Cite as: Afshar Ghasemi, K.: Implementing a non-isothermal interPhaseChangeFoam solver with a thermodynamic cavitation model. In Proceedings of CFD with OpenSource Software, 2023, Edited by Nilsson. H., http://dx.doi.org/10.17196/OS_CFD#YEAR_2023

CFD with OpenSource software

A course at Chalmers University of Technology
Taught by Håkan Nilsson

Implementing a non-isothermal interPhaseChangeFoam solver with a thermodynamic cavitation model

Developed for OpenFOAM-v2212

Author:
Keivan Afshar Ghasemi
Norwegian University of Science and Technology
keivan.a.ghasemi@ntnu.no

Peer reviewed by:
Prof. David Robert Emberson
Dr. Saeed Salehi
Martina Nobilo

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

January 15, 2024

Learning outcomes

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

The reader will learn:

How to use it:

- How to use the pre-existing `interPhaseChangeFoam` solver, in conjunction with the `interCondensatingEvaporatingFoam` solver, along with their phase change models to simulate a scenario featuring non-isothermal cavitation marked by considerable heat transfer rates.

The theory of it:

- The theory of `interPhaseChangeFoam` solver and its corresponding cavitation models.
- The theory of `interCondensatingEvaporatingFoam` solver and its corresponding boiling models.
- The main difference between a cavitation model and a boiling model.
- The theory of `ZwartExtended` cavitation model, an improved Zwart model for thermodynamic cavitations.

How it is implemented:

- How the phase change models (cavitation and boiling models) are implemented in OpenFOAM.
- How these phase change models are incorporated into the phase change solvers (`interPhaseChangeFoam` and `interCondensatingEvaporatingFoam` solvers).

How to modify it:

- How to integrate the Plesset-Zwick bubble dynamics equation to the Zwart cavitation model to develop `ZwartExtended` cavitation model.
- How to modify the `interCondensatingEvaporatingFoam` solver so that it can be considered as the non-isothermal counterpart to the `interPhaseChangeFoam` solver.
- How to implement the `thermalInterPhaseChangeFoam` solver and its corresponding `ZwartExtended` cavitation model.

Prerequisites

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

- How to run standard document tutorials like `condensatingVessel` and `cavitatingBullet` tutorials.
- Fundamentals of Computational Methods for Fluid Dynamics, Book by J. H. Ferziger and M. Peric.
- Basic C++ programming and how to customize an OpenFOAM solver and do top-level application programming.
- Fundamentals of bubble dynamics and cavitation modelings, and their numerical scheme implementations.

Contents

Nomenclature

Acronyms

`iCEF`	interCondensatingEvaporatingFoam
`iPCF`	interPhaseChangeFoam
`tIPCF`	thermalInterPhaseChangeFoam
CFD	Computational Fluid Dynamics
OpenFOAM	Open-source Field Operation and Manipulation
VOF	Volume of Fluid

English symbols

$\dot{m}$	Mass transfer rate	$\mathrm{kg}/(\mathrm{m}^3 \cdot \mathrm{s})$
$\Re$	Universal gas constant	$\mathrm{J}/(\mathrm{K} \cdot \mathrm{mol})$
a	Thermal diffusivity	m^2/s
c_p	Specific heat capacity at constant pressure	$\mathrm{J}/(\mathrm{kg} \cdot \mathrm{K})$
g	Gravitational acceleration	m/s^2
L	Latent heat of vaporization	J/kg
l	Physical length scale	m
M	Molecular weight	g/mol
p	Local pressure	Pa
R	Radius	m
T	Local temperature	K
t	Time	s
u_i	Cartesian velocity component in i-direction	m/s

Greek symbols

α	Volume fraction	
δ	Kronecker delta	
κ	Thermal conductivity	$\mathrm{J}/(\mathrm{s} \cdot \mathrm{m} \cdot \mathrm{K})$
λ	Molecular mean free path	m
μ	Dynamic viscosity	$\mathrm{kg}/(\mathrm{m} \cdot \mathrm{s})$
ν	Kinematic viscosity	m^2/s
ρ	Density	kg/m^3
σ	Surface tension	N/m

Superscripts

$+$	Condensation
$-$	Vaporization/Evaporation

Subscripts

∞	Free stream
i, j, k	Cartesian directions
0	Nucleus (effective initial)

B	Bubble
C	Condensation
E	Evaporation
l	Liquid phase
nuc	Nucleation
sat	Saturation
t	Turbulent
V	Vaporization
v	Vapor phase

Introduction

In the field of fluid dynamics, "cavitation" denotes the localized isothermal vaporization that appears when the local pressure drops below the saturation vapor pressure of the liquid. This occurrence is a byproduct of escalating fluid flow velocity. The vapor-filled cavities or bubbles formed within a cavitating flow experience implosion when the local pressure surpasses the vapor pressure. Characterized by this inward process, cavitation is often described using terms such as "implosion" or "nucleus collapse" [1]. This phenomenon holds critical significance, particularly in the study of fluid mechanics and related engineering applications.

Concerning the numerical modeling of the cavitation phenomenon, various methods exist to simulate this two-phase flow. One prominent approach is the Volume of Fluid (VOF) phase-fraction based interface capturing technique, widely employed in several OpenFOAM multiphase solvers. This method, recognized as a one-fluid formulation, derives transport equations for the mixture (comprising a fraction of both liquid and vapor phases distributed within each computational cell). Nevertheless, it factors in the volume fraction of each phase separately when solving the VOF equation as well as computing different parameters of transport equations, including density and thermophysical properties [2]. To model cavitating flows using this method, the mass transfer rates between liquid and vapor phases are calculated by considering the respective volume fractions of phases. Regarding OpenFOAM, three models characterizing these mass transfer rates have already been implemented, including `Kunz`, `Merkle`, and `SchnerrSauer` models, all of which will be briefly discussed in *Section 1.2*. However, in these three cavitation models, as outlined in the previous paragraph, the cavitation phenomenon is treated as an isothermal event. Consequently, not only the solver using these models is isothermal, but also the mass transfer rates are solely computed based on the difference between local and saturation vapor pressures, emphasizing the mechanical effect in the bubble generation process. While this is generally accurate, it may fall short if the superheated liquid approaches its boiling temperature, necessitating the inclusion of thermal effects or temperature fluctuations in the mass transfer rates equations, which would be the case for thermal fluids such as refrigerants [3].

The objective of this project is to enhance OpenFOAM's capability to employ the VOF method in simulating scenarios with the conditions described earlier, specifically thermodynamic cavitating flows. Within OpenFOAM, there are two pre-existing solvers tailored for cavitating flows, i.e. `cavitatingFoam` and `interPhaseChangeFoam` solvers. However, the former relies on a different two-phase flow model rather than VOF, while the latter, as mentioned briefly in the previous paragraph, is an isothermal solver using the previously discussed cavitation models. Consequently, modifications are needed. We need to not only introduce an energy transport equation to consider temperature changes in `interPhaseChangeFoam` but also implement a new thermodynamic cavitation model. In addition to these, OpenFOAM includes a non-isothermal phase change solver named `interCondensatingEvaporatingFoam`, designed for simulating condensation and evaporation in two-phase flows using the VOF method. Fortunately, from the numerical implementation standpoint, it aligns well with the `interPhaseChangeFoam` approach, with the primary difference being the set of models used to calculate the mass transfer rates. As a result, in the subsequent chapters, the `interCondensatingEvaporatingFoam` solver will serve as the base solver for developing the new non-isothermal `interPhaseChangeFoam` solver, referred to as `thermalInterPhaseChangeFoam`.

Chapter 1

Theory

The `interCondensatingEvaporatingFoam` (`iCEF`) solver was thoroughly explored in a student tutorial featured in the proceedings of CFD with OpenSource Software, 2022 [4]. Similarly, the `interPhaseChangeFoam` (`iPCF`) solver and its associated Zwart cavitation model, to be discussed later in this chapter, were covered in the tutorials for the proceedings 2013 and 2018, respectively [5, 6]. Therefore, this report will focus on the unique aspects of the source codes of each solver, emphasizing the distinctions that allow for the incorporation of the energy transport equation from `iCEF` into `iPCF`. Concluding this chapter, the Plesset-Zwick bubble dynamics equation will be introduced, accompanied by the presentation of the new thermodynamic cavitation model. This model, formulated based on the mentioned bubble dynamics equation, essentially constitutes a modified version of the Zwart model.

1.1 Comparing the equations of `iPCF` and `iCEF` solvers

At the outset, the governing equations of the two solvers will be investigated, considering their roles as incompressible solvers tailored for systems involving two immiscible fluids undergoing phase changes. Notably, cavitation defines the phase change in `iPCF`, whereas `iCEF` addresses condensation and evaporation. To streamline this chapter, all the parameters and their units used in the following equations are defined in the *Nomenclature* section of this report. The following equations outline mass conservation and momentum transport for these distinct scenarios.

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x_j}(\rho u_j) = 0 \tag{1.1}$$

$$\frac{\partial}{\partial t}(\rho u_j) + \frac{\partial}{\partial x_j}(\rho u_i u_j) = -\frac{\partial p}{\partial x_i} + \frac{\partial}{\partial x_j}\left((\mu + \mu_t)(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} - \frac{2}{3}\frac{\partial u_k}{\partial x_k}\delta_{ij}\ )\right) + \rho g \tag{1.2}$$

However, adhering to the OpenFOAM convention, the direct implementation of these two equations is not followed. Instead, there is an equation governing velocity (`UEqn`) and another for pressure (`pEqn`), a combination of Eq. (1.1) and Eq. (1.2); their implementations will be investigated later in this chapter. Besides these, an additional equation governs the VOF model, determining the volume fraction of `phase1`, i.e. liquid in this context, based on the mass transfer rates between liquid and vapor phases. Given that liquid serves as the carrier fluid (`phase1`) here, $\dot{m}^+$ corresponds to condensation, while $\dot{m}^-$ signifies vaporization/evaporation.

$$\frac{\partial(\rho_l \alpha_l)}{\partial t} + \frac{\partial(\rho_l \alpha_l u_j)}{\partial x_j} = \dot{m}^+ + \dot{m}^- \tag{1.3}$$

It is needless to state that the sum of phase fractions in the VOF model equals one ($\alpha_l + \alpha_v = 1$). The calculation of mass transfer rates, $\dot{m}^+$ and $\dot{m}^-$ in Eq. (1.3), depends on the selected phase change model, a topic that will be explored in greater detail in the subsequent sections. However, Eq. (1.3) is

also not implemented directly in OpenFOAM. The numerical derivation of the actual implementation of this equation is thoroughly explained at the beginning of *Section 2.2* of reference [4]; according to which, the OpenFOAM volume fraction equation (`alphaEqn`) is stated as

$$\frac{\partial \alpha_{\mathrm{l}}}{\partial t} + \frac{\partial (\alpha_{\mathrm{l}} u_j)}{\partial x_j} - \alpha_{\mathrm{l}} \frac{\partial u_j}{\partial x_j} = \alpha_l (\dot{V}_{\mathrm{V}} - \dot{V}_{\mathrm{C}}) + \dot{V}_{\mathrm{C}} \ , \tag{1.4}$$

where $\dot{V}_{\mathrm{C}}$ and $\dot{V}_{\mathrm{V}}$ are calculated as

$$\begin{cases} \dot{V}_{\mathrm{C}} = \left(\dfrac{1}{\rho_{\mathrm{l}}} - \alpha_{\mathrm{l}} (\dfrac{1}{\rho_{\mathrm{l}}} - \dfrac{1}{\rho_{\mathrm{v}}}) \right) \dot{m}_{\mathrm{C}} \\ \dot{V}_{\mathrm{V}} = \left(\dfrac{1}{\rho_{\mathrm{l}}} - \alpha_{\mathrm{l}} (\dfrac{1}{\rho_{\mathrm{l}}} - \dfrac{1}{\rho_{\mathrm{v}}}) \right) \dot{m}_{\mathrm{V}} \ . \end{cases} \tag{1.5}$$

In the above equations, subscripts $_{\mathrm{C}}$ and $_{\mathrm{V}}$ are used for $\dot{V}$ to emphasize the connection of this equation with the `iPCF` solver and, consequently, with condensation and vaporization phenomena.

