a volume around the region where a free surface should exist [1] as we can see in the Figure 2.3.

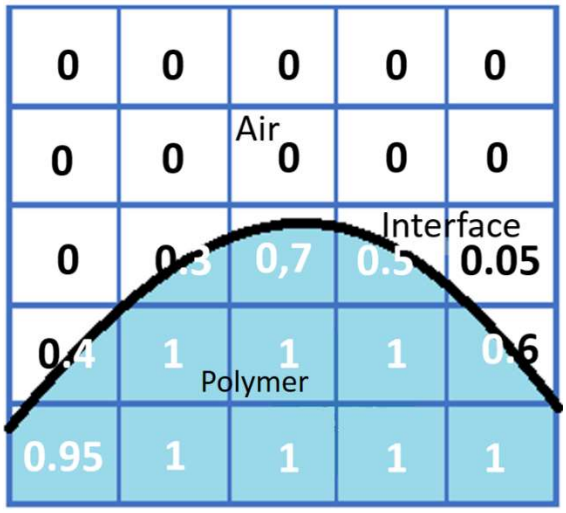

Figure 2.3: Example of free surface modelling using the VOF method.

Hence, physical properties can be calculated as weighted averages, considering α as the fluid phase fraction,

$$\rho = \rho_p \alpha + \rho_a (1-\alpha), \tag{18}$$

$$\eta = \eta_p \alpha + \eta_a (1-\alpha), \tag{19}$$

where ρ_p and ρ_a are the density of polymer and air, respectively, since we want to study the filling process in the presence of air and polymer. η_p and η_a represent the viscosity of the polymer and air, respectively.

The time dependence of α is governed by the following transport equation,

$$\frac{\partial \alpha}{\partial t} + u\frac{\partial \alpha}{\partial x} + v\frac{\partial \alpha}{\partial y} + w\frac{\partial \alpha}{\partial z} = 0. \tag{20}$$

Considering only the polymer phase fraction, Equation (20) can be expressed as,

$$\frac{\partial \alpha}{\partial t} + \nabla \cdot (\mathbf{u}_p \alpha) = 0 \tag{21}$$

and considering only the air phase fraction,

$$\frac{\partial(1-\alpha)}{\partial t}+\nabla\cdot[\mathbf{u}_a(1-\alpha)]=0 \tag{22}$$

To overcome the difficulty in obtaining a sharp interface, a new formulation for the evolution of the interface was proposed, introducing an extra term in the phase fraction function – the **artificial compression term**.

$$\frac{\partial\alpha}{\partial t}+\nabla\cdot(\mathbf{u}\alpha)+\underbrace{\nabla\cdot[\mathbf{u}_r\alpha(1-\alpha)]}_{artificial\ compression\ term}=0 \tag{23}$$

where $\mathbf{u}_r=\mathbf{u}_p-\mathbf{u}_a$ is the vector of relative velocity between the two fluids, also called as compression velocity, $\mathbf{u}$ is the mean velocity, calculated by a weighted average of the velocity between the two fluids [4],

$$\mathbf{u}=\alpha\mathbf{u}_p+(1-\alpha)\mathbf{u}_a \tag{24}$$

The momentum equation must be modified in order to consider the surface tension (σ). The surface tension at the liquid-gas interface generates a pressure gradient which results in a force,

$$f_\sigma=\sigma\,\mathrm{k}\nabla\alpha \tag{25}$$

where k is the mean curvature of free surface.

With Equations (3) and (4), we can decompose $\nabla\cdot\tau$ into a more appropriate form [4],

$$\nabla\cdot\tau=\nabla\cdot\eta\left[\nabla\mathbf{u}+(\nabla\mathbf{u})^T\right]=\nabla\cdot(\eta\nabla\mathbf{u})+(\nabla\mathbf{u})\cdot\nabla\eta \tag{26}$$

It is common to define a modified pressure (p_d) as,

$$p_d=p-\rho g\cdot x \tag{27}$$

due the existence of different phases [4].

With Eqs. (25), (26) and (27) we can rearrange momentum equation, Eq. (2), as,

$$\frac{\rho\mathbf{u}}{\partial t}+\nabla\cdot(\rho\mathbf{u}\mathbf{u})-\nabla\cdot(\eta\nabla\mathbf{u})-(\nabla\mathbf{u})\cdot\nabla\eta=-\nabla p_d-g\cdot x\nabla\rho+\sigma\mathrm{k}\nabla\alpha \tag{28}$$

3. Numerical method

The objective of this chapter is to demonstrate, using simple examples, how the numerical solution of such differential equations can be obtained.

We briefly describe some few steps that lead to the numerical solution, which is no more than obtaining the value of the dependent variable at specific points (the centre of the control volume) [29].

1. The first step consists on the definition of our physical domain, boundaries, etc;
2. Then, we have to describe the physical phenomenon using a set of differential equations;
3. The next step is to divide the domain into small control volumes (this stage is also known as meshing);
4. Next, we can discretize the governing equations (transform continuous operators into discrete ones). OpenFOAM® uses a finite volume method based methodology for the discretization procedure. In this method all the equations are integrated over the control volume (CV). At the end of this step we obtain a system of algebraic equations;
5. In order to obtain our numerical solution we need to solve our algebraic system of equations.

3.1. Finite Volume Method (FVM)

The FVM discrete process is the method most common used on computational fluid dynamics. To demonstrate how this method works we will take as an example, the scalar transport equation for a dependent variable ϕ.

$$\underbrace{\frac{\partial(\rho\phi)}{\partial t}}_{transient\ \ term} + \underbrace{\nabla\cdot(\rho\phi\mathbf{u})}_{convective\ \ term} = \underbrace{\nabla\cdot(\Gamma\nabla\phi)}_{diffusion\ \ term} + \underbrace{S_\phi}_{source\ \ term} \tag{29}$$

As mentioned before, in this method the studying object (the rectangular channel shown in Figure 3.1) is divided into several control volumes.

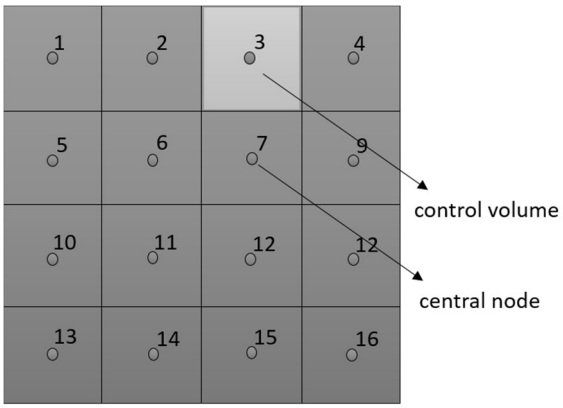

Figure 3.1: Division of a two-dimensional physical domain in 16 CVs.

The governing equations are then integrated over space (and time, for transient problems), allowing the conservation of properties for each cell [30],

$$\int_{CV} \frac{\partial(\rho\phi)}{\partial t} dV + \int_{CV} \nabla \cdot (\rho\phi \mathbf{u}) dV = \int_{CV} \nabla \cdot (\Gamma \nabla \phi) dV + \int_{CV} S_\phi dV \qquad (30)$$

Considering a steady problem, the transient term of Equation (30) disappears. If we consider an unsteady problem, Equation (30) needs to be integrated again, but over time.

In order to simplify the convective and diffusive terms, we use Gauss theorem, represented in the Figure 3.2, which transform the volume integrals into surface integrals. Basically, the theorem says that is possible to observe what happens to the variable in the surface instead of studying it in the whole volume.

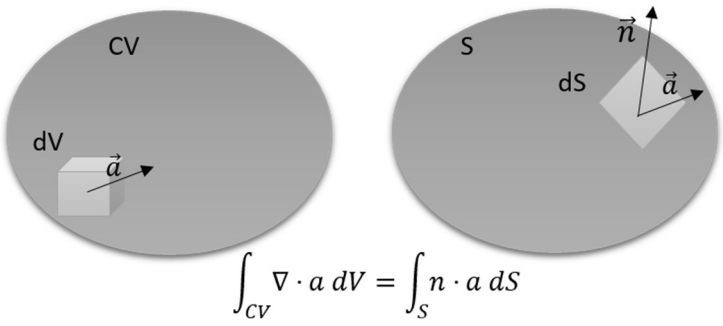

Figure 3.2: Gauss theorem scheme.

Applying Gauss theorem, Equation (30) (for a steady problem) can be rewritten as,

$$\int_S n \cdot (\rho\phi\mathbf{u})dS = \int_S n \cdot (\Gamma\nabla\phi)dS + \int_{CV} S_\phi dV \tag{31}$$

For each node we will have an equation similar to the one show before, and, these equations need to be discretized, resulting in a system of algebraic equations. The approximation of the integral is simple, and we can assume we know the values of the variables under the integral at the centre of the control volume faces. For example, for the control volume 1, we would have $\int_S n \cdot (\rho\phi\mathbf{u})dS \approx (\rho\phi u)_e - (\rho\phi u)_w + (\rho\phi v)_n - (\rho\phi v)_s$, and for the diffusive term $\int_S n \cdot (\Gamma\nabla\phi)dS \approx (\Gamma\nabla\phi)_e - (\Gamma\nabla\phi)_w + (\Gamma\nabla\phi)_n - (\Gamma\nabla\phi)_s$, where *e*, *w*, *n*, *s* stand for the east, west, north and south faces of the control volume. This means that we need to approximate $\mathbf{u}, \phi$ and $\nabla\phi$ (we assume Γ is constant) at the faces of the CV, since all variables are placed at the centre of the control volume.

For instance, considering that the area of each face (A) is the same for all CVs, as well as the distance between the nodes (δ), using Equation (31), the equation for ϕ at node 6, using the approximation given by the central difference method (CDS), is given by (we assume the velocity field is known):

$$\begin{aligned} &A\rho\left(\frac{\phi_7+\phi_6}{2}\right)u_e - A\rho\left(\frac{\phi_6+\phi_5}{2}\right)u_w + A\rho\left(\frac{\phi_2+\phi_6}{2}\right)v_n - A\rho\left(\frac{\phi_6+\phi_{11}}{2}\right)v_s = \\ &= A\Gamma\frac{(\phi_7-\phi_6)}{\delta} - A\Gamma\frac{(\phi_6-\phi_5)}{\delta} + A\Gamma\frac{(\phi_2-\phi_6)}{\delta} - A\Gamma\frac{(\phi_6-\phi_{11})}{\delta} + S\Delta V \end{aligned} \tag{32}$$

where, u and v are the *x* and *y* velocity components, respectively. If we consider that the source term does not depend of the variable in the node 6, $S\Delta V = S_u$, equation (32) can be simplified as,

$$\begin{aligned} &\underbrace{4\frac{A\Gamma}{\delta}}_{a_P}\phi_6 = \underbrace{\left(\frac{A\Gamma}{\delta} - \frac{A\rho u}{2}\right)}_{a_E}\phi_7 + \underbrace{\left(\frac{A\Gamma}{\delta} - \frac{A\rho u}{2}\right)}_{a_W}\phi_5 + \underbrace{\left(\frac{A\Gamma}{\delta} - \frac{A\rho u}{2}\right)}_{a_N}\phi_2 + \\ &+ \underbrace{\left(\frac{A\Gamma}{\delta} - \frac{A\rho u}{2}\right)}_{a_S}\phi_{11} + S_u \end{aligned} \tag{33}$$

Note that Equation (33) is written in its canonical form:

$$a_p\phi_p = a_E\phi_E + a_W\phi_W + a_N\phi_N + a_S\phi_S + S_u \tag{34}$$

where the indexes P, E, W, N, S are the discretized node, and the neighbours at East, West, North and South, respectively.

Equation (34) was obtained using approximation for the derivatives and velocity at the cell faces. In the OpenFOAM® computational library we can choose the discretization schemes to be used for each term of Equation (30) (file fvSchemes).

3.2. Numerical solution of the system of equations

As we said before, after discretization of Equation (31) we obtain a system of algebraic equations as illustrated in Figure 3.3.

$$\begin{bmatrix}
a_P & -a_E & 0 & 0 & -a_S & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-a_W & a_P & -a_E & 0 & 0 & -a_S & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -a_W & a_P & -a_E & 0 & 0 & -a_S & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -a_W & a_P & 0 & 0 & 0 & -a_S & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-a_N & 0 & 0 & 0 & a_P & -a_E & 0 & 0 & -a_S & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -a_N & 0 & 0 & -a_W & a_P & -a_E & 0 & 0 & -a_S & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -a_N & 0 & 0 & -a_W & a_P & -a_E & 0 & 0 & -a_S & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -a_N & 0 & 0 & -a_W & a_P & 0 & 0 & 0 & -a_S & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -a_N & 0 & 0 & 0 & a_P & -a_E & 0 & 0 & -a_S & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -a_N & 0 & 0 & -a_W & a_P & -a_E & 0 & 0 & -a_S & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -a_N & 0 & 0 & -a_W & a_P & -a_E & 0 & 0 & -a_S & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -a_N & 0 & 0 & -a_W & a_P & 0 & 0 & 0 & -a_S \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -a_N & 0 & 0 & 0 & a_P & -a_E & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -a_N & 0 & 0 & -a_W & a_P & -a_E & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -a_N & 0 & 0 & -a_W & a_P & -a_E \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -a_N & 0 & 0 & -a_W & a_P
\end{bmatrix}
\begin{bmatrix} \phi_1 \\ \phi_2 \\ \phi_3 \\ \phi_4 \\ \phi_5 \\ \phi_6 \\ \phi_7 \\ \phi_8 \\ \phi_9 \\ \phi_{10} \\ \phi_{11} \\ \phi_{12} \\ \phi_{13} \\ \phi_{14} \\ \phi_{15} \\ \phi_{16} \end{bmatrix}
=
\begin{bmatrix} S_u \\ S_u \\ S_u \\ S_u \\ S_u \\ S_u \\ S_u \\ S_u \\ S_u \\ S_u \\ S_u \\ S_u \\ S_u \\ S_u \\ S_u \\ S_u \end{bmatrix}$$

Figure 3.3: System of algebraic equations for the two-dimensional physical problem of 16 CVs.

The system of algebraic equations can be solved by calculating the inverse matrix (obtaining in this way the exact solution) or using iterative methods [31]. Due to the size of the system of algebraic equations, iterative methods are preferred, since they perform faster.

Observing momentum equation (Equation (2) in 2.1) we can discretize it for each one of the three velocity components, but we cannot solve it since we

do not know how to calculate the pressure field. There is no transport equation for pressure. In the same equation we want to obtain pressure and velocity fields, so we have a coupling problem.

A careful watch of the momentum equation reveals the existence of non-linear terms $\left(\frac{\partial}{\partial x}(\rho uu) = \frac{\partial}{\partial x}(\rho u^2)\right)$.

To overcome these problems we will use SIMPLE (Semi-Implicit Method for Pressure-Linked Equations) algorithm, as described in the next section.

3.2.1. SIMPLE algorithm

The SIMPLE algorithm was presented by Patankar and Spalding in 1972 [30], and essentially it is a procedure for the calculation of pressure and velocity, by guessing and correcting pressure and velocity fields.

For ease of understanding we are going to explain how this algorithm works in a staggered mesh where the variables are not all stored at the centre of the CV, as shown in the Figure 3.4. In this we already have velocity and the CV faces.

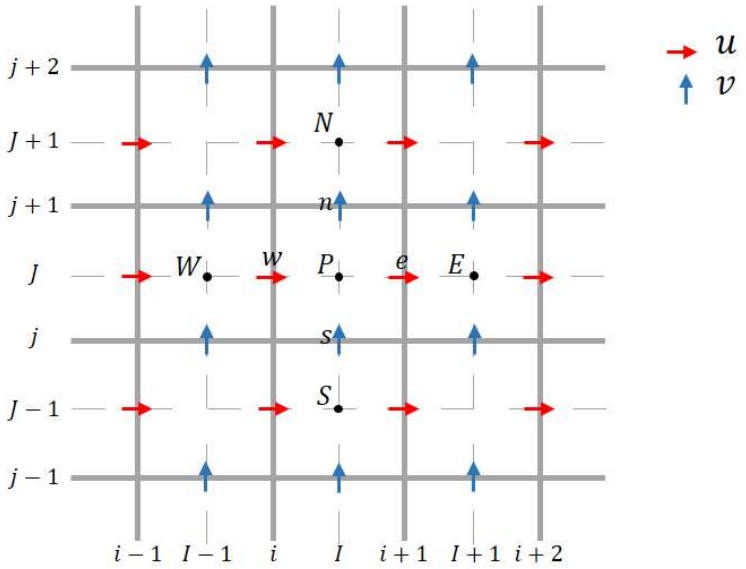

Figure 3.4: Representation of a staggered grid and its coordinates.

The discretized momentum equations for this case is given by,

$$
\begin{aligned}
a_{i,J}u^{*}_{i,J} &= \sum_{nb} a_{nb}u^{*}_{nb} - (p^{*}_{I-1,J} - p^{*}_{I,J})A_{i,J} + b_{i,J} \\
a_{I,j}v^{*}_{I,j} &= \sum_{nb} a_{nb}v^{*}_{nb} - (p^{*}_{I-1,J} - p^{*}_{I,J})A_{I,j} + b_{I,j}
\end{aligned}
\tag{35}
$$

for the two velocity components, *u* and *v*, respectively.

Note that the equations above are written in the canonical form (as shown in Equation (34)). u^{*} and v^{*} are the velocity components calculated from the initial guess pressure, p^{*}, b is the source term and the index nb represents the cell neighbours.

The methodology defines a new pressure (p') and velocity (u',v') correction field to achieve the correct fields (p,u,v),

$$
\begin{aligned}
p &= p^{*} + p' \\
u &= u^{*} + u' \\
v &= v^{*} + v'
\end{aligned}
\tag{36}
$$

The correction velocity field (u',v') is calculated from Equation (35), eliminating the sum term, being this suppression, the main simplification on the SIMPLE algorithm,

$$
\begin{aligned}
a_{i,J}u'_{i,J} &= (p'_{I-1,J} - p'_{I,J})A_{i,J} + b_{i,J} \\
a_{I,j}v'_{I,j} &= (p'_{I-1,J} - p'_{I,J})A_{I,j} + b_{I,j}
\end{aligned}
\tag{37}
$$

The whole SIMPLE algorithm can be seen in the flowchart of the Figure 3.5.

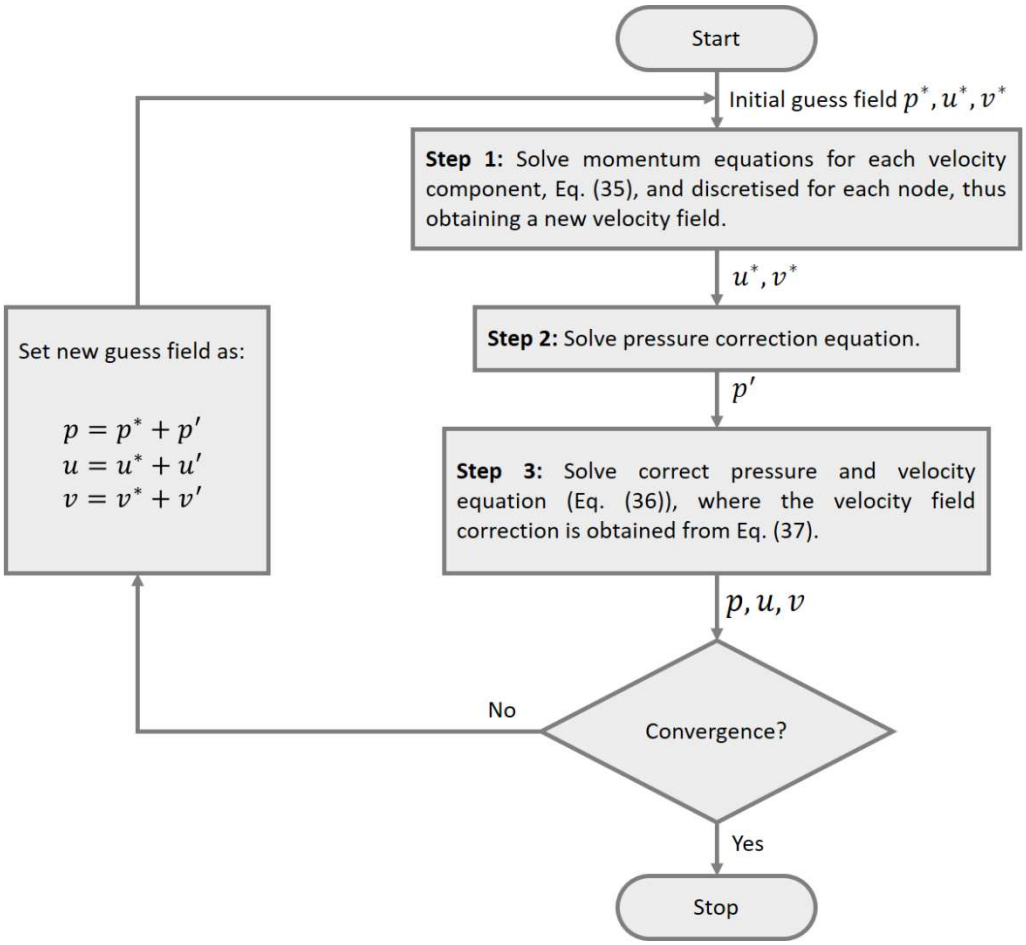

Figure 3.5: Flowchart of the SIMPLE algorithm.

4. The interFoam solver

The OpenFOAM® solver most suitable to solve the equations governing the injection moulding filling stage is called interFoam. It is an isothermal, multiphase solver for two incompressible and immiscible fluids, that uses the VOF interface capturing approach [1].

The filling stage of the injection moulding process is controlled by velocity and can be assumed as incompressible, since there is no compaction of the material in this phase. Initially we have air inside the cavity of the mould and gradually we fill it with molten polymer.

When setting up a case in OpenFOAM® we need the folders 0, constant and system. For the InterFoam solver the typical files (and folders) used are shown in Figure 4.1.

