Cite as: Lavari, M.: interPhaseChangeBubbleFoam: A hybrid Eulerian-Lagrangian solver to capture nuclei effects on hydrodynamic cavitation. In Proceedings of CFD with OpenSource Software, 2024, Edited by Nilsson. H., http://dx.doi.org/10.17196/OS_CFD#YEAR_2024

CFD with OpenSource software

A course at Chalmers University of Technology
Taught by Håkan Nilsson

interPhaseChangeBubbleFoam: A hybrid Eulerian-Lagrangian solver to capture nuclei effects on hydrodynamic cavitation

Developed for OpenFOAM-v2406

Author:
Mahdi Lavari
Worcester Polytechnic Institute
mlavari@wpi.edu

Peer reviewed by:
Prof. Aswin Gnanaskandan
Dr. Saeed Salehi
Muhammad Ahmad Raza

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 to 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 23, 2025

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 `interPhaseChangeFoam`.
- How to use the Lagrangian library in OpenFOAM.
- How to use the Rayleigh-Plesset equation from Chen's work [1].

The theory of it:

- The theory of the Homogeneous Mixture Model or Volume of Fluid method implemented in `interPhaseChangeFoam`.
- The theory of Lagrangian particle tracking.

How it is implemented:

- How the alpha (liquid volume fraction) transport equation is implemented in the code.

How to modify it:

- How to couple `interPhaseChangeFoam` with the Lagrangian library.
- How to couple the Lagrangian library with Rayleigh-Plesset equation.
- How to switch the vapor structures between the Eulerian and Lagrangian frameworks.

Prerequisites

The reader is expected to be familiar with the following topics to gain maximum benefit from this report:

- Fundamentals of multiphase flows, more specifically hydrodynamic cavitation.
- Fundamentals of the Homogeneous Mixture Model or Volume of Fluid with Finite Mass Transfer.
- PhD dissertations by Ghahramani [2] and Vallier [3] on coupling Lagrangian and Eulerian frameworks.
- Fundamentals of bubble dynamics and the solution to the Rayleigh-Plesset equation.

Acknowledgments

This project is dedicated to the cherished memory of my beloved father, Hassan Lavari, whose unwavering love, guidance, and sacrifices laid the foundation of my life's journey and the person I strive to become.

His belief in my abilities, encouragement through every step of my path, and the values he instilled in me have been a constant source of strength and inspiration. Even as he departed during the course of this project, his presence remained deeply rooted in my heart, motivating me to persevere and complete this work.

I am forever grateful for the countless ways he shaped my life, from his honesty, dignity, wisdom, and strength to his unconditional support. This work stands as a tribute to his enduring legacy and the profound impact he has had on my life.

May his soul rest in eternal peace.

Contents

Nomenclature

Acronyms

FMT	Finite Mass Transfer
L-E	Lagrangian-Eulerian
RPE	Rayleigh-Plesset Equation
VOF	Volume of Fluid

English symbols

$\dot{m}$	Source term for phase interaction	kg/s
$\dot{m}_c$	Mass transfer rate for condensation	kg/s
$\dot{m}_\mathrm{v}$	Mass transfer rate for vaporization	kg/s
$\dot{R}$	Bubble growth rate	m/s
$\dot{V}$	Volume transfer factor	m^3/kg
$\dot{V}_c$	Volume transfer factor for condensation	m^3/kg
$\dot{V}_\mathrm{v}$	Volume transfer factor for vaporization	m^3/kg
$\mathbf{U}$	Velocity vector	m/s
τ_{ij}	Stress tensor	Pa
F_d	Forces acting on the bubble	N
g	Gravitational acceleration	$\mathrm{m/s}^2$
p	Pressure	Pa
p_v	Vapor pressure	Pa
R	Bubble radius	m
R_0	Initial bubble radius	m
u_i	Velocity components	m/s

Greek symbols

α_N^c	Volume fraction of bubble nuclei in the liquid
α_l	Liquid volume fraction
α_v	Vapor volume fraction
β	Lagrangian liquid volume fraction
δ_{ij}	Kronecker delta
μ_l	Liquid viscosity $\mathrm{kg \cdot s/m}$
μ_m	Mixture viscosity $\mathrm{kg \cdot s/m}$
μ_v	Vapor viscosity $\mathrm{kg \cdot s/m}$
ρ_l	Liquid density $\mathrm{kg/m}^3$
ρ_m	Mixture density $\mathrm{kg/m}^3$
ρ_v	Vapor density $\mathrm{kg/m}^3$

Superscripts

i	counter
j	counter
k	counter

b	bubble
l	liquid
m	mixture
v	vapor

Subscripts

Eulerian A frame of reference fixed in space

Lagrangian A frame of reference moving with particles

Chapter 1

Introduction

Cavitation is the formation of vapor in a liquid that occurs when the pressure drops below the vapor pressure. Studying this phenomenon has been the subject of research across a wide range of applications, from medical devices to industrial equipment.

Numerous numerical approaches have been developed to study cavitation. The Homogeneous Mixture Model or Volume of Fluid(VOF) approach with Finite Mass Transfer(FMT) models have been proven to successfully capture large scale vapor cavities in a variety of cavitating flows [4]. However, one of the main challenges is understanding the inception process. In this regard, considering bubble dynamics and the presence of nuclei in the liquid is essential.

It is well-known that when the nuclei pass through a low-pressure region, the pressure inside the bubble, being much higher than the outside pressure, causes rapid bubble growth and, consequently, results in the formation of cavitating bubbles. Since the size of nuclei is typically in the micron-to-millimeter range, it is computationally expensive to apply the traditional Eulerian approaches to resolve the nuclei. A more efficient alternative is to track nuclei in the Lagrangian framework. Ghahramani *et al.* [5] tried to explore a similar concept to some extent. In this approach, nuclei are tracked in the Lagrangian framework until they reach a size that can be resolved using macroscale methods like VOF. A transition algorithm would then be required to switch between the Eulerian and Lagrangian frameworks.

This project aims to establish a foundation for accounting for the presence of nuclei, bubble dynamics, and the transition between the Lagrangian framework and the Eulerian framework. This goal can be achieved by coupling Eulerian and Lagrangian frameworks using OpenFOAM facilities. Therefore, the primary objective is to develop a comprehensive simulation approach that incorporates the Rayleigh-Plesset equation for bubble dynamics, integrates the Lagrangian library with the `interPhaseChangeFoam` solver, and enhances the solver to account for source terms in the continuity and momentum rate equations. The project is structured into the following main steps:

1. Integrate the Rayleigh-Plesset equation to account for changes in bubble diameter in Lagrangian library.
2. Couple the modified Lagrangian library with the `interPhaseChangeFoam`.
3. Modify `interPhaseChangeFoam` to include source terms.
4. Develop a transition mechanism between the Eulerian and Lagrangian frameworks.

Readers can visit link[1] to make comments or get the latest updates on the solver.

[1]The link to solver is https://github.com/mlavari/interPhaseChangeBubbleFoam.

Chapter 2

Theoretical Framework

The author will delve into each component of OpenFOAM used in this study. To streamline this work, all parameters and their units referenced in the following equations are listed in the Nomenclature section.

2.1 interPhaseChangeFoam

There are many useful explanations of `interPhaseChangeFoam` in the proceedings of this PhD course such as by Ghasemi [6], Asnagi [7], and Lu [8]. Additionally, Andersen [9] has provided an insightful guide on how to use the `interPhaseChangeFoam` solver. While the author reviews the theory behind `interPhaseChangeFoam` based on these works, explaining the solver's tutorial case would be redundant.

`interPhaseChangeFoam` is an isothermal VOF solver that uses three FMT models—namely Kunz, Merkel, and Schnerr-Sauer—to simulate cavitation when the pressure falls below the vapor pressure. The core concept of this method is to treat the entire domain as a single fluid with varying percentages of volume fraction of liquid and vapor. As a result, the mass and momentum conservation equations are solved for a single fluid which is a mixture of liquid and vapor. Therefore, the key distinction lies in the viscosity and density used in these equations, which are functions of fractions of liquid and vapor volumes. These fractions are governed by Reynolds transport equation for the volume fraction, augmented with a source term that accounts for condensation and vaporization in FMT models.

An effective approach to studying the Homogeneous Mixture Model, or the Volume of Fluid(VOF) method combined with Finite Mass Transfer(FMT), is to structure the analysis into two key components: VOF and FMT.

2.1.1 Volume of Fluid(VOF)

As mentioned earlier, the main concept in VOF is to consider a mixture of vapor and liquid. As a result, flow parameters such as density and viscosity must be defined in their mixture form. This mixture is constructed using the volume fraction of one of the two phases. Here, we use the liquid volume fraction, which represents the volume of liquid divided by the volume of the computational cell. Consequently, the vapor volume fraction is given by

$$\alpha_{\mathrm{v}} = 1 - \alpha_{\mathrm{l}}, \quad \alpha_{\mathrm{l}} \in [0,1], \quad \alpha_{\mathrm{v}} \in [0,1]. \tag{2.1}$$

The conservation of mass and momentum for VOF, assuming a Newtonian fluid, are expressed as

$$\frac{\partial \rho_{\mathrm{m}}}{\partial t} + \nabla \cdot (\rho_{\mathrm{m}} \mathbf{U}) = 0, \tag{2.2}$$

$$\frac{\partial}{\partial t}(\rho_{\mathrm{m}} u_j) + \frac{\partial}{\partial x_j}(\rho_{\mathrm{m}} u_i u_j) = -\frac{\partial p}{\partial x_i} + \frac{\partial}{\partial x_j}\left[(\mu_{\mathrm{m}} + \mu_{\mathrm{laminar}})\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} - \frac{2}{3}\frac{\partial u_k}{\partial x_k}\delta_{ij}\right)\right] + \rho_{\mathrm{m}} g. \quad (2.3)$$

Here,

$$\rho_{\mathrm{m}} = \alpha_{\mathrm{l}}\rho_{\mathrm{l}} + (1-\alpha_{\mathrm{l}})\rho_{\mathrm{v}}, \quad (2.4)$$

$$\mu_{\mathrm{m}} = \alpha_{\mathrm{l}}\mu_{\mathrm{l}} + (1-\alpha_{\mathrm{l}})\mu_{\mathrm{v}}. \quad (2.5)$$

The transport equation for volume fractions of liquid and vapor are given by

$$\frac{\partial \alpha_{\mathrm{l}}}{\partial t} + \nabla \cdot (\alpha_{\mathrm{l}}\mathbf{U}) = \frac{\dot{m}}{\rho_{\mathrm{l}}}, \quad (2.6)$$

$$\frac{\partial \alpha_{\mathrm{v}}}{\partial t} + \nabla \cdot (\alpha_{\mathrm{v}}\mathbf{U}) = -\frac{\dot{m}}{\rho_{\mathrm{v}}}. \quad (2.7)$$

Here, $\dot{m}$ is the source term representing phase interactions, such as evaporation or condensation, where FMT plays a role, which will be discussed in Section 2.1.2.

In OpenFOAM, `interPhaseChangeFoam` utilizes the PIMPLE algorithm, and the continuity equation acts as a constraint on the conservation of momentum rate equation. In an incompressible solver, the divergence of velocity is zero. However, in a cavitating flow, since there is generation and destruction of liquid and vapor, the divergence of velocity is no longer zero. By adding Eq. (2.6) and Eq. (2.7), we arrive at

$$\nabla \cdot \mathbf{U} = \left(\frac{1}{\rho_{\mathrm{l}}} - \frac{1}{\rho_{\mathrm{v}}}\right)\dot{m}. \quad (2.8)$$

Eq. (2.6) and Eq. (2.8) are implemented in `alphaEqn.H` and `pEqn.H`, respectively. From a numerical perspective, to make the solver more diagonally dominant, more stable, and to improve the interface capturing efficiency, minor modifications are made to these equations.

Regarding Eq. (2.6), by adding and subtracting $\alpha_{\mathrm{l}}\nabla \cdot \mathbf{U}$ (with the help of Eq. (2.8)) from the right-hand side of Eq. (2.6), we arrive at

$$\frac{\partial \alpha_{\mathrm{l}}}{\partial t} + \nabla \cdot (\alpha_{\mathrm{l}}\mathbf{U}) = \frac{\dot{m}}{\rho_{\mathrm{l}}} + \alpha_{\mathrm{l}}\nabla \cdot \mathbf{U} - \alpha_{\mathrm{l}}\left(\frac{1}{\rho_{\mathrm{l}}} - \frac{1}{\rho_{\mathrm{v}}}\right)\dot{m}. \quad (2.9)$$

Rearranging Eq. (2.9), we get

$$\frac{\partial \alpha_{\mathrm{l}}}{\partial t} + \nabla \cdot (\alpha_{\mathrm{l}}\mathbf{U}) = \left(\frac{1}{\rho_{\mathrm{l}}} - \alpha_{\mathrm{l}}\left(\frac{1}{\rho_{\mathrm{l}}} - \frac{1}{\rho_{\mathrm{v}}}\right)\right)\dot{m} + \alpha_{\mathrm{l}}\nabla \cdot \mathbf{U}. \quad (2.10)$$

By defining $\dot{V} = \left(\frac{1}{\rho_{\mathrm{l}}} - \alpha_{\mathrm{l}}\left(\frac{1}{\rho_{\mathrm{l}}} - \frac{1}{\rho_{\mathrm{v}}}\right)\right)$, and given that $\dot{m} = (\alpha_{\mathrm{l}}\dot{m}_{\mathrm{v}} + (1-\alpha_{\mathrm{l}})\dot{m}_c)$ represents the mass transfer rate for condensation and vaporization, Eq. (2.10) transforms to

$$\frac{\partial \alpha_{\mathrm{l}}}{\partial t} + \nabla \cdot (\alpha_{\mathrm{l}}\mathbf{U}) - \alpha_{\mathrm{l}}\nabla \cdot \mathbf{U} = \dot{V}\left(\alpha_{\mathrm{l}}\dot{m}_{\mathrm{v}} + (1-\alpha_{\mathrm{l}})\dot{m}_c\right). \quad (2.11)$$

By defining $\dot{V}_{\mathrm{v}} = \dot{V}\dot{m}_{\mathrm{v}}$ and $\dot{V}_c = \dot{V}\dot{m}_c$, and reorganizing Eq.(2.11), we arrive at

$$\frac{\partial \alpha_{\mathrm{l}}}{\partial t} + \nabla \cdot (\alpha_{\mathrm{l}}\mathbf{U}) - \alpha_{\mathrm{l}}\nabla \cdot \mathbf{U} = \alpha_{\mathrm{l}}\dot{V}_{\mathrm{v}} - \alpha_{\mathrm{l}}\dot{V}_c + \dot{V}_c. \quad (2.12)$$

Eq. (2.12) shows how the liquid volume fraction interface tracking is implemented in `alphaEqn.H` as shown in Listing 2.1.

Listing 2.1: Liquid volume fraction equation in `alphaEqn.H`

```
fvScalarMatrix alpha1Eqn
(
    fv::EulerDdtScheme<scalar>(mesh).fvmDdt(alpha1)
  + fv::gaussConvectionScheme<scalar>
    (
        mesh,
        phi,
        upwind<scalar>(mesh, phi)
    ).fvmDiv(phi, alpha1)
  - fvm::Sp(divU, alpha1)
 ==
    fvm::Sp(vDotvmcAlphal, alpha1)
  + vDotcAlphal
);
```

Regarding the continuity equation implemented in `interPhaseChangeFoam`, from this present course material, the discretized linearized momentum equation, with the pressure gradient retained, can be given by

$$a_P^u u_P + \sum_N a_N^u u_N = r - \nabla p. \tag{2.13}$$

Here, a_P^u and a_N^u are the diagonal and non-diagonal parts of the coefficient matrix, respectively. r is the source term, and p is the pressure. In OpenFOAM, the operator $\mathbf{H}(\mathbf{u})$ is introduced, which includes the source terms in addition to the non-diagonal terms of the coefficient matrix, and defined as

$$\mathbf{H}(\mathbf{u}) = r - \sum_N a_N^u u_N. \tag{2.14}$$

Substituting Eq. (2.14) into Eq. (2.13) yields

$$a_P^u u_P = \mathbf{H}(\mathbf{u}) - \nabla p \tag{2.15}$$

or

$$u_P = (a_P^u)^{-1} \left(\mathbf{H}(\mathbf{u}) - \nabla p\right). \tag{2.16}$$

Regarding the divergence of velocity, Eq. (2.8), substituting Eq. (2.16) into the divergence of velocity equation, Eq. (2.8), yields the pressure equation for incompressible flow given by

$$\nabla \cdot \left[(a_P^u)^{-1} \nabla p\right] - \nabla \cdot \left[(a_P^u)^{-1} \mathbf{H}(\mathbf{u})\right] - \left(\frac{1}{\rho_\text{l}} - \frac{1}{\rho_\text{v}}\right) \dot{m} = 0. \tag{2.17}$$

Eq. (2.17) is implemented within `pEqn.H`, as shown in Listing 2.2.