In addition to these three equations (continuity, momentum, and volume fraction), the `iCEF` solver incorporates an energy transport equation (`TEqn`, based on temperature), Eq. (1.6), to account for temperature variations. Notably, subscripts $_{\mathrm{C}}$ and $_{\mathrm{E}}$ are employed to underscore the association of this equation with the `iCEF` solver and, consequently, with condensation and evaporation phenomena.

$$\frac{\partial (\rho c_{\mathrm{p}} T)}{\partial t} + \frac{\partial}{\partial x_j} (\rho u_j c_{\mathrm{p}} T) = \frac{\partial}{\partial x_j} (\kappa \frac{\partial T}{\partial x_j}) - (\dot{m}_{\mathrm{C}} + \dot{m}_{\mathrm{E}}) L \tag{1.6}$$

In all the previously mentioned equations, the variables p, T, and all other unsubscripted parameters refer to the mean values (computational cell center). Therefore, all thermophysical properties of the mixture, such as ϕ, can be computed based on the properties of the liquid and vapor phases and their volume fractions, as

$$\phi = \alpha_{\mathrm{l}} \phi_{\mathrm{l}} + \alpha_{\mathrm{v}} \phi_{\mathrm{v}} = \alpha_{\mathrm{l}} \phi_{\mathrm{l}} + (1 - \alpha_{\mathrm{l}}) \phi_{\mathrm{l}} \ . \tag{1.7}$$

1.2 iPCF and iCEF phase change models

Henceforth, the phase change models of the `iPCF` solver will be referred to as cavitation models, and those of the `iCEF` solver will be denoted as boiling models. In both of these model classifications, the mass transfer rates have been computed for the aforementioned transport equations, i.e. `pEqn`, `TEqn`, and `alphaEqn`. However, there are two principal distinctions between these two types. First of all, as described in the *Introduction* chapter, the cavitation models treat the phase change phenomenon as an *isothermal* process, focusing on the inertia (mechanical) effect caused by the disparity between saturation and local pressures in their simulation equations. The saturation pressure, p_{sat}, is also considered constant in their corresponding equations. Conversely, the boiling models consider the phase change phenomenon as an *isobaric* process, concentrating on the heat transfer (thermal) effect arising from the difference between saturation and local temperatures in their simulation equations. The saturation temperature, T_{sat}, is also considered constant in their corresponding equations. Another key difference is from a mathematical perspective. Given that the `iCEF` solver, and consequently its boiling models, must solve an energy transport equation, the implementation of the boiling models includes an additional member function, `TSource()`, corresponding to the source term of Eq. (1.6), i.e. $(\dot{m}_{\mathrm{C}} + \dot{m}_{\mathrm{E}})L$.

Initially, to ensure that none of the available phase change models in OpenFOAM align with the goals of this project, all of them have been investigated in detail. As a result, their condensation and vaporization/evaporation rate equations are briefly outlined in the following section. It is evident that none of these models has taken into account the superposed effect of the thermal and mechanical impacts. It is also worth mentioning that the rates mentioned below are equivalent to the return values of `mDotAlphal()` member function of each of these models in OpenFOAM that eventually will be used to calculate the terms associated with the mass transfer rates in the volume fraction equation.

1.2.1 Cavitation models

- **Kunz**

  Originally designed to analyze sheet- and super-cavitation for predicting cavitating flows over blunt bodies, utilizes two empirical constants, C_{V} and C_{C}, which should be set according to each case. Additionally, it considers properties of the bulk flow, including the mean flow time scale, t_{∞}, equal to the ratio of blunt body diameter to free stream velocity, U_{∞} [7]. The OpenFOAM implementation of this model differs slightly from the original version.

$$\begin{cases} \dot{m}_{\mathrm{V}} = \dfrac{C_{\mathrm{V}}\rho_{\mathrm{v}}}{(\frac{1}{2}\rho_{\mathrm{l}}U_{\infty}^2)t_{\infty}} \min\left[p - p_{\mathrm{sat}}, 0\right] \\ \dot{m}_{\mathrm{C}} = \dfrac{C_{\mathrm{C}}\rho_{\mathrm{v}}}{t_{\infty}}\alpha_l^2 \dfrac{\max\left[p - p_{\mathrm{sat}}, 0\right]}{\max\left[p - p_{\mathrm{sat}}, 0.01p_{\mathrm{sat}}\right]} \end{cases} \tag{1.8}$$

- **Merkle**

  Initially designed for predicting sheet cavitations. The `Kunz` model employs the same vaporization rate as the `Merkle` model. So, as anticipated, this model also relies on two empirical constants and the same bulk flow properties [8].

$$\begin{cases} \dot{m}_{\mathrm{V}} = \dfrac{C_{\mathrm{V}}\rho_{\mathrm{l}}}{(\frac{1}{2}\rho_{\mathrm{v}}U_{\infty}^2)t_{\infty}} \min\left[p - p_{\mathrm{sat}}, 0\right] \\ \dot{m}_{\mathrm{C}} = \dfrac{C_{\mathrm{C}}}{(\frac{1}{2}U_{\infty}^2)t_{\infty}} \max\left[p - p_{\mathrm{sat}}, 0\right] \end{cases} \tag{1.9}$$

- **SchnerrSauer**

  Originally developed for simulating transient cavitations. This model, the most recent among these cavitation models, which will be used for the baseline test case, employs the simplified Rayleigh-Plesset bubble dynamics equations, similar to the Zwart model, to be discussed later, for calculating mass transfer rates [9]. In this model, apart from the empirical constants, calculations involve a nucleation site volume fraction, α_{nuc}, and a bubble radius, R_{B}. Detailed equations for these two parameters can be found in *Section 1.1.1* of reference [6].

$$\begin{cases} \dot{m}_{\mathrm{V}} = C_{\mathrm{V}}(1 + \alpha_{\mathrm{nuc}} - \alpha_{\mathrm{l}})\alpha_{\mathrm{l}} \dfrac{3\rho_{\mathrm{l}}\rho_{\mathrm{v}}}{\rho R_{\mathrm{B}}} \sqrt{\dfrac{2}{3\rho_{\mathrm{l}}} \dfrac{1}{|p - p_{\mathrm{sat}} + 0.01p_{\mathrm{sat}}|}} \min\left[p - p_{\mathrm{sat}}, 0\right] \\ \dot{m}_{\mathrm{C}} = C_{\mathrm{C}}\alpha_{\mathrm{l}}^2 \dfrac{3\rho_{\mathrm{l}}\rho_{\mathrm{v}}}{\rho R_{\mathrm{B}}} \sqrt{\dfrac{2}{3\rho_{\mathrm{l}}} \dfrac{1}{|p - p_{\mathrm{sat}} + 0.01p_{\mathrm{sat}}|}} \max\left[p - p_{\mathrm{sat}}, 0\right] \end{cases} \tag{1.10}$$

1.2.2 Boiling models

- **constant**

  The implementation of this model in OpenFOAM is based on the Lee phase change model. It employs two empirical coefficients, C_{E} and C_{C}, calculated from the ratio between well-known mass transfer intensity factors, r_{E} and r_{C}, and the saturation temperature, T_{sat} [4].

$$\begin{cases} \dot{m}_{\mathrm{E}} = -C_{\mathrm{E}}\rho_{\mathrm{l}} \max\left[T - T_{\mathrm{sat}}, 0\right] \\ \dot{m}_{\mathrm{C}} = C_{\mathrm{C}}\rho_{\mathrm{v}} \max\left[T_{\mathrm{sat}} - T, 0\right] \end{cases} \tag{1.11}$$

- **interfaceHeatResistance**

  The implementation of this model in OpenFOAM is based on the film boiling model of Hardt [10]. It utilizes the thermal transmittance of the fluid, R, as well as the vapor-liquid interface area, $A_{\mathrm{interface}}$, calculated based on $\alpha = 0.5$.

$$\begin{cases} \dot{m}_{\mathrm{E}} = -\dfrac{A_{\mathrm{interface}}R}{L(\alpha_{\mathrm{l}} + \mathrm{SMALL})} \max\left[T - T_{\mathrm{sat}}, 0\right] \\ \dot{m}_{\mathrm{C}} = \dfrac{A_{\mathrm{interface}}R}{L(\alpha_{\mathrm{v}} + \mathrm{SMALL})} \max\left[T_{\mathrm{sat}} - T, 0\right] \end{cases} \tag{1.12}$$

The reason that the $\dot{m}_\mathrm{E}$ equations include a minus sign while the $\dot{m}_\mathrm{V}$ equations do not is due to the use of the maximum function for both condensation and evaporation rates in boiling models. This is achieved by changing the position of T and T_sat, making the term $T - T_\mathrm{sat}$ positive for both equations. In contrast, cavitation models use the minimum function for the vaporization rate, resulting in the term $p - p_\mathrm{sat}$ being a negative term for the vaporization equation. Therefore, there is no need to add the minus sign artificially to the equation.

It is also essential to note that the empirical constants for the models mentioned, denoted as C_C and $C_\mathrm{V/E}$, are specific to each model and test case. The utilization of the same symbols across models does not imply uniform values or units for these constants.

1.3 A thermodynamic cavitation model (ZwartExtended)

In the field of bubble dynamics, a renowned equation, namely the Rayleigh-Plesset equation, is employed to predict alterations in bubble radius under the influence of a pressure differential. This equation relies on the thermophysical characteristics of the surrounding liquid, encompassing parameters such as kinematic viscosity and surface tension. The equation is expressed as [1]

$$\frac{p_\mathrm{v} - p}{\rho_\mathrm{l}} = R\frac{\mathrm{d}^2 R_\mathrm{B}}{\mathrm{d}t^2} + \frac{3}{2}\left(\frac{\mathrm{d}R_\mathrm{B}}{\mathrm{d}t}\right)^2 + \frac{4\nu_\mathrm{l}}{R_\mathrm{B}}\frac{\mathrm{d}R_\mathrm{B}}{\mathrm{d}t} + \frac{2\sigma}{\rho_\mathrm{l} R_\mathrm{B}} \,. \tag{1.13}$$

This equation offers versatility in simplification based on various considerations of the thermodynamic relationship between the vapor bubble and the surrounding liquid. In the subsequent discussion, we will investigate one of its most commonly applied simplifications, which gives rise to the Zwart cavitation model. Towards the end, this equation will be augmented by incorporating another bubble dynamics equation, thereby deriving a novel thermodynamic cavitation model.