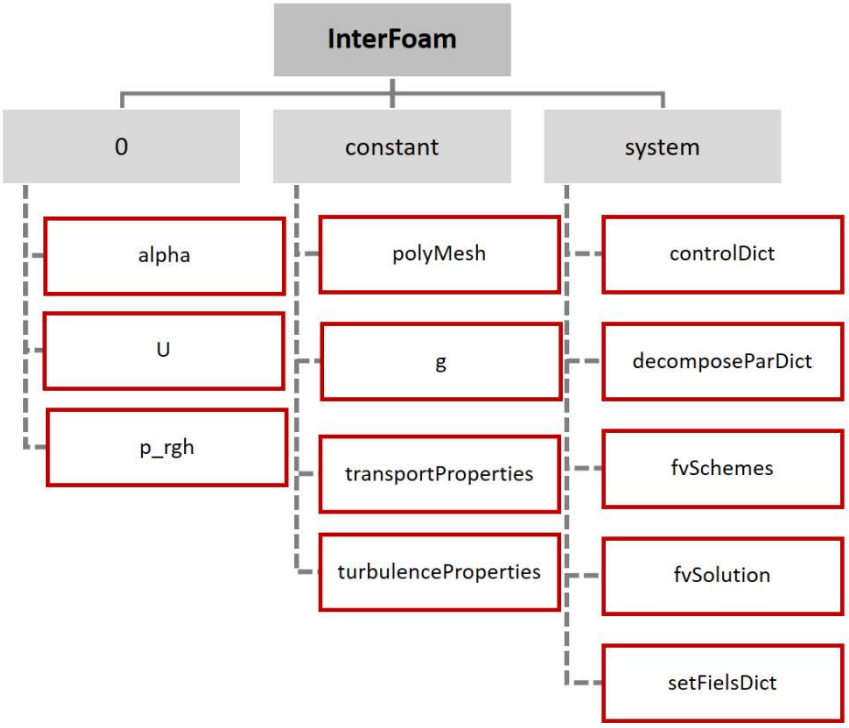

Figure 4.1: Directories and files for running a case using interFoam.

The purpose of the main files is:

- alpha: set up the interface boundary conditions;
- U: set up the velocity boundary conditions;
- p_rgh: set up the dynamic pressure boundary conditions;
- polyMesh: is actually a folder, within this folder there is a file called blockMeshDict where we can create the geometry;

– transportProperties: in this file we discriminate the properties of the two fluids under study (viscosity, density, surface tension, etc). It should be remarked that OpenFOAM® uses a cinematic viscosity, ν (dynamic viscosity (η) divided by the density).

– controlDict: in this file we control time settings like the maximum Courant number, time step, etc. The modifiable time step (δt) of the simulation is calculated according to the smaller dimension of the cell, δx, the magnitude of the velocity, $|U|$ and the Courant number (Co),

$$\delta t = \frac{Co \delta x}{|U|} \tag{38}$$

– fvSchemes: in this file we define the discretization schemes. The discretization schemes used in this work were central differences for the diffusive terms and (linear) upwind for the convective terms. For the time derivative, a simple Euler method was used;

– fvSolutions: we specify the methods for solving the system of algebraic equations. In this work, the SIMPLE method was used to couple velocity and pressure fields. For the numerical solution of the velocity we used the “smoothsolver”; for the pressure, the preconditioned conjugate gradient (PCG) method was used.

– setFieldsDict: define the initial spatial distribution of the phases involved in the simulation.

5. Air vents – a computational approach

In the injection moulding process, when the mould closes, some air remains inside. In many cases the air can escape through the parting line of the mould, but sometimes it cannot, and the part will not be completely filled, if the air does not get out. To overcome this problem, venting slits are necessary.

The venting gaps are of such small size that it becomes a problem to simulate the process, because, it is difficult to create such a refined mesh in the regions that model the air vents. In Figure 5.1 we can see a venting gap in a mould, and we can barely see its depth, because it is among 0.01 to 0.07 mm, depending on the used polymer. The other slit, described as vent relief, is used for letting the air out, after leaving the cavity through the venting gap.

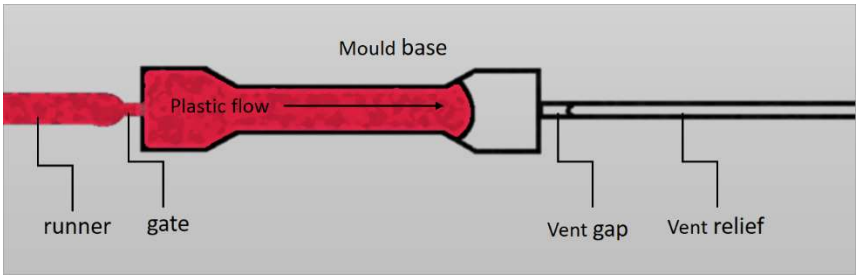

Figure 5.1: Scheme of a venting gap in a mould.

To use the real size air vents, we need to have a really refined mesh. The problem is that the mesh would become too heavy for the simulation to run smoothly. To overcome this problem we created a new boundary condition where the air can exit the mould but the material cannot (the code to implement this boundary condition is totally described in the Annex I).

Basically, in this new boundary condition we have the following:

- When the fraction of the material is higher than half at the last cell touching the venting gap, the velocity of the material at that boundary is set to zero (as happens in a wall), and a zero-gradient boundary condition is used for the pressure.

- If in the last cell touching the venting gap we have $\gamma < 0.5$, the pressure at that boundary is set to zero and a zero-gradient boundary condition is considered for the velocity, allowing the air to exit the mould (similar to an outlet boundary condition).

Another problem that requires our attention is where we should place this boundary condition in the mould. To overcome this problem and to

acknowledge the influence of the new boundary condition location, we did some two-dimensional simulations using four different locations for the air vents (showed in the next subsections).

5.1. Definition of the case study: geometry, meshes and data

The simulations performed to study the influence of the air exit region, were done using the geometry shown in Figure 5.2.

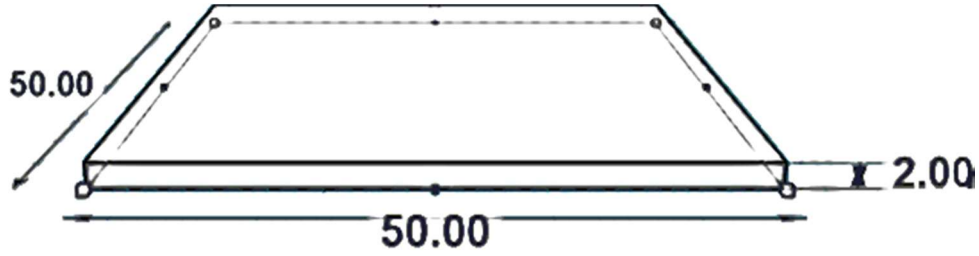

Figure 5.2: Geometry used to simulate the different locations of the new boundary condition. Dimensions: 50X50X2 [mm]

Besides the influence of the new boundary condition location, it is also important to be sure about the convergence of the numerical solutions. For that, we performed simulations using three different meshes, presented in Figure 5.3, where we also describe the number of divisions and the respective control volume dimensions.

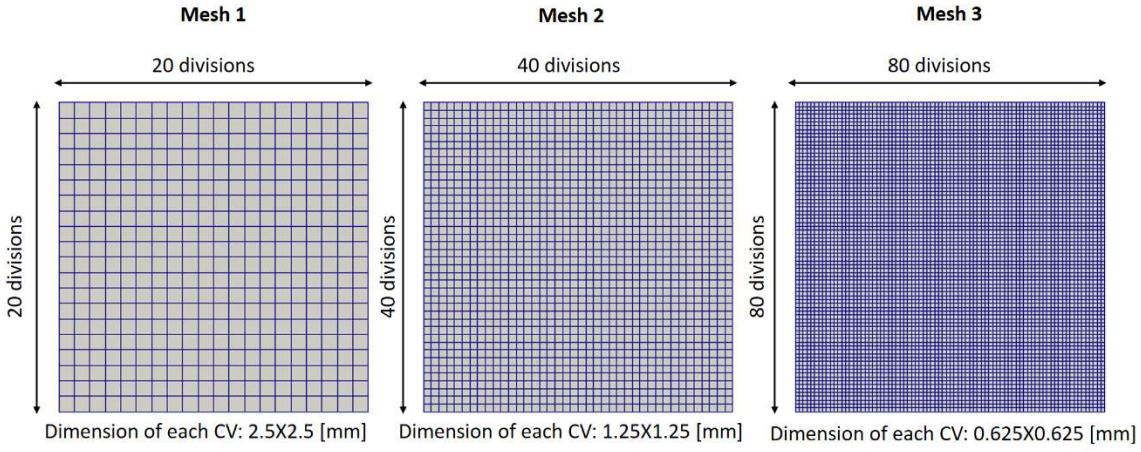

Figure 5.3: Meshes used to test the convergence of the method.

First, we studied the convergence of the solutions with two-dimensional simulations, considering a Newtonian fluid, and using the three meshes shown before. Then, we show the results obtained for a Newtonian fluid and for a

generalized Newtonian model (the Bird-Carreau model described in Equation (6)).

We, also performed three-dimensional simulations. For those simulations, a different number of cells were considered along thickness, and, we have concluded that using 10 cells, provides a reasonably accurate solution. Note that we need to take into account the simulation time when creating the meshes, therefore we simulated in parallel using a cluster (24 cores).

For the simulations with the Newtonian fluid, we have used the parameters shown in Table 1. Note, that the velocity corresponds to the fluid inlet velocity. This parameters were used by Chang and Yang [19] (they used a higher air viscosity to improve convergence of their method).

Throughout this work we are going to designate the polymer phase (note that in this case we are using a Newtonian model) as simply, fluid.

Table 1: Data used on Newtonian simulations.

	V [m^2/s]	**Density [kg/m^3]**	**Surface tension [kg/s^2]**	**Velocity [m/s]**
Fluid	0.2	1000	2.71E-2	6.33E-3
Air	8.12E-2	1.23		

For the simulations with inelastic fluid (parameters in Table 2) we have considered the same initial velocity and density of material and air.

Table 2: Bird-Carreau model parameters for the liquid phase.

	V_0 [m^2/s]	**V_∞ [m2/s]**	**m [s]**	**n**
Fluid	53	0	1.10	0.15

To obtain the parameters of the Bird-Carreau model, we fitted the model to a graph of viscosity vs. shear rate for polystyrene obtained from the study of Munstedt [32]. The graph we obtain after the adjustment is represented in Figure 5.4.

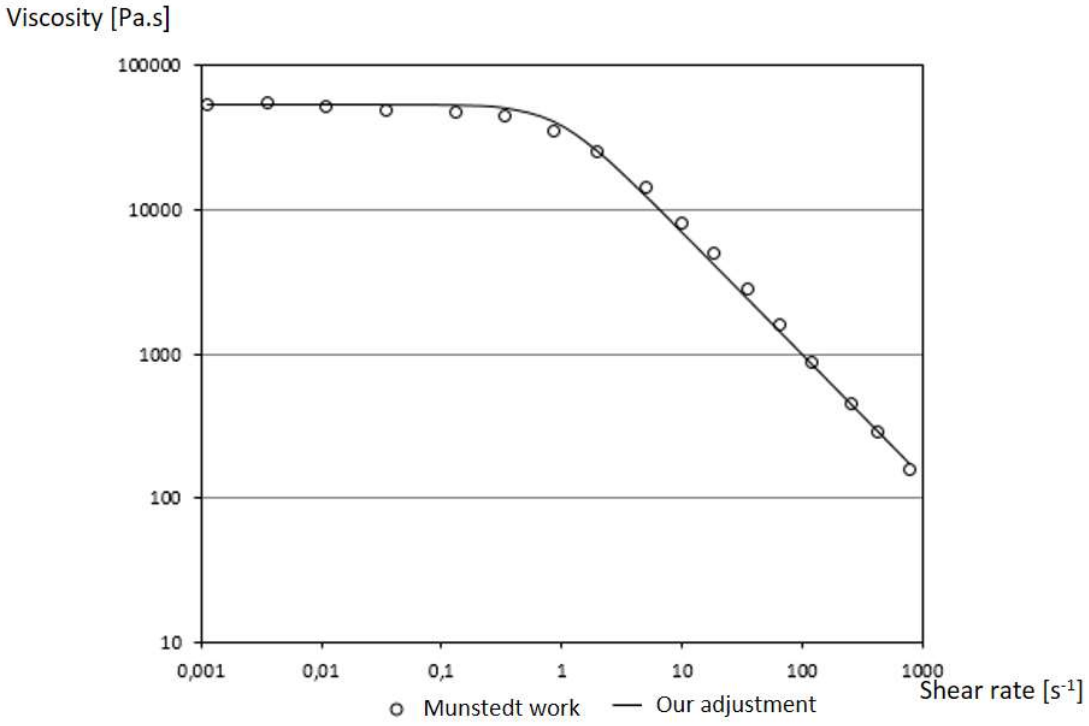

Figure 5.4: Fit of experimental data (symbols) using a Bird-Carreau model (full line).

We will show and comment the results obtained for the Newtonian and inelastic model, but, we cannot compare them since the characteristics of the two flows were obtained differently.

5.2. Simulation results without the air vents

In order to assess the original interFoam solver, we have tested first the case where the air and the material can both leave the part. In the scheme of the Figure 5.5 we see a schematic of the geometry and corresponding boundaries.

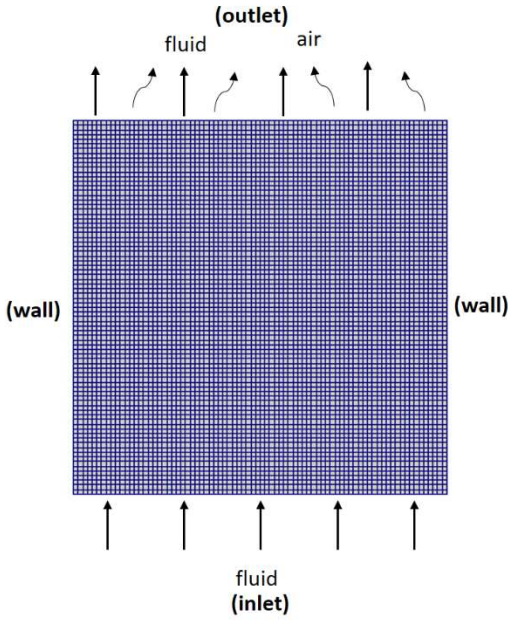

Figure 5.5: Geometry and boundary conditions used to test the interFoam solver.

5.2.1. Influence of the time step

First we did a study on the influence of the time step on the converged solution, in order to choose a time step that leads to an accurate solution, but caring in mind that a smaller time step increases the simulations time.

In Figure 5.6 we show the results obtained for three different time steps in the Mesh 1. In this mesh the time steps used seem to have a small influence on the final simulations results.

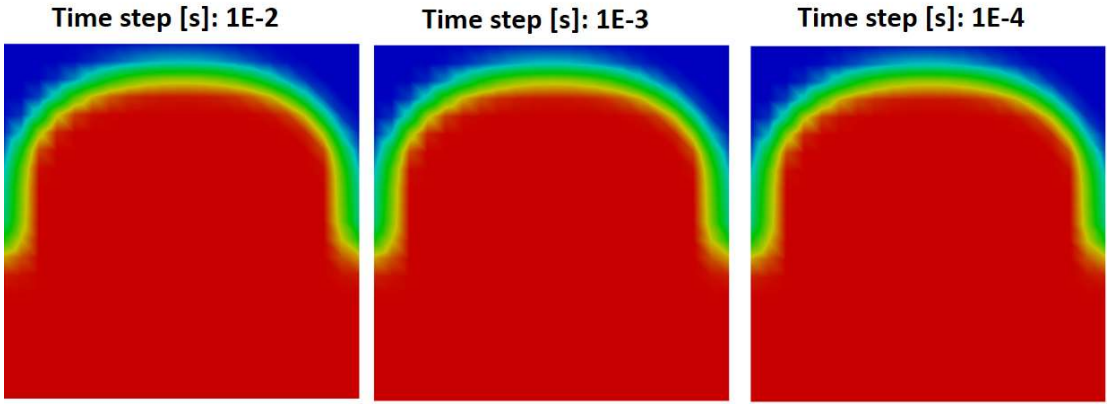

Figure 5.6: Mesh 1 results for different time steps (at t=5s).

From the Figure 5.7, we also see that the time step do not affect the simulations results, considering Mesh 2.

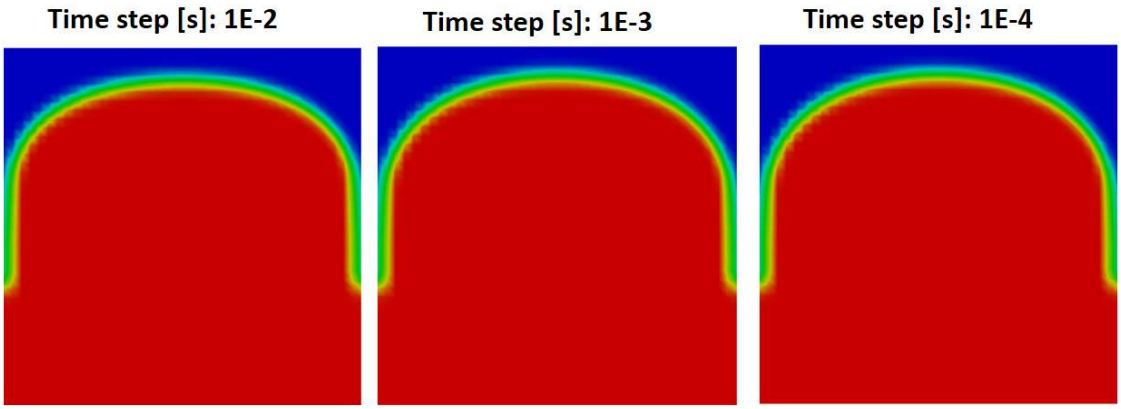

Figure 5.7: Mesh 2 results with different time step simulations, at 5s.

We only see differences between the different time steps in Mesh 3 (Figure 5.8). The time step of 0.01 [s] leads to a more plug flow front.

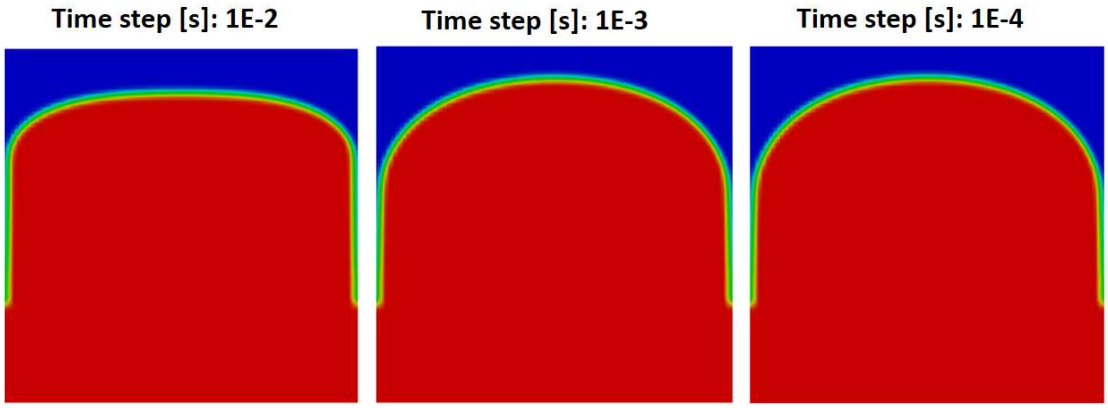

Figure 5.8: Mesh 3 results with different time step simulations, at 5s.

Almost all simulations during the next sections were done using a time step of 1E-3 seconds (with few exceptions).

5.2.2. Mesh convergence results

The fluid front, for the case where the air and the Newtonian liquid can go out at the *outlet* boundary (at 2, 4 and 6 seconds), is shown in Figure 5.9. It is possible to see that the Mesh 2 results are closer to the results obtained with Mesh 3, when compared to the results obtained with Mesh 1.

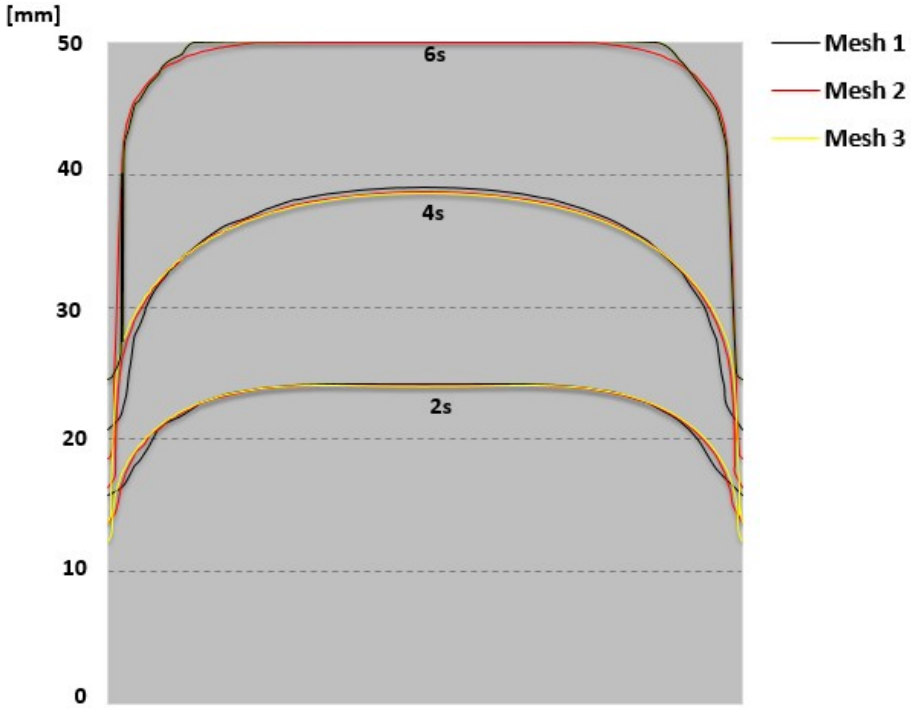

Figure 5.9: Fluid front at 2, 4 and 6 seconds.

In the simulations that follow, only Mesh 3 will be used.

We were expecting a profile of this type for the flow front, since there is no-slip along the wall, leading to a zero velocity of the material at the wall, and a maximum velocity occurring at the centre. Also, the fluid in the flow centre moves outwards, leading to the filling in the wall surfaces. This flow effect is known as fountain flow.

5.2.3. Bird-Carreau and Newtonian fluids

We can see in Figure 5.10 the flow front obtained for the Bird-Carreau and Newtonian fluids.