Listing 2.2: Continuity equation in `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)
);
```

2.1.2 Finite Mass Transfer(FMT)

In the above set of equations, the main component to close the equations, modeling finite mass transfer $\dot{m}$, needs to be addressed. In OpenFOAM, three main mass transfer models—namely Kunz, Merkel, and Schnerr-Sauer—are implemented in classes with the same name, all of which inherit from the `incompressibleTwoPhaseMixture` class. Since the subject of interest for this research is the Schnerr-Sauer model, the other two models will not be covered here.

2.1.2.1 Schnerr-Sauer

As Lavari *et al.* [10] investigated, this model employs a simplified Rayleigh-Plesset equation, and cavitation is initiated in regions where the pressure drops below the saturation vapor pressure, through the mass transfer source term which is given by

$$\dot{m}_c = C_c \alpha_\mathrm{l}(1-\alpha_\mathrm{l}) \frac{3\rho_\mathrm{l}\rho_\mathrm{v}}{\rho_\mathrm{m} R_B} \left(\frac{2}{3\rho_\mathrm{l}} |p - p_\mathrm{sat}| \right)^{1/2}, \quad p \geq p_\mathrm{sat}, \tag{2.18}$$

$$\dot{m}_\mathrm{v} = C_\mathrm{v} \alpha_\mathrm{l} (1 + \alpha_{Nc} - \alpha_\mathrm{l}) \frac{3\rho_\mathrm{l}\rho_\mathrm{v}}{\rho_\mathrm{m} R_B} \left(\frac{2}{3\rho_\mathrm{l}} |p - p_\mathrm{sat}| \right)^{1/2}, \quad p < p_\mathrm{sat}. \tag{2.19}$$

Here, $\dot{m}_c$ and $\dot{m}_\mathrm{v}$ represent the rates of condensation and vaporization, respectively. R_B and α_{Nc} denote the radius and volume fraction of bubble nuclei in the liquid. These parameters are determined as

$$\alpha_{Nc} = \frac{\pi n_0 d_{Nc}^3}{6} \left(1 + \frac{\pi n_0 d_{Nc}^3}{6} \right)^{1/2}, \tag{2.20}$$

$$R_B = \left(\frac{3}{4\pi n_0} \frac{1 + \alpha_{Nc} - \alpha}{\alpha} \right)^{1/3}. \tag{2.21}$$

Here, n_0 and d_{N_c} are pre-defined parameters, signifying the number of nuclei per liquid cubic meter and the diameter of nucleation sites, respectively.

2.2 Lagrangian library

Since the goal of this project is to track nuclei as Lagrangian entities, it is necessary to address the movement of bubbles in the fluid by using Lagrangian tracking. Within the context of Lagrangian tracking, there are different types of tracking based on the ratio of the total volume fraction of Lagrangian bubbles to the volume of the entire domain. Therefore, the nature of coupling between the Eulerian field and the Lagrangian field is determined by this ratio. According to a classification map proposed by Elghobashi [11], shown in Figure 2.1, a one-way coupling can be used for highly diluted flows with a volume fraction less than 10^{-6}. Within this range, the flow of the Eulerian fluid influences the bubble trajectories, but the bubbles have a negligible effect on the background flow. On the other hand, for volume fractions greater than 10^{-6} and less than 10^{-3}, the bubbles affect the background flow. At this point, a two-way coupling must be used to account for the additional feedback force exerted on the flow by the bubbles. Based on this concept, two-way coupling is essential within the context of cavitation and this project.

Moreover, for volume fractions exceeding the value 10^{-3}, the interactions between the bubbles themselves become significant. In such cases, four-way coupling is required, where the dynamics of the system are influenced by both bubble-bubble interactions and the feedback effects between the bubbles and the surrounding fluid. As a result, collision, breakup, and coalescence of the Lagrangian vapor structures should be considered, and this would be a subject of interest for future investigation.

In OpenFOAM, the Lagrangian directory, as shown in Listing 2.3, contains `basic` directory containing core classes that handle particle reading, writing, and storage. In addition, there is the `cloud` class concept in `basic` directory, which manages and manipulates a list of particles. In addition to `basic` directory, which is used by all other parts of Lagrangian library, two main concepts are pre-existing for tracking non-evaporating and isothermal particles in Lagrangian library. These are `solidParticle` class and `KinematicParcel` class within the intermediate directory. Lopez [12] examined the differences between `solidParticle` class and `KinematicParcel` class. The primary difference can be summarized in the number of particles that can be tracked. The `solidParticle` class has a preliminary design and track each particle individually, while `KinematicParcel` class can track parcels of particles that share the same size, velocity and trajectory within each parcel.

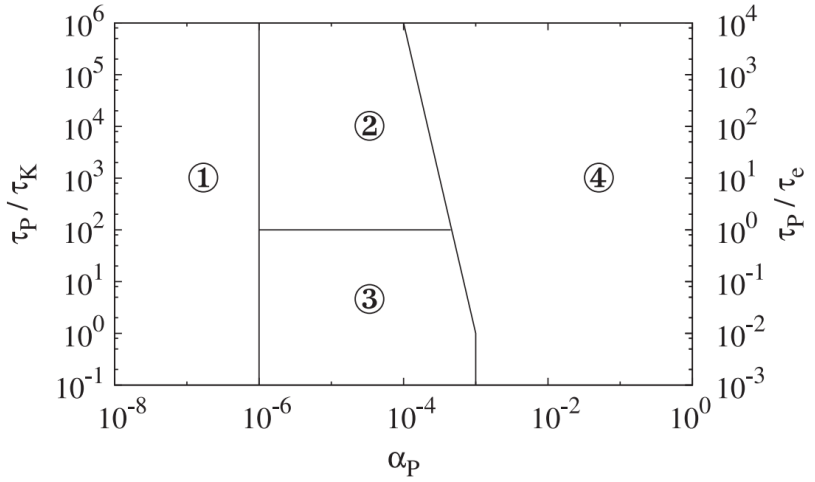


Figure 2.1: Classification of coupling schemes and interaction between particles and turbulence according to Elghobashi [11] for (1) one-way coupling, (2) two-way coupling where bubbles enhance turbulence production, (3) two-way coupling where bubbles enhance turbulence dissipation, and (4) four-way coupling

Listing 2.3: Lagrangian library directory

```
Lagrangian
    |-- DSMC
    |-- basic
    |-- coalCombustion
    |-- distributionModels
    |-- intermediate
    |-- molecularDynamics
    |-- solidParticle
    |-- spray
    '-- turbulence
```

There are many useful examples of using `solidParticle` tracking in the proceedings of this course. For instance, Andric [13] used the `solidParticle` library and introduced drag for non-spherical solid particles. However, due to the greater flexibility of tracking parcels of particles and the additional features of the `KinematicCloud` class, such as the inclusion of various injection models, pre-defined forces, patch interactions, and particle collision models, the `KinematicCloud` class has drawn more attention and, in turn, has been more developed. On the other hand, the simplicity of `solidParticle` class makes it easier to work with and modify from a programming perspective. Considering all of the above, studying and developing `KinematicCloud` class is of greater interest to this project.

2.2.1 KinematicCloud

The `KinematicCloud` directory resides in the `intermediate/clouds` directory alongside other cloud types, as shown in Table 2.1. All the cloud classes listed in the table are derived from the first item, `KinematicCloud` class. Items 2 and 3 deal with different collision methods, while the remaining items are reaction- or thermal-enabled `KinematicCloud` class. This project focuses only on `KinematicCloud` class, however, many other projects related to item 4-7 from the previous proceedings of this course can also be useful. Fjällborg [14] and Kampili [15] explored the heat transfer equation in other derivatives of `KinematicCloud` class and proposed various thermal equations for them. In addition to focusing on derivatives of `KinematicCloud` class, their work also offers valuable insights into `KinematicCloud` class itself, particularly Kampili's [15] study, which elaborates on the Eulerian and analytical calculations of ΔT, which is similar to ΔU used in `KinematicParcel` class

for applying force source terms. Regarding the forces acting on `KinematicParcel` class, Xu [16] provided sufficient information. For injection mechanisms in `KinematicCloud` class, reader can get valuable insights from the work of Nobile [17], although it focuses on `sprayFoam`.

Table 2.1: Cloud Types

Example	Cloud Name
1	KinematicCloud
2	CollidingCloud
3	MPPICCloud
4	ReactingCloud
5	ReactingHeterogeneousCloud
6	ReactingMultiphaseCloud
7	ThermoCloud

Within `intermediate/clouds` directory mentioned above, `KinematicCloud` class structure can be further divided into `derived` classes and `Templates` classes. In the `derived` directory, there is a file named `basicKinematicCloud.H`, which is solely the type definition of `KinematicCloud` class (with a capital K), showing the template class in OpenFOAM. The `basicKinematicCloud.H` file, as shown in Listing 2.4, reveals that `KinematicCloud` class is templated with `Cloud` class, which is further templated with `KinematicParcel` class. Therefore, `KinematicCloud` class inherits all features of `Cloud` class and `KinematicParcel` class.

Listing 2.4: A part of basicKinematicCloud.H