1.3.1 Zwart cavitation model

In the Rayleigh-Plesset equation, Eq. (1.13), under the conditions where the influence of the surface tension term as well as the kinematic viscosity term are negligible—that is respectively, when the Weber number ($We = \rho_\mathrm{l} U^2 l/\sigma$), representing the ratio of inertia to surface tension, is significantly greater than unity, and when the Reynolds number ($Re = \rho_\mathrm{l} U/\nu$), representing the ratio of inertia to viscosity, exceeds values typical of laminar flow—the Rayleigh-Plesset equation can be simplified to what is known as initial version of the Rayleigh equation [1, 11].

$$\frac{p_\mathrm{v} - p}{\rho_\mathrm{l}} = R\frac{\mathrm{d}^2 R_\mathrm{B}}{\mathrm{d}t^2} + \frac{3}{2}\left(\frac{\mathrm{d}R_\mathrm{B}}{\mathrm{d}t}\right)^2 \tag{1.14}$$

Moreover, in the scenario where the scale effect, as denoted by the second-order term in Eq. (1.14), can be disregarded—specifically, when the Knudsen number ($Kn = \lambda/l = \nu/l\sqrt{\pi M/2\Re T}$), representing the ratio of molecular mean free path to the physical length scale, is considerably less than unity—the Rayleigh equation, Eq. (1.14), can be further simplified to [11]

$$\frac{p_\mathrm{v} - p}{\rho_\mathrm{l}} = \frac{3}{2}\left(\frac{\mathrm{d}R_\mathrm{B}}{\mathrm{d}t}\right)^2 \rightsquigarrow \frac{\mathrm{d}R_\mathrm{B}}{\mathrm{d}t} = \sqrt{\frac{2}{3}\frac{p_\mathrm{v} - p}{\rho_\mathrm{l}}} \,. \tag{1.15}$$

This relation, which only considers the inertia (mechanical effect due to pressure difference), constitutes the foundational equation in bubble dynamics, extensively employed in various cavitation models, such as `SchnerrSauer` and Zwart models. The detailed equations and implementation of the Zwart model are comprehensively explained in reference [6]. However, in this report, the mass transfer rates are reiterated below, in Eq. (1.16), as the pressure-related terms of the new thermodynamic cavitation model, which are precisely aligned with those of the Zwart model.

Preceding the introduction of the Zwart equations, in distinguishing α_nuc and R_B within this model from the `SchnerrSauer` model, it is noteworthy that, in OpenFOAM implementations, these parameters are computed based on other field properties for the `SchnerrSauer` model. Conversely, in

the conventional application of the Zwart model, these parameters are treated as constants supplied to the solver.

$$\begin{cases} \dot{m}_{\mathrm{V}} = C_{\mathrm{V}} \dfrac{3\alpha_{\mathrm{nuc}}\rho_{\mathrm{v}}}{R_{\mathrm{B}}} \sqrt{\dfrac{2}{3\rho_{\mathrm{l}}} \dfrac{1}{|p - p_{\mathrm{sat}} + 0.01 p_{\mathrm{sat}}|}} \min\left[p - p_{\mathrm{sat}}, 0\right] \\ \dot{m}_{\mathrm{C}} = C_{\mathrm{C}} \dfrac{3\rho_{\mathrm{v}}}{R_{\mathrm{B}}} \sqrt{\dfrac{2}{3\rho_{\mathrm{l}}} \dfrac{1}{|p - p_{\mathrm{sat}} + 0.01 p_{\mathrm{sat}}|}} \max\left[p - p_{\mathrm{sat}}, 0\right] \end{cases} \tag{1.16}$$

1.3.2 ZwartExtended cavitation model

The Zwart cavitation model, grounded in the simplified Rayleigh bubble dynamics equation, Eq. (1.15), predicts well the initial stage of bubble growth during the bubble generation process. In this step, the dynamics are primarily influenced by the pressure disparity between the liquid and vapor phases. However, as the bubble generation progresses ($R_{\mathrm{B}} \gg R_0$), subsequent growth pivots on the heat transfer from the superheated liquid to the liquid-vapor interface, a factor overlooked in the Zwart model. Consequently, as superheated liquid approaches its boiling temperature (either by changing temperature or pressure), conventional cavitation models, which rely on inertia, encounter challenges in accurately estimating flow behavior [3].

This is where the Plesset-Zwick bubble dynamics equation can be beneficial. Originally designed to estimate changes in bubble radius by incorporating boiling heat transfer, this model considers the vapor bubble to be at saturated vapor temperature, while the surrounding liquid maintains its own temperature. Thus, the predicted bubble dynamics unfold as [12]

$$\frac{\mathrm{d}R_{\mathrm{B}}}{\mathrm{d}t} = \sqrt{\frac{3}{\pi}} \frac{\rho_{\mathrm{l}} c_{\mathrm{p}} (T - T_{\mathrm{sat}})}{\rho_{\mathrm{v}} L} \sqrt{\frac{a_{\mathrm{l}}}{t}} \,. \tag{1.17}$$

This equation is incorporated linearly into the simplified Rayleigh equation, Eq. (1.15), by Zhang Yao et al. [3] to consider the superposed effects of the inertia and heat transfer impacts for simulating thermodynamic cavitating flows. Initially, this new cavitation model, named the ZwartExtended model in this project, was applied across a broad range of water temperatures. However, Huashi Xu et al. [11] later adapted it by applying a new set of empirical constants for simulating cavitating flows of a superheated thermo-fluid, where the model again demonstrated satisfactory performance. According to the ZwartExtended model, condensation and vaporization rates can be calculated as

$$\begin{cases} \dot{m}_{\mathrm{V}} = C_{\mathrm{V}} \dfrac{3\alpha_{\mathrm{nuc}}\rho_{\mathrm{v}}}{R_{\mathrm{B}}} \left(\sqrt{\dfrac{2}{3} \dfrac{1}{\rho_{\mathrm{l}} |p - p_{\mathrm{sat}} + \mathrm{SMALL}|}} \min\left[p - p_{\mathrm{sat}}, 0\right] - \sqrt{\dfrac{3}{\pi}} \dfrac{\rho_{\mathrm{l}} c_{\mathrm{p}}}{\rho_{\mathrm{v}} L} \sqrt{\dfrac{a_{\mathrm{l}}}{t}} \max\left[T - T_{\mathrm{sat}}, 0\right] \right) \\ \dot{m}_{\mathrm{C}} = C_{\mathrm{C}} \dfrac{3\rho_{\mathrm{v}}}{R_{\mathrm{B}}} \left(\sqrt{\dfrac{2}{3} \dfrac{1}{\rho_{\mathrm{l}} |p - p_{\mathrm{sat}} + \mathrm{SMALL}|}} \max\left[p - p_{\mathrm{sat}}, 0\right] + \sqrt{\dfrac{3}{\pi}} \dfrac{\rho_{\mathrm{l}} c_{\mathrm{p}}}{\rho_{\mathrm{v}} L} \sqrt{\dfrac{a_{\mathrm{l}}}{t}} \max\left[T_{\mathrm{sat}} - T, 0\right] \right) \end{cases} . \tag{1.18}$$

In this model, additional parameters, namely the thermal diffusivity of the liquid phase, a_{l}, and the time variable, t, have been introduced beyond those featured in previous models. The incorporation of the temporal term could potentially introduce complexities in the application of these equations. Fortunately, a favorable circumstance arises. Given that bubble growth is predominantly influenced by temperature differences during the second stage of the cavitation process ($R_{\mathrm{B}} \gg R_0$), the time variable t in these equations can be effectively substituted with a constant value representing the bubble growth time. This substitution, as declared by Zhang Yao et al., ensures enhanced calculation convergence and stability of the model [3].

1.4 Comparing the source codes of iPCF and iCEF solvers

In this section, the detailed source codes of each solver will not be explored. Given the extensive nature of this detailed explanation, it is worth noting that the complete source codes investigations are available in the references mentioned earlier [4, 5, 6]. As declared at the outset of this chapter, the focus will be on highlighting the key distinctions between the iPCF and iCEF source codes. First of all, it is crucial to recognize that the iCEF solver is tailored for dynamic meshes, while the basic version of the iPCF solver is designed for static meshes. Consequently, most of the disparities observed in the primary solver files (.C) involve declarations of additional libraries and inclusions of extra terms and conditions for calculating the flux field (phi) in the iCEF solver. Conversely, in files like alphaEqn.H, alphaControls.H, alphaEqnSubCycle.H, and UEqn.H, contents are directly imported from the iPCF solver in the options file of the Make directory, obviating the need for detailed discussion in the subsequent sections. Moreover, since the upcoming chapter will employ the iCEF solver as the foundation for constructing the thermalInterPhaseChangeFoam (tIPCF) solver, the principal emphasis of this section will be on the supplementary terms present in the iPCF solver that need to be incorporated into the iCEF solver for the development of the new solver.

1.4.1 Transport equations (UEqn, pEqn, TEqn, and alphaEqn)

To facilitate a smoother exploration of the source codes, it is beneficial to initially focus on the main transport equations. However, it should be noted that the velocity and volume fraction equations (UEqn and alphaEqn) remain identical across these solvers, as previously mentioned. Therefore, the differences between the solvers can be assessed by examining the pEqn.H files. Two noticeable distinctions are evident in the pEqn.H files. The first difference lies in the definition of the phiHbyA field, as observable in the 4th lines of the following codes.

phiHbyA definition - iCEF/pEqn.H

```
1 surfaceScalarField phiHbyA
2     (
3         "phiHbyA",
4         (fvc::interpolate(HbyA) & mesh.Sf())
5       + fvc::interpolate(rho*rAU)*fvc::ddtCorr(U, phi)
6     );
```

phiHbyA definition - iPCF/pEqn.H

```
1 surfaceScalarField phiHbyA
2     (
3         "phiHbyA",
4         fvc::flux(HbyA )
5       + fvc::interpolate(rho*rAU)*fvc::ddtCorr(U, phi)
6     );
```

However, according to the definition of fvc::flux function, the flux of HbyA is calculated through the same expression as fvc::interpolate(HbyA) & mesh.Sf(). But in iCEF solver, this calculation is written in this way because, as stated, this solver is designed for dynamic meshes. The subsequent and principal distinction found in the pEqn.H files is the definition of the p_rghEqn matrices.

p_rghEqn definition - iCEF/pEqn.H

```
1 fvScalarMatrix p_rghEqn
2     (
3         fvc::div(phiHbyA)
4       - fvm::laplacian(rAUf, p_rgh)
5       ==
6         vDotv + vDotc
7     );
```

p_rghEqn definition - iPCF/pEqn.H

```
fvScalarMatrix p_rghEqn
    (
        fvc::div(phiHbyA) - fvm::laplacian(rAUf, p_rgh)
      - (vDotvP - vDotcP)*(mixture->pSat() - rho*gh)
      + fvm::Sp(vDotvP - vDotcP, p_rgh)
    );
```

As observed, the divergence and laplacian terms remain consistent in these equations, with the sole distinction residing in their source terms. As mentioned in the preceding sections, these source terms are linked to the mass transfer rates between phases, fundamentally computed according to the `mDot()` and `mDotP()` member functions of boiling and cavitation models, respectively. The rationale behind the direct inclusion of source terms into the pressure equation in the `iCEF` solver, as opposed to the implicit implementation in the `iPCF` solver, is that in the former scenario, the source terms do not depend on pressure. Consequently, throughout the pressure equation-solving process, their values remain constant. Conversely, in the latter scenario, the source terms are contingent on pressure, necessitating their inclusion through discretization to facilitate solver capability in solving the pressure equation.