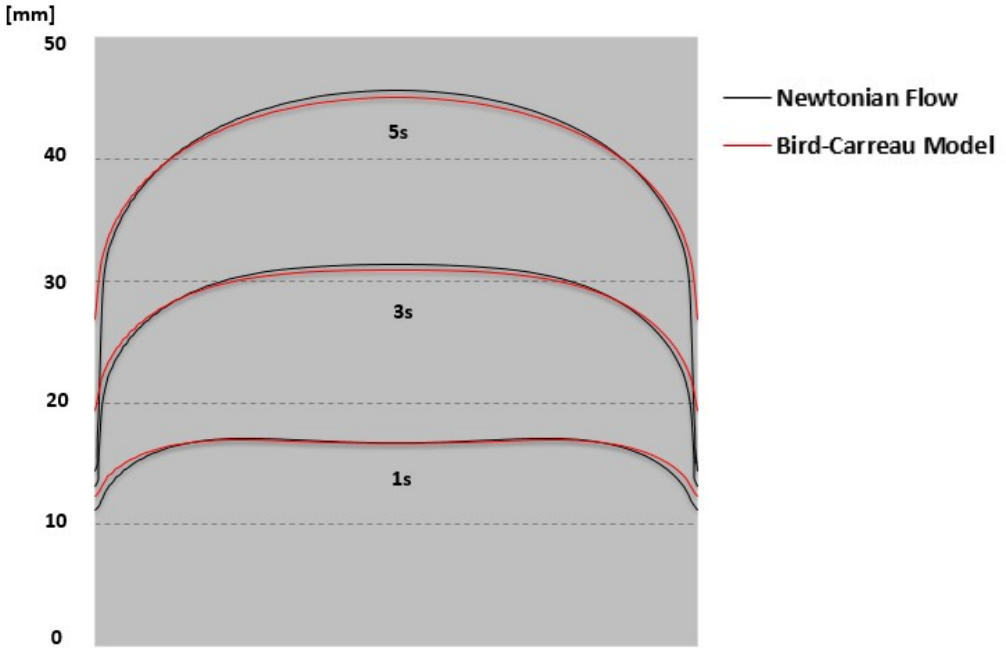

Figure 5.10: Flow front obtained for the Bird-Carreau and Newtonian fluids using the interFoam solver.

In Figure 5.11, we show the flow front for the three-dimensional simulations. It can be seen that a smooth velocity profile was obtained for both fluids.

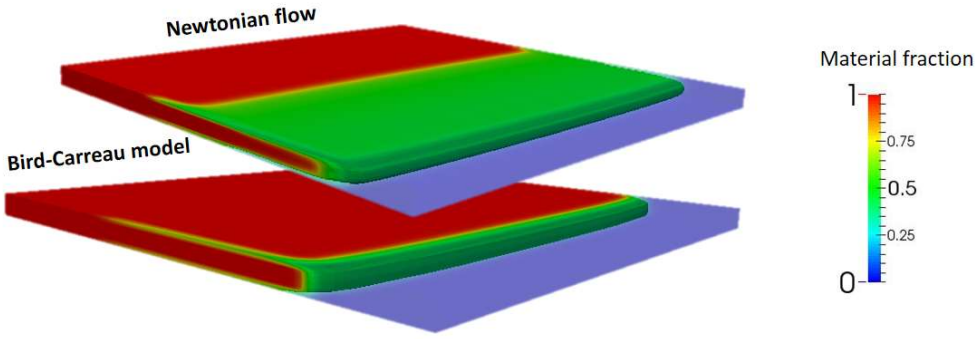

Figure 5.11: Flow front at 5s.

5.3. Modelling the air vents – first option

As said before we studied the influence of different venting zones. In this first option, the air escapes in the last filling zone, but this location is still seen as a wall by the fluid, as we can see in the Figure 5.12.

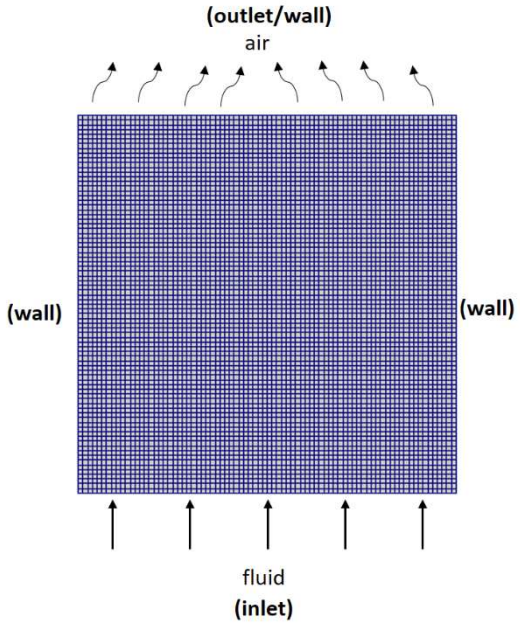

Figure 5.12: Representation of the boundary which lets the air out in the last filling zone.

5.3.1. Mesh convergence results

The simulation results for a Newtonian fluid, considering the three meshes presented before, are shown in Figure 5.13. Since, we are studying the influence of a new boundary condition that should model the air vents, it is pertinent to show the simulation results when the material reaches the final wall.

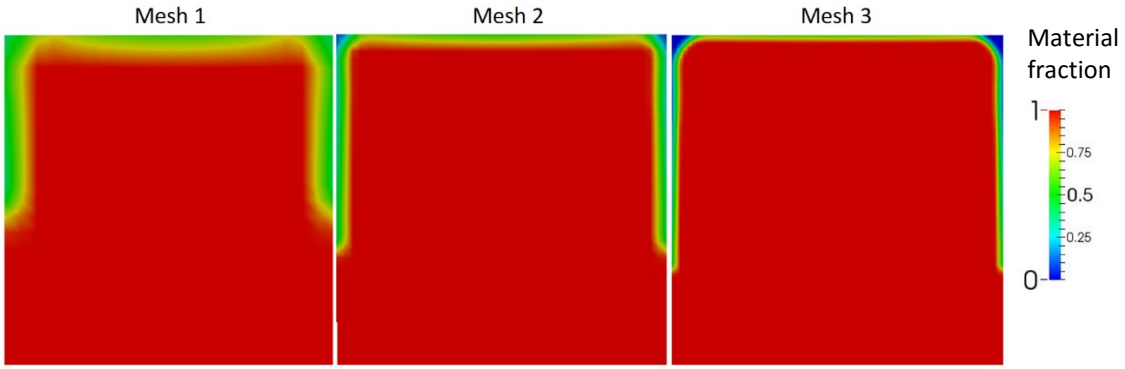

Figure 5.13: Simulation results when the material reaches the wall in three different meshes.

We can observe that the material cannot reach the corners of the final wall, and, this is more accentuated in Mesh 3. This happens because in the less refined meshes, the front line is more prone to diffusion, disguising the filling of

the box near the corners. According to the simulations results, we have obtained for meshes 1, 2 and 3 a filling of 95%, 96% and 98% respectively.

5.3.2. Bird-Carreau and Newtonian fluids

We can see in Figure 5.14 the flow front for both the Bird-Carreau and Newtonian fluids. Note that for the Bird-Carreau model we always obtain a more flatten interface at the centre of the channel. This is mainly due to the fact that in this region the shear rates are low, and therefore the fluid behaves like a solid (plug like profile).

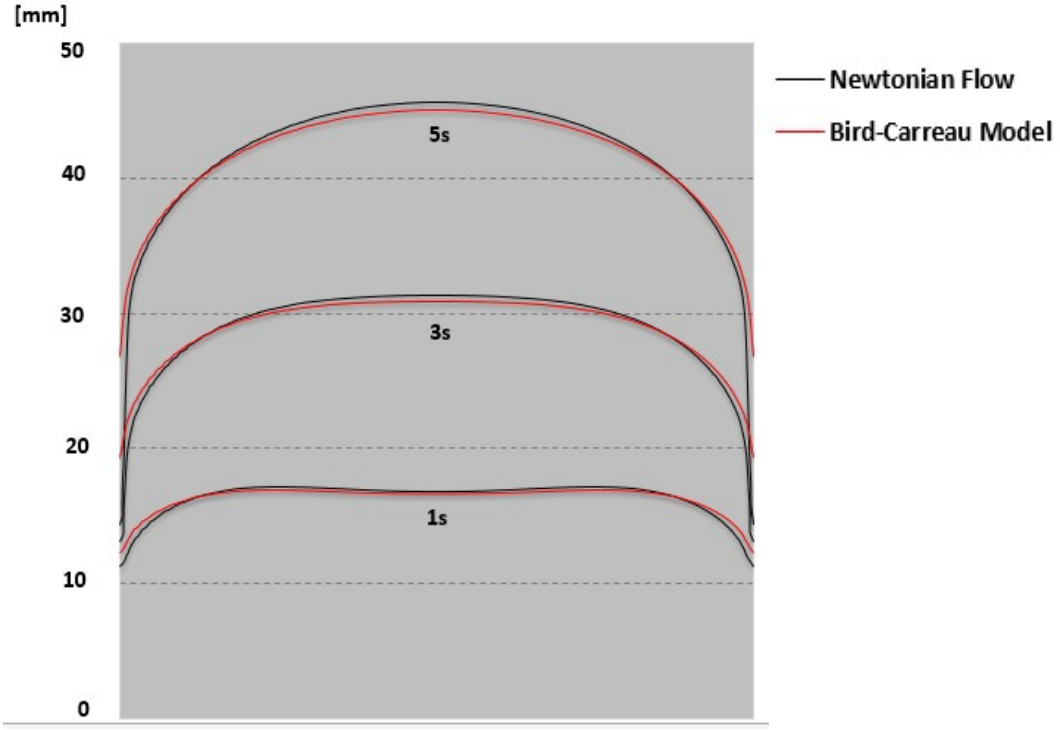

Figure 5.14: Flow front for the Bird-Carreau and Newtonian fluids – First option for the air exit boundary condition.

The three-dimensional results at t=5 s can be seen in the Figure 5.15. Again a smooth fluid front was obtained.

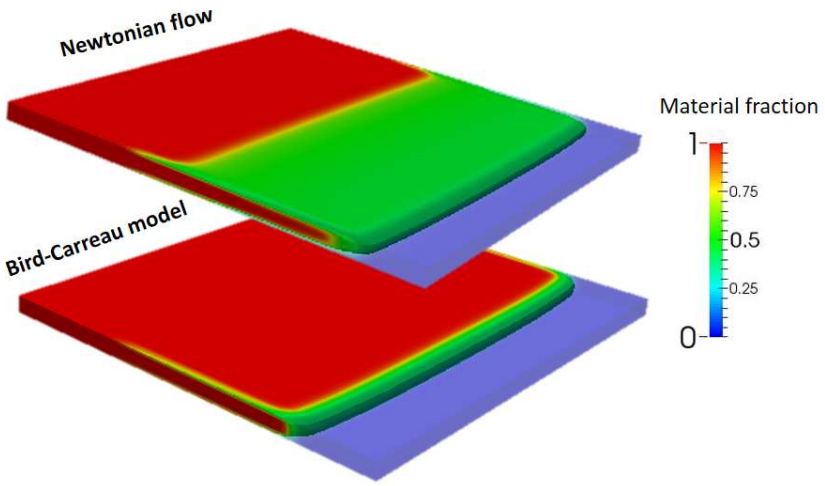

Figure 5.15: Flow front at 5s for the three-dimensional simulation.

5.4. Modelling the air vents – second Option

The second option we have considered to model the air vents is the possibility of the air to leave the channel through the bottom and lateral walls, as shown in Figure 5.16.

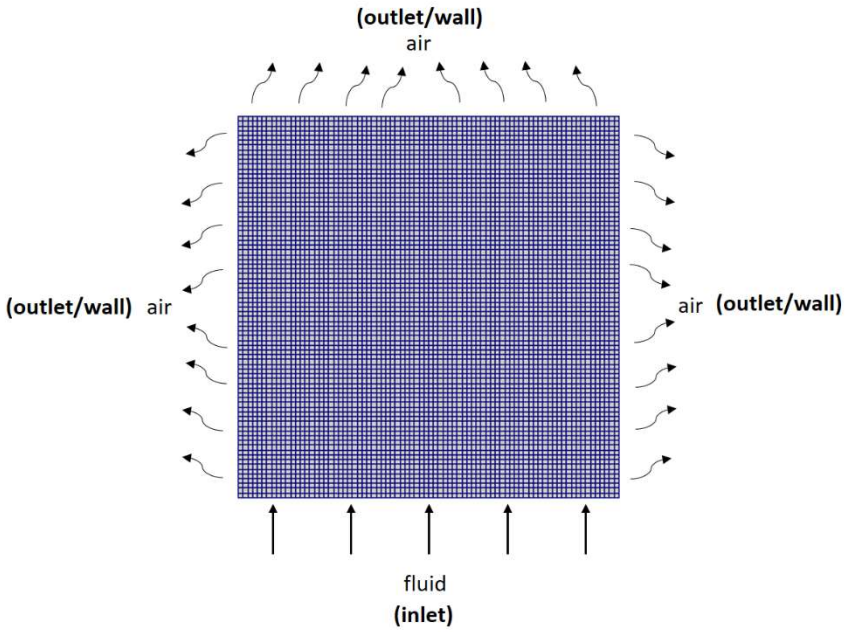

Figure 5.16: Representation of the boundaries at which the air can leave the mould.

5.4.1. Mesh converge results

The simulations results for the case where the air can exit the mould through the bottom and lateral walls (Figure 5.17), presents similarities with the case shown before.

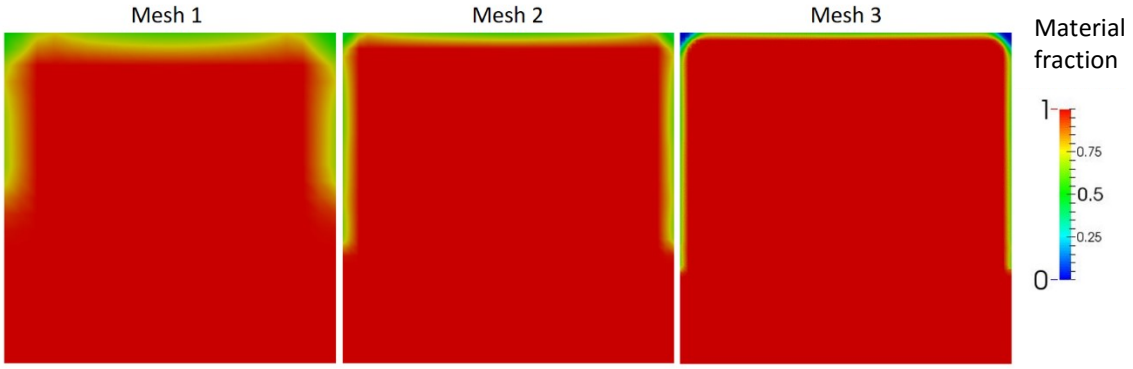

Figure 5.17: Simulation results in the three different meshes.

In this case the filling was about 97%, 98% and 99% for meshes 1, 2 and 3, respectively. This means a better filling percentage was obtained when compared to the *First Option*.

5.4.2. Bird-Carreau and Newtonian fluids

The flow front at three different simulation times using both the Bird-Carreau and Newtonian fluids are shown in Figure 5.18.

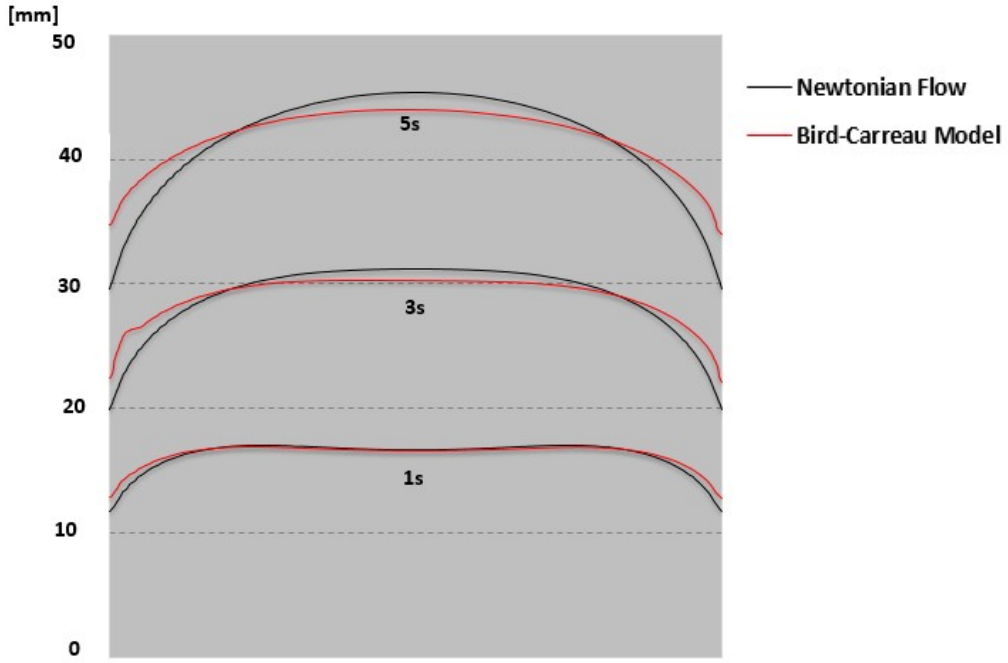

Figure 5.18: Flow front obtained for the Bird-Carreau and Newtonian fluids – second option.

The flow front for the Bird-Carreau model, is slightly different from the previous results. Therefore, this boundary condition is influencing the final results. We cannot forget that all the results are also influenced by the interpolation made by the visualization program (ParaView).

Figure 5.19 illustrates the flow front at 5s for the three-dimensional simulation. Until now, all three-dimensional simulations shown are similar.

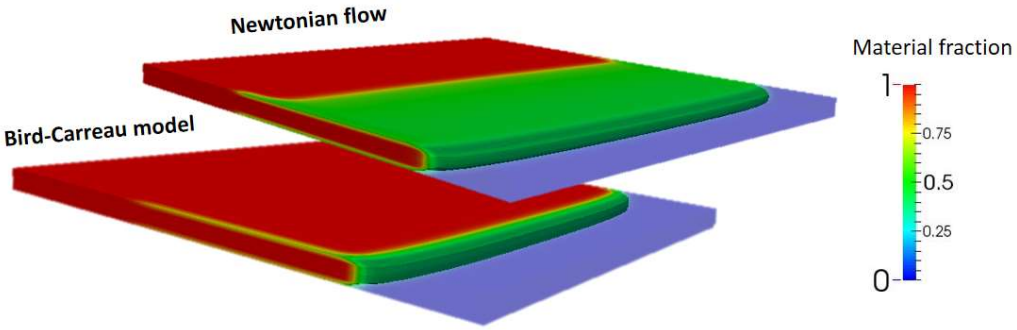

Figure 5.19: Three-dimensional flow front with the option two of the air exit, at 5s.

5.5. Modelling the air vents – third option

In this case, the wall parallel to the inlet wall is divided into three distinct zones and the air can escape from two of those three regions (as shown below).

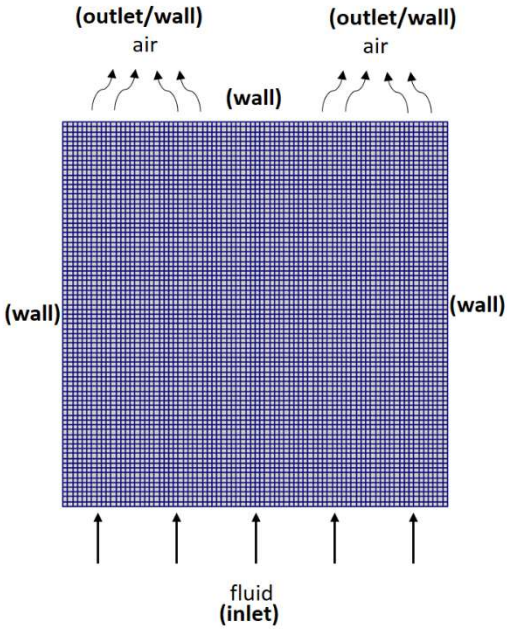

Figure 5.20: Third location of the air exit boundary conditions

5.5.1. Mesh convergence results

From the simulation results shown in Figure 5.21, we observe that the mould is not fully filled. This happens, because the fluid feels a restriction at the center of the channel (a wall) once the air reaches the wall, and it stays there, trapped.

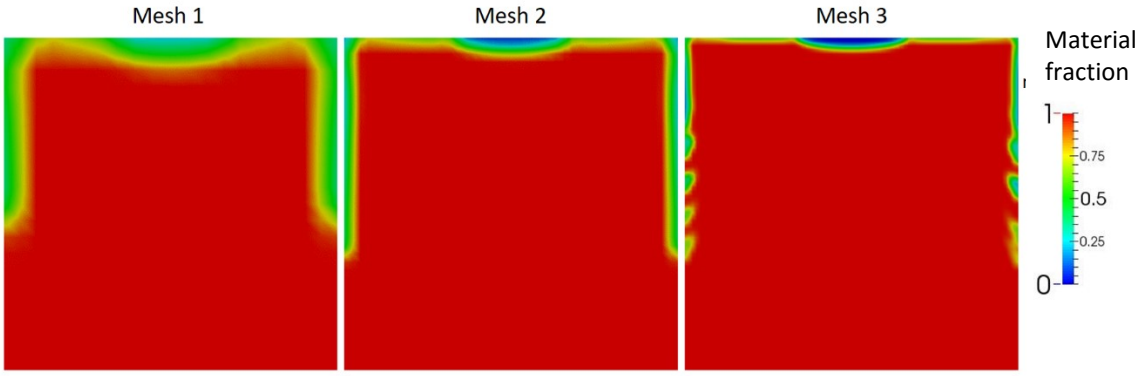

Figure 5.21: Simulation results obtained for three different meshes.

5.5.2. Bird-Carreau and Newtonian fluids

The flow front using the Bird-Carreau and Newtonian fluids is shown in Figure 5.22.

Some problems appeared at the flow front when using a Newtonian fluid. It looks like the flow front it is not symmetric, and we see oscillations near the wall (this problem is addressed in the next chapter).