```
namespace Foam
{
    typedef KinematicCloud
    <
        Cloud
        <
            basicKinematicParcel
        >
    > basicKinematicCloud;
}
```

In practice, the `KinematicCloud` class has several member functions, such as `evolve()` and `motion()`, which call other member functions like `inject()` in the `injectionModel` class and more importantly, `motion()` which calls `move()` function in the `Cloud` class. The `move()` function iterates over the Lagrangian entities. Within this loop, another `move()` function of the `KinematicParcel` class is called for each parcel. All calculations related to forces, new velocities, and new positions of the Lagrangian parcels are performed within this member function of `KinematicParcel` class. It is noteworthy that a similar concept is also applicable, to some extent, in `solidCloud` class, though with much simpler structure. In the `KinematicParcel` class, the Lagrangian equations for tracking the trajectory of bubbles account for the effect of forces on the bubbles, and would be defined as

$$\frac{\mathrm{d}x_\mathrm{b}}{\mathrm{d}t} = u_\mathrm{b}, \tag{2.22}$$

$$m_b \frac{\mathrm{d}u_\mathrm{b}}{\mathrm{d}t} = F_d. \tag{2.23}$$

2.3 Bubble Dynamics

To track nuclei and their diameter evolution, considering bubble dynamics is essential. Since `KinematicParcel` class only tracks parcels of particles, not parcels of bubbles, the change in diameter must be defined in `KinematicParcel` class, so nuclei size can evolve and, if required, subsequently transfer to the Eulerian field. The most well-known bubble dynamics equation, ignoring

compressibility effects, is the Rayleigh-Plesset equation. Here, the author focuses on implementing this equation.

2.3.1 Rayleigh Plesset Equation

The bubble growth is modeled using a localized form of the Rayleigh-Plesset equation given as [10]

$$\rho_{\mathrm{l}}\left(\frac{1}{2}R\ddot{R}+\frac{1}{2}\dot{R}^2\right)=p_{\mathrm{v}}+p_{g0}\left(\frac{R_0}{R}\right)^{3k}-p_{xR}-4\mu_{\mathrm{l}}\frac{\dot{R}}{R}-\frac{2\sigma}{R}. \tag{2.24}$$

Here, R, $\dot{R}$, and $\ddot{R}$ represent the radius of the bubble and its first and second temporal derivatives, respectively. The right-hand side of Eq. (2.24) includes the vapor pressure, p_{v}, and the dissolved gas pressure, p_{g0}, terms. Here, p_{g0} and R_0 denote the initial equilibrium gas pressure and radius, respectively. The exponent k is set to 1 for isothermal behavior and to γ (the gas polytropic constant) for adiabatic variations in the bubble's radius. Additionally, p_{xR} represents the surface-average pressure of the mixture over a concentric sphere with radius xR, where x is a user-defined factor and later would be called `averagingDistance` in solver, providing a representation of the local pressure around the bubble.

The implementation of the Rayleigh-Plesset equation was explored in the proceedings of this course. Ghahramani [18] implemented a Rayleigh-Plesset solver within `move()` member function of `solidParticle` class, while Chen [1] coded the Rayleigh-Plesset solver within `KinematicParcel` class, utilizing ODE solver provided by OpenFOAM. In this work, author adopted the same concept from Chen's work [1], with modifications to accommodate OpenFOAM-v2406, include localized Rayleigh-Plesset equation, adjust where Rayleigh-Plesset solver is located in the code, and modify the number of inputs as well as how they are read by the solver. As opposed to Chen's work, the author does not use radius; instead, only the diameter of the bubble is tracked. The dictionary containing bubble dynamics inputs is transferred to `KinematicParcel` class constructor, and each parcel has access to all the information from the `bubbleProperties` dictionary. Since the code and report from Chen [1] are available, the implementation of the Rayleigh-Plesset equation is not discussed here, instead, only the modifications are outlined. Here, the author names modified `KinematicCloud`, and `KinematicParcel` by incorporating localized Rayleigh-Plesset equation as `KinematicBubbleCloud` and `KinematicBubbleParcel`, respectively.

Chapter 3

Implementation of interPhaseChangeBubbleFoam

Although nuclei are being tracked in the Lagrangian framework to reduce computational cost, the Eulerian framework remains the primary platform for tracking both Eulerian bubbles (resolved structures/vapors) and Lagrangian bubbles (unresolved structures/vapors). In other words, while Lagrangian bubbles are not explicitly tracked in the Eulerian framework, they still exist and affect the conservation of mass and momentum within the Eulerian field. This oversight in previous hybrid Lagrangian-Eulerian models leads to violations of these conservation laws. Consequently, coupling both frameworks must involve a reliable and mathematically proven exchange of source terms. To address this, the author proposes a rigorous set of coupled equations to track both Lagrangian and Eulerian vapor structures.

In the proceedings of this course, many projects have focused on coupling the Eulerian and Lagrangian frameworks. For example, Vallier [19] initially attempted to couple Lagrangian particle tracking with `icoFoam`, and later, in their Ph.D. work [3], introduced algorithms for transitioning between the Eulerian and Lagrangian frameworks. However, their approach allowed Lagrangian and Eulerian vapor structures to coexist within the same cell, leading to unphysical results. Later, Ghahramani [2] identified this non-conservation of mass and introduced β_l, a Lagrangian liquid volume fraction, as a switch in the code to prevent the coexistence of Lagrangian and Eulerian vapors within the same cell. However, without adequately modifying the source terms, the model would fail to provide accurate source term values to the Eulerian field, making the proper tracking of nuclei unachievable. This work would bridge the gap by mathematically calculating and proving these source terms and implementing them in a code.

3.1 Theory

To make the multiscale Lagrangian-Eulerian cavitation approach more consistent and understandable, it can be viewed as resolved and unresolved flow: one continuous liquid phase (resolved liquid) and two dispersed vapor phases (i.e., vapor in the Eulerian framework (resolved structure/vapor) and vapor in the Lagrangian framework (unresolved structure/vapor)). However, some assumptions must be considered:

- The two dispersed vapors (resolved and unresolved vapors) share the same physical properties, such as density and viscosity.
- The two dispersed vapors (resolved and unresolved vapors) cannot coexist in the same location. In other words, there is no mixture of resolved and unresolved vapor at any point in the entire domain. However, the resolved liquid can form a mixture with both resolved and unresolved vapors.

- There is no slip velocity between the resolved liquid and resolved vapor, whereas slip conditions are considered between unresolved vapor and resolved liquid.

Therefore, the inherent concept is that when a structure is transferred to the Lagrangian framework, it should not be considered as being removed from the Eulerian field. It remains in the Eulerian field, but its position, mass, and momentum equations are solved within the Lagrangian framework.

Now, we take advantage of β_l for the unresolved liquid volume fraction as introduced by Ghahramani [2]. In each cell, where there is unresolved vapor in the Lagrangian framework, β_l takes values from 0 to less than 1, while α, which represents the resolved liquid fraction, remains 1. As a result, it can be given by

$$\beta_\mathrm{l} \in [0, 1), \quad \text{where} \quad \alpha_\mathrm{l} = 1. \tag{3.1}$$

On the other hand, α_l would be given by

$$\alpha_\mathrm{l} \in [0, 1], \quad \text{where} \quad \beta_\mathrm{l} = 1. \tag{3.2}$$

Using α_l and β_l based on the above assumptions, the definitions of density, viscosity, and momentum are reduced to

$$\rho_\mathrm{m} = (\alpha_\mathrm{l} + \beta_\mathrm{l} - 1)\rho_\mathrm{l} + (1 - \alpha_\mathrm{l})\rho_\mathrm{v} + (1 - \beta_\mathrm{l})\rho_\mathrm{v}, \tag{3.3}$$

$$\mu_\mathrm{m} = (\alpha_\mathrm{l} + \beta_\mathrm{l} - 1)\mu_\mathrm{l} + (1 - \alpha_\mathrm{l})\mu_\mathrm{v} + (1 - \beta_\mathrm{l})\mu_\mathrm{v}, \tag{3.4}$$

$$\rho_\mathrm{m} u_i = (\alpha_\mathrm{l} + \beta_\mathrm{l} - 1)\rho_\mathrm{l} u_i + (1 - \alpha_\mathrm{l})\rho_\mathrm{v} u_i + (1 - \beta_\mathrm{l})\rho_\mathrm{v} u_{Li}. \tag{3.5}$$

Here, u_{Li} is the unresolved velocity in the Lagrangian framework, which differs from u_i in the Eulerian field based on the slip assumption. By setting $\beta_\mathrm{l} = 1$, the well-known VOF equations are derived. By substituting Eq. (3.3) and Eq. (3.5) into Eq. (2.2), the mass conservation would be formed as

$$\begin{aligned}
&\left[\frac{\partial\left(\alpha_\mathrm{l}\rho_\mathrm{l} + (1-\alpha_\mathrm{l})\rho_\mathrm{v}\right)}{\partial t} + \frac{\partial\left(\alpha_\mathrm{l}\rho_\mathrm{l} u_i + (1-\alpha_\mathrm{l})\rho_\mathrm{v} u_i\right)}{\partial x_i}\right] \\
&+ \left[\frac{\partial\left(\beta_\mathrm{l}\rho_\mathrm{l} + (1-\beta_\mathrm{l})\rho_\mathrm{v}\right)}{\partial t} + \frac{\partial\left(\beta_\mathrm{l}\rho_\mathrm{l} u_i + (1-\beta_\mathrm{l})\rho_\mathrm{v} u_{Li}\right)}{\partial x_i}\right] \\
&- \rho_\mathrm{l}\left[\frac{\partial u_i}{\partial x_i}\right] = 0.
\end{aligned} \tag{3.6}$$

Where $\alpha_\mathrm{l} = 1$, the cell has unresolved vapor, and the divergence of velocity can be given as

$$\nabla \cdot \mathbf{U} = -\frac{1}{\beta_\mathrm{l}\rho_\mathrm{l}}\left[\frac{\partial(1-\beta_\mathrm{l})\rho_\mathrm{v}}{\partial t} + \frac{\partial\left((1-\beta_\mathrm{l})\rho_\mathrm{v} u_{Li}\right)}{\partial x_i}\right] - \frac{1}{\beta_\mathrm{l}}\left[\frac{\partial \beta_\mathrm{l}}{\partial t} + u_i\frac{\partial \beta_\mathrm{l}}{\partial x_i}\right]. \tag{3.7}$$

Where $\beta_\mathrm{l} = 1$, the cell has resolved vapor, and the divergence of velocity can be given as

$$\nabla \cdot \mathbf{U} = \frac{\rho_\mathrm{v} - \rho_\mathrm{l}}{\rho_\mathrm{v}}\left[\frac{\partial \alpha_\mathrm{l}}{\partial t} + \frac{\partial(\alpha_\mathrm{l} u_i)}{\partial x_i}\right]. \tag{3.8}$$

It should be noted that Eq. (3.8) is the same as Eq. (2.8). As a result, the divergence of velocity for this hybrid model can be given as

$$\nabla \cdot \mathbf{U} = \frac{\rho_\mathrm{v} - \rho_\mathrm{l}}{\rho_\mathrm{v}}\left[\frac{\partial \alpha_\mathrm{l}}{\partial t} + \frac{\partial(\alpha_\mathrm{l} u_i)}{\partial x_i}\right] - \frac{1}{\beta_\mathrm{l}\rho_\mathrm{l}}\left[\frac{\partial(1-\beta_\mathrm{l})\rho_\mathrm{v}}{\partial t} + \frac{\partial\left((1-\beta_\mathrm{l})\rho_\mathrm{v} u_{Li}\right)}{\partial x_i}\right] - \frac{1}{\beta_\mathrm{l}}\left[\frac{\partial \beta_\mathrm{l}}{\partial t} + u_i\frac{\partial \beta_\mathrm{l}}{\partial x_i}\right]. \tag{3.9}$$

Regarding the conservation of momentum, by substituting Eq.(3.3) and Eq. (3.5) into Eq. (2.3), we arrive at

$$\begin{aligned}\left[\frac{\partial\left(\alpha\rho_{\mathrm{l}}+(1-\alpha)\rho_{\mathrm{v}}\right)u_i}{\partial t}+\frac{\partial\left(\alpha\rho_{\mathrm{l}}+(1-\alpha)\rho_{\mathrm{v}}\right)u_iu_j}{\partial x_j}\right]-\left[\frac{\partial\left((1-\beta_{\mathrm{l}})\rho_{\mathrm{l}}u_i\right)}{\partial t}+\frac{\partial\left((1-\beta_{\mathrm{l}})\rho_{\mathrm{l}}u_iu_j\right)}{\partial x_j}\right]\\ =-\left[\frac{\partial\left((1-\beta_{\mathrm{l}})\rho_{\mathrm{v}}u_{Li}\right)}{\partial t}+\frac{\partial\left((1-\beta_{\mathrm{l}})\rho_{\mathrm{v}}u_{Li}u_j\right)}{\partial x_j}\right]\\ -\frac{\partial p}{\partial x_i}+\frac{\partial\tau_{ij}}{\partial x_j}+\rho_{\mathrm{m}}g_i.\end{aligned}\tag{3.10}$$

In previous works, Eq. (2.23) has been identified as the Lagrangian coupling source term. However, due to the change in volume of unresolved vapor and the assumption of the same density for both resolved and unresolved vapors, there is a rate of change of mass in the Lagrangian framework that has been neglected in those works. As a result, Eq. (2.23) no longer reflects the true behavior. Consequently, Eq. (2.23) from the Lagrangian framework should be modified, and its relation with the Eulerian framework should be explored.

To relate the Eulerian and Lagrangian frameworks, we use the definition of the total derivative to connect them. The total derivative of mass (for a cell volume) can be given as

$$\dot{m}_{\mathrm{bubble}}=\frac{1}{V}\frac{m_2-m_1}{t_2-t_1}=\frac{1}{V}\frac{\mathrm{D}m_{\mathrm{bubble}}}{\mathrm{D}t}\tag{3.11}$$

In the Eulerian framework, and by using the equation of mass conservation for vapor $\frac{1}{V}\frac{\mathrm{D}V}{\mathrm{D}t}=\nabla\cdot\mathbf{u}=\frac{\partial u_{L_i}}{\partial x_i}$, we end up with

$$\begin{aligned}\frac{1}{V}\frac{\mathrm{D}m_{\mathrm{bubble}}}{\mathrm{D}t}&=\frac{1}{V}\frac{\mathrm{D}(\rho_{\mathrm{v}}(1-\beta_{\mathrm{l}})V)}{\mathrm{D}t}\\ &=\rho_{\mathrm{v}}\frac{1}{V}\left[(1-\beta_{\mathrm{l}})\frac{\mathrm{D}V}{\mathrm{D}t}+V\frac{\mathrm{D}(1-\beta_{\mathrm{l}})}{\mathrm{D}t}\right]\\ &=\rho_{\mathrm{v}}(1-\beta_{\mathrm{l}})\frac{1}{V}\frac{\mathrm{D}V}{\mathrm{D}t}+\rho_{\mathrm{v}}\left[\frac{\partial(1-\beta_{\mathrm{l}})}{\partial t}+u_{L_i}\frac{\partial(1-\beta_{\mathrm{l}})}{\partial x_i}\right]\\ &=\left[(1-\beta_{\mathrm{l}})\frac{\partial(\rho_{\mathrm{v}}u_{L_i})}{\partial x_i}+\frac{\partial(1-\beta_{\mathrm{l}})\rho_{\mathrm{v}}}{\partial t}+u_{L_i}\frac{\partial(1-\beta_{\mathrm{l}})\rho_{\mathrm{v}}}{\partial x_i}\right]\\ &=\left[\frac{\partial(1-\beta_{\mathrm{l}})\rho_{\mathrm{v}}}{\partial t}+\frac{\partial(1-\beta_{\mathrm{l}})\rho_{\mathrm{v}}u_{L_i}}{\partial x_i}\right].\end{aligned}\tag{3.12}$$

Regarding the rate of change of momentum by taking the same approach, the total force (for a cell volume) acting over unresolved vapors is given by

$$\frac{F}{V}=\frac{1}{V}\frac{\mathrm{D}(m_{\mathrm{bubble}}u_{L_i})}{\mathrm{D}t}=\frac{1}{V}\frac{m_2u_{L_2}-m_1u_{L_1}}{t_2-t_1}=\frac{1}{V}\frac{\mathrm{D}V(1-\beta_{\mathrm{l}})\rho_{\mathrm{v}}u_{L_i}}{\mathrm{D}t}.\tag{3.13}$$

Again, in the Eulerian framework, we end up with

$$
\begin{aligned}
\frac{1}{V}\frac{\mathrm{D}V(1-\beta_\mathrm{l})\rho_\mathrm{v}u_{L_i}}{\mathrm{D}t} &= \frac{1}{V}\left[\frac{\partial(V(1-\beta_\mathrm{l})\rho_\mathrm{v}u_{L_i})}{\partial t} + u_j\frac{\partial(V(1-\beta_\mathrm{l})\rho_\mathrm{v}u_{L_i})}{\partial x_j}\right] \\
&= \frac{\rho_\mathrm{v}}{V}\left[(1-\beta_\mathrm{l})u_{L_i}\frac{\partial V}{\partial t} + V\frac{\partial((1-\beta_\mathrm{l})u_{L_i})}{\partial t} + (1-\beta_\mathrm{l})u_{L_i}u_j\frac{\partial V}{\partial x_j} + Vu_j\frac{\partial((1-\beta_\mathrm{l})u_{L_i})}{\partial x_j}\right] \\
&= \frac{\rho_\mathrm{v}}{V}(1-\beta_\mathrm{l})u_{L_i}\left[\frac{\partial V}{\partial t} + u_j\frac{\partial V}{\partial x_j}\right] + \rho_\mathrm{v}\left[\frac{\partial((1-\beta_\mathrm{l})u_{L_i})}{\partial t} + u_j\frac{\partial((1-\beta_\mathrm{l})u_{L_i})}{\partial x_j}\right] \\
&= (1-\beta_\mathrm{l})\rho_\mathrm{v}u_{L_i}\frac{1}{V}\frac{\mathrm{D}V}{\mathrm{D}t} + \rho_\mathrm{v}\left[\frac{\partial((1-\beta_\mathrm{l})u_{L_i})}{\partial t} + u_j\frac{\partial((1-\beta_\mathrm{l})u_{L_i})}{\partial x_j}\right] \\
&= (1-\beta_\mathrm{l})\rho_\mathrm{v}u_{L_i}\frac{\partial u_j}{\partial x_j} + \rho_\mathrm{v}\frac{\partial((1-\beta_\mathrm{l})u_{L_i})}{\partial t} + \rho_\mathrm{v}u_j\frac{\partial((1-\beta_\mathrm{l})u_{L_i})}{\partial x_j} \\
&= \frac{\partial(1-\beta_\mathrm{l})\rho_\mathrm{v}u_{L_i}}{\partial t} + \frac{\partial\left((1-\beta_\mathrm{l})\rho_\mathrm{v}u_{L_i}u_j\right)}{\partial x_j}.
\end{aligned}
\tag{3.14}
$$

Note that when $\beta_\mathrm{l} = 1$, both Eq. (3.14) and Eq. (3.12) become zero, and the equations automatically reduce to VOF set of equations. By substituting Eq. (3.14), Eq. (3.12), and Eq. (2.6) into Eq. (3.9) and Eq. (3.10), we end up with

$$
\nabla\cdot\mathbf{U} = \left(\frac{1}{\rho_\mathrm{l}} - \frac{1}{\rho_\mathrm{v}}\right)\dot{m} - \frac{1}{\beta_\mathrm{l}\rho_\mathrm{l}}\left[\dot{m}_\mathrm{bubble}\right] - \frac{1}{\beta_\mathrm{l}}\left[\frac{D\beta_\mathrm{l}}{Dt}\right], \tag{3.15}
$$

$$
\begin{aligned}
\left[\frac{\partial\left(\alpha\rho_\mathrm{l} + (1-\alpha)\rho_\mathrm{v}\right)u_i}{\partial t} + \frac{\partial\left(\alpha\rho_\mathrm{l} + (1-\alpha)\rho_\mathrm{v}\right)u_iu_j}{\partial x_j}\right] - \left[\frac{\partial\left((1-\beta_\mathrm{l})\rho_\mathrm{l}u_i\right)}{\partial t} + \frac{\partial\left((1-\beta_\mathrm{l})\rho_\mathrm{l}u_iu_j\right)}{\partial x_j}\right] \\
= -\left[\frac{F}{V}\right] \\
- \frac{\partial p}{\partial x_i} + \frac{\partial\tau_{ij}}{\partial x_j} + \rho_\mathrm{m}g_i.
\end{aligned}
\tag{3.16}
$$

The above equations demonstrate consistency between both frameworks. Consequently, Eq. (3.15) is implemented in this work. However, to simplify the process of implementation within a code, the author defines γ as the product of α_l and β_l, $\gamma = \alpha_\mathrm{l}\beta_\mathrm{l}$, such that density and viscosity are given by

$$
\rho_\mathrm{m} = \gamma\rho_\mathrm{l} + (1-\gamma)\rho_\mathrm{v}, \tag{3.17}
$$

$$
\mu_\mathrm{m} = \gamma\mu_\mathrm{l} + (1-\gamma)\mu_\mathrm{v}, \tag{3.18}
$$

$$
\rho_\mathrm{m}u_i = \gamma\rho_\mathrm{l}u_i + (1-\alpha_\mathrm{l})\rho_\mathrm{v}u_i + (1-\beta_\mathrm{l})\rho_\mathrm{v}u_{L_i}. \tag{3.19}
$$

Now, Eq. (3.16) can be expressed in a compact form–hence, simplifying the implementation within a code–given as

$$
\left[\frac{\partial\left(\gamma\rho_\mathrm{l} + (1-\alpha)\rho_\mathrm{v}\right)u_i}{\partial t} + \frac{\partial\left(\gamma\rho_\mathrm{l} + (1-\alpha)\rho_\mathrm{v}\right)u_iu_j}{\partial x_j}\right] = -\left[\frac{F}{V}\right] - \frac{\partial p}{\partial x_i} + \frac{\partial\tau_{ij}}{\partial x_j} + \rho_\mathrm{m}g_i. \tag{3.20}
$$

3.2 Implementing KinematicBubbleCloud

It was emphasized that the `KinematicCloud` class is being used. However, to make this class suitable for tracking bubbles, Rayleigh-Plesset equation should be included. The author has utilized concepts from Chen's work [1], enabling the reader to follow the steps in that study and implement a Rayleigh-Plesset solver in their own work. Here, the author only explains the differences.

Listing 3.1: Rayleigh-Plesset Class Directory

```
RPEqu/
    |-- RPEqu.H
    |-- RPEquI.H
    `-- bubbleDynamics.H
```

The Rayleigh-Plesset class is structured as shown in Listing 3.1. The `RPEqu.H` file contains the declarations of variables and functions to solve the Rayleigh-Plesset equation, while `RPEquI.H` includes definitions of inline functions. Readers are encouraged to refer to Chen's work [1] for further details. The `bubbleDynamics.H` file, as shown in Listing 3.2, acts as a coupling agent, gathering inputs from `KinematicParcel` class, solving, and sending data to the Rayleigh-Plesset class (`RPEqu.H`) through the member function `setValues()` of the `RPEqu` class. This structure facilitates modular and efficient integration of the Rayleigh-Plesset model into `KinematicParcel` class.

Listing 3.2: bubbleDynamics.H file

```
//-ML: Set tracking values from cells
this->setCellValues(cloud, td, this->averagingDistance());
//-ML: Create the selected ODE system solver
RPEqu_.setValues(
                td.pc(),
                td.rhoc(),
                td.muc(),
                this->bubbleSigma(),
                this->p0(),
                this->R0(),
                this->pv(),
                this->bubbleKappa()
                );


dictionary solverType;
solverType.add("solver", this->ODESolverType());

autoPtr<ODESolver> RPEquSolver = ODESolver::New(RPEqu_, solverType);
scalar xStart = this->age();
const scalar dx = this->RPdT();

//-ML: Initial radius and radiusGrowthRate
scalarField yStart(RPEqu_.nEqns());
yStart[0] = (this->d()/2);
yStart[1] = this->R_dot_;

//-ML: Integration initial step
scalar dxEst = 1e-10;
scalar xEnd = 0;

//-ML: Required to store dydx
scalarField dyStart(RPEqu_.nEqns());
const label n = dt/dx;