In summary, both of these terms need to be incorporated into the pressure equation of the new solver. However, due to the straightforward parsing of the condensation and vaporization rates' equations in the `ZwartExtended` model into pressure and temperature terms, the new cavitation model will feature two distinct functions: `mDotT()` and `mDotP()`. This approach aims to facilitate the discretization process of the pressure equation.

Aside from the pressure equation, the `TEqn`, which is the temperature equation, is exclusive to the `iCEF` solver, and a detailed exploration can be found in the reference [4]. The sole parameter that establishes a connection between this equation and the phase change models is the source term, calculated through the `TSource()` member function of the phase change model.

1.4.2 Solvers' main algorithms (`iPCF.C`, `iCEF.C`, and `createFields.H` files)

Apart from the general distinctions highlighted at the beginning of this section, the primary disparity between the main algorithms of these two solvers lies in how they create the pressure field. As evident in the following source codes, in the `iPCF` solver, the pressure field is directly generated using the `p_rgh` field by incorporating the hydraulic pressure, `rho*gh`. Consequently, the `iPCF` solver only necessitates reading the `p_rgh` dictionary from the `0` directory of a test case. Conversely, in the `iCEF` solver, the pressure field is constructed based on the `p` dictionary. This is essential as it serves as a required field for the thermodynamic library of the `iCEF` solver, enabling the creation of a `thermo` object specific to the temperature and pressure fields of a test case.

p definition - iCEF/createFields.H

```
// Creating e based thermo
autoPtr<twoPhaseMixtureEThermo> thermo
(
    new twoPhaseMixtureEThermo(U, phi)
);

...

volScalarField& p = thermo->p();
```

p definition - iPCF/createFields.H

```
volScalarField p
(
    IOobject
    (
        "p",
        runTime.timeName(),
        mesh,
```

```
        IOobject::NO_READ,
        IOobject::AUTO_WRITE
    ),
    p_rgh + rho*gh
);
```

Furthermore, two additional `volScalarField` and a `dimensionedScalar` objects are generated in the `createFilds.H` file of the `iCEF` solver, namely `kappaEff`, `rhoCp`, and `Prt`. These fields are necessary for the energy transport equation of the solver.

1.4.3 Phase change models and their directories

Considering the phase change directory of the `iCEF` solver, named `temperaturePhaseChangeTwoPhaseMixtures`, it contains the following directories shown in a tree diagram.

```
temperaturePhaseChangeTwoPhaseMixtures
    ├── Make
    ├── "A Boiling Model"
    ├── temperaturePhaseChangeTwoPhaseMixtures
    ├── thermoIncompressibleTwoPhaseMixture
    └── twoPhaseMixtureEThermo
```

While, the phase change directory of the `iCPF` solver, named `phaseChangeTwoPhaseMixtures`, contains these folders,

```
phaseChangeTwoPhaseMixtures
    ├── Make
    ├── "A Cavitation Model"
    └── phaseChangeTwoPhaseMixture
```

As evident from the above tree diagrams, apart from the `phaseChangeTwoPhaseMixture` and `temperaturePhaseChangeTwoPhaseMixtures` classes, which are basically the same, the phase change directory of the `iCEF` solver includes two additional directories related to temperature. Specifically, the phase change library of the `iCEF` solver, instead of utilizing the `incompressibleTwoPhaseMixture` class employed by the `iPCF` solver, employs the `thermoIncompressibleTwoPhaseMixture` class. This class is essentially inherited from the `incompressibleTwoPhaseMixture` class. Notably, it adds the capability to read values of thermophysical properties pertinent to the energy equation, such as thermal conductivities, specific heat capacities, and formation enthalpies of both phases. These values are sourced from the `transportProperties` dictionary, in addition to properties like kinematic viscosity and density, which are already being read by the parent class.

Additionally, the `twoPhaseMixtureEThermo` class, which inherits from both the `basicThermo` and `thermoIncompressibleTwoPhaseMixture` classes, is responsible for calculating the thermophysical properties of the mixture (liquid and vapor). It also computes their effective parameters, which are the combination of laminar and turbulent values. Moreover, this class reads the constant saturation temperature, denoted as `Tsat`, from the `thermophysicalProperties` dictionary of the respective test case. While the constant saturation pressure, denoted as `pSat`, is read by the `phaseChangeTwoPhaseMixture` class in the `iPCF` solver, from the `transportProperties` dictionary, which is the only difference between the main phase change classes of these two solvers.

Chapter 2

Implementing the `thermalInterPhaseChangeFoam` solver and its corresponding `ZwartExtended` cavitation model

As explained in the preceding chapter, the developments of the `tIPCF` solver and its corresponding `ZwartExtended` cavitation model are anchored in the `iCEF` solver and its `constant` boiling model. Consequently, the implementation process begins by duplicating the entire `iCEF` solver directory into the user directory and subsequently making the necessary file adjustments, including changing the name of the files, removing the redundant boiling model (`interfaceHeatResistance`), and adding the required libraries to the `Make` directory.

Preparing the base directory for implementing the new solver and its cavitation model

```
foam
cp -r --parents "applications/solvers/multiphase/interCondensatingEvaporatingFoam/"
     "$WM_PROJECT_USER_DIR"
ufoam
cd applications/solvers/multiphase
mv interCondensatingEvaporatingFoam thermalInterPhaseChangeFoam
cd thermalInterPhaseChangeFoam
mv interCondensatingEvaporatingFoam.C thermalInterPhaseChangeFoam.C
sed -i s/interCondensatingEvaporatingFoam/thermalInterPhaseChangeFoam/g thermalInterPhaseChangeFoam.C
sed -i s/interCondensatingEvaporatingFoam/thermalInterPhaseChangeFoam/g Make/files
sed -i s/FOAM_APPBIN/FOAM_USER_APPBIN/g Make/files
cd temperaturePhaseChangeTwoPhaseMixtures
rm -r interfaceHeatResistance
mv constant ZwartExtended
mv ZwartExtended/constant.C ZwartExtended/ZwartExtended.C
mv ZwartExtended/constant.H ZwartExtended/ZwartExtended.H
sed -i s/constant/ZwartExtended/g ZwartExtended/ZwartExtended.H
sed -i s/constant/ZwartExtended/g ZwartExtended/ZwartExtended.C
sed -i s/constant/ZwartExtended/g Make/files
sed -i s/interfaceHeatResistance.*//g Make/files
sed -i s/libphaseTemperatureChangeTwoPhaseMixtures/libthermalPhaseChangeTwoPhaseMixtures/g Make/files
sed -i s/FOAM_LIBBIN/FOAM_User_LIBBIN/g Make/files
cd ..
```

Following the above steps, the core directory is prepared for the integration of new lines of code. To start, the essential libraries need to be included into the `options` files of the `Make` directories for both the main solver and the `temperaturePhaseChangeTwoPhaseMixtures` library. As depicted in the subsequent list, initially, the `saturationModel` library must be added to the original `options` file, as the new cavitation model will utilize a variable saturation temperature and pressure to

calculate mass transfer rates. Exactly the same lines (under both EXE_INC and EXE_LIBS) should also be added to the options file of the Make directory inside the phase change library. Finally, due to the change in the name of the phase change library to thermalPhaseChangeTwoPhaseMixtures in the preparation steps, the new solver needs access to the new library in the user directory, which is the rationale for adding the first two lines under EXE_LIBS.

Solver main directory - Make/options

```
interPhaseChangeFoam = $(FOAM_SOLVERS)/multiphase/interPhaseChangeFoam
interFoam = $(FOAM_SOLVERS)/multiphase/interFoam
VoF = $(FOAM_SOLVERS)/multiphase/VoF

EXE_INC = \
    -I$(interPhaseChangeFoam) \
    -I$(interFoam) \
    -I$(VoF) \
    -ItemperaturePhaseChangeTwoPhaseMixtures/lnInclude \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/fvOptions/lnInclude\
    -I$(LIB_SRC)/meshTools/lnInclude \
    -I$(LIB_SRC)/sampling/lnInclude \
    -I$(LIB_SRC)/dynamicFvMesh/lnInclude \
    -I$(LIB_SRC)/thermophysicalModels/basic/lnInclude \
    -I$(LIB_SRC)/transportModels \
    -I$(LIB_SRC)/transportModels/twoPhaseMixture/lnInclude \
    -I$(LIB_SRC)/transportModels/incompressible/lnInclude \
    -I$(LIB_SRC)/transportModels/interfaceProperties/lnInclude \
    -I$(LIB_SRC)/TurbulenceModels/turbulenceModels/lnInclude \
    -I$(LIB_SRC)/TurbulenceModels/incompressible/lnInclude \
    -I$(LIB_SRC)/phaseSystemModels/reactingEuler/saturationModels/lnInclude

EXE_LIBS = \
    -L$(FOAM_USER_LIBBIN) \
    -lthermalPhaseChangeTwoPhaseMixtures \
    -lfiniteVolume \
    -lfvOptions \
    -lmeshTools \
    -lsampling \
    -ldynamicFvMesh \
    -ltwoPhaseMixture \
    -ltwoPhaseProperties \
    -linterfaceProperties \
    -lincompressibleTransportModels \
    -lturbulenceModels \
    -lincompressibleTurbulenceModels \
    -lfluidThermophysicalModels \
    -lsaturationModel
```

With all necessary changes made in the base directory, the implementation process of the new solver and cavitation model can now commence. The implementation will proceed in two primary steps: initially, the development of the phase change library in *Section 2.1* as the basis for the main solver, followed by the development of the main solver in *Section 2.2*, which is basically just a modification of one of the transport equations.

2.1 Implementing ZwartExtended cavitation model

To facilitate a smooth implementation process, all necessary changes to each directory will be discussed in separate sub-sections. The procedure will advance sequentially from a base class to its derived classes, and subsequently to the next base class, and so forth. As nothing has been altered in the thermoIncompressibleTwoPhaseMixture, which serves as the parent class for twoPhaseMixtureEThermo, the changes in this derived class can be examined as the initial step.