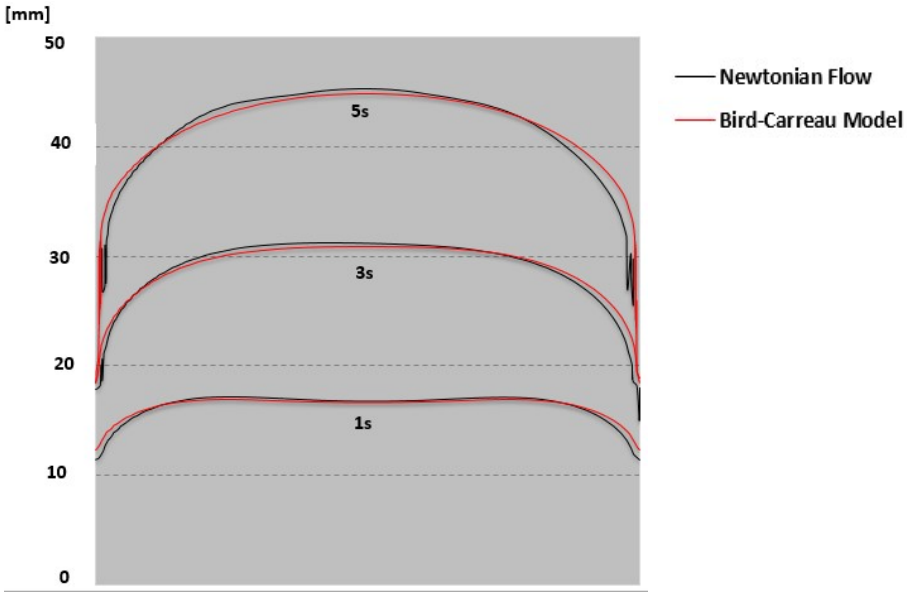

Figure 5.22: Flow front for the Bird-Carreau and Newtonian fluids – third option.

Figure 5.23 shows the flow front obtained for the three-dimensional simulation. We can also see an oscillation at the lateral wall for the Newtonian simulation

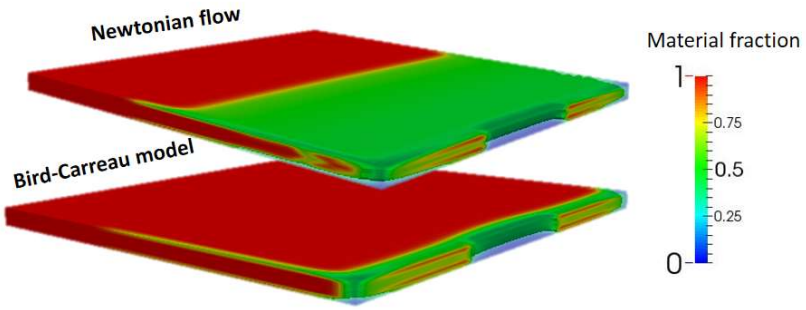

Figure 5.23: Three-dimensional simulation considering the third option for the location of the air exit boundary conditions.

5.6. Modelling the air vents – fourth option

The last attempt to study the influence of the *air exit* boundary condition location on the fluid front development is presented in the figure below.

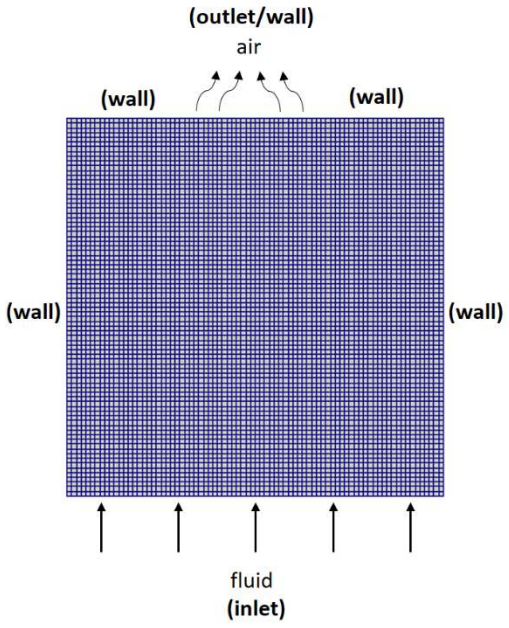

Figure 5.24: Representation of the fourth option for the *air exit* boundary condition.

5.6.1. Mesh convergence results

In this case, the fluid front reaches the air exit and closes the mould. Therefore, the air stays trapped inside the mould, resulting in a non-finished filling. We cannot forget that the fluid may only push the air, being unable to compress it.

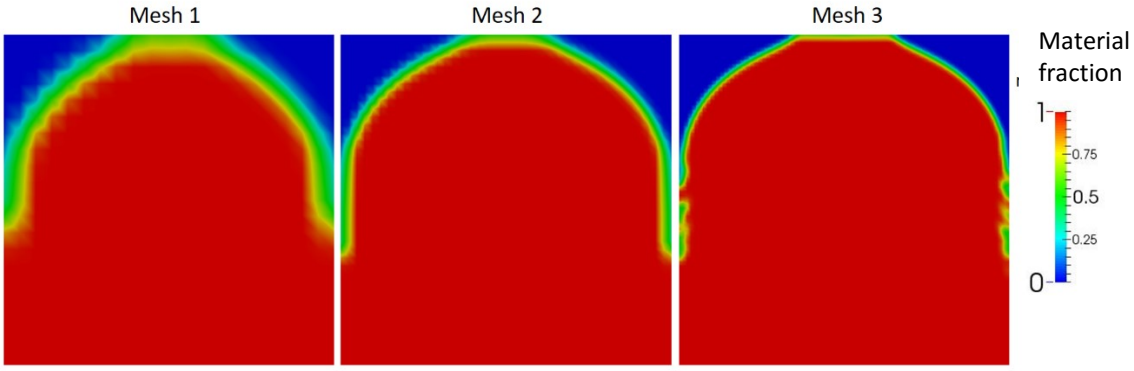

Figure 5.25: Simulation results using the fourth option for the air exit boundary condition.

5.6.2. Bird-Carreau and Newtonian fluids

The flow front using both Bird-Carreau and Newtonian fluids is shown in Figure 5.26.

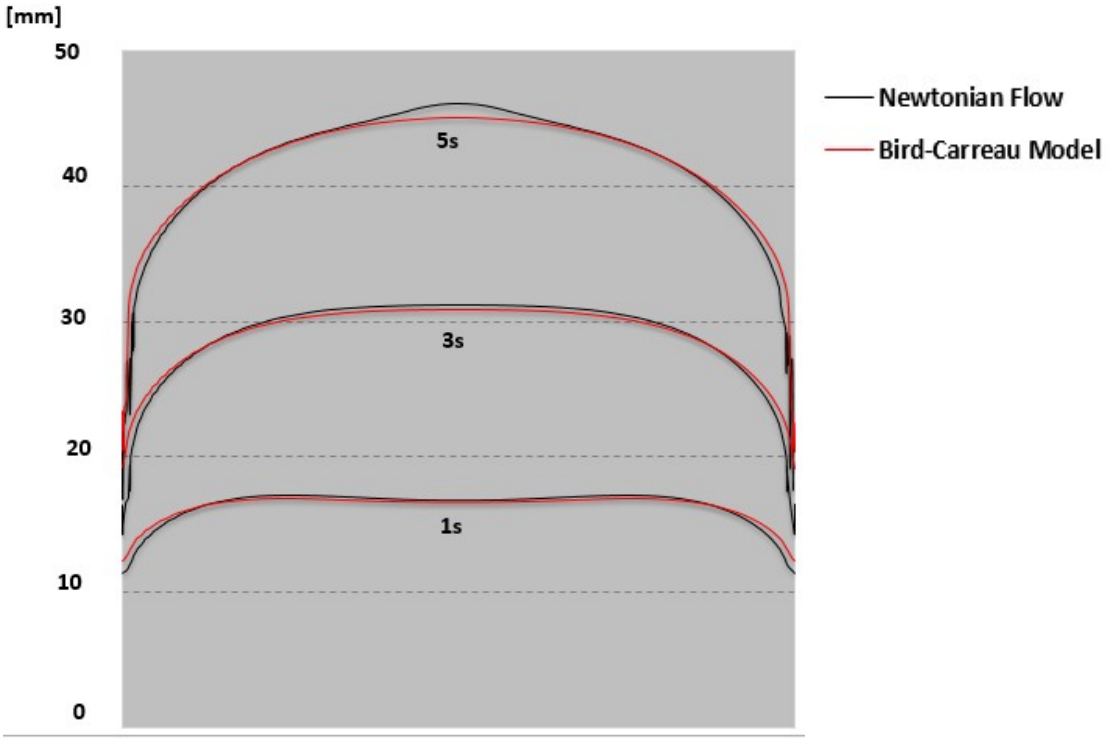

Figure 5.26: Flow front for the Bird-Carreau and Newtonian fluids - fouth option for the location of the air exit boundary condition.

The small bump obtained for the flow front of the Newtonian fluid at t=5 seconds, was not obtained for the Bird-Carreau fluid. Note that for both 2D and 3D simulations, some oscillations appear at the wall.

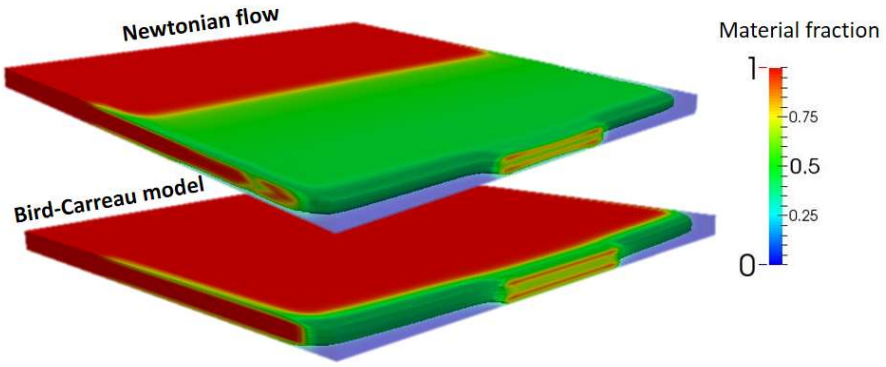

Figure 5.27: Three-dimensional simulation, when the material touches the wall, when using the fourth option for the air exit boundary condition location.

6. Oscillations of the interface near the wall

In Chapter 5 we have presented results using different location of the air exit/air vents boundary conditions, and, we saw that some simulations revealed oscillations of the interface near the wall, as lustrated in Figure 6.1.

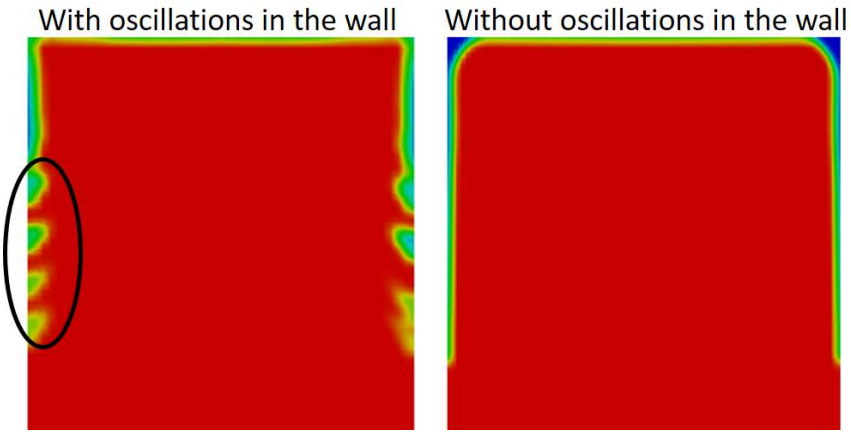

Figure 6.1: Representation of a simulation with and without oscillations near the wall.

We were expecting a progressive filling of the fluid near the wall as the fluid front goes backwards (fountain flow), not the appearance of those filling oscillations in the wall.

To discover the origin of the oscillations, we did a small study to understand if it is a problem of the solver or if it appears with some of the conditions we have used.

First, from observation, we noticed that those oscillations only appear in the most refined mesh, which it is normal, since we have a more refined and sharp solution. Our second move was to decrease the time step of the simulations of the most refined mesh to see if the solution would be the same or not. The results obtained are presented in Figure 6.2.

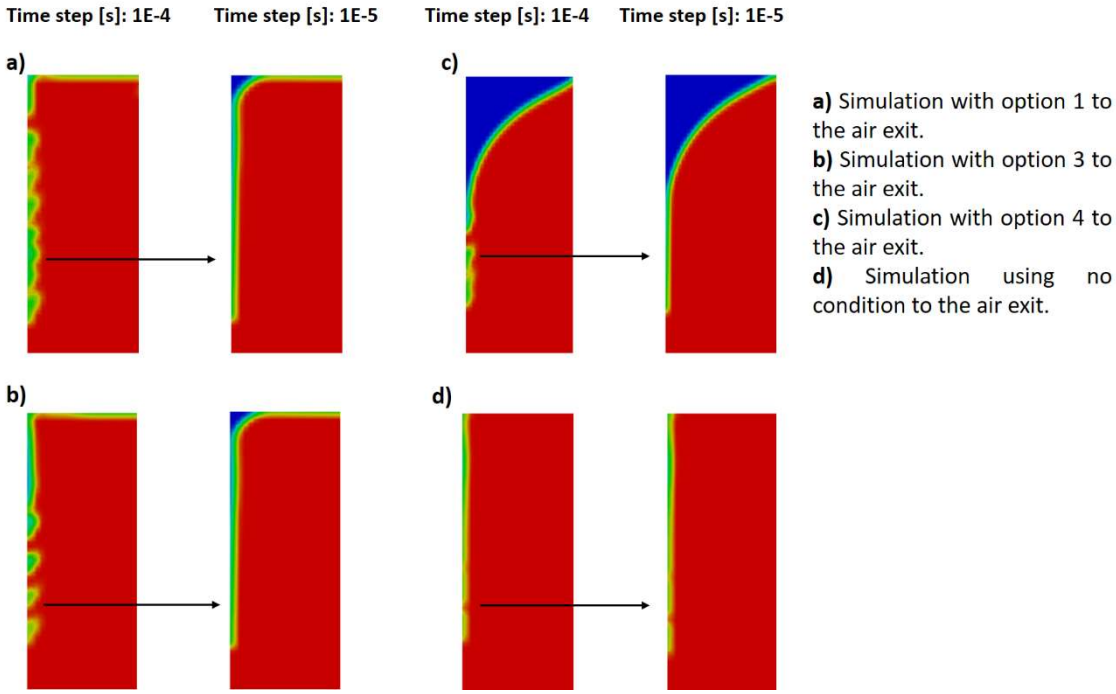

Figure 6.2: Simulations results near the wall with different boundary conditions and for different time step.

The solutions of the simulations with a smaller time step presented improvements compared with the time step of 1E-4 seconds. When we let both air and fluid leave through the outlet we still observe some oscillations. Another fact we can take from the results is that when these oscillations appear, we have a higher percentage of filling. This is more visible in subfigures **a)** and **b)** of Figure 6.2. This occurrence makes us question if the problem is because of velocity of the interface near the wall

As an example, in Figure 6.3 we can see the velocity field and the flow front at t=5 seconds of simulation for the third option of air vents location.

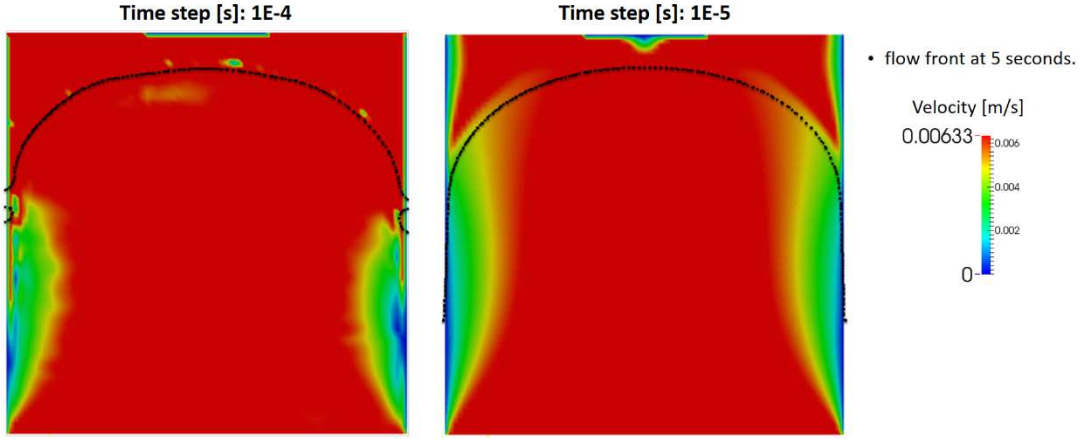

Figure 6.3: Velocity field at t=5 seconds.

Note that, “on top” of the flow front line we only have air. We see many differences between the velocity profiles using a different time step. We should have zero velocity in the wall and we see it well using the time step of 1E-5 seconds, but using the higher time step we have velocity fluctuations at the wall. We did not see, these velocity variations at the wall, in the case that we let the air and material exit, maybe because the oscillations are small.

It should be remarked that we barely observe oscillations using the Bird-Carreau Model. The difference between the two flows is the viscosity, with Newtonian flow we have a constant viscosity, but with the Bird-Carreau we obtained a viscosity profile, as the one exhibited in Figure 6.4.

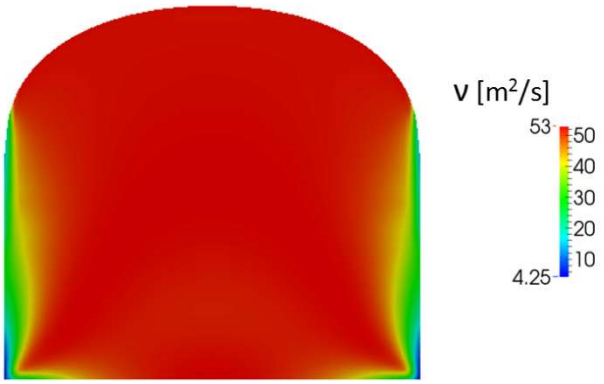

Figure 6.4: Viscosity profile with the Bird-Carreau model, at 5s.

We conclude that we have a viscosity in the wall greater than the used in the Newtonian flow (0.2 m^2/s), and this can be also influencing the appearance of oscillations. Therefore, we performed simulations with a Newtonian fluid with higher viscosity (the results are shown in the Figure 6.5). We can clearly see that we have fewer oscillations when we have a higher viscosity.

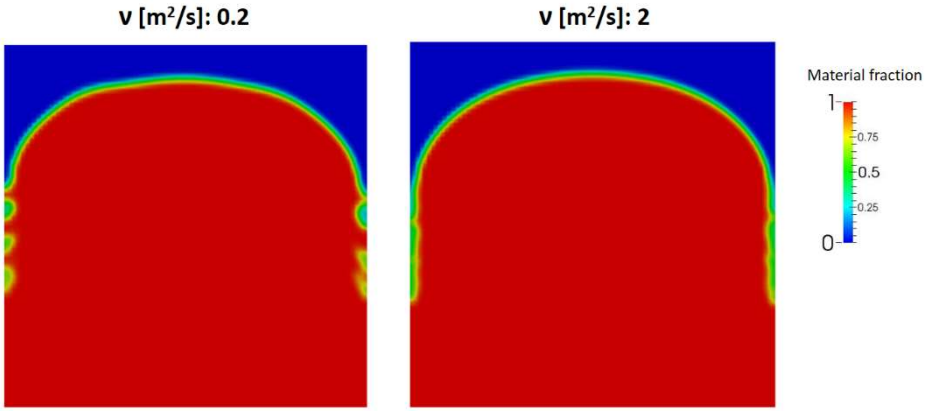

Figure 6.5: Flow front at t=5s considering different viscosity values.

The velocity profile of the simulation using a greater viscosity shows also, less oscillations near the wall, as expected (Figure 6.6).

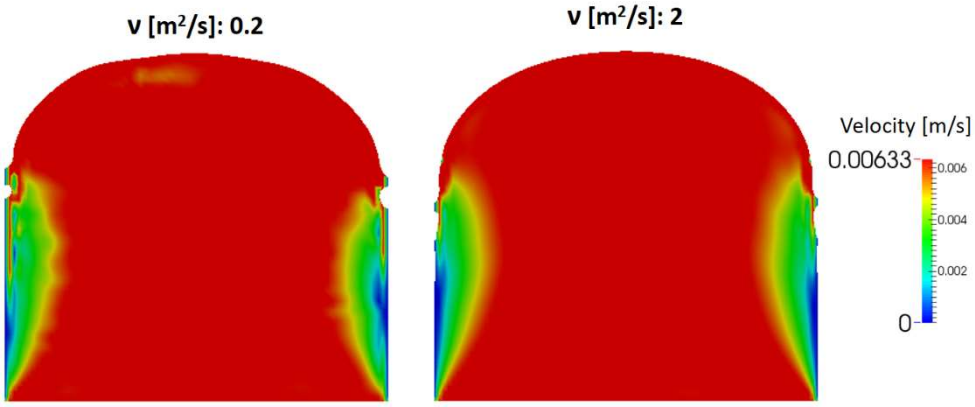

Figure 6.6: Velocity profile using Newtonian fluids with different viscosities, at 5s.

It seems that the solver produces more stable results when the viscosity in the wall is higher leading also to a better velocity field, and consequently less oscillations near the wall.

Besides those oscillations, we still have good results in the flow front. It seems that the decrease of the time step is not always the solution to remove these oscillations, but, due the limited time to perform this work, this problem could not be studied in detail.

7. Non-isothermal flow with temperature-dependent viscosity

As said before, interFoam is an isothermal solver, but, the injection moulding process is a non-isothermal process. For that reason we have implemented the energy equation in the interFoam solver and we called it viscousHeatingSolver (the implementation of this solver is totally described in the Annex II).

Many authors, for instance Khor *et al.* [33], Tie, Dequn and Huamin [34], Zhou, Yan and Zhang [35] and Yang *et al.* [36] use the Cross model, an inelastic model, modified with the Arrhenius equation (to add the temperature influence on the flow development) to study the injection moulding filling stage. The implementation of this model as well as the implementation of the temperature dependence can be seen in the Annex I.