//-ML: Integration loop
for (label i=0; i<n; i++)
{
    xEnd = xStart + dx;
    RPEqu_.derivatives(xStart, yStart, dyStart);
    RPEquSolver->relTol() = 1e-5; //-ML:For runge kuta
    RPEquSolver->solve(xStart, xEnd, yStart, dxEst);
    xStart = xEnd;
}

this->d_ = (yStart[0]*2);
this->R_dot_ = yStart[1];

this->setCellValues(cloud, td);
```

The code shown in Listing 3.2 begins by initializing cell values from the field mesh using `setCellValues`, which extracts properties like pressure, density, and viscosity for use in calculations. The Rayleigh-Plesset class is configured with fluid properties (`td.rhoc`, `td.muc`) and bubble-specific parameters such as surface tension (`this->bubbleSigma`), vapor pressure (`this->pv`), and initial bubble radius (`this->R0`). These parameters are passed to Rayleigh-Plesset class using `RPEqu_.setValues`. An ODE solver is created based on the user-defined solver type (`this->ODESolverType`), which is used to integrate Rayleigh-Plesset equation. After initial conditions for the integration are set, the simulation start time (`xStart`) is derived from the bubble's age, and the ODE solver time step (`dx`) is defined by `this->RPdT()`. The initial bubble radius and growth rate are initialized using the current diameter (`this->d()`) and growth rate (`this->R_dot_`), respectively. Then, the main integration loop iteratively solves ODE system over a defined time interval (`dt/dx`). At each iteration, the code computes Rayleigh-Plesset derivatives, updates the state variables, and integrates the system forward using the configured ODE solver (`RPEquSolver`). The results are stored in `yStart`, which contains the updated bubble radius and growth rate. After completing the integration, the updated bubble diameter (`this->d_`) and growth rate (`this->R_dot_`) are calculated from the solution. Finally, the `setCellValues` function is called again to update the field values. This process tracks the bubble's evolution over time while maintaining consistency with the surrounding fluid dynamics.

3.2.1 Coupling the Rayleigh-Plesset class with KinematicCloud

To start with, user may copy and paste `KinematicCloud` directories, shown in Listing 3.3, from the directory `src/lagrangian/intermediate` and add the term `Bubble` after `Kinematic` to create new classes. The `KinematicCloud` class is renamed to `KinematicBubbleCloud` class to emphasize bubble-specific simulations. These name modifications should be performed in both the directory and file names, as well as within the contents of these files.

Listing 3.3: kinematicCloud/Parcel Directories

```
intermediate/
    |-- clouds
    |   |-- Templates
    |   |   |-- CollidingCloud
    |   |   |-- KinematicBubbleCloud
    |   |   `-- MPPICCloud
    |   |-- baseClasses
    |   |   `-- kinematicBubbleCloud
    |   `-- derived
    |       |-- basicKinematicBubbleCloud
    |       |-- basicKinematicBubbleCollidingCloud
    |       `-- basicKinematicBubbleMPPICCloud
    |-- lnInclude
    `-- parcels
        |-- Templates
        |   |-- CollidingParcel
        |   |   `-- CollisionRecordList
        |   |       |-- PairCollisionRecord
        |   |       `-- WallCollisionRecord
        |   |-- KinematicBubbleParcel
        |   `-- MPPICParcel
        `-- derived
            |-- basicKinematicBubbleCollidingParcel
            |-- basicKinematicBubbleMPPICParcel
            `-- basicKinematicBubbleParcel
```

Now, the `clouds` directory includes templates such as `CollidingCloud`, `KinematicBubbleCloud`, and `MPPICCloud`, along with base classes. Derived implementations, such as `basicKinematicBubbleCloud` and its Colliding and MPPIC variants, enhance flexibility for different use cases. Similarly, `parcels` directory mirrors this structure, featuring templates and derived classes like `basicKinematicBubbleParcel`, which accommodates kinematic bubble behaviors with additional collision and MPPIC functionality. Notably, in `make` directory, the linking between these new cloud

classes and Lagrangian library in OpenFOAM source directory should be established, as they rely on functionalities from the Lagrangian library, including submodels such as injection.

The next step is to incorporate the Rayleigh-Plesset class into these new `KinematicBubbleCloud/Parcel` classes. The author introduces new variables as shown in Listing 3.4 for the `KinematicParcel` class and modifies the constructors of `KinematicParcel` class to read these values from the `KinematicBubbleCloudProperties` located in the `constant` directory. Regarding the pressure of the carrier phase, the constructor of `KinematicBubbleCloud` class has been modified to accept pressure as input and pass it to `KinematicBubbleParcel` class. Therefore, `constant/kinematicBubbleCloudProperties` accommodates the additional dictionary `bubbleProperties`, as shown in Listing 3.5.

Listing 3.4: New inputs to kinematicBubbleParcel

```
//- ML: Declarition of Class RPEqu
#include "RPEqu.H"

//- ML: Adding Pressure [Pa] for Rayleigh Plesset Eqaution
scalar pc_;

//- ML: instantiation of class RPEqu
RPEqu RPEqu_;

//- ML: Growth rate of radius [m/s]
scalar R_dot_;

//- ML: Dictionary of cloud properties
dictionary cloudProperties_;

//- ML: Dictionary of bubble properties
dictionary bubbleProperties_;

//- ML: flag for activation of bubble dynamics
label bubble_activation_;

//- ML: surface tension for bubble dynamics
scalar bubbleSigma_;

//- ML: initial pressure for bubble dynamics
scalar p0_;

//- ML: initial radius for bubble dynamics
scalar R0_;

//- ML: vapour pressure for bubble dynamics
scalar pv_;

//- ML: time step for each loop of RP solver
scalar RPdT_;

//- ML: kappa isotropic value for bubble dynamics
scalar bubbleKappa_;

//- ML: Averaging distance for bubble dynamics
scalar averagingDistance_;

//- ML: ODE solver type for bubble dynamics
word ODESolverType_;
```

Listing 3.5: kinematicBubbleCloudProperties

```
bubbleProperties
{
    //- ML: flag for activation of bubble dynamics
    bubble_activation       true;
    //- ML: surface tension for bubble dynamics
    bubbleSigma             0.07;
    //- ML: initial pressure for bubble dynamics
    p0                      101325;
    //- ML: initial radius for bubble dynamics
    R0                      1e-06;
    //- ML: vapour pressure for bubble dynamics
    pv                      2300;
    //- ML: time step for each loop of RP solver-important
    RPdT                    5.0e-6;
    //- ML: kappa isotropic value for bubble dynamics
    bubbleKappa             1.4;
    //- ML: Averaging distance for bubble dynamics
    averagingDistance       5;
    //- ML: ODE solver type for bubble dynamics
    ODESolverType           RKF45;
}
```

Next step would be finding the correct location to add `bubbleDynamics.H` to `KinematicBubbleCloud` class. Therefore, understanding the main workflow in `KinematicBubbleCloud` class is crucial. The main member of function in `KinematicBubbleCloud` class is `evolve()` member function. When the `evolve()` function is called, it calls the `solve()` function in which a call to `evolveCloud()` is initiated. The `evolveCloud()` member function triggers a call to the `motion()` member function. This, in turn, calls the `move()` function of the `Cloud` class, which contains a loop. Inside this loop, there is a call to the `move()` member function of `KinematicBubbleParcel` class. In the latter `move()` function, the `calc()` function is called to determine the new velocity of the bubble, where the `calcVelocity` function is invoked, and then new source terms for the Eulerian field are calculated. Since the source terms need to be updated based on Eq. (3.11) and Eq. (3.13), which should be calculated after the bubble radius change from the Rayleigh-Plesset equation, the integration of `bubbleDynamics.H` is placed right before the calculation of the new source terms and inside the `calcVelocity` member function of `KinematicBubbleParcel` class.

The integration of `bubbleDynamics.H` and the modification of `dUTrans` based on Eq. (3.13) can be seen in Listing 3.6. The `calcVelocity` function is a member function of the `KinematicBubbleParcel` class, responsible for calculating the velocity of a bubble parcel based on various forces acting on it. It first extracts the forces from the surrounding using the `calcCoupled` and `calcNonCoupled` methods, which compute the coupled and non-coupled forces, respectively. The effective mass of the bubble is then calculated, accounting for potential added virtual mass. The function computes the velocity update by integrating the velocity components over the time step `dt`, and the new velocity is determined by combining these forces. As shown in Listing 3.6, if bubble activation is enabled, additional bubble dynamics are included through the `bubbleDynamics.H` file, allowing for modifications to the radius and mass. Consequently, the source term `dUTrans` is adjusted to reflect the changes in momentum transfer. Finally, the function returns the updated velocity to the `calc` member function of the `KinematicBubbleParcel` class, where other modifications to the source terms are handled.

Listing 3.6: A part of calcVelocity in KinematicBubbleParcel

```
    // Calculate the new velocity and the momentum transfer terms
    vector Unew = U_ + deltaU;

    //- ML: Include bubble dynamics
    if(this->bubble_activation())
    {
        #include "bubbleDynamics.H"
    }

```

```
    //dUTrans -= massEff*deltaUcp;
    vector Unew2 = U_ + deltaUcp;
    scalar mass2 = this->mass();
    const scalar massEff2 = forces.massEff(p, ttd, mass2);//-ML: In case virtual mass transfer is added
    dUTrans -= ((massEff2*Unew2)-(massEff*U_)); //-ML: Modify the source term to make it according to the equations
```

As shown in Listing 3.7, `calc` function in the `KinematicBubbleParcel` class is responsible for calculating the velocity and mass-related properties of a bubble parcel at each time step. It begins by defining local properties such as the initial number of particles, mass, and Reynolds number. The momentum source terms are initialized, and the particle velocity is computed using the `calcVelocity` function. After calculating the new velocity, as part of the modification based on the mass generation term or Eq. (3.11), the change in bubble mass is computed.

In Listing 3.7, the author defines an approach to distribute source terms among the cells the bubble is in. If the bubble size is smaller than the size of a single cell, the cell takes all mass and momentum source terms. Otherwise, the terms are distributed among the cells the bubble covers. The step-by-step explanation in the function involves accumulating source terms for the carrier phase, starting with the bubble's position and radius. It explores neighboring cells within the bubble's radius, accumulating information about each cell's distance from the bubble's center. The Gaussian weight is calculated for each cell based on its distance from the bubble, and the total weight within the radius is summed. The bubble's volume is distributed among neighboring cells based on the normalized Gaussian weight. Finally, the momentum transfer rate and the rate change of mass for each neighboring cell are updated, reflecting the bubble's interaction with the carrier phase. At the top level, to send the source terms to the Eulerian solver, new functions should be defined inside `KinematicBubbleCloud` class.

Listing 3.7: A part of calc in KinematicBubbleParcel

```
// Calculate new particle velocity
this->U_ =
    calcVelocity(cloud, td, dt, Re, td.muc(), mass0, Su, dUTrans, Spu);

this->U_ += this->UCorrect_;

//-ML: Mass generation in cell due to RP equation
scalar mDotB = (mass() - mass0);

// Accumulate carrier phase source terms
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
if (cloud.solution().coupled())
{

    //-ML: Find position and radius of bubble
    vector posB = this->position();
    scalar radB = this->d() / 2;
    label startCell = this->cell();

    //-ML: DynamicList to store cells and their distances
    DynamicList<std::pair<label, scalar>> cellsWithDistances;

    //-ML: Add the start cell to the list
    vector cellCenter = this->mesh().cellCentres()[startCell];
    cellsWithDistances.append(std::make_pair(startCell, mag(cellCenter - posB)));  //-ML: Add the first cell with distance

    //-ML: Create a set to avoid re-visiting cells
    labelHashSet visitedCells;
    visitedCells.insert(startCell);

    //-ML: Dynamic list to store all neighbors for further exploration
    DynamicList<label> cells(0);
    cells.append(this->mesh().cellCells()[startCell]);

```

```
35
36      //-ML: Start outer loop to explore neighbors
37      while (cells.size() > 0)
38      {
39          //-ML: New dynamic list to store newly found neighbors within the radius
40          DynamicList<label> newNeighbours;
41
42          //-ML: Loop over all current neighbors in the cells list
43          forAll(cells, i)
44          {
45              label cellI = cells[i];
46              if (!visitedCells.found(cellI))  //-ML: Check if cellI is already visited
47              {
48                  cellCenter = this->mesh().cellCentres()[cellI];
49
50                  //-ML: Check if this neighbor cell's center is within the radius
51                  scalar distance = mag(cellCenter - posB);  //-ML: Calculate distance to posB
52
53                  if (distance <= (radB))
54                  {
55                      cellsWithDistances.append(std::make_pair(cellI, distance));  //-ML: Store the cell
       and its distance
56                      newNeighbours.append(this->mesh().cellCells()[cellI]);  //-ML: Add neighbors to
       the list
57                  }
58
59                  visitedCells.insert(cellI);  //-ML: Mark cell as visited
60              }
61          }
62
63          //-ML: Sort the cells based on their distance to posB
64          std::sort(cellsWithDistances.begin(), cellsWithDistances.end(),
65                  [](const std::pair<label, scalar>& a, const std::pair<label, scalar>& b)
66                  {
67                      return a.second < b.second;  //-ML: Compare distances
68                  });
69
70          //-ML: Continue to the next iteration with newNeighbours
71          cells.clear();
72          cells.transfer(newNeighbours);
73      }
74
75      scalar sigma = radB / this->LE_deviation();  //-ML: Gaussian standard deviation
76      scalar totalWeightWithinRadius = 0.0;
77
78      //-ML: Calculate the Gaussian weight for each cell and accumulate total weight within radius
79      List<scalar> gaussianWeights(cellsWithDistances.size());
80
81      forAll(cellsWithDistances, j)
82      {
83
84          scalar gaussFactor;
85
86          if (cellsWithDistances.size() == 1) {
87              gaussFactor = 1.0;  //-ML: Set gaussFactor to 1 if there is only one cell
88          } else {
89              scalar distance = cellsWithDistances[j].second;  //-ML: Get distance from pair
90              //-ML: Calculate Gaussian weight for this cell
91              gaussFactor = exp(-0.5 * pow(distance / sigma, 2.0));
92          }
93          gaussianWeights[j] = gaussFactor;
94
95          //-ML: Accumulate total weight
96          totalWeightWithinRadius += gaussFactor;
97      }
98
99      //-ML: Distribute the bubble volume using normalized Gaussian weights
100     forAll(cellsWithDistances, j) {
```

```
101
102         label cellJ = cellsWithDistances[j].first;
103
104         //-ML: Normalize the Gaussian weight and compute `vi_j` for this cell
105         scalar normalizedWeight = gaussianWeights[j] / totalWeightWithinRadius;
106         //-ML: Update momentum transfer
107         cloud.UTrans()[cellJ] += normalizedWeight*np0*dUTrans;
108
109         //-ML: Update momentum transfer coefficient
110         cloud.UCoeff()[cellJ] += normalizedWeight*np0*Spu;
111
112         //-ML: Update Return rate of change of mass
113         cloud.mDotBubble()[cellJ] += normalizedWeight*np0*mDotB;
114
115     }
116 }
```

3.2.2 Source terms in KinematicBubbleCloud

The variables `mDotBubble_` and `Dbetal_`, according to Eq. (3.15), and their respective call functions are implemented inside the `KinematicBubbleCloud` class. The way these terms are defined and initialized in `KinematicBubbleCloud` class can be seen in the accompanying files, and readers are referred to these files. The use of the member function `mDotBubble()` can be seen in line 113 of Listing 3.7. The use of the member function `Dbetal()` can be seen in the `evolveCloud` member function of `KinematicBubbleCloud` class, as shown in Listing 3.8. In the latter, in line 31 of Listing 3.8, the calculation of `Dbetal()` according to Eq. (3.15) is presented. This line incorporates updates related to `Dbetal()` for the divergence of velocity equation, calculated as $(\beta_{\mathrm{l}} - \beta_{\mathrm{l,\,old}})/\Delta t$, where Δt is the simulation time step. It is noteworthy that this update occurs after calling `motion`, which eventually leads to the `move` function of `KinematicBubbleParcel` class.

Listing 3.8: A part of evolveCloud in KinematicBubbleCloud

```
1     if (solution_.transient())
2     {
3         label preInjectionSize = this->size();
4
5         this->surfaceFilm().inject(cloud);
6
7         // Update the cellOccupancy if the size of the cloud has changed
8         // during the injection.
9         if (preInjectionSize != this->size())
10        {
11            updateCellOccupancy();
12            preInjectionSize = this->size();
13        }
14
15        //-ML = Store betal before resetting it
16        betal_old() = betal();
17        //-ML = Reset the beta
18        betal() = 1.0;
19
20        injectors_.inject(cloud, td);
21
22        //-ML = Reset the beta; should be reset after injection
23        // since in injection, we have move funciton triggered
24        betal() = 1.0;
25
26        // Assume that motion will update the cellOccupancy as necessary
27        // before it is required.
28        cloud.motion(cloud, td);
29
30        //-ML = DBeta for non-free-divergence equations
31        Dbetal() = (betal() - betal_old()) / (this->db().time().deltaT());
32
```

```
33          stochasticCollision().update(td, solution_.trackTime());
34      }
```

3.2.3 Modifying setCellValues function in KinematicBubbleParcel

In the original `KinematicParcel` class in Lagrangian library, to move the bubble, forces must be calculated. To do so, pressure, velocity, density, and viscosity from the Eulerian cell are used. The `KinematicCloud` class employs the `trackingData` class and the `setCellValues` member function of `KinematicParcel` class to set these values. However, since the original `KinematicParcel` class was designed to track particles, these values were interpolated from the cell center to the particle's center. On the other hand, to track bubble dynamics and movement, an average of these properties over the bubble's surface is required.

As a result, a significant modification to the `setCellValues` function in the `KinematicBubbleParcel` class, shown in Listing 3.9, involves enhancing its approach to determining Eulerian parameters such as density, pressure, velocity, and viscosity from the carrier phase. This is achieved by calculating averaged physical properties over a spherical surface at a user-defined distance from the bubble's position. The updated implementation evaluates these properties across the entire bubble surface. By sampling values on the bubble's surface and averaging them, the method provides a more realistic and practical representation of the carrier phase's influence on the bubble. The `setCellValues` function calculates these averaged properties by defining the bubble's position and radius and, sampling points uniformly on the bubble's spherical surface using azimuthal (θ) and polar (ϕ) angles. At each sampled position, the function interpolates velocity, density, viscosity, and pressure from the mesh cells and accumulates valid data. The use of `setCellValues` can be seen in lines 2 and 49 of Listing 3.2.

Listing 3.9: setCellValues in KinematicBubbleParcel