2.1.1 twoPhaseMixtureEThermo

As mentioned at the beginning of this chapter, the saturation pressure and temperature in this new cavitation model need to be variable. Consequently, it is not only essential to have access to the built-in saturation models of OpenFOAM by including its library in this class, but also, when calculating the saturation pressure and temperature based on the local temperature and pressure (`pSat(T)` and `Tsat(p)`), it is necessary to restrict the values of the local temperature and pressure within the range from the triple point up to the critical point of the fluid. This ensures that the values of the saturation model remain reasonable and no `Floating point exception` error would appear. Therefore, three parts of this class need to be modified: the header, the protected data, and then the public member functions that provide access to these protected data. The modifications are as follows.

twoPhaseMixtureEThermo.H

```
...

#include "saturationModel.H"

...

protected:

    // Protected Data

        //- Saturation model
        autoPtr<saturationModel> saturationModelPtr_;

        //- Critical pressure [Pa]
        dimensionedScalar Pc_;

        //- Critical temperature [K]
        dimensionedScalar Tc_;

        //- Triple-point pressure [Pa]
        dimensionedScalar Pt_;

        //- Triple-point temperature [K]
        dimensionedScalar Tt_;

...

        // Access to thermodynamic state variables

            //- Return the saturation temperature based on the chosen model
            virtual tmp<volScalarField> Tsat
            (
                const volScalarField& p
            ) const;

            //- Return the saturation pressure based on the chosen model
            virtual tmp<volScalarField> pSat
            (
                const volScalarField& T
            ) const;

            //- Return const-access to the critical pressure
            const dimensionedScalar& Pc() const
            {
                return Pc_;
            }

            //- Return const-access to the critical temperature
            const dimensionedScalar& Tc() const
            {
                return Tc_;
```

```
            }

            //- Return const-access to the triple-point pressure
            const dimensionedScalar& Pt() const
            {
                return Pt_;
            }

            //- Return const-access to the triple-point temperature
            const dimensionedScalar& Tt() const
            {
                return Tt_;
            }
...
```

As a result of these adjustments, the constructor of this class and its `read()` member function must be modified in the `twoPhaseMixtureEThermo.C` file. The `read()` function should be modified to ensure that the required triple- and critical-point temperatures and pressures are provided inside the `thermophysicalProperties` dictionary. Additionally, two member functions, `Tsat` and `pSat`, declared in the header file, should be defined here.

twoPhaseMixtureEThermo.C

```
...

// * * * * * * * * * * * * * * * * Constructor * * * * * * * * * * * * * * * * //

Foam::twoPhaseMixtureEThermo::twoPhaseMixtureEThermo
(
    const volVectorField& U,
    const surfaceScalarField& phi
)
:
    basicThermo(U.mesh(), word::null),
    thermoIncompressibleTwoPhaseMixture(U, phi),

    saturationModelPtr_
    (
        saturationModel::New
        (
            static_cast<const basicThermo&>(*this).subDict("saturationModel"),
            U.mesh()
        )
    ),
    Pc_("Pc", dimPressure, static_cast<const basicThermo&>(*this)),
    Tc_("Tc", dimTemperature, static_cast<const basicThermo&>(*this)),
    Pt_("Pt", dimPressure, static_cast<const basicThermo&>(*this)),
    Tt_("Tt", dimTemperature, static_cast<const basicThermo&>(*this))
{}


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

...

Foam::tmp<Foam::volScalarField> Foam::twoPhaseMixtureEThermo::Tsat
(
    const volScalarField& p
) const
{
    return saturationModelPtr_->Tsat(p);
}


Foam::tmp<Foam::volScalarField> Foam::twoPhaseMixtureEThermo::pSat
(
    const volScalarField& T
```

```
) const
{
    return saturationModelPtr_->pSat(T);
}


bool Foam::twoPhaseMixtureEThermo::read()
{
    if (basicThermo::read() && thermoIncompressibleTwoPhaseMixture::read())
    {
        basicThermo::readEntry("Pc", Pc_);
        basicThermo::readEntry("Tc", Tc_);
        basicThermo::readEntry("Pt", Pt_);
        basicThermo::readEntry("Tt", Tt_);

        return true;
    }

    return false;
}


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

2.1.2 temperaturePhaseChangeTwoPhaseMixture

Within this class, the necessary modifications pertain to the member functions. As discussed in the previous chapter, the equations of the new cavitation model need to be split into temperature and pressure terms. Hence, instead of a single function for calculating the source term of the pressure equation, there will be two (the following source code includes comments providing descriptions for each of these terms). It is worth noting that the member functions `mDotP()` and `mDotT()`, along with their derived functions `vDotP()` and `vDotT()`, are implemented as virtual functions. This design choice has been implemented to enable each phase change model to calculate these functions according to its distinct definitions. The following adjustments are required in the source code of this class.

temperaturePhaseChangeTwoPhaseMixture.H

```
...

    // Member Functions

        ...

        //- Return the pressure term of the mass condensation and vaporisation
        //  rates as coefficients to multiply (p - pSat)
        virtual Pair<tmp<volScalarField>> mDotP() const = 0;

        //- Return the temperature term of the mass condensation and
        //  vaporisation rates as coefficients
        virtual Pair<tmp<volScalarField>> mDotT() const = 0;

        ...

        //- Return the pressure term of the volumetric condensation and
        //  vaporisation rates as coefficients
        virtual Pair<tmp<volScalarField>> vDotP() const;

        //- Return the temperature term of the volumetric condensation and
        //  vaporisation rates as coefficients
        virtual Pair<tmp<volScalarField>> vDotT() const;

...
```

temperaturePhaseChangeTwoPhaseMixture.C

```
...

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

...

Foam::Pair<Foam::tmp<Foam::volScalarField>>
Foam::temperaturePhaseChangeTwoPhaseMixture::vDotP() const
{
    dimensionedScalar pCoeff(1.0/mixture_.rho1() - 1.0/mixture_.rho2());
    Pair<tmp<volScalarField>> mDotP = this->mDotP();

    return Pair<tmp<volScalarField>>(pCoeff*mDotP[0], pCoeff*mDotP[1]);
}

Foam::Pair<Foam::tmp<Foam::volScalarField>>
Foam::temperaturePhaseChangeTwoPhaseMixture::vDotT() const
{
    dimensionedScalar pCoeff(1.0/mixture_.rho1() - 1.0/mixture_.rho2());
    Pair<tmp<volScalarField>> mDotT = this->mDotT();

    return Pair<tmp<volScalarField>>(pCoeff*mDotT[0], pCoeff*mDotT[1]);
}

...
```

2.1.3 ZwartExtended

The ZwartExtended class, which inherits from the temperaturePhaseChangeTwoPhaseMixture class and has been developed based on the constant class, has undergone extensive modifications, with several new member data and functions implemented. Additionally, the return values of the pre-existing functions have also been altered. Therefore, instead of isolating the modifications within the constant base class, according to the convention of writing the tutorials of this course, the entire new cavitation class will be presented in Appendix A.

2.2 Implementing thermalInterPhaseChangeFoam solver

Regarding implementing the tIPCF solver, fortunately, there is not much that needs to be changed in the source code. At this phase of the project, the only modification required is to the source term of the pressure equation, pEqn, as stated in *Section 1.4.1*. This is because all other source terms in TEqn and alphaEqn have already been addressed in the implementation of the member functions of the ZwartExtended cavitation model, and no further adjustments need to be done to employ them in these transport equations.

pEqn.H

```
...

// Update the pressure BCs to ensure flux consistency
    constrainPressure(p_rgh, U, phiHbyA, rAUf);

    Pair<tmp<volScalarField>> vDotP = mixture->vDotP();
    const volScalarField& vDotcP = vDotP[0]();
    const volScalarField& vDotvP = vDotP[1]();

    Pair<tmp<volScalarField>> vDotT = mixture->vDotT();
    const volScalarField& vDotcT = vDotT[0]();
    const volScalarField& vDotvT = vDotT[1]();

    volScalarField limitedT (min(max(T, thermo->Tt()), thermo->Tc()));

```

```
    while (pimple.correctNonOrthogonal())
    {
        fvScalarMatrix p_rghEqn
        (
            fvc::div(phiHbyA) - fvm::laplacian(rAUf, p_rgh)
          - (vDotvP - vDotcP)*(thermo->pSat(limitedT) - rho*gh)
          + fvm::Sp(vDotvP - vDotcP, p_rgh)
          ==
            vDotvT + vDotcT
        );

...
```

At this stage, all that is required is to first compile the phase change library and then compile the main solver, which can be achieved by executing the specified commands below. After that, if everything is done correctly, there will be a new library named `libthermalPhaseChangeTwoPhaseMixtures.so` in the `$WM_PROJECT_USER_DIR/platforms/*/lib` directory, and a new solver named `thermalInterPhaseChangeFoam` in the `$WM_PROJECT_USER_DIR/platforms/*/bin` directory.

Compiling the new solver and its cavitation model

```
ufoam
cd applications/solvers/multiphase/thermalInterPhaseChangeFoam
wclean
wclean temperaturePhaseChangeTwoPhaseMixtures
wmake temperaturePhaseChangeTwoPhaseMixtures
wmake
```

Chapter 3

Test cases and results

As the test case for the new solver and cavitation model, the original tutorial of the `iPCF` solver was chosen, which involves modeling the cavitating flow over a bullet moving through water. Several reasons influenced this decision. First, there was a need for a reliable base case to which the results of the new models could be compared. Additionally, for the tutorial test cases within this course, the simulation runtime needed to be relatively short, making it impractical to use a real engineering application for assessing the new models. Considering that the original tutorial is a laminar case, this simplified the comparison process, as there was no need to account for turbulent parameters in the final results. Overall, the only modification to the original tutorial was to increase the temperature of the liquid, bringing it close to its boiling temperature at the free stream pressure.

3.1 Setting up the test case for `iPCF` solver

As mentioned, all that needs to be done is to duplicate the original tutorial of the `iPCF` solver, named `cavitatingBullet`, and then adjust the liquid temperature and its thermophysical properties, by going through the subsequent relevant sections of the following files. These modifications need to be done for the `iPCF` test case with high temperature (363 K).

Copying the original tutorial

```
run
cp -r "$WM_PROJECT_DIR/tutorials/multiphase/interPhaseChangeFoam/cavitatingBullet/"
    "cavitatingBullet_HighTemperature"
```

- **Modifying the saturation pressure and transport properties of the phases**

`constant/transportProperties` dictionary

```
pSat            69783;   // Saturation pressure at 363 K

sigma           0.061;   // Water surface tension at 363 K

water // Saturated liquid water properties at 363 K (CoolProp)
{
    transportModel  Newtonian;
    nu              3.26e-07;
    rho             965.396;
}

vapour // Saturated vapour water properties at 363 K (CoolProp)
{
    transportModel  Newtonian;
    nu              2.817e-05;
    rho             0.422;
}
...
```

- **Changing the number of subdomains for parallel simulation**

system/decomposeParDict dictionary

```
numberOfSubdomains 8;
method          simple;

coeffs
{
    n           (1 2 4);
}
```

- **Changing the Allrun script for running in parallel mode**

Allrun file

```
...
runApplication decomposePar

# Run the solver
# runApplication $(getApplication)
runParallel $(getApplication)
#------------------------------------------------------------------------------
```

3.2 Setting up the test case for tIPCF solver

Configuring the test case for the tIPCF solver requires additional steps. First, as noted in *Section 1.4.2*, the iCEF solver reads the pressure dictionary directly from the 0 directory. Therefore, for the tICPF solver, which is also derived from the iCEF solver, this pressure dictionary needs inclusion in the 0 directory. Additionally, since this new solver deals with the energy equation, it requires a temperature dictionary in the 0 directory. Besides the fields dictionaries, as mentioned in *Section 2.1*, this new cavitation model needs to read additional constants from the phaseChangeProperties, thermophysicalProperties, and transportProperties dictionaries. These new entries must be included in the mentioned dictionaries. Also, due to the energy equation and the dynamic mesh nature of this solver, several entries need to be added to the dictionaries of the system directory. All these will be detailed in the following.

Preparing the original tutorial

```
run
cp -r "$WM_PROJECT_DIR/tutorials/multiphase/interPhaseChangeFoam/cavitatingBullet/"
     "thermalCavitatingBullet"
cd thermalCavitatingBullet
cp 0.orig/p_rgh 0.orig/p
cp 0.orig/alpha.water 0.orig/T
touch constant/phaseChangeProperties
touch constant/thermophysicalProperties
```

- **Pressure field**
  (Header is included because the object needs to be changed from p_rgh to p)

0.orig/p dictionary

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

```

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

internalField   uniform 1e05;

boundaryField
{
    inlet
    {
        type            calculated;
        value           $internalField;
    }

    outlet
    {
        type            calculated;
        value           $internalField;
    }

    walls
    {
        type            symmetry;
    }

    bullet
    {
        type            calculated;
        value           $internalField;
    }
}


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

- **Temperature field**
  (Header is included because the `object` needs to be changed from `alpha.water` to `T`)

`0.orig/T` dictionary

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

dimensions      [0 0 0 1 0 0 0];

internalField   uniform 363;

boundaryField
{
    inlet
    {
        type            fixedValue;
        value           $internalField;
    }

    outlet
    {
        type            inletOutlet;
        inletValue      $internalField;
    }

    walls
    {
```

```
        type            symmetry;
    }

    bullet
    {
        type            zeroGradient;
    }
}


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

- **Phase change properties**
  (Header is included because this file was created as a blank file)

  The specific coefficients for the new cavitation model should be included in this dictionary. These coefficients are derived from references [3, 6, 11].