The modified Cross model is given by,

$$\eta(|\dot{\gamma}|,T) = \frac{\eta_0(T)}{1+\left(\frac{\eta_0(T)\,|\dot{\gamma}|}{\tau^*}\right)^{1-n}} \tag{39}$$

where $\dot{\gamma}$ is the shear rate and n and τ^* are material constants. The zero shear viscosity, $\eta_0(T)$, is given by the Arrhenius equation, at the reference temperature (T_{ref}),

$$\eta_0(T) = \eta_0 e^{-b(T-T_{ref})} \tag{40}$$

b, is also a material constant.

7.1. Definition of the case study: geometry, mesh and data

To test the new non-isothermal solver, as well as the temperature-dependent viscosity model we compare our simulations results with the work of Wang, Li and Han [23]. The geometry used to perform such simulations is presented in the Figure 7.1.

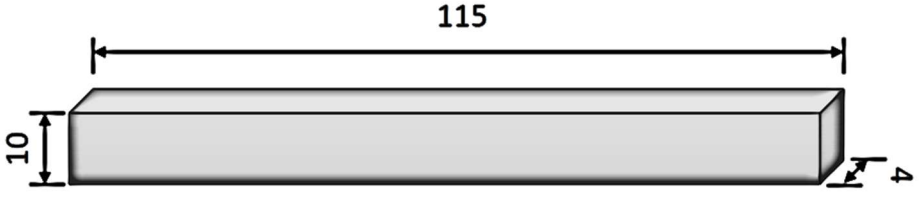

Figure 7.1: Two-dimensional geometry. Dimensions: 115X10X4 [mm].

The parameters of the modified Cross model used for the material were taken from the work of Wang, Li and Han, and are presented in Table 3. Note that we use the kinematic viscosity instead of the dynamic viscosity.

Table 3: Parameters of the modified Cross model.

	ν_0 [m^2/s]	τ^* [m^2/s]	n	b [K^{-1}]	T_{ref} [K]
Fluid	3.105	90.941	0.21	5.8E-3	433

The study of Wang, Li and Han refers nothing about the simulation of the air phase, but in this work we consider this phase, with a viscosity of 15.11E-6 m^2/s [37].

The thermal properties and the density used for the fluid [23] and the air [37] are exhibited in the following table.

Table 4: Density and thermal properties of the fluid and the air.

	k [(kg·m) /s^3]	C_p [$m^2/(s^2 \cdot K)$]	ρ [kg/m^3]
Fluid	0.355	2300	920
Air	0.025	1005	1.205

To perform this case study we used the two-dimensional mesh represented in Figure 7.2.

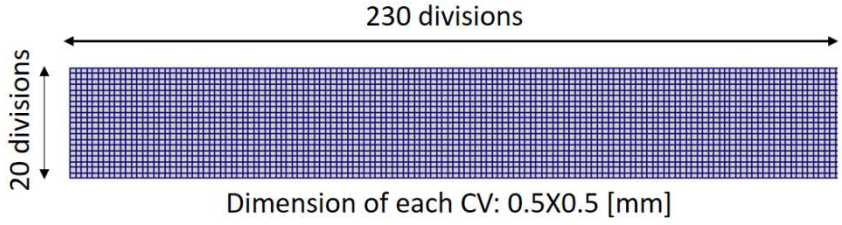

Figure 7.2: Mesh used in the non-isothermal simulations.

The boundary conditions for this case are shown in Figure 7.3. As we can see, the inlet velocity of the fluid is 0.1 m/s and the fluid enters the channel at a temperature of 465 Kelvin. Before the filling, the material inside the mould is at 295 K (room temperature), and the remaining boundaries are set to T=315 K.

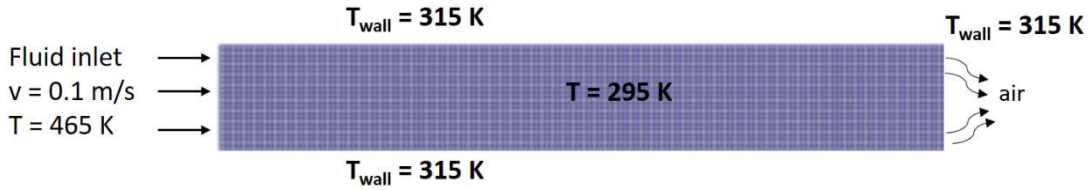

Figure 7.3: Representation of the boundary conditions.

Note that in this simulation we use the boundary condition that allows the air to exit the mould but not the material.

7.2. Simulation results

In Figure 7.4 we show the simulation results regarding the flow front. We can see that our flow front is much more curved. The work of Wang, Li and Han does not consider the air, but we do. We did two experiments without and with surface tension (2.77E-2 kg/m^2), we also tested different air viscosities and nothing changed the flow front geometry. Besides, our mesh is more refined, maybe that is why the difference between our simulations and theirs.

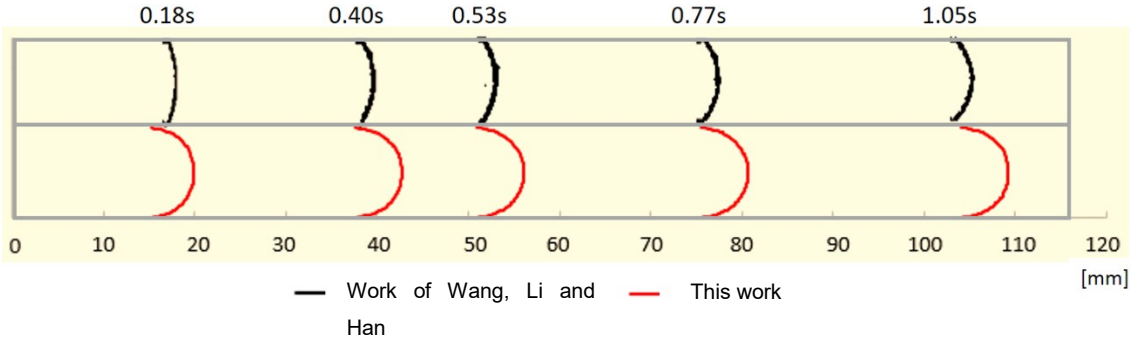

Figure 7.4: Flow front using the Cross model modified with Arrhenius equation.

Regarding, the temperature field, we can see the results in Figure 7.5. Our result seems to be similar to the ones obtained by Wang. We observe a gradient of temperature that is sharp in the zone near the side wall (this indicates the material is cooling faster in this region due to the smaller temperature imposed at the wall). Our modified solver seems to be working fine. In the future further tests need to be performed to validate the solver.

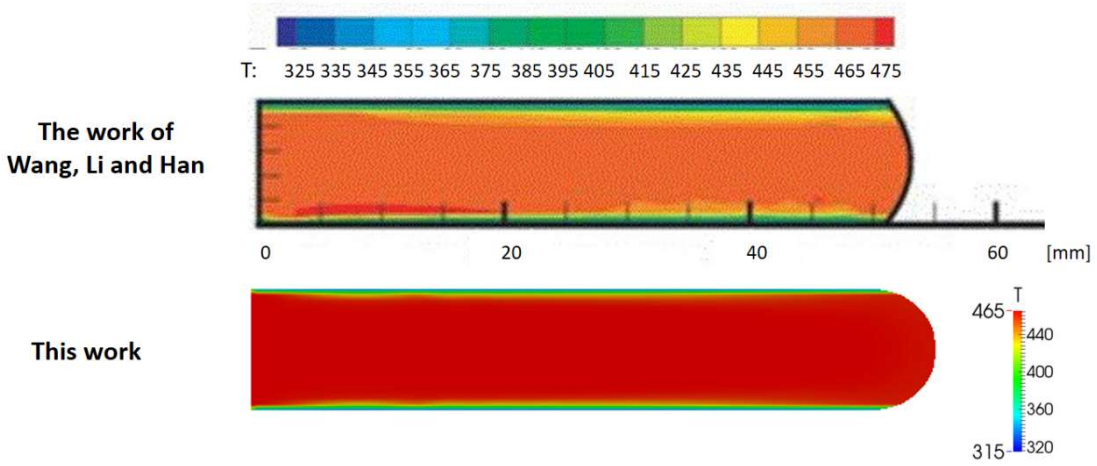

Figure 7.5: Simulation results of the temperature field at 0.53s. Comparison with the work of Wang, Li and Han.

8. Viscoelastic simulations

The interFoam solver was modified so that it could deal with viscoelastic fluids – viscoelasticInterFoam (see Annex III). Due to time limitations we could only perform really simple simulations considering an Oldroyd-B model, low polymer viscosity, and using Mesh 2. Some tests were performed with different parameters and different models (for example the PTT model), but we faced severe convergence problems (mainly due the high polymer viscosities tested).

The parameters used in this study are: $\eta_s = 1.(185)$, $\eta_p = 0.(814)$, $\lambda = 0.(6)$.

Figure 8.1 shows the results obtained for a case study where the fluid and air can both leave the mould through the bottom boundary.

The results are only shown for t=1, 2, and 3 seconds, from left to right (these simulations are still running).

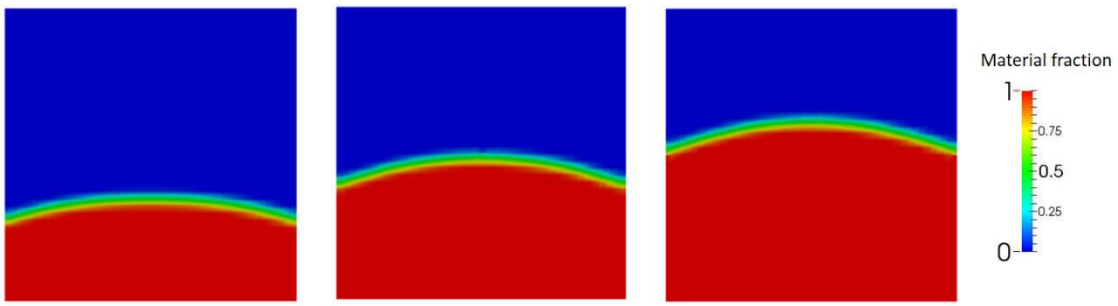

Figure 8.1: Fluid front evolution for the Oldroyd-B model (from left to right – t=1,2,3 [s]).

Based on the preliminary tests we have performed, the viscoelasticInterFoam solver seems to be inadequate to model two fluids with highly different viscosities, but, more mathematical studies need to be performed in order to evaluate the performance of this technique (the one employed by interFoam) under sharp differences at the interface.

9. Three dimensional simulation using a real injected part

In this chapter we simulate the injection moulding filling stage using a real injected part, a tensile specimen, considering its feeding system too.

9.1. Definition of the case study: geometry, mesh and material properties

To perform the simulations we use a tensile specimen with the dimensions illustrated in Figure 9.1.

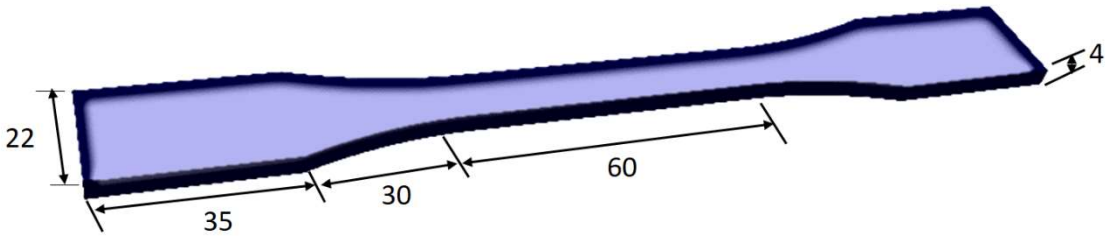

Figure 9.1: Dimension of the tensile specimen in millimetres.

The feeding system dimensions are shown in Figure 9.2.

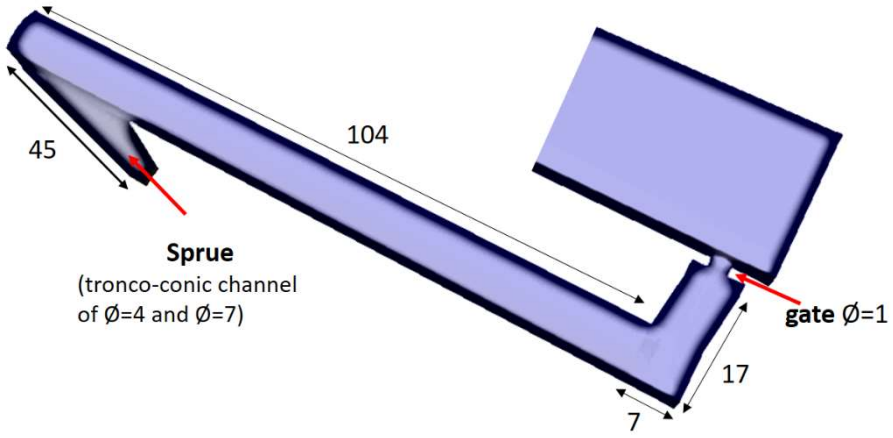

Figure 9.2: Dimension of the feeding system of the part in millimetres.

We tried to make a mesh as most uniform as possible, but, because of the complexity of geometry, this was not possible. Anyway, our maximum control volume size is of 0.5 [mm] (edge).

To perform Newtonian simulations, we used the data of Table 5, and we consider that the air can leave the mould through the last filling zone. Note that the velocity in Table 5 corresponds to the injection velocity at the inlet.

Table 5: Data used in the simulation of real injected part.

	ν [m^2/s]	Density [kg/m^3]	Surface tension [kg/s^2]	Velocity [m/s]
Fluid	0.2	1000	2.77E-2	5E-2
Air	15.11E-6	1.205		

We also perform simulations with inelastic fluid, using the Bird-Carreau model. The parameters were presented in Table 2.

9.2. Newtonian simulation results

In Figure 9.3. we show the results obtained at t= 22.21s, and, we can see that the appearance of oscillations near the wall could not be avoided. In this case the oscillations are more pronounced because we simulated a three-dimensional case, thus we have oscillations also in the top and bottom walls.

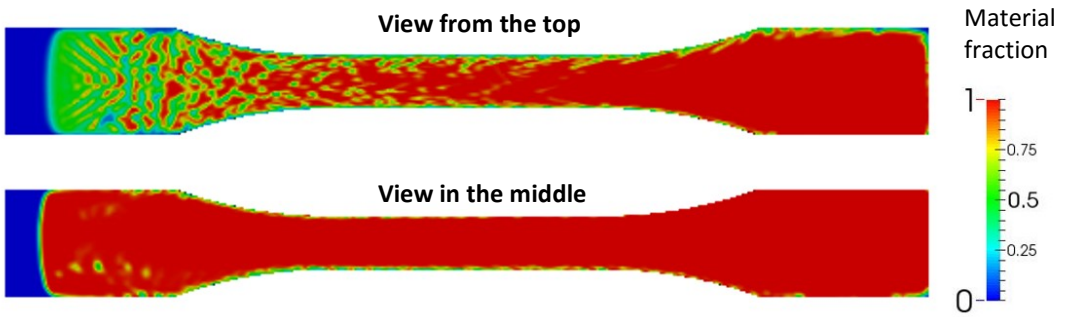

Figure 9.3: Simulation at 22.21s.

After 22.21s, some unfilled zones appear that make no sense, we can clearly see it in the Figure 9.4. This is also due to oscillations.

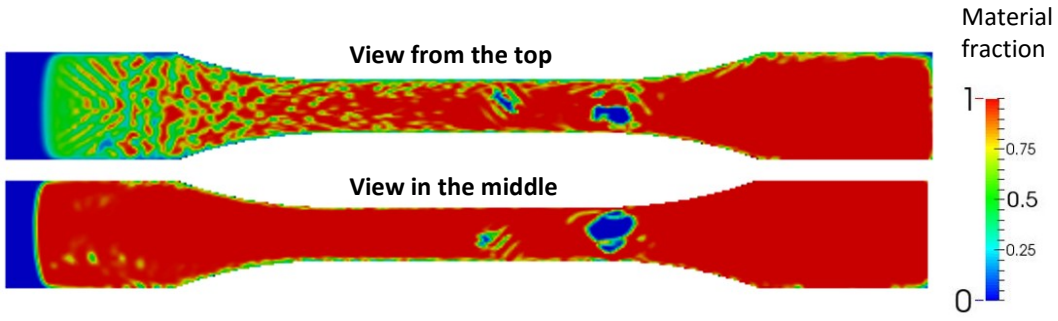

Figure 9.4: Simulation at 22.29s

When we see the whole domain together, in Figure 9.5 we can see that something wrong happened with the numerical solution of the material phase

fraction. Some cells are filled with material over six hundred percent, of course this is impossible.

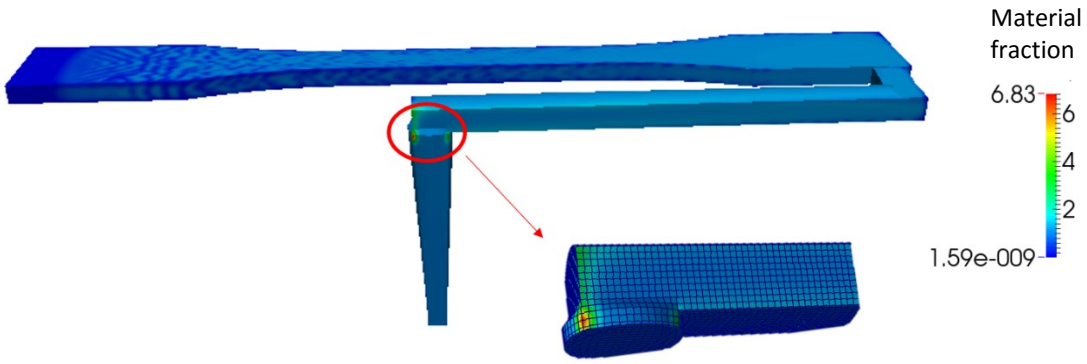

Figure 9.5: Simulation result in the whole domain.

We tried to refine the mesh and the time step, but, the simulations are still slowly running.

9.3. Simulation results using the Bird-Carreau model

In Figure 9.6 is shown the simulation result at 18 seconds. This is the last time with a good result, from there, the simulation explodes.

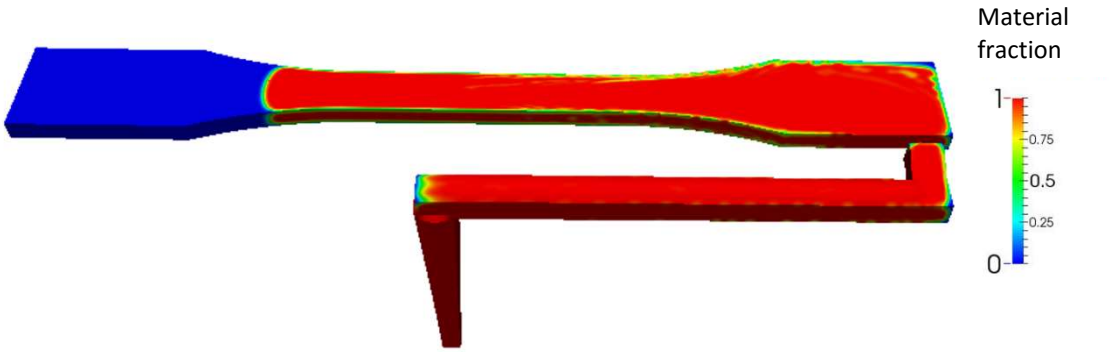

Figure 9.6: Simulation result at 18s.

As we can see, with the Bird-Carreau model, the oscillations decrease considerably comparing with the Newtonian results.

The last result of the simulation is illustrated in Figure 9.7. In this case, the increase of the material fraction led to higher residuals, and therefore leading to the end of the simulation.

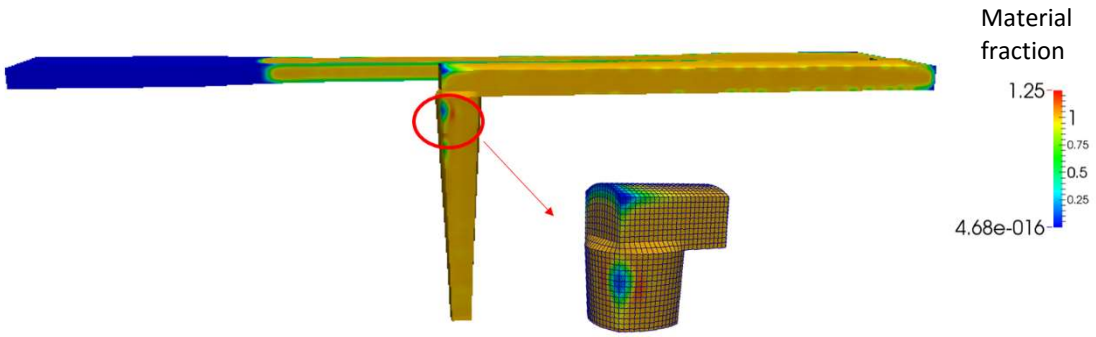

Figure 9.7: Simulation result at 18.15s.

In the Newtonian simulations the material fraction starts to increase from the 7 seconds, but using the Bird-Carreau model only from the 18.15 seconds the same happened. The error seems to appear in the same place of the geometry, what makes us think that can be a geometry imperfection. If we look close to the Figure 9.6, we see that, the place that at 18.15 seconds has higher material fraction, is completely filled at 18 seconds, what makes us think that can be a problem of the software, too.

We tried to simulate from the 18 seconds with a lower time step (1E-6 seconds), but we also obtained an increase of the material fraction.

10. Conclusion

We succeed in the implementation of a boundary condition capable of modeling the air vents.

We concluded that the last filling zone/boundary is the best region, for the fluid to leave the mould. We obtained a higher filling percentage when the air leaves through the bottom and side boundaries.

After obtaining our preliminary results we observed oscillations near the wall which should not exist. These oscillations only appear with the Newtonian flow, what makes us think that the used solver is better when the viscosity in the wall is higher.

The study of a non-isothermal flow was also performed, showing the good implementation of the energy equation in the interFoam solver. The results were compared with another numerical and experimental work, and they were qualitatively similar.

We have also extended the interFoam solver to deal with viscoelastic fluids, but, we could only perform one simulation, considering the Oldroyd-B model with a low viscosity.

The simulation with a real complex geometry revealed some problems in the phase fraction profile, for the mesh we have tested. The weird part is that the residuals of all equations were low in the Newtonian simulation. The simulation with the Bird-Carreau model led to a better result but it presented the same problem in the phase fraction profile.