```
1  template<class ParcelType>
2  template<class TrackCloudType>
3  void Foam::KinematicBubbleParcel<ParcelType>::setCellValues
4  (
5      TrackCloudType& cloud,
6      trackingData& td,
7      const scalar distance
8  )
9  {
10     //-ML: Initialise finding fields
11     vector posB= this->position();
12     scalar R_sphere = distance * (this->d()/2); // Define the radius of the sphere
13     //-ML: Number of divisions in theta (azimuthal angle) and phi (polar angle)
14     label numTheta = 10;
15     label numPhi = 10;
16     vector USum(0, 0, 0);
17     scalar rhoSum = 0.0;
18     scalar muSum = 0.0;
19     scalar pLSum = 0.0;
20     label validSamples = 0; //-ML: Counter for valid samples
21
22     //-ML: Loop over the spherical surface with uniform sampling
23     for (label i = 0; i < numTheta; i++)
24     {
25         scalar thetaBubble = (2 * constant::mathematical::pi * i) / numTheta; //-ML: Azimuthal angle
26
27         for (label j = 0; j < numPhi; j++)
28         {
29             scalar phiBubble = (constant::mathematical::pi * j) / (numPhi - 1); //-ML: Polar angle
30
31             //-ML: Compute the unit vector components explicitly
32             scalar x = sin(phiBubble) * cos(thetaBubble);
33             scalar y = sin(phiBubble) * sin(thetaBubble);
34             scalar z = cos(phiBubble);
35
```

```
            //-ML: Create the vector using the computed components
            vector unitVec(x, y, z);

            //-ML: Scale by the sphere radius and shift by the center
            vector samplePos = posB + R_sphere * unitVec;

            //-ML: Find the cell that contains this position
            label cellSample = cloud.mesh().findCell(samplePos);

            //-ML: Check if a valid cell was found
            if (cellSample != -1)
            {
                //-ML: Interpolate values at the sample position
                vector USample = td.UInterp().interpolate(samplePos, cellSample);
                scalar rhoSample = td.rhoInterp().interpolate(samplePos, cellSample);
                scalar muSample = td.muInterp().interpolate(samplePos, cellSample);
                scalar pLSample = td.pInterp().interpolate(samplePos, cellSample);

                //-ML: Accumulate the values
                USum += USample;
                rhoSum += rhoSample;
                muSum += muSample;
                pLSum += pLSample;

                //-ML: Increment valid sample counter
                validSamples++;
            }
        }
    }

    //-ML: Calculate the average values over the surface
    scalar avgRho = (validSamples > 0) ? (rhoSum / validSamples) : 0.0;
    vector avgU = (validSamples > 0) ? (USum / validSamples) : Foam::Vector<double>::zero;
    scalar avgMu = (validSamples > 0) ? (muSum / validSamples) : 0.0;
    scalar avgPL = (validSamples > 0) ? (pLSum / validSamples) : 0.0;

    td.rhoc() = avgRho;
    if (td.rhoc() < cloud.constProps().rhoMin())
    {
        if (debug)
        {
            WarningInFunction
                << "Limiting observed density in cell " << this->cell()
                << " to " << cloud.constProps().rhoMin() <<  nl << endl;
        }

        td.rhoc() = cloud.constProps().rhoMin();
    }
    td.Uc() = avgU;
    td.muc() = avgMu;
    td.pc() = avgPL;
}
```

3.3 Modification of interPhaseChangeBubbleFoam

After modifying `KinematicBubbleCloud` class and `KinematicBubbleParcel` class, the `interPhaseChangeFoam` solver also needs to be updated. The reader may copy and paste the `interPhaseChange` directory from the `multiphase` directories within the `solvers` directory in OpenFOAM. It is recommended to rename the file, directory, and content of `interPhaseChangeFoam` to `interPhaseChangeBubbleFoam`. Moreover, ensuring the correct compilation of the new solver before proceeding with further changes is advisable.

Within `interPhaseChangeBubbleFoam`, a directory named `bubbleDynamics` can be created to include the `KinematicBubbleCloud` class and to host the Rayleigh-Plesset equation class files, `RPEqu`.

In addition to these changes, the name and content of `phaseChangeTwoPhaseMixtures` are updated to `hybridPhaseChangeTwoPhaseMixtures`. Additionally, within the `bubbleDynamics` directory, a subdirectory named `transitionAlgorithm` is reserved for future use. An example of the file structure is shown in Listing 3.10.

Listing 3.10: interPhaseChangeFoam proposed directory

```
interPhaseChangeBubbleFoam
    |-- Make
    |-- bubbleDynamics
    |   |-- Make
    |   |-- RPEqu
    |   |-- clouds
    |   |   |-- Templates
    |   |   |   |-- CollidingCloud
    |   |   |   |-- KinematicBubbleCloud
    |   |   |   `-- MPPICCloud
    |   |   |-- baseClasses
    |   |   |   `-- kinematicBubbleCloud
    |   |   `-- derived
    |   |       |-- basicKinematicBubbleCloud
    |   |       |-- basicKinematicBubbleCollidingCloud
    |   |       `-- basicKinematicBubbleMPPICCloud
    |   |-- lnInclude
    |   |-- parcels
    |   |   |-- Templates
    |   |   |   |-- CollidingParcel
    |   |   |   |   `-- CollisionRecordList
    |   |   |   |       |-- PairCollisionRecord
    |   |   |   |       `-- WallCollisionRecord
    |   |   |   |-- KinematicBubbleParcel
    |   |   |   `-- MPPICParcel
    |   |   `-- derived
    |   |       |-- basicKinematicBubbleCollidingParcel
    |   |       |-- basicKinematicBubbleMPPICParcel
    |   |       `-- basicKinematicBubbleParcel
    |   `-- transitionAlgorithm
    `-- hybridPhaseChangeTwoPhaseMixtures
        |-- Kunz
        |-- Make
        |-- Merkle
        |-- SchnerrSauer
        |-- hybridPhaseChangeTwoPhaseMixture
        `-- lnInclude
```

3.3.1 Implementing Beta and Gamma

The first change to `interPhaseChangeBubbleFoam` is to introduce β_l and γ into the solver according to Eq. (3.1). It is more useful to introduce these two variables in the same place where α_l is defined. In the `createField.H` file, we can see that α_l is defined as a member function of the `phaseChangeTwoPhaseMixture` class. As we delve deeper into this class, we observe that it inherits from `incompressibleTwoPhaseMixture` class, which, in turn, inherits from the `twoPhaseMixture` class. Inside the `twoPhaseMixture` class, we find that α_l is being read from the dictionary located in the time directories.

As a result, we copied and pasted the `incompressibleTwoPhaseMixture`, `twoPhaseMixture`, and `phaseChangeTwoPhaseMixture` classes into the `interPhaseChangeBubbleFoam` directory and renamed them to `HybridIncompressibleTwoPhaseMixture`, `HybridTwoPhaseMixture`, and `HybridPhaseChangeTwoPhaseMixture`, respectively. In Listing 3.11, two new variables and their functions are defined and declared, and one function is defined to update the value of γ.

Listing 3.11: Variables are added in HybridTwoPhaseMixture class

```
//-ML: The Lagrangian phase-fraction
        volScalarField beta1_;
        volScalarField beta2_;
//-ML: The Hybrid phase-fraction
        volScalarField gamma1_;
        volScalarField gamma2_;


//-ML: Return Hybrid phase-fraction of phase 2
        void update_gamma()
        {
            gamma1_ = alpha1_ * beta1_;
            gamma2_ = 1.0 - gamma1_;
        }
```

Another modification is in the `HybridIncompressibleTwoPhaseMixture` class, as it is where the laminar mixture viscosity, μ_{laminar} in Eq. (2.3), is defined and returned to the momentum equation in `UEqn.H`. In this class, all instances where α_1 is used are replaced with γ to align it with Eq. (3.18). Readers may refer to the accompanying files to see these changes.

Now, the changes in the `interPhaseChangeBubbleFoam` solver should be made. Inside the `createField.H` file, the definition of `rho` is updated to use `gamma` (γ) instead of `alphal` (α_1). The changes use variables, `beta` and `gamma`, to handle phase fractions and mixing properties. Specifically, `beta1`, `beta2`, `gamma1`, and `gamma2` are initialized as references to fields in the `mixture` object. The code ensures that the phase fractions are consistent by enforcing relationships such as $\alpha_2 = 1 - \alpha_1$ and $\beta_2 = 1 - \beta_1$. The mixture density `rho` is redefined using `gamma` values as $\rho = \gamma_1\rho_1 + \gamma_2\rho_2$, where `rho1` and `rho2` represent the densities of the two phases. Additionally, whenever there is a calculation affecting density, the `updateGamma()` function should be called to ensure consistency in viscosity and density calculations.

Listing 3.12: Modifications in createField.H

```
//- ML: Adding the beta and gamma
volScalarField& beta1(mixture->beta1());
volScalarField& beta2(mixture->beta2());
volScalarField& gamma1(mixture->gamma1());
volScalarField& gamma2(mixture->gamma2());

//- ML: Double check values
alpha2 = 1.0 - alpha1;
beta2 = 1.0 - beta1;

const dimensionedScalar& rho1 = mixture->rho1();
const dimensionedScalar& rho2 = mixture->rho2();

// Need to store rho for ddt(rho, U)
volScalarField rho
(
    IOobject
    (
        "rho",
        runTime.timeName(),
        mesh,
        IOobject::READ_IF_PRESENT
    ),
    gamma1*rho1 + gamma2*rho2
);
rho.oldTime();
```

3.4 Coupling of interPhaseChangeBubbleFoam with KinematicBubbleCloud

Prior to compiling and using the code, the `KinematicBubbleCloud` class and `interPhaseChangeBubbleFoam` should be coupled. There are works such as by Vallier [19, 20] that demonstrated how to couple `interFoam` with the `solidParticle` class, and later, a more improved coupling with the `solidParticle` class was also presented by Ghahramani [21]. However, Ghahramani [5], in their Ph.D. thesis, demonstrated the coupling of `KinematicCloud` class with `interPhaseChangeFoam`. In this study, the author will show how to link these two classes, `KinematicBubbleCloud` and `interPhaseChangeBubbleFoam`. In order to couple them, the first step would be including the declaration of the class `basicKinematicBubbleCloud.H` in `interPhaseChangeBubbleFoam`, as shown in Listing 3.13.

Listing 3.13: Modifications in interPhaseChangeBubbleFoam

```
#include "fvOptions.H"

#include "basicKinematicBubbleCloud.H" //-ML : declarition of basicKinematicBubbleCloud

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

Another modification is in `createField.H`, as shown in Listing 3.14. The code begins with defining a scalar field, `muc`, representing the dynamic viscosity of the fluid, which is calculated as the product of the kinematic viscosity (`turbulence->nu()`) and mixture density (`rho`). Then, it creates a cloud of bubbles, with the name specified either through an argument or defaulting to `KinematicBubbleCloud`. Finally, the `basicKinematicBubbleCloud` object, namely `parcels`, is constructed by supplying parameters such as density (`rho`), velocity (`U`), dynamic viscosity (`muc`), pressure (`p`), and gravitational acceleration (`g`) to the constructor of the `KinematicBubbleCloud` class.

Listing 3.14: Modifications in createField.H

```
//-ML: Creating elements important for lagrangian bubbles
volScalarField muc
(
    IOobject
    (
        "muc",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::AUTO_WRITE
    ),
    turbulence->nu()*rho
);

//- ML: Creating clouds
const word kinematicBubbleCloudName
(
    args.getOrDefault<word>("cloud", "kinematicBubbleCloud")
);

Info<< "Constructing Cloud: " << kinematicBubbleCloudName << endl;

basicKinematicBubbleCloud parcels
(
    kinematicBubbleCloudName,
    rho,
    U,
    muc,
    p,//-ML:  It is added for bubble Dynamics-Rayleigh Plesset Equation
    g
);
```

Another major change is including the source terms from the Lagrangian framework into the Eulerian framework. According to Eq. (3.20), the inclusion of `parcels.SU(U)` should be added to the momentum equation in `UEqn.H`, as shown in Listing 3.15. In addition, source terms `parcels.SU_mDotBubble()` and `parcels.Dbetal()` according to Eq. (3.15) are included in `pEqn.H`, as shown in Listing 3.16.

Listing 3.15: Modifications in UEqn.H

```
1     fvVectorMatrix UEqn
2     (
3         fvm::ddt(rho, U)
4       + fvm::div(rhoPhi, U)
5       - fvm::Sp(fvc::ddt(rho) + fvc::div(rhoPhi), U)
6       - fvm::ddt(rho2Beta2, U) //-ML: subtract effects of 1-beta1
7       - fvm::div(rho2PhiBeta2, U) //-ML: subtract effects of 1-beta1
8       + turbulence->divDevRhoReff(rho, U)
9       + parcels.SU(U) //-ML: Return the momentum source from the lagrangian framework
10    );
```

Listing 3.16: Modifications in pEqn.H

```
1         fvScalarMatrix p_rghEqn
2         (
3             fvc::div(phiHbyA) - fvm::laplacian(rAUf, p_rgh)
4           - (EuvDotcvP)*(mixture->pSat() - rho*gh)  //-ML: p_rgh + rho*gh
5           + fvm::Sp(EuvDotcvP, p_rgh)
6           + invRho1Beta*parcels.SU_mDotBubble() //-ML: Rate of change of bubble mass
7           + invBeta*parcels.Dbetal() //-ML: Total derivative of betal
8         );
```

Finally, the last step involves adding the `evolve` member function of `basicKinematicBubbleCloud` in `interPhaseChangeBubbleFoam` to manipulate the parcels, such as moving, injecting, and updating source terms, as shown in line 4 of Listing 3.17. There are other minor modifications in `pEqn.H`, `UEqn.H`, and `alphaEqn.H`, which readers can find by referring to the accompanying files.

Listing 3.17: interPhaseChangeBubbleFoam.H

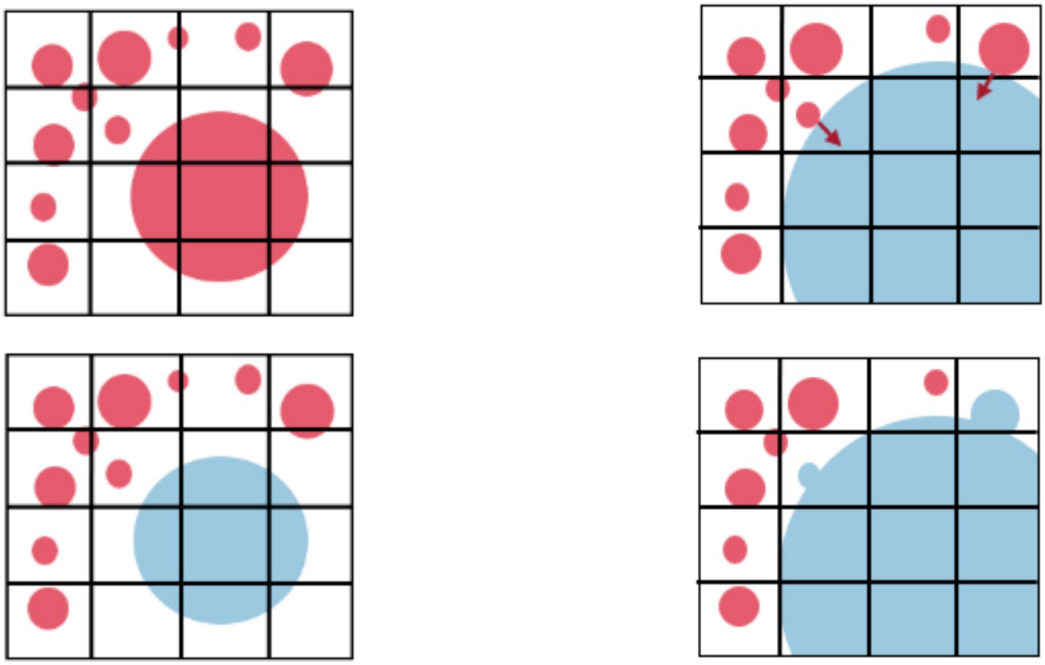
```
1         //-ML: Adding evolution of cloud
2         Info<< "Evolution of Cloud: "<< parcels.name() << endl;
3
4         parcels.evolve();
5         //-ML: update the gamma value from multiplication of alpha1_*beta1_
6         mixture->update_gamma();
7
8         runTime.write();
```

3.5 Lagrangian to Eulerian framework transition

The main aim of the project is to inject nuclei as unresolved vapor and then track their size and movement. However, the nuclei's size may reach a point where it is less computationally expensive to track them within the Eulerian framework than to keep them in the Lagrangian framework, moreover, unresolved vapors may get close to the interface of resolved vapor. In this regard, the author implemented an algorithm to transfer Lagrangian bubbles when they meet certain criteria, as can be seen in Lavari *et al.* [10]. The algorithm includes:

1. Transition from the Lagrangian to Eulerian framework due to the expansion of the vapor structure beyond a size threshold, as depicted in Figure 3.1a.