`constant/phaseChangeProperties` dictionary

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      phaseChangeProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

phaseChangeTwoPhaseModel ZwartExtended;

ZwartExtendedCoeffs
{
    Cc          2e-03;
    Cv          10;
    alphaNuc    5e-04;
    rNuc        1e-06;
    tG          5e-03;
    TInf        363;
}

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

- **Thermophysical properties**
  (Header is included because this file was created as a blank file)

  Within this dictionary, it is necessary to include the temperature and pressure values corresponding to the triple- and critical-point, as explained in *Section 2.1.1*. Simultaneously, the coefficients for the well-known Antoine saturation model, derived from reference [13], should also be added. However, it is worth noting that the coefficients in this reference were originally formulated for a different version of the Antoine saturation model. As a result, modifications were made to adapt the coefficients for compatibility with the OpenFOAM implementation of this model, which relies on natural logarithms, pressure expressed in Pascal, and temperature in Kelvin.

`constant/thermophysicalProperties` dictionary

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      thermophysicalProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
```

```
//- Water critical- and triple-point pressures and temperatures (CoolProp)
Pc    2.2064e7;
Tc    647.096;

Pt    611.655;
Tt    273.16;

/*
saturationModel
{
    type    constant;
    pSat    69783;
    Tsat    363;
}
*/

saturationModel
{
    type    Antoine;
    A       23.196;
    B       -3816.447;
    C       -46.13;
}

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

- **Transport properties**

  As evident in the following, within each phase dictionary, namely `water` and `vapour`, four entries pertaining to the energy equation have been added. Consequently, the `transportProperties` dictionary for the new solver and its cavitation model mirrors the structure found in the `iCEF` original tutorial, i.e. `condensatingVessel`.

`constant/transportProperties` dictionary

```
phases          (water vapour);

sigma           0.061; // Water surface tension at 363 K

water // Saturated liquid water properties at 363 K (CoolProp)
{
    transportModel  Newtonian;
    nu              3.26e-07;
    rho             965.396;

    Cp              4205.135;
    Cv              3821.188;
    kappa           0.673; // Thermal conductivity [J/s/m/K]
    hf              376408.297;
}

vapour // Saturated vapour water properties at 363 K (CoolProp)
{
    transportModel  Newtonian;
    nu              2.817e-05;
    rho             0.422;

    Cp              2042.423;
    Cv              1531.255;
    kappa           0.024;
    hf              2659285.627;
}

Prt             0.7;

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

- **Changing the default solver and adding two new entries in control dictionary**

system/controlDict dictionary

```
...

application     thermalInterPhaseChangeFoam;

...

maxAlphaCo      1.0;

maxDeltaT       1e-02;


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

- **Including the required numerical schemes for the temperature field**

system/fvSchemes dictionary

```
...

divSchemes
{
...
    div(rhoCpPhi,T)     Gauss linearUpwind grad(T);
...
    div((muEff*dev(T(grad(U))))  Gauss linear;
    div((interpolate(cp)*rhoPhi),T) Gauss linearUpwind grad(T);
}

...

wallDist
{
    method          meshWave;
}
```

- **Including the required control entries for the energy equation**

system/fvSolution dictionary

```
...

solvers
{
    ".*(rho|rhoFinal)"
    {
        solver          diagonal;
    }

...

    "T.*"
    {
        solver          smoothSolver;
        smoother        symGaussSeidel;
        tolerance       1e-6;
        relTol          0.0;
    }

...

}

...
```

3.3 Results

In this section, the outcomes of the three different cases will be presented and compared to each other. Firstly, the results of the original `iPCF` tutorial will be examined. Secondly, the case configuration for simulating the `iPCF` tutorial under elevated temperatures will be explored. It is noteworthy that, despite the fact that the energy equation is not solved by the `iPCF` solver, the impact of increased temperature will be manifested in altered saturation pressure and other transport properties of the liquid. Lastly, the same case will be analyzed using the `tIPCF` solver. Notably, the `outlet` boundary will be reached by the cavitation of the flow at higher temperatures around 9 ms, exerting influence on domain properties thereafter. Thus, only the results up to this critical time will be presented in the subsequent discussion. Moreover, given that the original `iPCF` tutorial, as depicted in the following figures, was tailored for a short cavitating flow over the bullet, the mesh resolution in the latter portion of the domain is inadequately refined, thereby inducing anomalous characteristics in the tail of the cavitating flow for cases with higher saturation temperature at 8 ms.

As outlined in the preceding setting sections, the cavitation model utilized for the `iPCF` solver is the `SchnerrSauer` model, while for the `tIPCF` solver, the `ZwartExtended` model is employed. To visualize the simulation results using `ParaView`, certain considerations should be noted. Firstly, since the simulations were conducted in parallel mode without reconstruction, it is essential to select the `Decomposed Case` type under the `Properties` section of a given test case in `ParaView`, as opposed to the default `Reconstructed Case` option. Secondly, due to the placement of the bullet in the center of the domain, visualizing a cross-sectional representation of the domain is preferable that can be achieved by using either `Clip` or `Slice` filters.

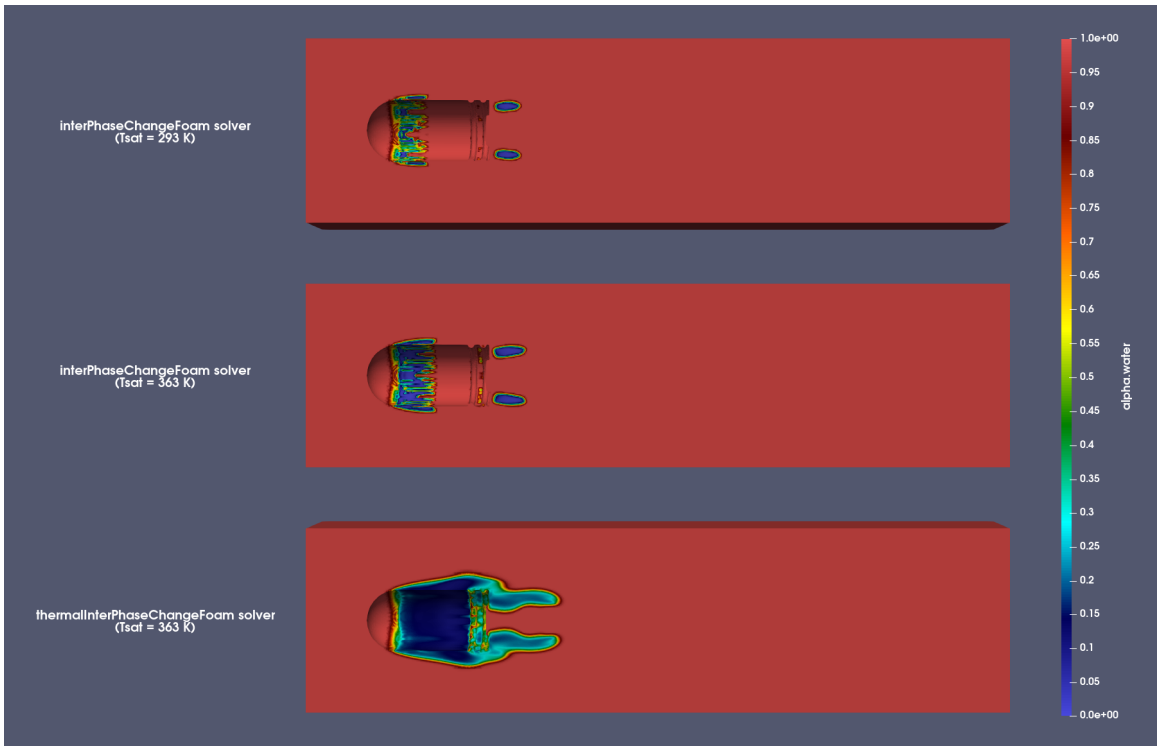

Figure 3.1: Volume fraction of the carrier fluid at 1 ms with different solvers and under different conditions

In Figure 3.1, the results are presented in three rows: the first row displays the outcomes of the `iPCF` solver at a temperature of 293 K (original tutorial). Moving to the second row, the results of the same solver under an elevated temperature of 363 K are showcased. Lastly, in the third row, the outcomes of the `tIPCF` solver under the identical temperature as the second row, 363 K, are

illustrated. It is important to note that the bulk fluid (free stream) pressure is held constant at 1 bar across all these cases. Observing the figures, it becomes evident that initially, the `ZwartExtended` cavitation model (third row) predicts higher cavitation in the flow with a high saturation temperature (close to its boiling point) in comparison to the `Schnerrsauer` model (second row). Furthermore, comparing the `Schnerrsauer` model in two different conditions (first and second rows) illustrates that increasing the saturation temperature of the flow does not initially significantly increase the cavitation rate for this model.

However, as depicted in Figure 3.2, although the `Schnerrsauer` model initially projected lower vaporization, it predicted higher vapor fractions after 8 ms compared to the `ZwartExtended` model behind the bullet (second and third rows). In other words, while the `ZwartExtended` model forecasts elevated vaporization rates at the beginning, it also predicts heightened condensation rates after the establishment of the flow.