One of the targets of this work was to perceive the performance of the solver, using the new implemented boundary conditions. The observed oscillations are a real problem of the solver that we need to understand, because it can be the reason of other nonphysical solutions.

We believe that OpenFOAM® is a good software, but, interFoam needs further validation, because, of its difficulty in solving two completely different phases.

Future Work

Further validation tests with the used models, namely the Cross model modified with the Arrhenius equation.

Additional tests with viscoelastic fluids should be performed, which can take into account the evolution of the stresses in the regions where we have a singularity.

A future work would be to consider more refined meshes and smaller time steps, and also, the filling using a viscoelastic models with a real complex geometry.

Annex I – Code used to create the new boundary conditions (air vents)

To create the new boundary condition, to let the air exit the mould but not the polymer, we have to create a code that basically says what happens to velocity and pressure at the boundary under study. In order to create this code we need to create in the user directory two folders, one for the pressure and one for velocity. The content of the folder for pressure is shown in Figure A. 1: Content for the pressure folder.

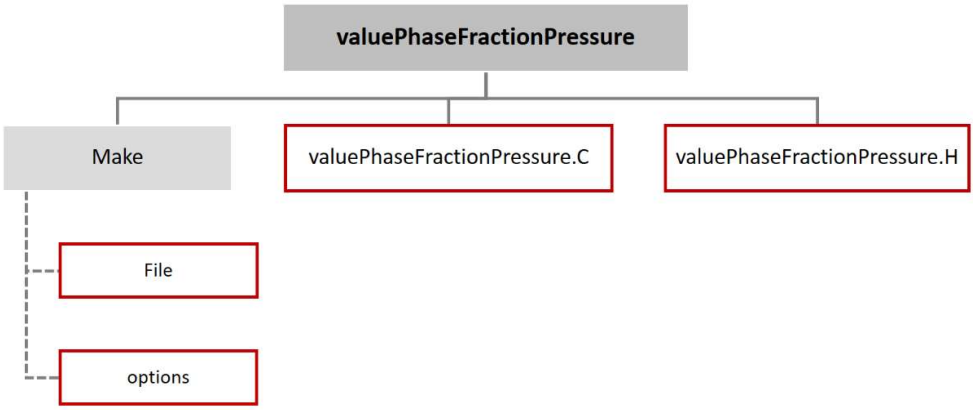

Figure A. 1: Content for the pressure folder.

The content of the folder for the velocity is the same but instead of writing pressure, we write velocity. We are only going to show the code for the pressure boundary, for the velocity is the same; we just have to be careful, because the velocity is a vector field, not a scalar one.

In the file valuePhaseFractionPressure.H, we must have the class names and their function. The added code is the following,

```
#ifndef valuePhaseFractionPressure_H
#define valuePhaseFractionPressure_H
#include "mixedFvPatchFields.H“
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
namespace Foam
{
/*---------------------------------------------------------------------------*\
Class valuePhaseFractionPressure Declaration
\*---------------------------------------------------------------------------*/
class valuePhaseFractionPressure
:
public mixedFvPatchScalarField
{
word alphaName_;                    //- Name of the phase-fraction field
```

```
        public:
        TypeName("valuePhaseFractionPressure");          //- Runtime type information

        //- Construct from patch and internal field
        valuePhaseFractionPressure
        (
                const fvPatch&,
                const DimensionedField<scalar, volMesh>&
        );
        //- Construct from patch, internal field and dictionary
         valuePhaseFractionPressure
        (
                const fvPatch&,
                const DimensionedField<scalar, volMesh>&,
                const dictionary&
        );
//- Construct by mapping given outletPhaseMeanVelocityFvPatchScalarField onto a new patch
        valuePhaseFractionPressure
        (
                const valuePhaseFractionPressure&,
                const fvPatch&,
                const DimensionedField<scalar, volMesh>&,
                const fvPatchFieldMapper&
        );
        //- Construct as copy
        valuePhaseFractionPressure
        (
        const valuePhaseFractionPressure&
        );
        //- Construct and return a clone
        virtual tmp<fvPatchScalarField> clone() const
        {
        return tmp<fvPatchScalarField>
        (
        new valuePhaseFractionPressure(*this)
        );
    }
        //- Construct as copy setting internal field reference
        valuePhaseFractionPressure
          (
                const valuePhaseFractionPressure&,
                const DimensionedField<scalar, volMesh>&
           );
```

```
        //- Construct and return a clone setting internal field reference
        virtual tmp<fvPatchScalarField> clone
        (
         const DimensionedField<scalar, volMesh>& iF
        ) const
         {
        return tmp<fvPatchScalarField>
        (
                    new valuePhaseFractionPressure
                   (
                            *this,
                            iF
                   )
        );
    }
        //- Update the coefficients associated with the patch field
        virtual void updateCoeffs();
        //- Write
        virtual void write(Ostream&) const;
        };
        }
```

In the file valuePhaseFractionPressure.C we have the main code,

```
        #include "valuePhaseFractionPressure.H“
        #include "volFields.H“
        #include "addToRunTimeSelectionTable.H“
        #include "fvPatchFieldMapper.H“
        #include "surfaceFields.H“
// * * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * * //
        Foam::valuePhaseFractionPressure
        ::valuePhaseFractionPressure
        (
                const fvPatch& p,
                  const DimensionedField<scalar, volMesh>& iF
        )
        :
        mixedFvPatchField<scalar>(p, iF),
        alphaName_("none")
        {
```

```
        refValue() = 0.0;
        refGrad() = 0.0;
        valueFraction() = 0.0;
}
Foam::valuePhaseFractionPressure
::valuePhaseFractionPressure
(
        const valuePhaseFractionPressure& ptf,
        const fvPatch& p,
        const DimensionedField<scalar, volMesh>& iF,
        const fvPatchFieldMapper& mapper
)
:
mixedFvPatchField<scalar>(ptf, p, iF, mapper),
alphaName_(ptf.alphaName_)
{}
Foam::valuePhaseFractionPressure
::valuePhaseFractionPressure
(
        const fvPatch& p,
        const DimensionedField<scalar, volMesh>& iF,
        const dictionary& dict
)
:
mixedFvPatchField<scalar>(p, iF),
alphaName_(dict.lookup("alpha"))
{
refValue() = 0.0;
refGrad() = 0.0;
valueFraction() = 0.0;
 if (dict.found("value"))
{
        fvPatchScalarField::operator=
        (
        scalarField("value", dict, p.size())
        );
}
else
{
        fvPatchScalarField::operator=(patchInternalField());
}
}
```

```
Foam::valuePhaseFractionPressure
::valuePhaseFractionPressure
(
 const valuePhaseFractionPressure& ptf
)
:
mixedFvPatchField<scalar>(ptf),
alphaName_(ptf.alphaName_)
{}
Foam::valuePhaseFractionPressure
::valuePhaseFractionPressure
(
            const valuePhaseFractionPressure& ptf,
           const DimensionedField<scalar, volMesh>& iF
)
:
mixedFvPatchField<scalar>(ptf, iF),
alphaName_(ptf.alphaName_)
{}
// * * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * * * * //
void Foam::valuePhaseFractionPressure::updateCoeffs()
{
if (updated())
{
return;
}
 const fvPatchField<scalar>& alphap =
 patch().lookupPatchField<volScalarField, scalar>
(
           alphaName_
);
scalarField alphaCellp(alphap.patchInternalField());
forAll(patch(), faceI)
```

The next lines of the code are different for pressure and velocity files.

For the pressure file,

```
{
        if (alphaCellp[facel] > 0.5)
        {
        refGrad()[facel] = 0.0;
        valueFraction()[facel] = 0.0;
        }
        else
        {
        refValue()[facel] = 0.0;
        valueFraction()[facel] = 1.0;
        }
}
```

For the velocity file,

```
{
        if (alphaCellp[facel] > 0.5)
        {
        refValue()[facel] = vector::zero;
        valueFraction()[facel] = 1.0;
        }
        else
        {
        refGrad()[facel] = vector::zero;
        valueFraction()[facel] = 0.0;
        }
}
```

These code lines says what was already explained in the main text:

"*Basically, in this new boundary condition we have the following:*

- when the fraction of the material is higher than half at the last cell touching the venting gap, the velocity of the material at that boundary is set to zero (as happens in a wall), and a zero-gradient boundary condition is used for the pressure.

- If in the last cell touching the venting gap we have $\gamma < 0.5$*, the pressure at that boundary is set to zero and a zero-gradient boundary condition is considered for the velocity, allowing the air to exit the mould (similar to an outlet boundary condition).*"

The remaining code stays the same, remembering always that velocity is a vector field, not a scalar field as pressure is.

```
        }
        mixedFvPatchField<scalar>::updateCoeffs();
        }
        void Foam::valuePhaseFractionPressure::write
        (
        Ostream& os
        ) const
        {
        fvPatchField<scalar>::write(os);
        os.writeKeyword("alpha") << alphaName_
                << token::END_STATEMENT << nl;
        writeEntry("value", os);
        }
```

```
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
namespace Foam
{
makePatchTypeField
(
        fvPatchScalarField,
        valuePhaseFractionPressure
);
}
```

In the options we add the existent libraries that we need to take in order to compile our new boundary condition.

```
EXE_INC = \
        -I$(LIB_SRC)/triSurface/lnInclude \
        -I$(LIB_SRC)/meshTools/lnInclude \
        -I$(LIB_SRC)/finiteVolume/lnInclude
LIB_LIBS = \
        -lOpenFOAM \
        -ltriSurface \
        -lmeshTools \
        -lfiniteVolume
```

In the file named files we write,

```
valuePhaseFractionPressure.C

LIB = $(FOAM_USER_LIBBIN)/libvaluePhaseFractionPressure
```

Regarding the velocity we call the new boundary condition as valuePhaseFractionU.

Once we created a new boundary condition, we need to do some changes in our running cases. Those modifications are presented in Figure A. 2: Changes in the running files.

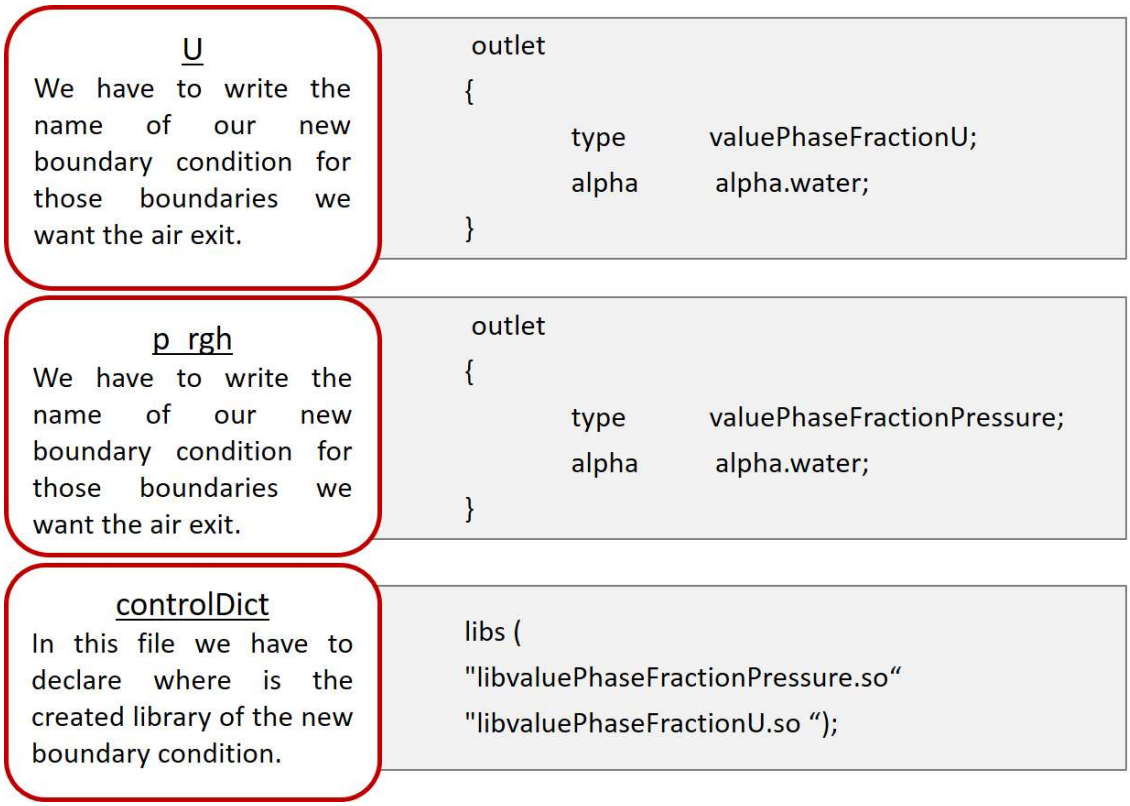

Figure A. 2: Changes in the running files.

Annex II – Implementation of the energy equation

For the InterFoam to able to deal with non-isothermal fluids, we need to add to this solver the energy equation (Eq. (17)).

Modification of the transport model

Since we need to introduce new thermal properties, the transportModels library needs to be modified. The files that we have to adjust are inside the directory incompressibleTwoPhaseMixture. We, also, want to add a new temperature-dependent viscosity model, the Cross model modified by Arrhenius equation, so we have to also change the files inside the viscosityModels directory. The path to access the files that we need to change (or create) is shown in Figure A. 3: Directories to reach the needed files..

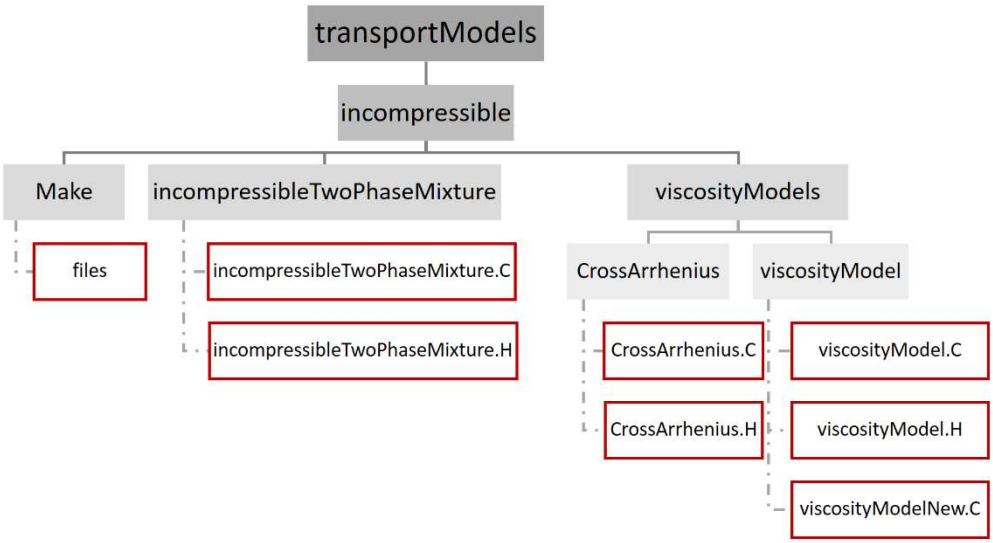

Figure A. 3: Directories to reach the needed files.

First of all, we have to copy the incompressible model to the user folder and clean all the dependencies already existing, using the commands,

```
cp –r $WM_PROJECT_DIR/src/transportModels $WM_PROJECT_USER_DIR/src
wclean
```

As we can see in Equation (17), we have two new constants that need to be introduced, C_p and k in the file twoPhaseMixture.H, defining them as a scalar.

```
        dimensionedScalar rho1_;
        dimensionedScalar rho2_;
//----------------------------Modified----------------------------//
        dimensionedScalar cp1_;
        dimensionedScalar cp2_;
        dimensionedScalar k1_;
        dimensionedScalar k2_;
//-----------------------------End----------------------------//
```

Then, in the same file, we add,

```
        const dimensionedScalar& rho2() const
        {
        return rho2_;
        };
//----------------------------Modified----------------------------//
        const dimensionedScalar& cp1() const
        {
        return cp1_;
        };
        const dimensionedScalar& cp2() const
        {
        return cp2_;
        };
        const dimensionedScalar& k1() const
        {
        return k1_;
        };
        const dimensionedScalar& k2() const
        {
        return k2_;
        };
//-----------------------------End----------------------------//
```

Lastly, we write a line, which basically says to return the face-interpolated thermal conductivity of the mixture,

```
        temp<surfaceScalarField> nuf() const;
//----------------------------Modified----------------------------//
        temp<surfaceScalarField> kappaf() const;
//-----------------------------End----------------------------//
```

The first change that we do in the file two twoPhaseMixture.C is,

```
rho2_("rho", dimDensity, nuModel2_->viscosityProperties().lookup("rho")),
//----------------------------Modified----------------------------//
cp1_(nuModel1_->viscosityProperties().lookup ("cp")),
cp2_(nuModel1_->viscosityProperties().lookup ("cp")),
k1_(nuModel1_->viscosityProperties().lookup (“k")),
k2_(nuModel1_->viscosityProperties().lookup (“k")),
//-----------------------------End-----------------------------//
```

We have now to add the thermal conductivity of the mixture, which is given by the following equation,

$$k = \alpha k_1 + (1-\alpha)\alpha k_2 \tag{41}$$

```
"nuf ",
(
alpha1f*rho1_*fvc::interpolate(nuModel1_->nu())
 + (scalar(1) - alpha1f)*rho2_*fvc::interpolate(nuModel2_->nu())
)/(alpha1f*rho1_ + (scalar(1) - alpha1f)*rho2_)
)
 );
}
//----------------------------Modified----------------------------//
Foam::temp<Foam::surfaceScalarField>
Foam::incompressibleTwoPhaseMixture::kappaf() const
{
        const surfaceScalarField alpha1f
        (
         min(max(fvc::interpolate(alpha1_), scalar(0)), scalar(1))
         );

         return temp<surfaceScalarField>
        (
         new surfaceScalarField
        (
        "kappaf",
        alpha1f*k1_
       + (scalar(1) - alpha1f)*k2_
        )
        );
//-----------------------------End-----------------------------//
```

The final modification is

```
nuModel2_->viscosityProperties().lookup("rho") >> rho2_;
//---------------------------Modified---------------------------//
 nuModel1_->viscosityProperties().lookup("cp") >> cp1_;
 nuModel2_->viscosityProperties().lookup("cp") >> cp2_;
 nuModel1_->viscosityProperties().lookup("k") >> k1_;
 nuModel2_->viscosityProperties().lookup("k") >> k2_;
//-----------------------------End-----------------------------//
```

Before compiling the new transport model, we want to add a new viscosity model, thus we must create a folder named CrossArrhenius inside the viscosityModels directory. Inside that folder we can create a file called CrossArrhenius.H, with the following code,

```
#infndef CrossArrhenius_H
#define CrossArrhenius_H
#include "viscosityModel.H"
#include "dimensionedScalar.H"
#include "volFields.H"
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
namespace Foam
{
namespace viscosityModels
{
/*---------------------------------------------------------------------------*\
          Class CrossArrhenius Declaration
\*---------------------------------------------------------------------------*/
class CrossArrhenius
:
  public viscosityModel
{
 // Private data
    dictionary CrossArrheniusCoeffs_;
    dimensionedScalar nuref_;   → η0
    dimensionedScalar b_;       → b
    dimensionedScalar Tref_;    → Tref
   dimensionedScalar t1_;       → τ*
   dimensionedScalar n_;        → n

   volScalarField nu_;
```

Parameters of the modified Cross model

```
// Private Member Functions
        //- Calculate and return the laminar viscosity
        tmp<volScalarField> calcNu() const;
public:
        //- Runtime type information
        TypeName("CrossArrhenius");

// Constructors
        //- Construct from components
        CrossArrhenius
        (
        const word& name,
        const dictionary& viscosityProperties,
        const volVectorField& U,
        const surfaceScalarField& phi,
        const volScalarField& T
         );
 // Destructor
   ~CrossArrhenius()
   {}
// Member Functions
    //- Return the laminar viscosity
   tmp<volScalarField> nu() const
     {
         return nu_;
     }
          //- Return the laminar viscosity for patch
      tmp<scalarField> nu(const label patchi) const
      {
        return nu_.boundaryField()[patchi];
      }
     //- Correct the laminar viscosity
     void correct()
     {
            nu_ = calcNu();
      }
 //- Read transportProperties dictionary
      bool read(const dictionary& viscosityProperties);
};
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
} // End namespace viscosityModels
} // End namespace Foam
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
#endif
```

In the same folder, we create another file called CrossArrhenius.C, it is in this file that we define the modified Cross model. All the information present in this file is,

```
#include "CrossArrhenius.H“
#include "addToRunTimeSelectionTable.H“
#include "surfaceFields.H“
// * * * * * * * * * * * * * * * Static Data Members * * * * * * * * * * * * * * * //
        namespace Foam
        {
        namespace viscosityModels
        {
        defineTypeNameAndDebug(CrossArrhenius, 0);
        addToRunTimeSelectionTable
        (
                viscosityModel,
                CrossArrhenius,
                dictionary
        );
        }
        }
// * * * * * * * * * * * * * * Private Member Functions * * * * * * * * * * * * * * //
        Foam::tmp<Foam::volScalarField>
        Foam::viscosityModels::CrossArrhenius::calcNu() const
        {
                const volScalarField& T=U_.mesh().lookupObject<volScalarField>("T");
                return
                (nuref_*exp(-b_*(T-Tref_)))/(scalar(1)+pow((((nuref_*exp(-b_*(T-
                Tref_)))*strainRate())/t1_), scalar(1)-n_));
        }
// * * * * * * * * * * * * * * * * * Constructors * * * * * * * * * * * * * * * //
        Foam::viscosityModels::CrossArrhenius::CrossArrhenius
        (
                const word& name,
                const dictionary& viscosityProperties,
                const volVectorField& U,
                const surfaceScalarField& phi,
                const volScalarField& T
        )
:
```

```
        viscosityModel(name, viscosityProperties, U, phi,T),
        CrossArrheniusCoeffs_(viscosityProperties.subDict(typeName + "Coeffs")),
        nuref_(CrossArrheniusCoeffs_.lookup("nuref")),
        b_(CrossArrheniusCoeffs_.lookup("b")),
        Tref_(CrossArrheniusCoeffs_.lookup("Tref")),
        t1_(CrossArrheniusCoeffs_.lookup("t1")),
        n_(CrossArrheniusCoeffs_.lookup("n")),

        nu_
        (
        IOobject
        (
                name,
                U_.time().timeName(),
                U_.db(),
                IOobject::NO_READ,
                IOobject::AUTO_WRITE
        ),
        calcNu()
        )
        {}
// * * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * * * * * * //
        bool Foam::viscosityModels::CrossArrhenius::read
        (
                const dictionary& viscosityProperties
        )
        {
        viscosityModel::read(viscosityProperties);
        CrossArrheniusCoeffs_ = viscosityProperties.subDict(typeName + "Coeffs");
        CrossArrheniusCoeffs_.lookup("nuref") >> nuref_;
        CrossArrheniusCoeffs_.lookup("b") >> b_;
        CrossArrheniusCoeffs_.lookup("Tref") >> Tref_;
        CrossArrheniusCoeffs_.lookup("t1") >> t1_;
        CrossArrheniusCoeffs_.lookup("n") >> n_;
        return true;
        }
```