2. Coalescence of microscale vapor structures with macroscale structures, as depicted in Figure 3.1b.



(a) Transition from Lagrangian to Eulerian due to expansion [10]

(b) Transition from Lagrangian to Eulerian due to Coalescence [10]

Figure 3.1: Lagrangian to Eulerian framework transition

In `interPhaseChangeBubbleFoam/bubbleDynamics` directory, the author created a directory dedicated to transition algorithms with the same name as the directory. Inside this directory, there are two files. One of them is `LagrangianToEulerian.H`, and its content is shown in Listing A.1. This code traces the Lagrangian bubbles within a computational mesh, identifies the bubble's location and its neighboring cells, redistributes bubble volume using Gaussian weights, and deactivates bubble tracing under above-mentioned criteria.

The code begins by determining the bubble's position (`posB`) and radius (`radB`). The initial computational cell containing the bubble is identified, and its distance from the bubble center is computed. A `DynamicList` is used to store pairs of cell labels and their distances that the bubble covers. The algorithm loops over all cells while ensuring no cells are revisited by tracking visited cells. Neighboring cells are explored iteratively, adding cells within $2 \times$ `radB` to the list, sorted by distance. This factor of 2 is chosen to ensure that 100 percent of the vapor is distributed.

The volume distribution uses Gaussian weights based on the bubble's center and the distances of neighboring cells. For a given cell j, the Gaussian weight is computed as

$$w_j = \exp\left(-\frac{1}{2}\left(\frac{\mathrm{d}_j}{\sigma}\right)^2\right). \tag{3.21}$$

Here, d_j is the distance between the center of cell j and the bubble's position, and $\sigma = \frac{r_B}{\text{deviation}}$ is the Gaussian standard deviation, with r_B as the bubble radius. To ensure volume conservation and smooth distribution, which are critical for accurately modeling physical interactions, the normalized weight is used to compute the fraction of the bubble volume assigned to each cell. This is given by

$$\tilde{w}_j = \frac{w_j}{W}. \tag{3.22}$$

Here, W is the total weight within the radius, defined as

$$W = \sum_j w_j. \tag{3.23}$$

Finally, the normalized weight is used to compute the fraction of the bubble volume assigned to each cell, defined as

$$v_{i,j} = \tilde{w}_j \cdot \frac{V_B}{V_j}. \tag{3.24}$$

Here, $v_{i,j}$ is the volume fraction of the bubble assigned to cell j. V_B is the total volume of the bubble, and V_j is the volume of cell j.

In addition, the code checks proximity to the interface by evaluating α_l in nearby cells. If even one cell within the bubble's radius contains α_l below a set threshold, the bubble is considered to be close to Eulerian structures, potentially triggering deactivation. Another feature that has been added is the ability to check the Lagrangian bubbles in a certain area of interest, the limits of which are defined by a box. When the user enables the box limits, tracking of Lagrangian bubbles would be restricted out of the box boundaries, reducing the computational cost of the simulation.

In order to couple `LagrangianToEulerian.H` with `kinematicBubbleParcel` class, the author defined some controllers/variables and their calling functions, as well as initialized them in the constructor of `kinematicBubbleParcel` class. All of this happens inside the `kinematicBubbleParcel` class, as shown in Listing 3.18.

Listing 3.18: `New inputs in kineamticBubbleParcel.H`

```
            //- ML: Dictionary of Transition_Algorithm
            dictionary Transition_Algorithm_;

            //- ML: flag for activation of EulerianToLagrangian
            label EulerianToLagrangian_activation_;

            //- ML: flag for activation of LagrangianToEulerian
            label LagrangianToEulerian_activation_;

            //- ML:sigma = bubbleRadius / deviation
            scalar LE_deviation_;

            //- ML:Minimum liquid volume fraction value that lagrangian cell can occupy
            scalar LE_minCellOccupancy_;

            //- ML:Number of cells that trigger the transtion from Lagrangian to Eulerian
            scalar LE_cellThreshold_;

            //- ML: Set threshold for interface proximity that trigger the transtion from Lagrangian
    to Eulerian
            scalar LE_alphaThreshold_;

            //- ML: Minimum bubble radius threshold that will be tracked
            scalar LE_bubbleSizeThreshold_;

            //- ML: Define the box boundaries for tracking- Only lagrangian inside this box will be
    tracked
            label LE_boxCheckEnabled_;

            //- ML: the top-left corner coordinates
            vector LE_boxTopLeftCorner_;

            //- ML: the bottom-right corner coordinates
            vector LE_boxBottomRightCorner_;
```

Prior to including `LagrangianToEulerian.H`, it is important to note that β_l should be filled and updated with the Lagrangian vapor volume fraction over cells that unresolved vapor covers. Moreover, β_l should be transferred to the Eulerian framework by correcting α_l in case the Lagrangian bubble is deactivated. Thus, α_l and β_l should exchange values inside the `KinematicBubbleParcel`

class, meaning that the constructor of `KinematicBubbleCloud` class should be modified to accept the α_l and β_l references from `interPhaseChangeBubbleFoam`. Listing 3.19 shows how to use the new constructor of `basicKinematicBubbleCloud` class in the `createField.H`. Readers are referred to the accompanying files to see all details of the changes in the constructor of `KinematicBubbleCloud` class.

Listing 3.19: Modifications in createField.H

```
basicKinematicBubbleCloud parcels
(
    kinematicBubbleCloudName,
    rho,
    U,
    muc,
    p,//-ML:  It is added for bubble Dynamics-Rayleigh Plesset Equation
    alpha1,//-ML: It is added to modify liquid volume fraction
    beta1,//-ML: It is added to modify liquid volume fraction
    g
);
```

3.5.1 Coupling LagrangianToEulerian.H with KinematicBubbleParcel

As emphasized earlier, the main member function inside the `KinematicBubbleParcel` is the `move()` member function, thus `LagrangianToEulerian.H` is added at the end of the `move()` function. The final lines of the `move()` function is changed with the addition of `LagrangianToEulerian.H`, as shown in line 19 of Listing 3.20.

Listing 3.20: `Modifications in KineamticBubbleParcel.H`

```
        p.age() += dt;

        if (p.active() && p.onFace())
        {
            ttd.keepParticle = cloud.functions().postFace(p, ttd);
        }

        ttd.keepParticle = cloud.functions().postMove(p, dt, start, ttd);

        if (p.active() && p.onFace() && ttd.keepParticle)
        {
            p.hitFace(s, cloud, ttd);
        }
    }

    //- ML: add transition from lagrangian to eulerian
    if(this->LagrangianToEulerian_activation())
    {
        #include "LagrangianToEulerian.H"
    }

    return ttd.keepParticle;
}
```

The final change involves defining a dictionary named `Transition_Algorithm` inside the `constant-/kinematicBubbleCloudProperties`. This dictionary is populated as shown in Listing 3.21.

Listing 3.21: `Modifications in kinematicBubbleCloudProperties.H`

```
Transition_Algorithm
{
    //- ML: flag for activation of EulerianToLagrangian
    EulerianToLagrangian_activation         true;

    //- ML: flag for activation of EulerianToLagrangian
    LagrangianToEulerian_activation         true;
```

```
8
9       //- ML:sigma = bubbleRadius / deviation-> Gaussian standard deviation (for 99% volume within
        radius)
10      LE_deviation                            3.0;
11
12      //- ML:Minimum liquid volume fraction value that lagrangian cell can occupy
13      LE_minCellOccupancy                     0.3;
14
15      //- ML:Number of cells that trigger the transtion from Lagrangian to Eulerian
16      LE_cellThreshold                        20000;
17
18      //- ML: Set threshold for interface proximity that trigger the transtion from Lagrangian to
        Eulerian
19      LE_alphaThreshold                       0.5;
20
21      //- ML: Minimum bubble radius threshold that will be tracked
22      LE_bubbleSizeThreshold                  5e-07;
23
24      //- ML: Define the box boundaries for tracking- Only lagrangian inside this box will be tracked
25      LE_boxCheckEnabled                      false;
26
27      LE_boxTopLeftCorner                (0.0 0.0 0.0);  // the top-left corner coordinates at the back
          face
28
29      LE_boxBottomRightCorner            (1.0 1.0 1.0);  // the bottom-right corner coordinates at the
        front face
30  }
```

3.6 Eulerain to Lagrangian framework transition

The Eulerian-resolved vapors may shrink to small sizes that cannot be tracked with the cell size, as shown in Figure 3.2. One good practice is to transfer them into the Lagrangian framework instead of ignoring them, as they may collapse and generate re-entering jets, ultimately influencing the pressure and velocity fields. Vallier [3] introduced an algorithm to detect Eulerian structures, check their size, and, if needed, transfer them to the Lagrangian framework. However, the author found this algorithm challenging to parallelize. In contrast, for the type of problems under consideration, the domain could be divided between different processors, making parallelization necessary.

Heinrich *et al.* [22] used Connected Component Labeling(CCL) for nozzle atomization, where small Eulerian elements are converted to Lagrangian parcels to reduce computational cost, and the code is fully parallelized. This conversion from the Eulerian framework to the Lagrangian framework resides within the `phaseCoupling` class inside the `/src/libAtomization` directory[1]. Since the original `phaseCoupling` class is designed for finding liquid structures, and in this project, vapor structures are the focus of interest, some modifications were made. Specifically, only the `update()` member function of the `phaseCoupling` class was extracted and modified to accept Eulerian vapor instead of Eulerian liquid. Additionally, the injection part was completely redesigned to follow the criteria shown in Figure 3.2 for injecting Lagrangian vapor into the Lagrangian cloud. The modified code is written in `EulerianToLagrangian.H`, which resides in the `bubbleDynamics` directory.

By examining `EulerianToLagrangian.H`, as shown in Listing A.2, the process begins with the definition of two thresholds. The first is `alpha1Lim`, which acts as a cutoff for α_l to distinguish between vapor and liquid. The second is `minCells`, which specifies the minimum number of cells required to consider a structure as significant. Additionally, a scalar field `vofID_` is initialized to store the volume ID of each computational cell. This field is set to zero values initially and employs corrected boundary conditions.

A `globalIndex` object is utilized to map local cell indices to global indices, enabling the identification of cells across processors in parallel simulations. The algorithm iterates through all cells, assigning a unique volume ID to connected regions that satisfy the phase fraction threshold. Starting from an unmarked cell with α_l below the threshold, it identifies neighboring cells recursively using

[1]The link to the original code is https://github.com/ElsevierSoftwareX/SOFTX_2020_30.

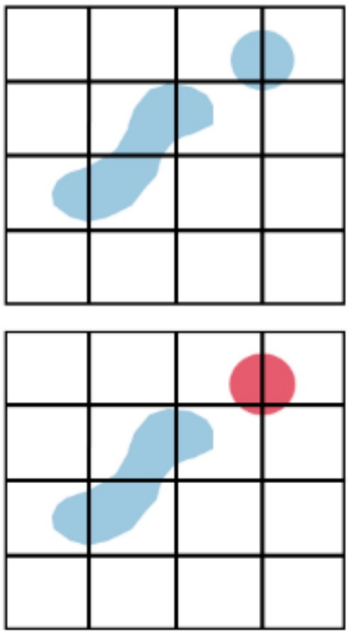


Figure 3.2: Transition from the Eulerian to Lagrangian framework due to an insufficient number of grids to resolve the vapor structure [10]

dynamic lists to explore and mark all connected cells meeting the same condition. These identified cells are assigned the same volume ID, and the process continues until all relevant cells are processed.

To support the parallel processing, the code ensures consistency of the `vofID_` field across coupled patches such as processor boundaries. This involves verifying the volume IDs on both sides of a patch and merging differing IDs when necessary. The merging process is repeated a number of times proportional to the logarithm of the number of processors to achieve complete consistency.

Once the volume IDs are finalized, the code renumbers them into a compact sequential order. It calculates various properties for each identified structure, including the number of cells, total volume, centroid position, and average velocity. These properties are determined by iterating over all cells and aggregating the relevant data based on the volume ID of each cell. The results are then `reduced` across all processors to account for parallel computation.

The code outputs details about each identified structure, such as the number of cells, volume, position, and velocity. For small structures below the `minCells` threshold, the code adjusts α_l and computes additional properties like the center position and average velocity. These properties are subsequently injected into a Lagrangian cloud to represent the bubbles in a discrete Lagrangian framework. Finally, corrections are applied to ensure the proper placement of the injected particles within the computational mesh.

3.6.1 Coupling EulerianToLagrangian.H with interPhaseChangeBubble

The `EulerianToLagrangian.H` file should be coupled within `interPhaseChangeBubble` class, as it is responsible for updating resolved vapors. Since `EulerianToLagrangian.H` injects bubbles into the Lagrangian framework, it needs to be added prior to the `evolve` function in `interPhaseChange-Bubble`. Listing 3.22 illustrates the location of this implementation in `interPhaseChangeBubble`.

Listing 3.22: `Modifications in interPhaseChangeBubbleFoam.H`

```
1        //-ML: Transition from Eulerian to Lagrangian
2        if(EulerToLagrang_activation)
3        {
4            #include "EulerianToLagrangian.H"
5        }
```

```
        mixture->update_gamma();

        //-ML: Adding evolution of cloud
        Info<< "Evolution of Cloud: "<< parcels.name() << endl;

        parcels.evolve();
        //-ML: update the gamma value from multiplication of alpha1_*beta1_
        mixture->update_gamma();

        runTime.write();
```

To enable control over the Eulerian-to-Lagrangian transition, additional controllers and variables are introduced within `Transition_Algorithm` in the `constant/kinematicBubbleCloudProperties` file. These modifications are presented in Listing 3.23. The initialization of these variables is included in the `createFields.H` file, as shown in Listing 3.24.

Listing 3.23: `Modifications in constant/kinematicBubbleCloudProperties`

```
Transition_Algorithm
{
    //- ML: flag for activation of EulerianToLagrangian
    EulerianToLagrangian_activation         true;

    //- ML:Number of cells that trigger the transtion from Eulerian to Lagrangian
    EL_cellThreshold                        4;

    //- ML: Set threshold for tracking structures with alpha below this Threshold
    EL_alphaThreshold                       0.999;

    //- ML: flag for activation of EulerianToLagrangian
    LagrangianToEulerian_activation         false;
...
```

Listing 3.24: `Modifications in createFields.H`

```
//-ML: Set the Euler to Lagrang controllers
Info<< "Set Eulerian-to-Lagrangian Controllers " <<  endl;
IOdictionary EulerToLagrang
(
    IOobject
    (
        "kinematicBubbleCloudProperties",
        mesh.time().constant(),
        mesh,
        IOobject::MUST_READ,
        IOobject::NO_WRITE
    )
);

dictionary readEulerToLagrang(EulerToLagrang.subDict("Transition_Algorithm"));

//- ML: Flag for activation of EulerianToLagrangian
label EulerToLagrang_activation(readEulerToLagrang.lookupOrDefault<bool>("
    EulerianToLagrangian_activation", 0));
Info << "Set the EulerToLagrang_activation: "  <<  EulerToLagrang_activation  << endl;

//- ML: Number of cells that trigger the transtion from Eulerian to Lagrangian
scalar EL_cellThreshold(readEulerToLagrang.lookupOrDefault<scalar>("EL_cellThreshold", 0));
Info << "Set the EL_cellThreshold: "  <<  EL_cellThreshold  << endl;

//- ML: Set threshold for tracking structures with alpha below this Threshold
scalar EL_alphaThreshold(readEulerToLagrang.lookupOrDefault<scalar>("EL_alphaThreshold", 0));
Info << "Set the EL_alphaThreshold: "  <<  EL_alphaThreshold  << endl;
```

Chapter 4

Test cases and results

In order to evaluate the different features of the new solver `interPhaseChangeBubbleFoam`, four test cases have been designed. Careful consideration is made to design the cases in such a way that they can be run in a short time, making it impractical to use in a real engineering application. However, readers can modify these cases to make the tutorials suitable for their problem of interest. Four different abilities tested in present cases are as follows:

1. Case 1 verifies the interface tracking between unresolved and resolved vapors and merges the unresolved with the resolved vapors if the interface is tracked, as shown in Figure 3.1b.
2. Case 2 transfers unresolved vapor to resolved vapor when its size goes beyond a user-defined threshold as depicted in Figure 3.1a.
3. Case 3 transfers resolved vapor to unresolved vapor when its size goes below a user-defined threshold as depicted in Figure 3.2.
4. Case 4 checks the deletion of Lagrangian bubbles out of the region of interest, or user-defined box.