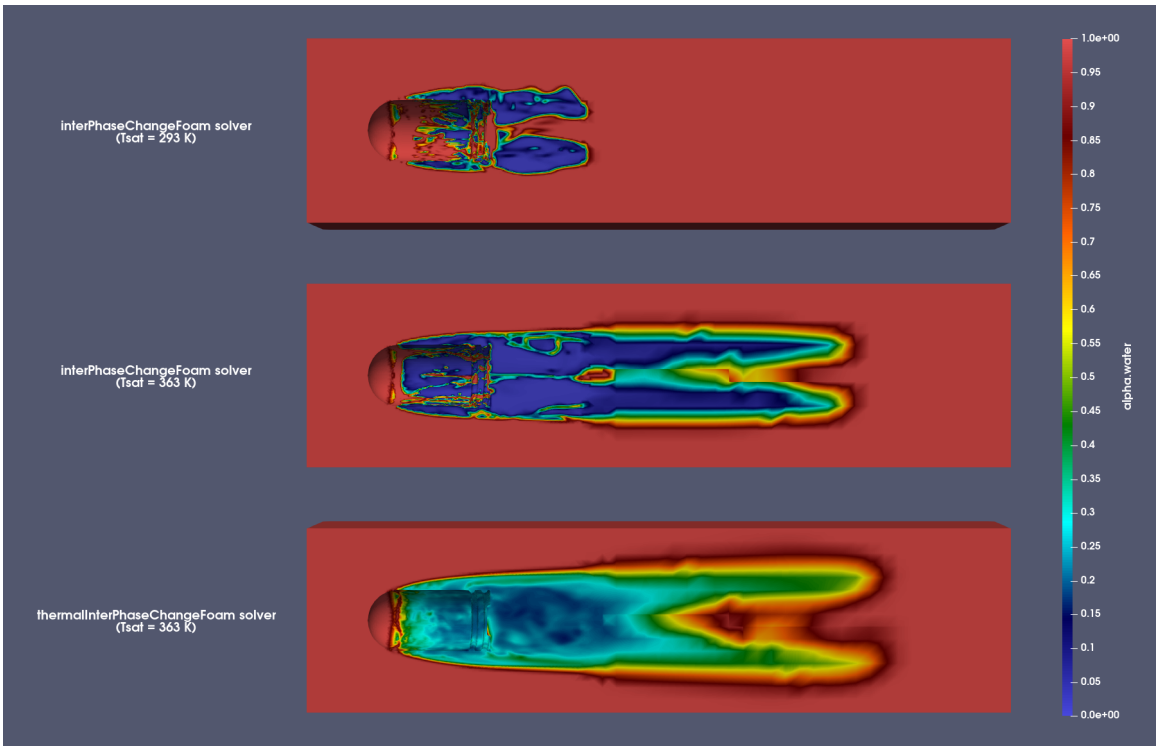

Figure 3.2: Volume fraction of the carrier fluid at 8 ms with different solvers and under different conditions

Finally, it is noteworthy that computing the pressure coefficient of the bullet, denoted as $C_\mathrm{p} = \frac{p-p_\infty}{\frac{1}{2}\rho_\infty U^2}$, which closely corresponds to the cavitation number in this scenario, does not reveal a substantial difference between the employed cavitation models across the bullet. However, the difference between the computed coefficients decreases over time, just like the appearance of the volume fraction contours in the above figures. Therefore, the author's perspective at this stage is that for a more meaningful comparison between models, their application in simulating two scenarios is crucial: first, simulating cavitating flows involving thermo-fluids, such as refrigerants, and then simulating cavitating internal flows. In the former cases, the thermal impact of the cavitation phenomenon will be more significant due to the thermophysical properties of the fluid, and in the latter cases, a highly informative criterion, such as the discharge coefficient of the nozzle, can be calculated, providing a more effective means of distinguishing between these models.

Bibliography

[1] C. E. Brennen, *Cavitation and Bubble Dynamics*. New York: Oxford University Press, Jan. 1995.

[2] Y. Sun, Z. Guan, and K. Hooman, "Cavitation in Diesel Fuel Injector Nozzles and its Influence on Atomization and Spray," *Chemical Engineering & Technology*, vol. 42, no. 1, pp. 6–29, 2019. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/ceat.201800323.

[3] Z. Yao, L. Xian-Wu, Jibin, L. Shu-Hong, W. Yu-Lin, and X. Hong-Yuan, "A Thermodynamic Cavitation Model for Cavitating Flow Simulation in a Wide Range of Water Temperatures," *Chinese Physics Letters*, vol. 27, p. 016401, Jan. 2010.

[4] Y. Sun, "Description of interCondensatingEvaporatingFoam and implementation of SGS term into volume fraction equation," *In Proceedings of CFD with OpenSource Software, 2022, Edited by Nilsson. H.* http://dx.doi.org/10.17196/OS_CFD#YEAR_2022.

[5] A. Asnagi, "interPhaseChangeFoam tutorial and PANS turbulence model," *In Proceedings of CFD with OpenSource Software, 2013, Edited by Nilsson. H.* http://dx.doi.org/10.17196/OS_CFD#YEAR_2013.

[6] M. Jansson, "Implementing a Zwart-Gerber-Belamri cavitation model," *In Proceedings of CFD with OpenSource Software, 2018, Edited by Nilsson. H.* http://dx.doi.org/10.17196/OS_CFD#YEAR_2018.

[7] R. F. Kunz, D. A. Boger, D. R. Stinebring, T. S. Chyczewski, J. W. Lindau, H. J. Gibeling, S. Venkateswaran, and T. R. Govindan, "A preconditioned Navier–Stokes method for two-phase flows with application to cavitation prediction," *Computers & Fluids*, vol. 29, pp. 849–875, Aug. 2000.

[8] C. Merkle, J. Feng, and P. Buelow, "Computational modeling of the dynamics of sheet cavitation," 1998.

[9] G. Schnerr Professor Dr.-Ing.habil, "Physical and Numerical Modeling of Unsteady Cavitation Dynamics," May 2001.

[10] S. Hardt and F. Wondra, "Evaporation model for interfacial flows based on a continuum-field representation of the source terms," *Journal of Computational Physics*, vol. 227, pp. 5871–5895, May 2008.

[11] H. u, T. ang, and Z. he, "Flow in fuel nozzles under cavitation and flash-boiling conditions," *AIP Advances*, vol. 12, no. 5, p. 055218, 2022. _eprint: https://doi.org/10.1063/5.0089755.

[12] M. S. Plesset and S. A. Zwick, "The Growth of Vapor Bubbles in Superheated Liquids," *Journal of Applied Physics*, vol. 25, pp. 493–500, May 2004.

[13] *Appendix A: Useful Tables and Charts*, pp. 303–317. John Wiley Sons, Ltd, 2012.

Study questions

1. What criteria are utilized to classify a phase change phenomenon as cavitation?
2. What sets apart a cavitation model from a boiling model at a fundamental level?
3. How are cavitation and boiling models implemented in OpenFOAM's computational framework?
4. What constitutes the bubble dynamics equations, and what functions do they serve in the modeling of the cavitation phenomenon?
5. What is the significance of considering the combined impact of mechanical and thermal influences in the cavitation process?
6. In the context of cavitation, how can thermal effects be appropriately accounted for?
7. How a thermodynamic cavitation model can be implemented in OpenFOAM? What additional libraries need to be considered in comparison to the required libraries of the built-in cavitation models in OpenFOAM?
8. How a non-isothermal counterpart for `interPhaseChangeFoam` solver can be implemented in OpenFOAM?

Appendix A

ZwartExtended cavitation model source code

A.1 ZwartExtended.H file

ZwartExtended.H file

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | www.openfoam.com
     \\/     M anipulation  |
-------------------------------------------------------------------------------
    Copyright (C) 2016-2019 OpenCFD Ltd.
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

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

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

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

Class
    Foam::temperaturePhaseChangeTwoPhaseMixture::ZwartExtended

Description
    This model is based on a thermodynamic cavitation model, which considers
    the superposed effect of inertia (mechanical effect due to pressure
    gradient) and heat transfer (thermal effect due to temperature gradient)
    impacts in calculating the mass transfer rates.

    Reference:
    \verbatim
        Z. Yao, L. Xian-Wu, Jibin, L. Shu-Hong, W. Yu-Lin, and X. Hong-Yuan,
          A   Thermodynamic Cavitation Model for Cavitating Flow Simulation in a
        Wide Range of Water Temperatures,    Chinese Phys. Lett., vol. 27, no. 1,
        p. 016401, Jan. 2010, doi: 10.1088/0256-307X/27/1/016401.
    \endverbatim
```

```
42
43  SourceFiles
44      ZwartExtended.C
45
46  \*---------------------------------------------------------------------------*/
47
48  #ifndef ZwartExtended_H
49  #define ZwartExtended_H
50
51  #include "temperaturePhaseChangeTwoPhaseMixture.H"
52
53  // * * * * * * * * * * * * * * * * * *  * * * * * * * * * * * * * * * * * * //
54
55  namespace Foam
56  {
57  namespace temperaturePhaseChangeTwoPhaseMixtures
58  {
59
60  /*---------------------------------------------------------------------------*\
61                             Class ZwartExtended
62  \*---------------------------------------------------------------------------*/
63
64  class ZwartExtended
65  :
66      public temperaturePhaseChangeTwoPhaseMixture
67  {
68      // Private data
69
70          //- Condensation rate coefficient [-]
71          dimensionedScalar Cc_;
72
73          //- Vapourisation rate coefficient [-]
74          dimensionedScalar Cv_;
75
76          //- Nucleation site volume-fraction [-]
77          dimensionedScalar alphaNuc_;
78
79          //- Nucleation site radius [m]
80          dimensionedScalar rNuc_;
81
82          //- Bubble growth time [s]
83          dimensionedScalar tG_;
84
85          //- Bulk temperature (liquid) [K]
86          dimensionedScalar TInf_;
87
88          //- Return the limited pressures of both phases for calculating Tsat
89          tmp<volScalarField> pTsat
90          (
91              const volScalarField& p
92          ) const;
93
94          //- Return the limited temperature for calculating pSat
95          tmp<volScalarField> TpSat
96          (
97              const volScalarField& T
98          ) const;
99
100         //- Part of the condensation and vaporisation rates [s/m2]
101         tmp<volScalarField> pCoeff
102         (
103             const volScalarField& p,
104             const volScalarField& T
105         ) const;
106
107         //- Part of the condensation and vaporisation rates [J/m3/s/K]
108         tmp<volScalarField> TCoeff() const;
109
```