The next modifications are in the files inside of viscosityModel folder. Since we now have to consider the temperature field, everytime the following code line appears,

```
        const volVectorField& U,
        const surfaceScalarField& phi,
```

we only need to add,

```
const volScalarField& T
```

The last file that will suffer some changes is the file named File inside the Make directory. We add the next line to the Files file in order to compile the new transport model with our created viscosity model.

```
viscosityModels/CrossArrhenius/CrossArrhenius.C
```

Finally, since we need to compile the new library, and instead of doing it in the OpenFOAM directory, is better to compile in the user directory, thus we need to modify the last line of the Files file as,

```
LIB = $(FOAM_USER_LIBBIN)/libmyincompressibleTransportModels
```

To compile the library we need to type the following command,

```
wmake libso
```

Modification of the interFoam solver

After we compile the new transport model, we have to modify the interFoam solver. The differences between the interFoam and the new solver created from it, that we call viscousHeatingSolver, are exhibited in Figure A. 4

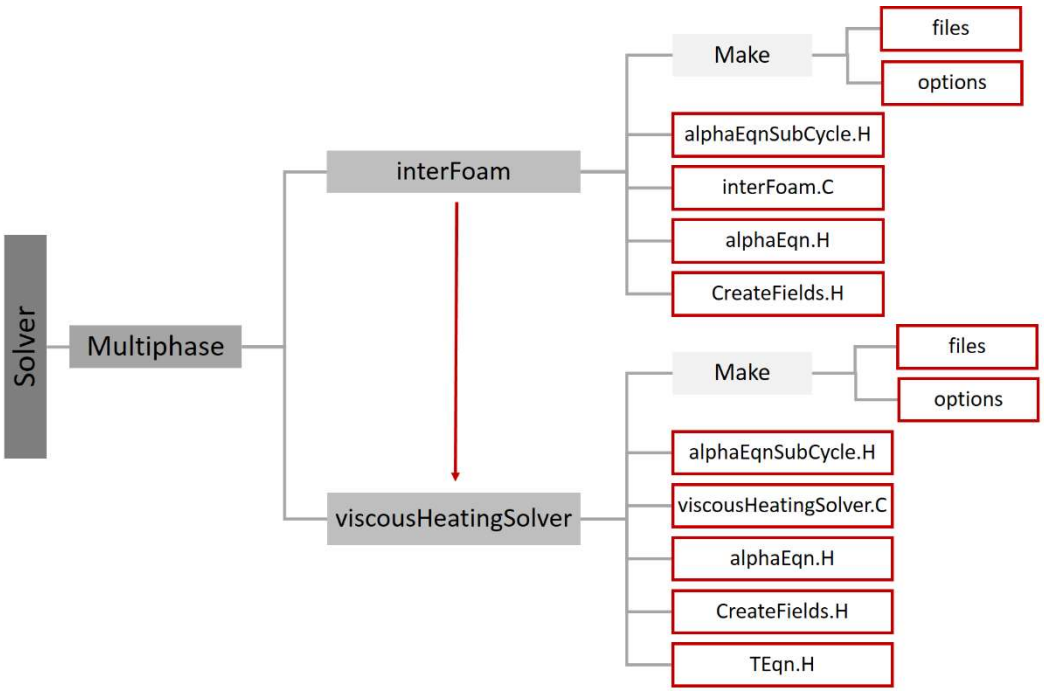

Figure A. 4: Files that need to be modified.

It is possible to observe that we need to create a new file designated by TEqn.H, but before doing any modifications, we also have to copy the interFoam solver to the user directory.

```
cp –r $FOAM_APP/solvers/multiphase $WM_PROJECT_USER_DIR/applications/solvers/multiphase
```

We should then, create temperature field in the file createFields.H, by adding the following lines of code,

```
                "U",
                runTime.timeName(),
                mesh,
                IOobject::MUST_READ,
                IOobject::AUTO_WRITE
                ),
                mesh
    );
    //----------------------------Modified----------------------------//
     Info<< "Reading field T\n" << endl;
    volScalarField T
    (
        IOobject
        (
            "T",
            runTime.timeName(),
            mesh,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
        ),
        mesh
    );
    //-------------------------------End-------------------------------//
```

And declare the specific heat capacity, since we are going to add the equation that needs this term.

```
const dimensionedScalar& rho2 = twoPhaseProperties.rho2();
//----------------------------Modified----------------------------//
const dimensionedScalar& cp1 = twoPhaseProperties.cp1();
const dimensionedScalar& cp2 = twoPhaseProperties.cp2();
//-----------------------------End-----------------------------//
```

The last modification in this file is the description of the specific heat capacity of the mixture,

$$\rho C_p = \alpha\rho_1 C_{p1} + (1-\alpha)\rho_2 C_{p2} \tag{42}$$

and the flux heat,

$$\rho\varphi C_p = \alpha\rho_1\varphi C_{p1} + (1-\alpha)\rho_2\varphi C_{p2} \tag{43}$$

```
        p_rgh = p - rho*gh;
}
//----------------------------Modified----------------------------//
 Info<< "Reading / calculating rho*cp\n" << endl;
volScalarField rhocp
(
        IOobject
        (
                "rho*cp",
                runTime.timeName(),
                mesh,
                IOobject::NO_READ,
                IOobject::NO_WRITE
        ),
        alpha1*rho1*cp1 + (scalar(1) - alpha1)*rho2*cp2,
        alpha1.boundaryField().types()
);
rhocp.oldTime();
```

The φ term is the velocity interpolated to the cell faces.

```
Info<< "Reading / calculating rho*phi*cp\n" << endl;
surfaceScalarField rhoPhiCpf
(
        IOobject
        (
                "rho*phi*cpf",
                runTime.timeName(),
                mesh,
                IOobject::NO_READ,
                IOobject::NO_WRITE
        ),
        rhoPhi*cp1
);
//------------------------------End-----------------------------//
```

Equation (42) is defined in the alphaEqnSubCycle.H.

As well as Equation (43) in the alphaEqn.H file.

```
rhoPhi = tphiAlpha()*(rho1 – rho2) + phi*rho2;
//-----------------------------Modified----------------------------//
rhoPhiCpf = tphiAlpha*(rho1*cp1 – rho2*cp2) + phi*rho2*cp2;
//------------------------------End-----------------------------//
```

Afterwards we need to create an empty file with the name TEqn.H to add the energy equation,

```
volTensorField gradU = fvc::grad(U);
volScalarField nu = twoPhaseProperties.nu();
volTensorField tau = nu * (gradU + gradU.T());
surfaceScalarField kappaf = twoPhaseProperties.kappaf();

fvScalarMatrix Teqn
(
 fvm::ddt(rhocp, T)                    // → ρC_p ∂T/∂t
+ fvm::div(rhoPhiCpf, T)               // → ρC_p u · ∇T
- fvm::laplacian(kappaf, T)            // → ∇ · (k∇T)
== rho * (tau && gradU)                //viscous heat dissipation term
);

TEqn.solve();
```

The file interFoam.C must be renamed to viscousHeatingSolver.C and must contain the following line,

```
        #include "UEqn.H"
//----------------------------Modified----------------------------//
        #include "TEqn.H"
//-----------------------------End-----------------------------//
```

The file named Files, inside the Make directory, should only contain the following lines of code,

```
viscousHeatingSolver.C

EXE = $(FOAM_USER_APPBIN)/viscousHeatingSolver
```

Since we need to connect this new solver with the modified library of the transportModels, we need to add the next lines in the options file inside the Make directory,

```
EXE_INC = -ggdb3 \
        -I$(LIB_SRC)/transportModels/twoPhaseMixture/lnInclude \
         -I$(LIB_SRC)/transportModels \
//----------------------------Modified----------------------------//
-I$(WM_PROJECT_USER_DIR)/src/transportModels/incompressible/lnInclude \
//-----------------------------End-----------------------------//


EXE_LIBS = \
//----------------------------Modified----------------------------//
        -L$(FOAM_USER_LIBBIN) \
        -lmyincompressibleTransportModels \
//-----------------------------End-----------------------------//
```

Finally, we rename the solver interFoam to viscousHeatingSolver and we compile it by using the following command,

```
wmake
```

Modifications on the files for cases using the viscousHeatingSolver

Once we change the transportModels library and the interFoam solver, some of the files that we need to run the cases have to be modified.

The folders that will suffer some change are shown in Figure A. 5. The folder 0 has the boundary conditions in the initial time state. Since viscousHeatingSolver solves the temperature equation, we need to have a new file named T. We add new thermal properties, and their values are written in the transportProperties file. One of the purposes of controlDict file is calling the used solver, so we need to change this file. The fvSchemes file need to be modified, once there are new terms to be discretised, as well as the fvSolution file, since we have to add a new solution method for the linear system resulting from the temperature equation discretization.

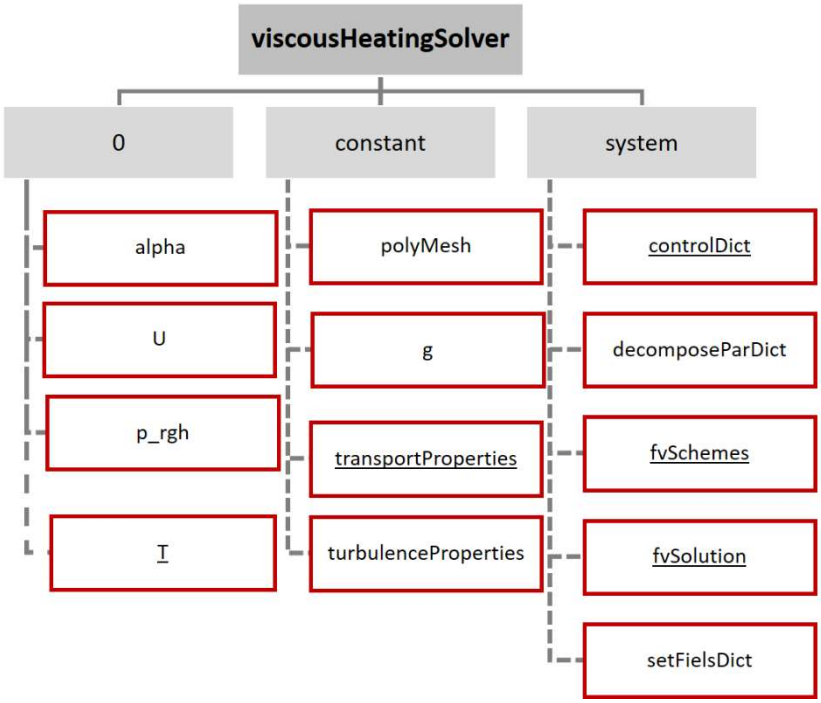

Figure A. 5: The underlined files are the ones who suffer changes.

The temperature file of the initial time has the same functionality of the velocity and pressure file, but we have to be careful that temperature is a scalar field and we have to define our boundaries in Kelvin units, as seen in the Figure A. 6: The T file in the folder 0..

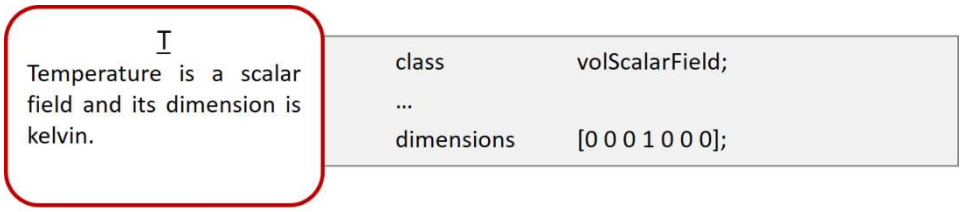

Figure A. 6: The T file in the folder 0.

In the folder *constant* we modified the transportProperties file as shown in Figure A. 7

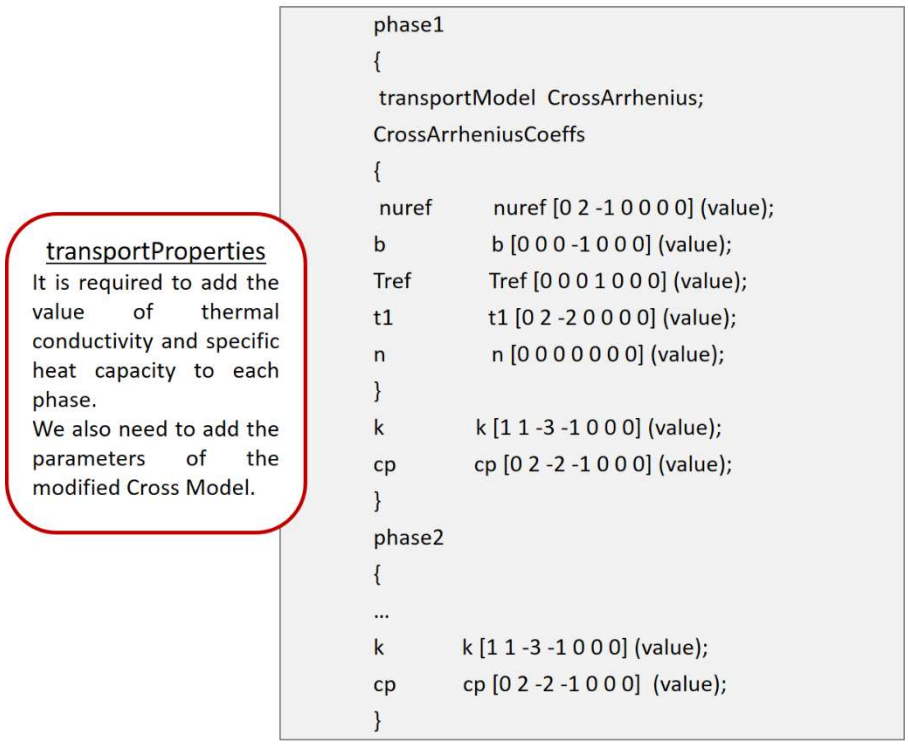

Figure A. 7: Adjustments in the transportProperties file.

The changes in the files inside the system directory are presented in Figure A. 8.

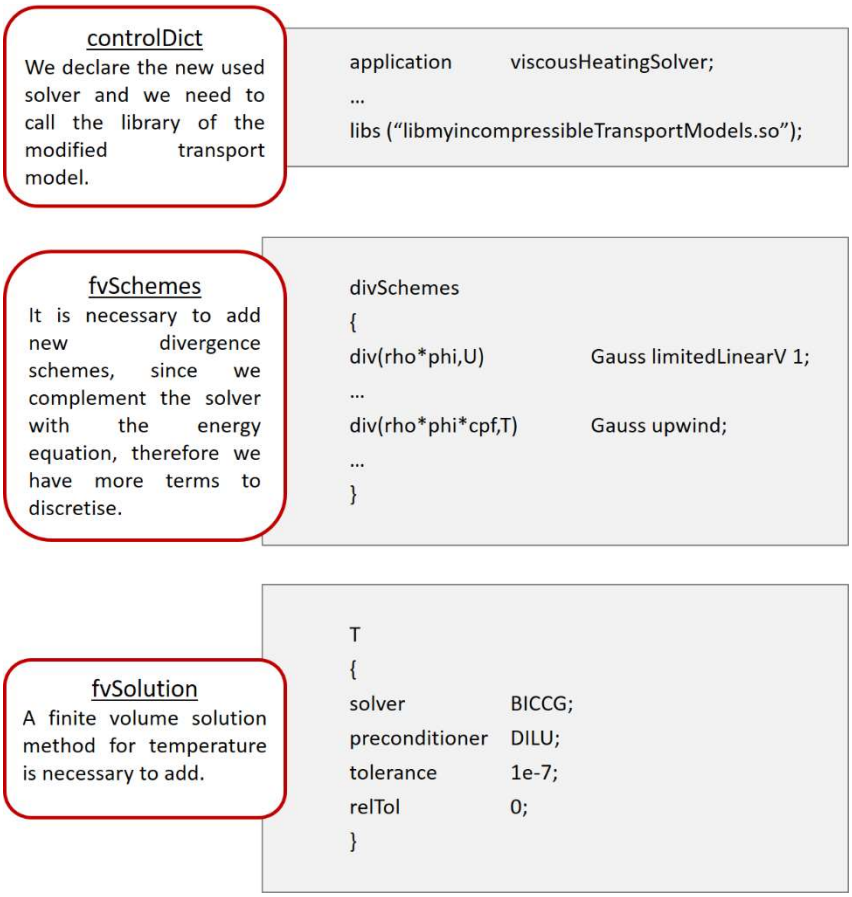

Figure A. 8: Modifications made in the files inside the system folder.

Annex III – Implementation of the viscoelasticInterFoam

For the InterFoam to able to deal with viscoelastic fluids, we need to do some changes.

Modification on the transport model

In Figure A. 9 we can see the files which need to be modify or created in the original transport model used by the interFoam solver

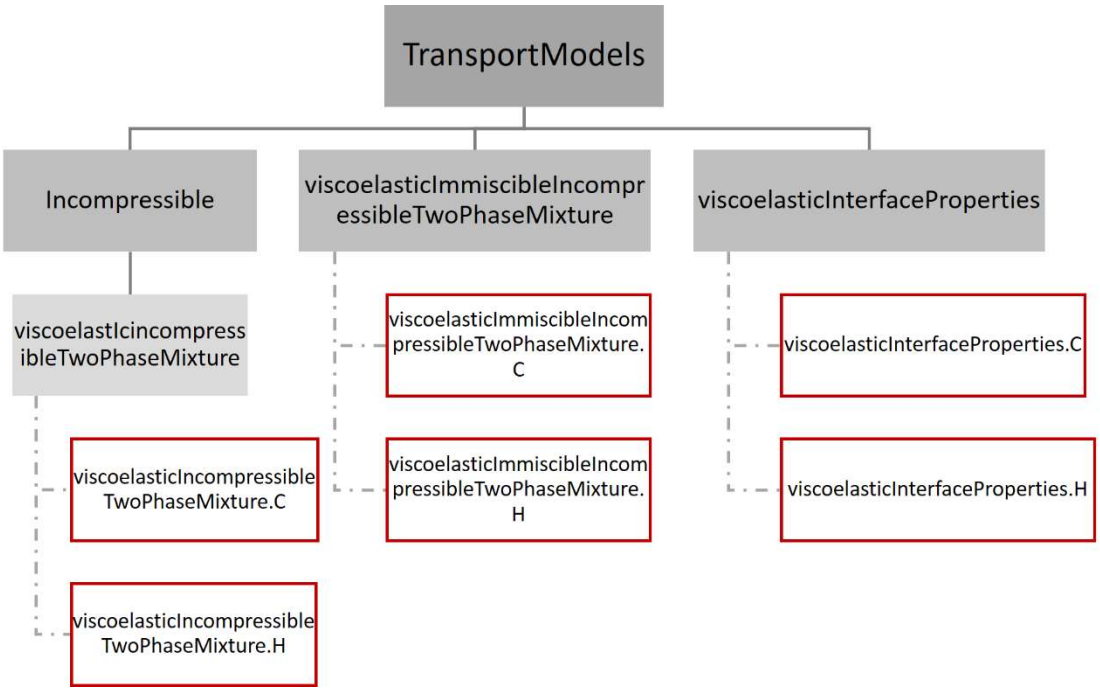

Figure A. 9 - Files that need to be modify.