Readers can use the accompanying files to run all the cases. The step-by-step process is to first source OpenFOAM-v2406 and then run the `Allrun` script to compile the solver and execute all the cases. In each case study directory, there is a ParaView state file that can be used to import the pipelines used to visualize the results in this work.

4.1 Case study 1: checkingInterFace

Case 1 consists of a 3D box in which the bottom half is filled with water and the upper half with air. The size of the box is 0.1 m in the x-direction, 0.2 m in the y-direction, and 0.1 m in the z-direction, as shown in Figure 4.1. The number of cells in the x-direction is 50, in the y-direction is 100 and in the z-direction is 50. The cells have a uniform length of 0.002 m everywhere, resulting in a total number of 250,000 cells. Boundary conditions are defined as `leftWall`, `rightWall`, `lowerWall`, `backWall`, and `frontWall`, with an atmosphere patch at the top of the box.

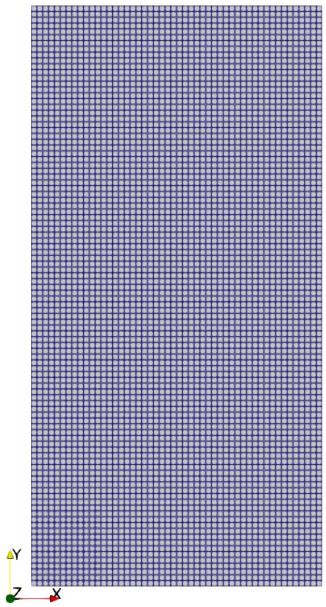


Figure 4.1: z-plane at z= 0.05 of the mesh

The `setFields` utility is used to fill the lower half of the box with water as shown in Figure 4.2. The simulation was run for 0.1 s with a time step of 0.001 s and a write interval of 0.005 s. At time 0 s, a bubble is injected using the `manualInjection` model at position the (0.05, 0.085, 0.05), which then rises due to the buoyancy force.

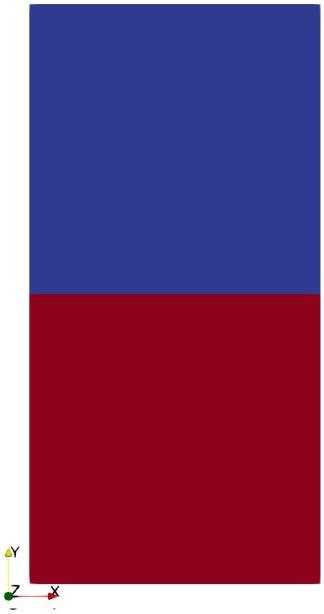


Figure 4.2: z-plane at z= 0.05 of domain; red represents water and blue represents air

The configuration for `bubbleProperties` and `Transition_Algorithm` dictionaries can be seen in Listing 4.1. The flag for activation of bubble dynamics was disabled, so the Rayleigh-Plesset

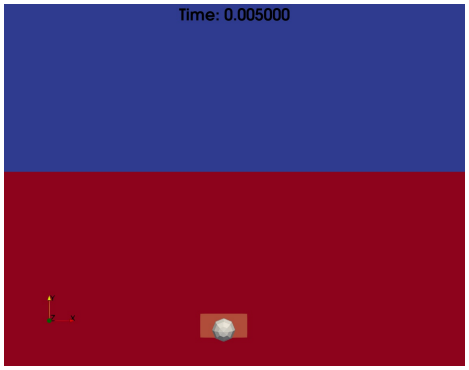
equation did not perform, and the change in bubble diameter was deactivated. All files related to this case can be found in the accompanying files.

Listing 4.1: Configuration of `bubbleProperties` and `Transition_Algorithm` in `kinematicBubbleCloudProperties`

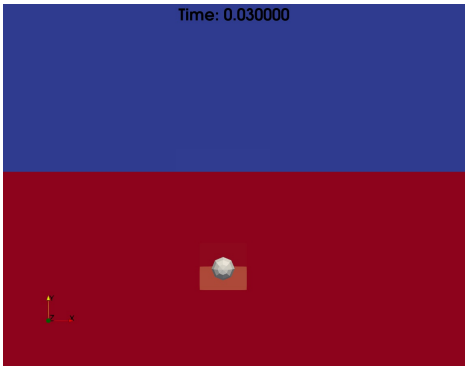
```
bubbleProperties
{
    //- ML: flag for activation of bubble dynamics
    bubble_activation       false;
    //- ML: surface tension for bubble dynamics
    bubbleSigma             0.07;
    //- ML: initial pressure for bubble dynamics
    p0                      101325;
    //- ML: initial radius for bubble dynamics
    R0                      1e-06;
    //- ML: vapour pressure for bubble dynamics
    pv                      2300;
    //- ML: time step for each loop of RP solver-important
    RPdT                    5.0e-6; //1.0e-7 ;
    //- ML: kappa isotropic value for bubble dynamics
    bubbleKappa             1.4;
    //- ML: Averaging distance for bubble dynamics
    averagingDistance       2;//5;
    //- ML: ODE solver type for bubble dynamics
    ODESolverType           RKF45;
}

Transition_Algorithm
{
    //- ML: flag for activation of EulerianToLagrangian
    EulerianToLagrangian_activation         true;

    //- ML:Number of cells that trigger the transtion from Eulerian to Lagrangian
    EL_cellThreshold                        0;

    //- ML: Set threshold for tracking structures with alpha below this Threshold
    EL_alphaThreshold                       0.9;

    //- ML: flag for activation of LagrangianToEulerian
    LagrangianToEulerian_activation         true;

    //- ML:sigma = bubbleRadius / deviation-> Gaussian standard deviation (for 99% volume within radius)
    LE_deviation                            3.0;

    //- ML:Minimum liquid volume fraction value that lagrangian cell can occupy
    LE_minCellOccupancy                     0.3;

    //- ML:Number of cells that trigger the transtion from Lagrangian to Eulerian
    LE_cellThreshold                        20000;

    //- ML: Set threshold for interface proximity that trigger the transtion from Lagrangian to Eulerian
    LE_alphaThreshold                       0.5;

    //- ML: Minimum bubble radius threshold that will be tracked
    LE_bubbleSizeThreshold                  5e-07;

    //- ML: Define the box boundaries for tracking- Only lagrangian inside this box will be tracked
    LE_boxCheckEnabled                      false;

    LE_boxTopLeftCorner                 (0.0 0.0 0.0);  // the top-left corner coordinates at the back face

    LE_boxBottomRightCorner             (1.0 1.0 1.0);  // the bottom-right corner coordinates at the front face
}
```

4.1.1 Results

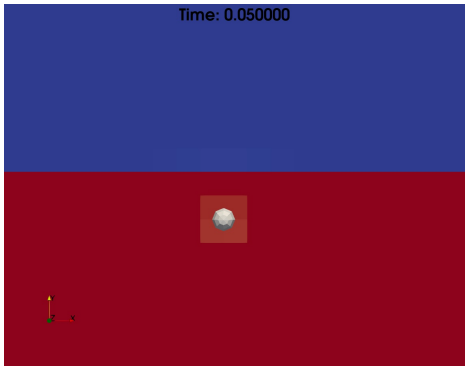
As can be seen in Figure 4.3, the bubble rises and at time 0.065 s, where it touches the interface of air and water, it is removed from the Lagrangian framework, and its effects remain as the Eulerian volume fraction, α_1.



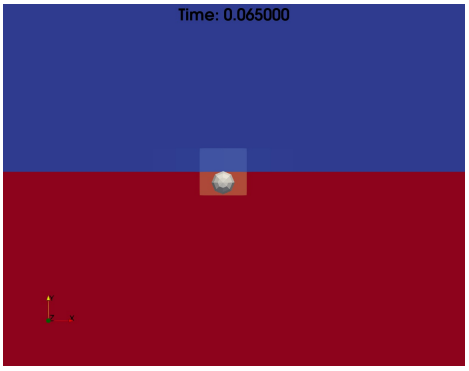
(a) Time 0.005 s



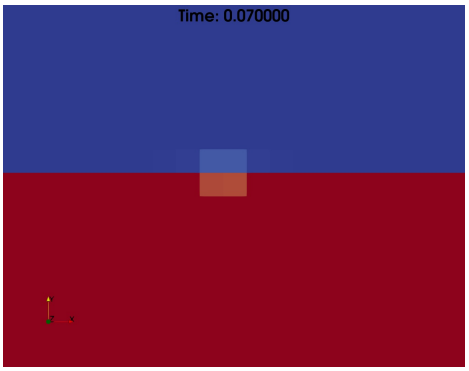
(b) Time 0.03 s



(c) Time 0.05 s



(d) Time 0.065 s



(e) Time 0.07 s

Figure 4.3: z-planes at z= 0.05 of the mesh at different time steps; the figures show the volume fraction contours with the spherical Lagrangian bubble

4.2 Case study 2: checkingLagrangianCellThreshold

In case study 2, one nucleus is injected inside a box. Since the pressure inside the box is less than the vapor pressure (2300 Pa), as the Lagrangian bubble moves, its radius increases. A threshold value of 44 is set, above which this unresolved vapor transfers to the Eulerian framework. In this regard, this case is the main case study in this project as it covers the primary goal of the project. This case consists of a 3D box with dimensions of 0.3 m in the x-direction, 0.1 m in the y-direction, and 0.1 m in the z-direction, as shown in Figure 4.4. The number of cells in the x-direction is 60, in the y-direction is 20, and in the z-direction is 20. The cells have a uniform length of 0.005 m everywhere, resulting in a total of 24,000 cells. Boundary conditions are defined as `inlet` at the left face of the domain, `outlet` at the right face, `lowerWall`, `backWall`, `frontWall`, and `topWall` at the top of the box.

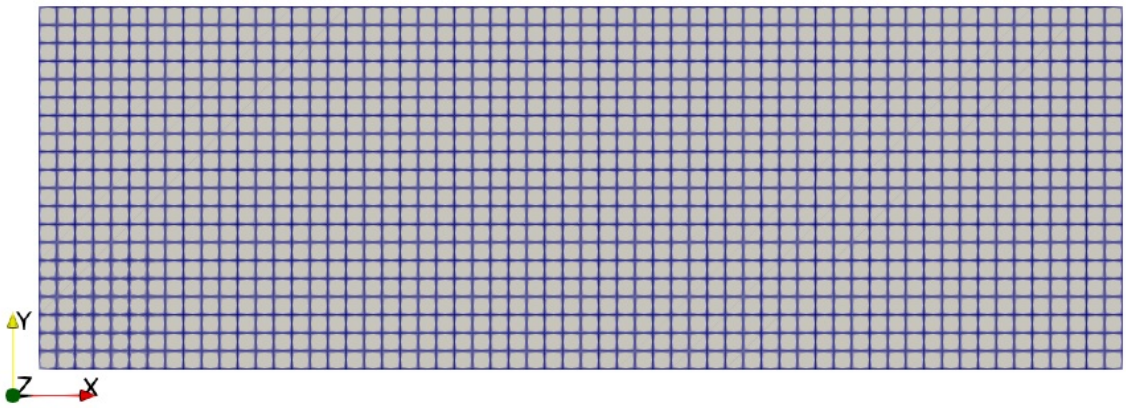


Figure 4.4: z-plane of the mesh

A pressure gradient is set in the domain from 2000 Pa at the `inlet` to 1000 Pa at the `outlet`, as shown in Figure 4.5. The simulation was run for 0.25 s with a time step of 0.0005 s and a write interval of 0.001 s. At time 0.02 s, a bubble was injected using `manualInjection` model at the position (0.0 0.05 0.05) with an initial diameter and velocity of 0.001 m and 2 m/s, receptively. All forces on the nucleus are disabled, thus it moves with a prescribed motion.

Figure 4.5: z-plane of domain; pressure distribution inside the domain

The configuration for `bubbleProperties` and `Transition_Algorithm` dictionaries can be seen in Listing 4.2. Bubble dynamics is activated in this case, along with the transition from Lagrangian to Eulerian with an `LE_cellThreshold` value of 44. All files related to this case can be found in the accompanying files.

Listing 4.2: Configuration of `bubbleProperties` and `Transition_Algorithm` in `kinematicBubbleCloudProperties`

```
bubbleProperties
{
    //- ML: flag for activation of bubble dynamics
    bubble_activation       true;
    //- ML: surface tension for bubble dynamics
    bubbleSigma             0.07;
    //- ML: initial pressure for bubble dynamics
    p0                      101325;
    //- ML: initial radius for bubble dynamics
    R0                      1e-06;
    //- ML: vapour pressure for bubble dynamics
    pv                      2300;
    //- ML: time step for each loop of RP solver-important
    RPdT                    5.0e-6;
    //- ML: kappa isotropic value for bubble dynamics
    bubbleKappa             1.4;
    //- ML: Averaging distance for bubble dynamics
    averagingDistance       5;//5;
    //- ML: ODE solver type for bubble dynamics
    ODESolverType           RKF45;
}

Transition_Algorithm
{
    //- ML: flag for activation of EulerianToLagrangian
    EulerianToLagrangian_activation         true;

    //- ML:Number of cells that trigger the transtion from Eulerian to Lagrangian
    EL_cellThreshold                        0;

    //- ML: Set threshold for tracking structures with alpha below this Threshold
    EL_alphaThreshold                       0.999;

    //- ML: flag for activation of LagrangianToEulerian
    LagrangianToEulerian_activation         true;

    //- ML:sigma = bubbleRadius / deviation-> Gaussian standard deviation (for 99% volume within radius)
    LE_deviation                            3.0;

    //- ML:Minimum liquid volume fraction value that lagrangian cell can occupy
    LE_minCellOccupancy                     0.3;

    //- ML:Number of cells that trigger the transtion from Lagrangian to Eulerian
    LE_cellThreshold                        44;

    //- ML: Set threshold for interface proximity that trigger the transtion from Lagrangian to Eulerian
    LE_alphaThreshold                       0.5;

    //- ML: Minimum bubble radius threshold that will be tracked
    LE_bubbleSizeThreshold                  5e-07;

    //- ML: Define the box boundaries for tracking- Only lagrangian inside this box will be tracked
    LE_boxCheckEnabled                      false;

    LE_boxTopLeftCorner                 (0.0 0.0 0.0);  // the top-left corner coordinates at the back face

    LE_boxBottomRightCorner             (1.0 1.0 1.0);  // the bottom-right corner coordinates at the front face
}
```

4.2.1 Results

As shown in Figure 4.6, the nucleus with a diameter of 0.001 m is injected at time 0.2 s and moves from left to right. At time 0.124 s, the number of cells that the unresolved vapor covers exceeds the threshold value of 44, and at time 0.125 s, it transfers to the Eulerian framework. The Lagrangian vapor is then removed from the domain, however, its α_l remains in the domain and flows with the mainstream, driven by the pressure difference between the `inlet` and `outlet`.

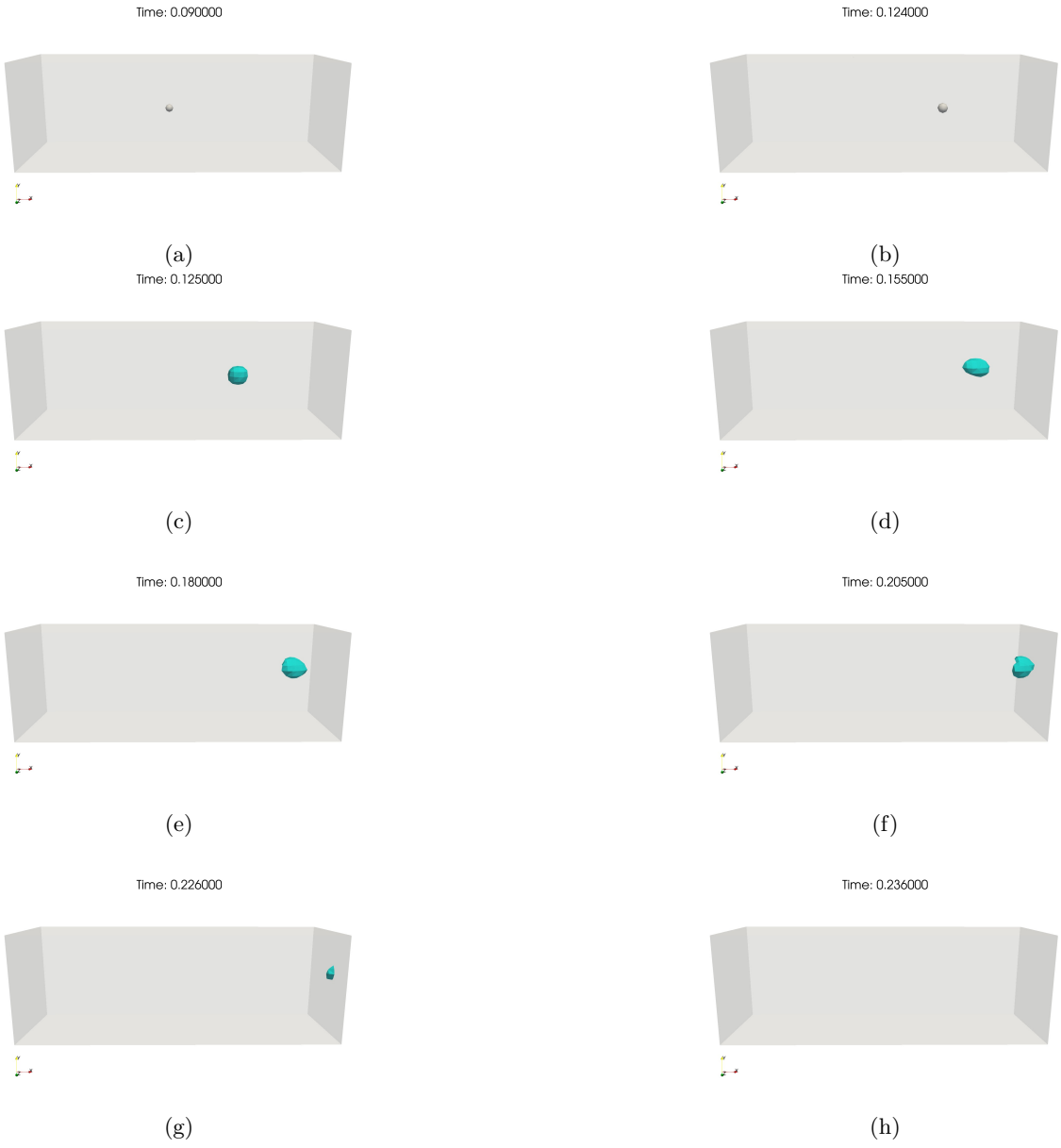


Figure 4.6: Evolution of nucleus at different times for case 2

4.3 Case study 3: checkingEulerianCellThreshold

Case 3 consists of a tank filled with water with an opening or a nozzle at the bottom, and air is injected from bottom, as shown in Figure 4.7. The cells have a uniform length of 0.01 m everywhere, resulting in a total number of 126000 cells. Boundary conditions are defined as `inlet`, `outlet`, `nozzleWall`, `lowerWall`, `leftWall`, `rightWall`, `backWall`, and `frontWall`.