```
110
111 public:
112
113     //- Runtime type information
114     TypeName("ZwartExtended");
115
116
117     // Constructors
118
119         //- Construct from components
120         ZwartExtended
121         (
122             const thermoIncompressibleTwoPhaseMixture& mixture,
123             const fvMesh& mesh
124         );
125
126
127     //- Destructor
128     virtual ~ZwartExtended() = default;
129
130
131     // Member Functions
132
133         //- Return the mass condensation and vaporisation rates as a
134         //  coefficient to multiply (1 - alphal) for the condensation rate
135         //  and a coefficient to multiply  alphal for the vaporisation rate
136         virtual Pair<tmp<volScalarField>> mDotAlphal() const;
137
138         //- Return the pressure term of the mass condensation and vaporisation
139         //  rates as coefficients to multiply (p - pSat)
140         virtual Pair<tmp<volScalarField>> mDotP() const;
141
142         //- Return the temperature term of the mass condensation and
143         //  vaporisation rates as coefficients
144         virtual Pair<tmp<volScalarField>> mDotT() const;
145
146         //- Source for T equarion
147         virtual tmp<fvScalarMatrix> TSource() const;
148
149         //- Correct the ZwartExtended phaseChange model
150         virtual void correct();
151
152         //- Read the transportProperties dictionary and update
153         virtual bool read();
154 };
155
156
157 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
158
159 } // End namespace temperaturePhaseChangeTwoPhaseMixtures
160 } // End namespace Foam
161
162 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
163
164 #endif
165
166 // ************************************************************************* //
```

A.2 ZwartExtended.C file

ZwartExtended.C file

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | www.openfoam.com
     \\/     M anipulation  |
-------------------------------------------------------------------------------
    Copyright (C) 2016-2020 OpenCFD Ltd.
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

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

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

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

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

#include "ZwartExtended.H"
#include "addToRunTimeSelectionTable.H"
#include "fvcGrad.H"
#include "twoPhaseMixtureEThermo.H"
#include "fvmSup.H"
#include "mathematicalConstants.H"

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

namespace Foam
{
namespace temperaturePhaseChangeTwoPhaseMixtures
{
    defineTypeNameAndDebug(ZwartExtended, 0);
    addToRunTimeSelectionTable
    (
        temperaturePhaseChangeTwoPhaseMixture,
        ZwartExtended,
        components
    );
}
}

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

Foam::temperaturePhaseChangeTwoPhaseMixtures::ZwartExtended::ZwartExtended
(
    const thermoIncompressibleTwoPhaseMixture& mixture,
    const fvMesh& mesh
)
:
    temperaturePhaseChangeTwoPhaseMixture(mixture, mesh),
    Cc_("Cc", dimless, optionalSubDict(type() + "Coeffs")),
    Cv_("Cv", dimless, optionalSubDict(type() + "Coeffs")),
    alphaNuc_("alphaNuc", dimless, optionalSubDict(type() + "Coeffs")),
    rNuc_("rNuc", dimLength, optionalSubDict(type() + "Coeffs")),
```

```
64      tG_("tG", dimTime, optionalSubDict(type() + "Coeffs")),
65      TInf_("TInf", dimTemperature, optionalSubDict(type() + "Coeffs"))
66  {
67      correct();
68  }
69
70  // * * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * * * * * * //
71
72  Foam::tmp<Foam::volScalarField>
73  Foam::temperaturePhaseChangeTwoPhaseMixtures::ZwartExtended::pTsat
74      (
75          const volScalarField& p
76      ) const
77  {
78      const twoPhaseMixtureEThermo& thermo =
79          refCast<const twoPhaseMixtureEThermo>
80          (
81              mesh_.lookupObject<basicThermo>(basicThermo::dictName)
82          );
83
84      volScalarField limitedP
85      (
86          "pTsat",
87          min(max(p, thermo.Pt()), thermo.Pc())
88      );
89
90      if (mesh_.time().outputTime())
91      {
92          limitedP.write();
93      }
94
95      return
96          tmp<volScalarField>(new volScalarField(limitedP));
97  }
98
99  Foam::tmp<Foam::volScalarField>
100 Foam::temperaturePhaseChangeTwoPhaseMixtures::ZwartExtended::TpSat
101     (
102         const volScalarField& T
103     ) const
104 {
105     const twoPhaseMixtureEThermo& thermo =
106         refCast<const twoPhaseMixtureEThermo>
107         (
108             mesh_.lookupObject<basicThermo>(basicThermo::dictName)
109         );
110
111     volScalarField limitedT
112     (
113         "TpSat",
114         min(max(T, thermo.Tt()), thermo.Tc())
115     );
116
117     if (mesh_.time().outputTime())
118     {
119         limitedT.write();
120     }
121
122     return
123         tmp<volScalarField>(new volScalarField(limitedT));
124 }
125
126
127 Foam::tmp<Foam::volScalarField>
128 Foam::temperaturePhaseChangeTwoPhaseMixtures::ZwartExtended::pCoeff
129     (
130         const volScalarField& p,
131         const volScalarField& T
```

```
    ) const
{
    const twoPhaseMixtureEThermo& thermo =
        refCast<const twoPhaseMixtureEThermo>
        (
            mesh_.lookupObject<basicThermo>(basicThermo::dictName)
        );

    const volScalarField pSat = thermo.pSat(this->TpSat(T));

    return
        (3*mixture_.rho2())*sqrt(2/(3*mixture_.rho1()))
        /(rNuc_*sqrt(mag(p - pSat) + SMALL*pSat));
}


Foam::tmp<Foam::volScalarField>
Foam::temperaturePhaseChangeTwoPhaseMixtures::ZwartExtended::TCoeff() const
{
    volScalarField limitedAlpha1
    (
        min(max(mixture_.alpha1(), scalar(0)), scalar(1))
    );

    //- Specific heat capacity of the mixture [J/kg/K]
    volScalarField cp
    (
        limitedAlpha1*mixture_.Cp1()
      + (scalar(1) - limitedAlpha1)*mixture_.Cp2()
    );

    //- Thermal diffusivity of the liquid phase [m2/s]
    const dimensionedScalar a1 =
        mixture_.kappa1()/(mixture_.Cp1()*mixture_.rho1());

    return
        (3*mixture_.rho1())*cp*sqrt(3*a1)
        /(rNuc_*sqrt(constant::mathematical::pi*tG_));
}


Foam::Pair<Foam::tmp<Foam::volScalarField>>
Foam::temperaturePhaseChangeTwoPhaseMixtures::ZwartExtended::mDotAlphal() const
{
    const volScalarField& p = mesh_.lookupObject<volScalarField>("p");
    const volScalarField& T = mesh_.lookupObject<volScalarField>("T");

    volScalarField pCoeff(this->pCoeff(p, T));
    volScalarField TCoeff(this->TCoeff());

    const twoPhaseMixtureEThermo& thermo =
        refCast<const twoPhaseMixtureEThermo>
        (
            mesh_.lookupObject<basicThermo>(basicThermo::dictName)
        );

    const volScalarField pSat = thermo.pSat(this->TpSat(T));
    const volScalarField Tsat = thermo.Tsat(this->pTsat(p));

    const dimensionedScalar p0(dimPressure, Zero);
    const dimensionedScalar T0(dimTemperature, Zero);

    //- Latent heat of vaporization [J/kg]
    dimensionedScalar L = mixture_.Hf2() - mixture_.Hf1();

    return Pair<tmp<volScalarField>>
    (
        Cc_*(pCoeff*max(p - pSat, p0) + TCoeff*max(Tsat - T, T0)/L),
```

```
        Cv_*alphaNuc_*(pCoeff*min(p - pSat, p0) - TCoeff*max(T - Tsat, T0)/L)
    );
}


Foam::Pair<Foam::tmp<Foam::volScalarField>>
Foam::temperaturePhaseChangeTwoPhaseMixtures::ZwartExtended::mDotP() const
{
    const volScalarField& p = mesh_.lookupObject<volScalarField>("p");
    const volScalarField& T = mesh_.lookupObject<volScalarField>("T");

    volScalarField pCoeff(this->pCoeff(p, T));

    volScalarField limitedAlpha1
    (
        min(max(mixture_.alpha1(), scalar(0)), scalar(1))
    );

    const twoPhaseMixtureEThermo& thermo =
        refCast<const twoPhaseMixtureEThermo>
        (
            mesh_.lookupObject<basicThermo>(basicThermo::dictName)
        );

    const volScalarField pSat = thermo.pSat(this->TpSat(T));

    return Pair<tmp<volScalarField>>
    (
        Cc_*(scalar(1) - limitedAlpha1)*pos0(p - pSat)*pCoeff,
       -Cv_*alphaNuc_*limitedAlpha1*neg(p - pSat)*pCoeff
    );
}


Foam::Pair<Foam::tmp<Foam::volScalarField>>
Foam::temperaturePhaseChangeTwoPhaseMixtures::ZwartExtended::mDotT() const
{
    const volScalarField& p = mesh_.lookupObject<volScalarField>("p");
    const volScalarField& T = mesh_.lookupObject<volScalarField>("T");

    volScalarField TCoeff(this->TCoeff());

    volScalarField limitedAlpha1
    (
        min(max(mixture_.alpha1(), scalar(0)), scalar(1))
    );

    const twoPhaseMixtureEThermo& thermo =
        refCast<const twoPhaseMixtureEThermo>
        (
            mesh_.lookupObject<basicThermo>(basicThermo::dictName)
        );

    const volScalarField Tsat = thermo.Tsat(this->pTsat(p));

    const dimensionedScalar T0(dimTemperature, Zero);

    dimensionedScalar L = mixture_.Hf2() - mixture_.Hf1();

    volScalarField mDotTV
    (
        "mDotTV", Cv_*alphaNuc_*limitedAlpha1*TCoeff*max(T - Tsat, T0)/L
    );
    volScalarField mDotTC
    (
        "mDotTC", Cc_*(scalar(1) - limitedAlpha1)*TCoeff*max(Tsat - T, T0)/L
    );

```

```
268     if (mesh_.time().outputTime())
269     {
270         mDotTC.write();
271         mDotTV.write();
272     }
273
274     return Pair<tmp<volScalarField>>
275     (
276         tmp<volScalarField>(new volScalarField(mDotTC)),
277         tmp<volScalarField>(new volScalarField(-mDotTV))
278     );
279 }
280
281
282 Foam::tmp<Foam::fvScalarMatrix>
283 Foam::temperaturePhaseChangeTwoPhaseMixtures::ZwartExtended::TSource() const
284 {
285
286     const volScalarField& p = mesh_.lookupObject<volScalarField>("p");
287     const volScalarField& T = mesh_.lookupObject<volScalarField>("T");
288
289     volScalarField pCoeff(this->pCoeff(p, T));
290     volScalarField TCoeff(this->TCoeff());
291
292     tmp<fvScalarMatrix> tTSource
293     (
294         new fvScalarMatrix
295         (
296             T,
297             dimEnergy/dimTime
298         )
299     );
300
301     fvScalarMatrix& TSource = tTSource.ref();
302
303     volScalarField limitedAlpha1
304     (
305         min(max(mixture_.alpha1(), scalar(0)), scalar(1))
306     );
307
308     const twoPhaseMixtureEThermo& thermo =
309         refCast<const twoPhaseMixtureEThermo>
310         (
311             mesh_.lookupObject<basicThermo>(basicThermo::dictName)
312         );
313
314     const volScalarField pSat = thermo.pSat(this->TpSat(T));
315     const volScalarField Tsat = thermo.Tsat(this->pTsat(p));
316
317     const dimensionedScalar p0(dimPressure, Zero);
318
319     dimensionedScalar L = mixture_.Hf2() - mixture_.Hf1();
320
321     const volScalarField VcoeffP
322     (
323         Cv_*alphaNuc_*limitedAlpha1*pCoeff*min(p - pSat, p0)*L
324     );
325
326     const volScalarField CcoeffP
327     (
328         Cc_*(scalar(1) - limitedAlpha1)*pCoeff*max(p - pSat, p0)*L
329     );
330
331     const volScalarField VcoeffT
332     (
333         Cv_*alphaNuc_*limitedAlpha1*TCoeff*pos(T - Tsat)
334     );
335     const volScalarField CcoeffT
```

```
336     (
337         Cc_*(scalar(1) - limitedAlpha1)*TCoeff*pos(Tsat - T)
338     );
339
340     TSource =
341       - (VcoeffP + CcoeffP)
342       + fvm::Sp(VcoeffT, T) - VcoeffT*Tsat
343       + fvm::Sp(CcoeffT, T) - CcoeffT*Tsat;
344
345     return tTSource;
346 }
347
348
349 void Foam::temperaturePhaseChangeTwoPhaseMixtures::ZwartExtended::correct()
350 {
351 }
352
353
354 bool Foam::temperaturePhaseChangeTwoPhaseMixtures::ZwartExtended::read()
355 {
356     if (temperaturePhaseChangeTwoPhaseMixture::read())
357     {
358         subDict(type() + "Coeffs").readEntry("Cc", Cc_);
359         subDict(type() + "Coeffs").readEntry("Cv", Cv_);
360         subDict(type() + "Coeffs").readEntry("alphaNuc", alphaNuc_);
361         subDict(type() + "Coeffs").readEntry("rNuc", rNuc_);
362         subDict(type() + "Coeffs").readEntry("tG", tG_);
363         subDict(type() + "Coeffs").readEntry("TInf", TInf_);
364
365         return true;
366     }
367
368     return false;
369 }
370
371
372 // ************************************************************************* //
```