The Incompressible folder already exists in the original transport model, and also the folder inside it, named viscoelasticIncompressibleTwoPhase-Mixture, but in the original transport model is called IncompressibleTwoPhase-Mixture. The files inside the IncompressibleTwoPhaseMixture need to be deleted and we have to add the viscoelasticIncompressibleTwoPhaseMixture.C file,

```
#include "viscoelasticIncompressibleTwoPhaseMixture.H“
#include "addToRunTimeSelectionTable.H“
#include "surfaceFields.H“
#include "fvc.H“
namespace Foam
{
defineTypeNameAndDebug(viscoelasticIncompressibleTwoPhaseMixture, 0);
}
void Foam::viscoelasticIncompressibleTwoPhaseMixture::calcEta()
{
const volScalarField limitedAlpha1
(
        "limitedAlpha1",
        min(max(alpha1_, scalar(0)), scalar(1))
 );
if (viscosityType_ == "ShearThinning")
{
volScalarField shearRate_ = sqrt(2.0)*mag(symm(fvc::grad(U_)));
volScalarField etaL_ = muInf_ +     ((mu0_ - muInf_)/pow((scalar(1) +
pow((k_*shearRate_),b_)),a_));
etaS_ = limitedAlpha1*etaL_ + (scalar(1) - limitedAlpha1)*etaS2_;
}
else
{
etaS_ = limitedAlpha1*etaS1_ + (scalar(1) - limitedAlpha1)*etaS2_;
}
etaP_ = limitedAlpha1*etaP1_ + (scalar(1) - limitedAlpha1)*etaP2_;
}
Foam::viscoelasticIncompressibleTwoPhaseMixture::viscoelasticIncompressibleTwoPha
seMixture
(
        const volVectorField& U,
        const surfaceScalarField& phi
)
:
IOdictionary
(
        IOobject
        (
        "transportProperties",
        U.time().constant(),
        U.db(),
        IOobject::MUST_READ_IF_MODIFIED,
```

```
            IOobject::NO_WRITE
            )
),
twoPhaseMixture(U.mesh(), *this),
nuModel1_
(
viscosityModel::New
(
            "nu1",
            subDict(phase1Name_),
            U,
            phi
)
),
nuModel2_
(
            viscosityModel::New
            (
             "nu2",
            subDict(phase2Name_),
            U,
            phi
            )
),
rho1_("rho", dimDensity, nuModel1_->viscosityProperties().lookup("rho")),
rho2_("rho", dimDensity, nuModel2_->viscosityProperties().lookup("rho")),
rheologyType_(nuModel1_->viscosityProperties().lookup("rheology")),
viscosityType_(nuModel1_->viscosityProperties().lookup("viscosityType")),
etaS1_(nuModel1_->viscosityProperties().lookup("etaS")),
etaS2_(nuModel2_->viscosityProperties().lookup("etaS")),
etaP1_(nuModel1_->viscosityProperties().lookup("etaP")),
etaP2_(nuModel2_->viscosityProperties().lookup("etaP")),
lambda1_(nuModel1_->viscosityProperties().lookup("lambda")),
lambda2_(nuModel2_->viscosityProperties().lookup("lambda")),
Alpha1_(nuModel1_->viscosityProperties().lookup("Alpha")),
Alpha2_(nuModel2_->viscosityProperties().lookup("Alpha")),
mu0_(nuModel1_->viscosityProperties().lookup("mu0")),
muInf_(nuModel1_->viscosityProperties().lookup("muInf")),
a_(nuModel1_->viscosityProperties().lookup("a")),
b_(nuModel1_->viscosityProperties().lookup("b")),
 k_(nuModel1_->viscosityProperties().lookup("k")),
U_(U),
phi_(phi),
etaS_
```

```
        (
                IOobject
                (
                "etaS",
                U_.time().timeName(),
                 U_.db()
                ),
                 U_.mesh(),
                 etaS1_
                ),
        etaP_
        (
                IOobject
                (
                "etaP",
                 U_.time().timeName(),
                U_.db()
                ),
                U_.mesh(),
                etaP1_
        ),
        tau_
        (
                 IOobject
                (
                "tau",
                U.time().timeName(),
                U.mesh(),
                IOobject::MUST_READ,
                IOobject::AUTO_WRITE
                ),
                U.mesh()
        )
        {
        calcEta();
        }
        Foam::tmp<Foam::fvVectorMatrix>Foam::viscoelasticIncompressibleTwoPhaseMixture:
:divTau() const
        {
        volScalarField etaPEff = etaP_;
        volTensorField S = tau_ - etaPEff* fvc::grad(U_);
        return
        (
```

```
        fvc::div(S, "div(tau)")
        + fvm::laplacian( (etaPEff + etaS_), U_, "laplacian(etaPEff+etaS,U)")
        );
        }
        void Foam::viscoelasticIncompressibleTwoPhaseMixture::correct()
        {
        calcEta();
        volTensorField L = fvc::grad(U_);
        volTensorField C = tau_ & L;
        volSymmTensorField twoD = twoSymm(L);
        const volScalarField limitedAlpha1
        (
                min(max(alpha1_, scalar(0)), scalar(1))
        );
        volScalarField lambda_  = limitedAlpha1*lambda1_ + (scalar(1) - limitedAlpha1)*
lambda2_;
        if (rheologyType_ == "Oldroyd-B" )
        {
        fvSymmTensorMatrix tauEqn
        (
                fvm::ddt(tau_)
                + fvm::div(phi_, tau_)
        ==
                etaP_/lambda_*twoD
                + twoSymm(C)
                        - fvm::Sp(1/lambda_, tau_)
        );
        tauEqn.relax();
        tauEqn.solve();
                volTensorField S = tau_ - etaP_* fvc::grad(U_);
        }
        else
        {
        volScalarField Alpha_  = limitedAlpha1*Alpha1_ + (scalar(1) - limitedAlpha1)*Alpha2_;

        fvSymmTensorMatrix tauEqn
        (
                (lambda_*etaP_)*fvm::ddt(tau_)
                + (lambda_*etaP_)*(fvm::div(phi_, tau_))
                ==
                sqr(etaP_)*twoD
                + (lambda_*etaP_)*twoSymm(C)
                - (Alpha_*lambda_)*(symm(tau_ & tau_))
                - fvm::Sp(etaP_, tau_)
```

```
);
        tauEqn.relax();
        tauEqn.solve();
}
}
bool Foam::viscoelasticIncompressibleTwoPhaseMixture::read()
{
        if (regIOobject::read())
        {
                if
                (
                 nuModel1_().read
                (
                subDict(phase1Name_ == "1" ? "phase1": phase1Name_)
                )
                && nuModel2_().read
                (
                subDict(phase2Name_ == "2" ? "phase2": phase2Name_)
                )
                )
        {
nuModel1_->viscosityProperties().lookup("rho") >> rho1_;
nuModel2_->viscosityProperties().lookup("rho") >> rho2_;
nuModel1_->viscosityProperties().lookup("rheology") >> rheologyType_;
nuModel1_->viscosityProperties().lookup("etaS") >> etaS1_;
nuModel2_->viscosityProperties().lookup("etaS") >> etaS2_;
nuModel1_->viscosityProperties().lookup("etaP") >> etaP1_;
nuModel2_->viscosityProperties().lookup("etaP") >> etaP2_;
nuModel1_->viscosityProperties().lookup("lambda") >> lambda1_;
nuModel2_->viscosityProperties().lookup("lambda") >> lambda2_;
nuModel1_->viscosityProperties().lookup("Alpha") >> Alpha1_;
nuModel2_->viscosityProperties().lookup("Alpha") >> Alpha2_;
nuModel1_->viscosityProperties().lookup("viscosityType") >> viscosityType_;
nuModel1_->viscosityProperties().lookup("mu0") >> mu0_;
 nuModel1_->viscosityProperties().lookup("muInf") >> muInf_;
nuModel1_->viscosityProperties().lookup("a") >> a_;
nuModel1_->viscosityProperties().lookup("b") >> b_;
nuModel1_->viscosityProperties().lookup("k") >> k_;
return true;
}
else
{
return false;
```

```
}
}
else
{
 return false;
}
}
```

We also add the viscoelasticIncompressibleTwoPhaseMixture.H file,

```
#ifndef viscoelasticIncompressibleTwoPhaseMixture_H
#define viscoelasticIncompressibleTwoPhaseMixture_H
#include "incompressible/transportModel/transportModel.H“
#include "incompressible/viscosityModels/viscosityModel/viscosityModel.H“
#include "twoPhaseMixture.H“
#include "IOdictionary.H“
#include "dimensionedScalar.H“
#include "volFields.H“
#include "fvm.H“
#include "fvc.H“
#include "fvMatrices.H“
namespace Foam
{
class viscoelasticIncompressibleTwoPhaseMixture
:
        public IOdictionary,
        public transportModel,
        public twoPhaseMixture
{
protected:
        autoPtr<viscosityModel> nuModel1_;
        autoPtr<viscosityModel> nuModel2_;
        dimensionedScalar rho1_;
        dimensionedScalar rho2_;
        const volVectorField& U_;
        const surfaceScalarField& phi_;
        word rheologyType_;
        dimensionedScalar etaS1_;
        dimensionedScalar etaS2_;
        dimensionedScalar etaP1_;
        dimensionedScalar etaP2_;
        dimensionedScalar lambda1_;
        dimensionedScalar lambda2_;
```

```
            dimensionedScalar Alpha1_;
             dimensionedScalar Alpha2_;
            dimensionedScalar mu0_;
            dimensionedScalar muInf_;
            dimensionedScalar a_;
            dimensionedScalar b_;
            dimensionedScalar k_;
            volScalarField etaS_;
            volScalarField etaP_;
            volSymmTensorField tau_;
            void calcEta();
            public:
            TypeName("viscoelasticIncompressibleTwoPhaseMixture");
viscoelasticIncompressibleTwoPhaseMixture
(
const volVectorField& U,
const surfaceScalarField& phi
);
virtual ~viscoelasticIncompressibleTwoPhaseMixture()
{}
const viscosityModel& nuModel1() const
{
return nuModel1_();
}
const viscosityModel& nuModel2() const
{
            return nuModel2_();
}
const dimensionedScalar& rho1() const
{
            return rho1_;
}
const dimensionedScalar& rho2() const
{
            return rho2_;
};
const volVectorField& U() const
{
            return U_;
}
 virtual tmp<volScalarField> nu() const
{
            notImplemented("virtual tmp<volScalarField> nu() const");
```

```
            return tmp<volScalarField>();
        }
        virtual tmp<scalarField> nu(const label patchi) const
          {

            notImplemented(" virtual tmp<volScalarField> nu(Foam::label) const");
            return tmp<scalarField>();
        }
        virtual void correctViscosity()
        {
            calcEta();
        }
        virtual void correct();
        virtual tmp<volSymmTensorField> tau() const
        {
            return tau_;
        }
        virtual tmp<fvVectorMatrix> divTau() const;
        virtual bool read();
        };
        }
        #endif
```

Then, we create the viscoelasticImmiscibleIncompressibleTwoPhase-Mixture folder and to add the viscoelasticImmiscibleIncompressibleTwoPhase-Mixture.C file,

```
#include "viscoelasticImmiscibleIncompressibleTwoPhaseMixture.H“
Foam::viscoelasticImmiscibleIncompressibleTwoPhaseMixture::
viscoelasticImmiscibleIncompressibleTwoPhaseMixture
(
    const volVectorField& U,
    const surfaceScalarField& phi
)
:
viscoelasticIncompressibleTwoPhaseMixture(U, phi),
 viscoelasticInterfaceProperties(alpha1(), U, *this)
{}
```

and the viscoelasticImmiscibleIncompressibleTwoPhase-Mixture.H file,

```
#ifndef viscoelasticImmiscibleIncompressibleTwoPhaseMixture_H
#define viscoelasticImmiscibleIncompressibleTwoPhaseMixture_H
#include "viscoelasticIncompressibleTwoPhaseMixture.H“
#include "viscoelasticInterfaceProperties.H“
namespace Foam
{
class viscoelasticImmiscibleIncompressibleTwoPhaseMixture
:
public viscoelasticIncompressibleTwoPhaseMixture,
public viscoelasticInterfaceProperties
{
public:
viscoelasticImmiscibleIncompressibleTwoPhaseMixture
(
        const volVectorField& U,
        const surfaceScalarField& phi
);
virtual ~viscoelasticImmiscibleIncompressibleTwoPhaseMixture()
{}
virtual void correct()
{
        viscoelasticIncompressibleTwoPhaseMixture::correctViscosity();
        viscoelasticInterfaceProperties::correct();
}
virtual void correctStress()
{
        viscoelasticIncompressibleTwoPhaseMixture::correct();
}
};
}
#endif
```

Afterwards, we rename the file interfaceProperties (that exists in the original transport model) as viscoelasticInterfaceProperties, Then, we rename the file interfaceProperties.C as viscoelasticInterfaceProperties.C and we add the following lines,

```
{
        calculateK();
}
  //----------------------------Modified----------------------------//
Foam::tmp<Foam::surfaceScalarField>
Foam::viscoelasticInterfaceProperties::surfaceTensionForce() const
{
        return fvc::interpolate(sigmaK())*fvc::snGrad(alpha1_);
}
Foam::tmp<Foam::volScalarField>
Foam::viscoelasticInterfaceProperties::nearInterface() const
{
        return pos(alpha1_ - 0.01)*pos(0.99 - alpha1_);
}
//-----------------------------End-----------------------------//
```

The last step is to rename the file interfaceProperties.H as viscoelasticInterfaceProperties.H and add the following lines,

```
 return sigma_*K_;
}
//----------------------------Modified----------------------------//
tmp<surfaceScalarField> surfaceTensionForce() const;
tmp<volScalarField> nearInterface() const;
//-----------------------------End-----------------------------//
```

Modification on the interFom solver

In the interFoam solver we just need to modify the creatFiles file, rewriting the lines 32 and 33 as,

```
Info<< "Reading transportProperties\n" << enddl;
viscoelasticImmiscibleIncompressibleTwoPhaseMixture mixture(U, phi);
```

The interFoam solver needs to be rename to viscoelasticInterFoam, as well as the file interFoam.C file should be rename to viscoelasticInterFoam.C, and to have the following code,

```
#include "fvCFD.H“
#include "CMULES.H“
#include "subCycle.H“
#include "viscoelasticImmiscibleIncompressibleTwoPhaseMixture.H“
#include "turbulenceModel.H“
#include "interpolationTable.H“
#include "pimpleControl.H“
#include "fvIOoptionList.H“
#include "fixedFluxPressureFvPatchScalarField.H“
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
int main(int argc, char *argv[])
{
#include "setRootCase.H"
#include "createTime.H"
#include "createMesh.H"
pimpleControl pimple(mesh);
#include "initContinuityErrs.H"
#include "createFields.H"
#include "readTimeControls.H"
#include "createPrghCorrTypes.H"
#include "correctPhi.H"
#include "CourantNo.H"
#include "setInitialDeltaT.H"
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
Info<< "\nStarting time loop\n" << endl;
while (runTime.run())
{
#include "readTimeControls.H"
#include "CourantNo.H"
#include "alphaCourantNo.H"
#include "setDeltaT.H"
runTime++;
Info<< "Time = " << runTime.timeName() << nl << endl;
while (pimple.loop())
{
#include "alphaControls.H"
#include "alphaEqnSubCycle.H"
mixture.correct();
 #include "UEqn.H"
while (pimple.correct())
{
#include "pEqn.H"
```

```
        }
        mixture.correctStress();
        }
        runTime.write();
        Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
        << "  ClockTime = " << runTime.elapsedClockTime() << " s"
        << nl << endl;
        }
        Info<< "End\n" << endl;
        return 0;
        }
```

References

[1] C. J. Greenshields, “OpenFoam User Guide,” 2015.

[2] P. Kennedy, “Development of Injection Molding Simulation,” in *Injection Molding Technology and Fundamentals*, Munich, Hanser Publishers, 2009, pp. 553-558.

[3] S. Posey, “Computational Resource Centre,” 05 June 2013. [Online]. Available: https://www.acrc.a-star.edu.sg/docs/NV_ACRC_05Jun13-Part3.pdf. [Accessed 06 June 2016].

[4] S. M. Damián, “Description and utilization of interFoam multiphase solver”.

[5] H. Zhou, Z. Hu, Y. Zhang and D. Li, “Numerical Implementation for the Filling and Packing Simulation,” in *Computer Modeling for Injection Molding: Simulation, Optimization, and Control*, Hubei, John Wiley & Sons, Inc., 2013, pp. 71-128.

[6] M. Kamal and S. Kenig, “The Injection Molding of Thermoplastics Part I: Theoretical Model,” *Polymer Engineering and Science ,* vol. 12, no. 4, pp. 294-301, 1972.

[7] J. L. Berger and C. Gogos, “A Numerical Simulation of the Cavity Filling Process with PVC in Injection Molding,” *Polymer Engineering and Science,* vol. 13, no. 2, pp. 102-112, 1973.

[8] G. Williams and H. A. Lord, “Mold-Filling Studies for the Injection Molding of Thermoplastic Materials. Part I: The Flow of Plastic Materials in Hot- and Cold-Walled Circular Channels,” *Polymer Engineering and Science,* vol. 15, no. 8, pp. 553-568, 1975.

[9] P. C. Wu, C. Huang and C. G. Gogos, “Simulation of the Mold-Filing Process,” *Polymer Engineering and Science,* vol. 14, no. 3, pp. 223-230, 1974.

[10] H. A. Lord and G. Williams, “Mold-Filling Studies for the Injection Molding of Thermoplastic Materials. Part II: The Transient Flow of Plastic Materials in the Cavities of Injection-Molding Dies,” *Polymer Engineering and Science,* vol. 15, no. 569-582, 1975.

[11] J. A. P. de Oliveira, *Análise numérica de tensões induzidas pelo escoamento não isotérmico de um polímero no preenchimento de cavidades de paredes finas,* Porto Alegre, 2012.

[12] S. Richardson, “Hele Shaw flows with a free boundary produced by the injection of fluid into a narrow channel,” *Journal of Fluid Mechanics,* vol. 56, no. 4, pp. 609-618, 1972.

[13] E. Broyer, Z. Tadmor and C. Gutfinger, “Filling of a Rectangular Channel with a Newtonian Fluid,” *Israel Journal of Technology,* vol. 11, no. 4, pp. 189-193, 1973.

[14] J. L. White, “Fluid Mechanical Analysis of Injection Mold Filling,” *Polymer Engineering and Science,* vol. 15, no. 1, pp. 44-50, 1975.

[15] M. R. Kamal, Y. Kuo and P. H. Doan, “The Injection Molding Behavior of Thermoplastics in Thin Rectangular Cavities,” *Polimer Engineering and Science,* vol. 15, no. 12, pp. 863-868, 1975.

[16] C. A. Hieber and S. F. Shen, “A Finite-element/Finite-difference Simulation of the Injection-Molding Fillig Process,” *Journal of Non-Newtonian Fluid Mechanics,* vol. 7, no. 1, pp. 1-32, 1980.

[17] J. F. Hétu, D. M. Gao, A. Garcia-Rejon and G. Salloum, “3D Finite Element Method for the Simulation of the Filling Stage in Injection Molding,” *Polymer Enginneering and Science,* vol. 38, no. 2, pp. 223-236, 1998.

[18] E. Pichelin and T. Coupez, “Finite element solution of the 3D mold filling problem for viscous incompressible fluid,” *Computer Methods in Applied Mechanics and Engineering,* vol. 163, no. 4, pp. 359-371, 1998.

[19] R.-Y. Chang and W.-H. Yang, “Numerical simulation of mold filling in injection molding using a three-dimensional finite volume approach,” *International Journal for Numerical Methods in Fluids,* vol. 37, no. 2, pp. 125-148, 2001.

[20] B. Yan, H. Zhou and D. Li, “Numerical simulation of the filling stage for plastic injection moulding based on the Petrov–Galerkin methods,” *Journal of Engineering Manufacture,* vol. 221, no. 8, pp. 1573-1577, 2007.

[21] M. D. Azaman, S. M. Sapuan, S. Sulaiman, E. S. Zainudin and A. Khalina, “Numerical simulation analysis of the in-cavity residual stress

distribution of lignocellulosic (wood) polymer composites used in shallow thin-walled parts formed by the injection moulding process," *Elsevier,* vol. 55, no. 1, pp. 381-386, 2013.

[22] N. H. Kim and A. I. Isayev, "Birefringence and interface in sequential co-injection molding of amorphous polymers: Simulation and experiment," *Polymer Engineering & Science,* vol. 55, no. 1, pp. 88-106, 2015.

[23] W. Wang, X. Li and X. Han, "Numerical Simulation and Experimental Verification of the Filling Stage in Injection Molding," *Polymer Engineering and Science,* vol. LVI, no. 8, pp. 42-51, 2012.

[24] N. Phan-Thien and R. Tanner, "A new constitutive equation derived from network theory," *Journal of Non-Newtonian Fluid Mechanics (1977),* no. 2, pp. 353-365, 1976.

[25] H. Giesekus, "A simple constitutive equation for polymer fluids based on the concept of deformation-dependent tensor mobility," *Journal of Non-Newtonian Fluid Mechanics (1982),* no. 11, pp. 69-109, 1981.

[26] R. B. Bird, P. J. Dotson and N. L. Johnson, "Polymer solution rheology based on a finitely extensible bead-spring chain model," *Journal of Non-Newtonian Fluid Mechanics (1980),* no. 7, pp. 213-235, 1980.

[27] H. Zhou, Computer modeling for injection molding: simulation, optimization, and control, China: John Wiley & Sons, Inc., 2013.

[28] C. W. Hirt and B. D. Nicholas, "Volume of Fluid (VOF) Method for the Dynamics of Free Boundaries," *Journal of Computacional Physics (1981),* no. 39, pp. 201-225, 1979.

[29] F. Moukalled, L. Mangani and M. Darwish, "The Discretization Process," in *The Finite Volume Method in Computational Fluid Dynamics An Advanced Introduction with OpenFOAM and Matlab*, Switzerland, Springer, 2016, pp. 85-101.

[30] H. K. Versteeg and W. Malalasekera, An introduction to Computational Fluid Dynamics The Finite Volume Method, Essex: Longman Scientific & Technical, 1995.

[31] L. J. L. Ferrás, "Condições Fronteiras nas Interfaces - Descontinuidade no Campo de Velocidades," (Master's thesis), Porto, 2007.

[32] H. Munstedt, “Rheological properties and molecular structure of polymer melts,” *Soft Matter,* vol. VII, no. 6, pp. 2273-2283, 2011.

[33] C. Y. Khor, Z. M. Ariff, F. C. Ani, M. A. Mujeebu, M. K. Abdullah, M. Z. Abdullah and M. A. Joseph, “Three-dimensional numerical and experimental investigations on polymer rheology in meso-scale injection molding,” *Elsevier,* vol. I, no. 37, p. 131_139, 2010.

[34] G. Tie, L. Dequn and Z. Huamin, “Three-dimensional finite element method for the filling simulation of injection molding,” *Engineering with Computers,* vol. I, no. 21, pp. 289-295, 2006.

[35] H. Zhou, B. Yan and Y. Zhang, “3D filling simulation of injection molding based on the PG method,” *Journal of Materials Processing Thechnology,* vol. I, no. 204, pp. 475-480, 2008.

[36] B. Yang, J. Ouyang, C. Liu and Q. Li, “Simulation of Non-isothermal Injection Molding for a Non-Newtonian Fluid by Level Set Method,” *Chinese Journal of Chemical Engineering,* vol. IV, no. 18, pp. 600-608, 2010.

[37] “The Engineering ToolBox,” [Online]. Available: http://www.engineeringtoolbox.com/air-properties-d_156.html. [Accessed 31 May 2016].

[38] H. Zhou, Z. Hu and D. Li, “Mathematical models for the filling and packing simulation,” in *Computer modeling for injection molding*, Hubei, John Wiley & Sons, Inc., 2013, pp. 51-68.

[39] L. Silva, J.-F. Agassant and T. Coupez, “Three-Dimensional Injection Molding Simulation,” in *Injection Molding: Technology and Fundamentals*, Munich, Hanser Publishers, 2009, pp. 599-650.

[40] H. Weller, C. Greenshields and M. Janssens, “The OpenFOAM Foundation,” OpenFOAM, 10 December 2014. [Online]. Available: http://openfoam.org/release/1-0/. [Accessed 31 May 2016].