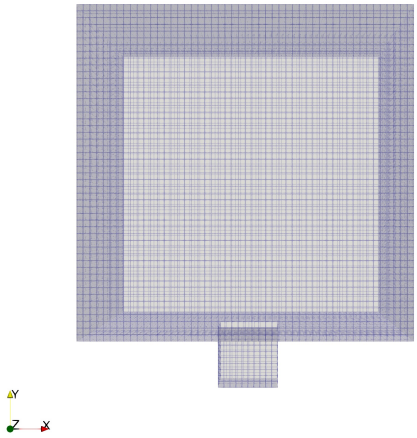


Figure 4.7: Tank with opening from bottom

The `setFields` utility has been used to fill a small part of the nozzle with air as shown in Figure 4.8. The simulation was run for 0.025 s with a time step of 0.0005 s and a write interval of 0.001 s.

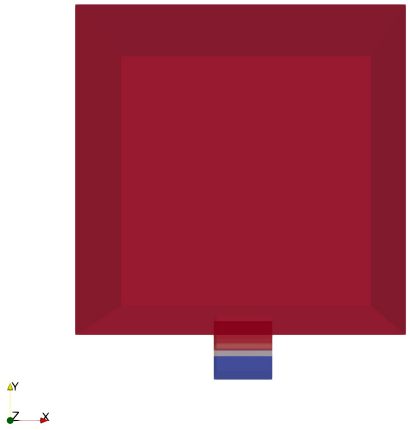


Figure 4.8: Red represents water and blue represents air

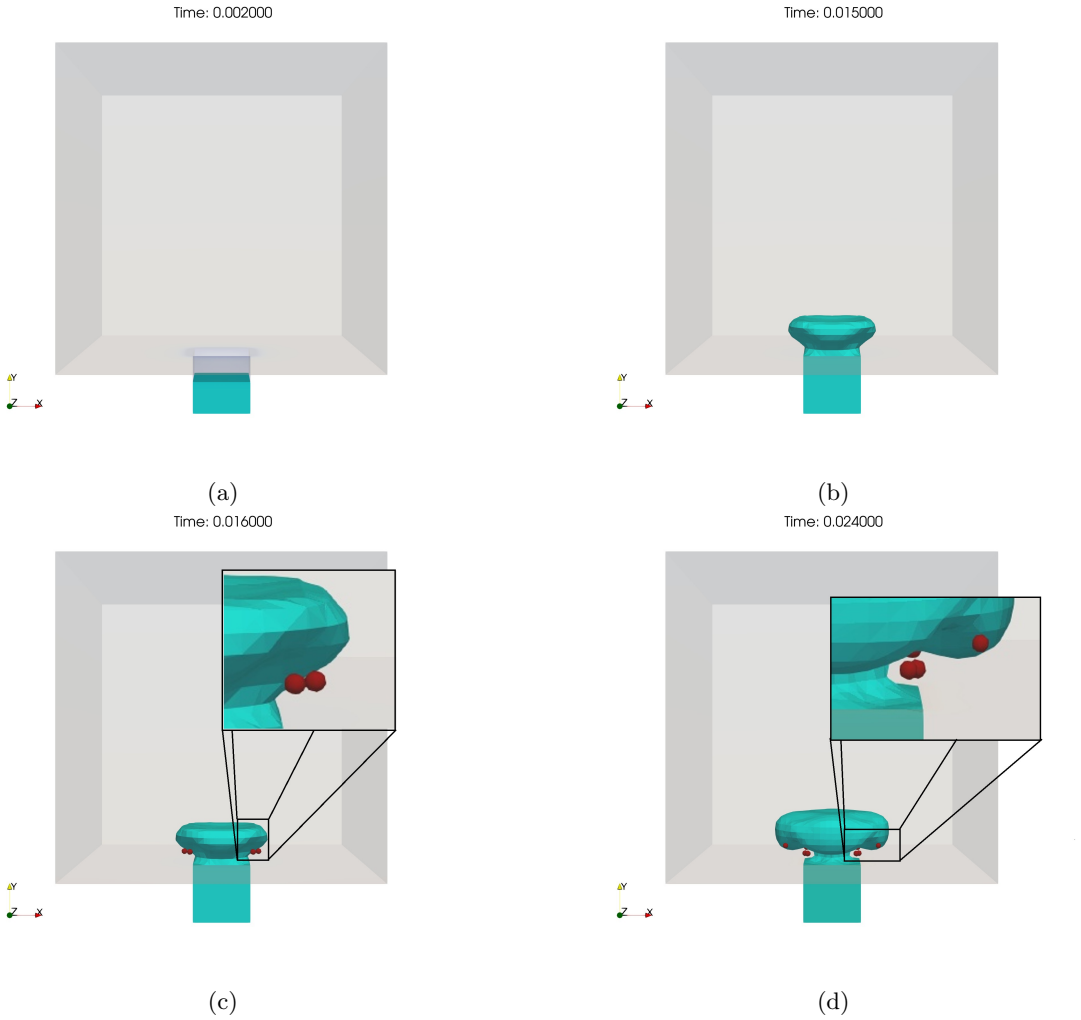
The configuration for `bubbleProperties` and `Transition_Algorithm` dictionaries is shown in Listing 4.3. The flag for the activation of bubble dynamics was disabled, so the Rayleigh-Plesset model did not operate, and changes in bubble diameter were deactivated. Additionally, the flag for the activation of `LagrangianToEulerian` was disabled, hence there was no transition from the Lagrangian to the Eulerian framework. The threshold for Eulerian-to-Lagrangian transition was set to 4. Therefore, whenever a resolved structure size dropped below 4 cells, an unresolved structure with the same size and velocity at the center of the previous resolved structure was injected. All files related to this case can be found in the accompanying files.

Listing 4.3: Configuration of `bubbleProperties` and `Transition_Algorithm` in `kinematicBubbleCloudProperties`

```
bubbleProperties
{
    //- ML: flag for activation of bubble dynamics
    bubble_activation       false;
    //- ML: surface tension for bubble dynamics
    bubbleSigma             0.07;
    //- ML: initial pressure for bubble dynamics
    p0                      101325;
    //- ML: initial radius for bubble dynamics
    R0                      1e-06;
    //- ML: vapour pressure for bubble dynamics
    pv                      2300;
    //- ML: time step for each loop of RP solver-important
    RPdT                    5.0e-6; //1.0e-7 ;
    //- ML: kappa isotropic value for bubble dynamics
    bubbleKappa             1.4;
    //- ML: Averaging distance for bubble dynamics
    averagingDistance       5;//5;
    //- ML: ODE solver type for bubble dynamics
    ODESolverType           RKF45;
}

Transition_Algorithm
{
    //- ML: flag for activation of EulerianToLagrangian
    EulerianToLagrangian_activation         true;

    //- ML:Number of cells that trigger the transtion from Eulerian to Lagrangian
    EL_cellThreshold                        4;

    //- ML: Set threshold for tracking structures with alpha below this Threshold
    EL_alphaThreshold                       0.999;

    //- ML: flag for activation of LagrangianToEulerian
    LagrangianToEulerian_activation         false;

    //- ML:sigma = bubbleRadius / deviation-> Gaussian standard deviation (for 99% volume within radius)
    LE_deviation                            3.0;

    //- ML:Minimum liquid volume fraction value that lagrangian cell can occupy
    LE_minCellOccupancy                     0.3;

    //- ML:Number of cells that trigger the transtion from Lagrangian to Eulerian
    LE_cellThreshold                        44;

    //- ML: Set threshold for interface proximity that trigger the transtion from Lagrangian to Eulerian
    LE_alphaThreshold                       0.5;

    //- ML: Minimum bubble radius threshold that will be tracked
    LE_bubbleSizeThreshold                  5e-07;

    //- ML: Define the box boundaries for tracking- Only lagrangian inside this box will be tracked
    LE_boxCheckEnabled                      false;

    LE_boxTopLeftCorner                 (0.0 0.0 0.0);  // the top-left corner coordinates at the back face

    LE_boxBottomRightCorner             (1.0 1.0 1.0);  // the bottom-right corner coordinates at the front face
}
```

4.3.1 Results

As shown in Figure 4.9, the airflow enters the tank through the nozzle. When the fluid expands into the water above the opening, it introduces small vapor structures smaller than the predefined threshold of 4 cells. At time 0.016 s, the first resolved vapors are introduced, and as the simulation progresses, the number of unresolved vapors increases.



(a) (b) (c) (d)

Figure 4.9: Injection of air into the water from bottom of the tank. The Eularian structures are represented in cyan, while the Lagrangian structures are represented in red.

4.4 Case study 4: checkingLEBox

Since the number of unresolved bubbles can reach thousands, tracking them may become computationally expensive. In such cases, defining a zone of interest to track the Lagrangian bubbles within that region can be a practical approach. Here, we use the same mesh and configuration as in case study 2, with one difference: around 200 bubbles were injected into the domain using the `patchInjection` model. A limitation zone was defined by introducing a box with top left corner at (0.0, 0.1, 0.0) and bottom right corner at (0.2, 0.0, 0.1).

The configuration for the `bubbleProperties` and `Transition_Algorithm` dictionaries is shown in Listing 4.4. The flag for the activation of bubble dynamics was disabled, so the Rayleigh-Plesset model did not get activated and hence the changes in the bubble diameter are zero. All files related to this case can be found in the accompanying files.

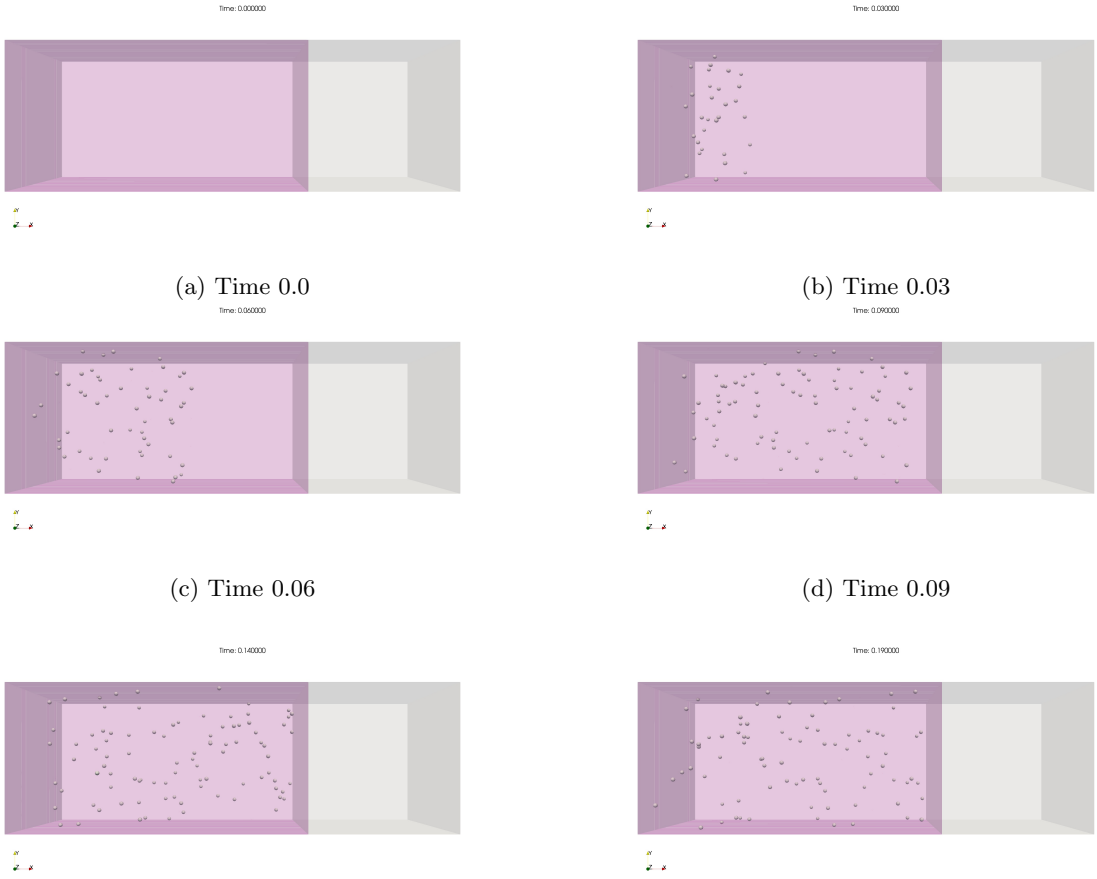
Listing 4.4: Configuration of `bubbleProperties` and `Transition_Algorithm` in `kinematicBubbleCloudProperties`

```
bubbleProperties
{
    //- ML: flag for activation of bubble dynamics
    bubble_activation       false;
    //- ML: surface tension for bubble dynamics
    bubbleSigma             0.07;
    //- ML: initial pressure for bubble dynamics
    p0                      101325;
    //- ML: initial radius for bubble dynamics
    R0                      1e-06;
    //- ML: vapour pressure for bubble dynamics
    pv                      2300;
    //- ML: time step for each loop of RP solver-important
    RPdT                    5.0e-6; //1.0e-7 ;
    //- ML: kappa isotropic value for bubble dynamics
    bubbleKappa             1.4;
    //- ML: Averaging distance for bubble dynamics
    averagingDistance       5;//5;
    //- ML: ODE solver type for bubble dynamics
    ODESolverType           RKF45;
}

Transition_Algorithm
{
    //- ML: flag for activation of EulerianToLagrangian
    EulerianToLagrangian_activation         true;

    //- ML:Number of cells that trigger the transtion from Eulerian to Lagrangian
    EL_cellThreshold                        0;

    //- ML: Set threshold for tracking structures with alpha below this Threshold
    EL_alphaThreshold                       0.999;

    //- ML: flag for activation of LagrangianToEulerian
    LagrangianToEulerian_activation         true;

    //- ML:sigma = bubbleRadius / deviation-> Gaussian standard deviation (for 99% volume within
     radius)
    LE_deviation                            3.0;

    //- ML:Minimum liquid volume fraction value that lagrangian cell can occupy
    LE_minCellOccupancy                     0.3;

    //- ML:Number of cells that trigger the transtion from Lagrangian to Eulerian
    LE_cellThreshold                        20000;

    //- ML: Set threshold for interface proximity that trigger the transtion from Lagrangian to
     Eulerian
    LE_alphaThreshold                       0.5;

```

```
49      //- ML: Minimum bubble radius threshold that will be tracked
50      LE_bubbleSizeThreshold                  5e-07;
51
52      //- ML: Define the box boundaries for tracking- Only lagrangian inside this box will be tracked
53      LE_boxCheckEnabled                      true;
54
55      LE_boxTopLeftCorner                 (0.0 0.1 0.0);  // the top-left corner coordinates at the back
          face
56
57      LE_boxBottomRightCorner             (0.2 0.0 0.1);  // the bottom-right corner coordinates at the
         front face
58  }
```

4.4.1 Results

As shown in Figure 4.10, a front of approximately 200 bubbles is injected from the patch `inlet`. As the bubbles move outside the pink box, they are removed from the domain before reaching the outlet.



(a) Time 0.0

(b) Time 0.03

(c) Time 0.06

(d) Time 0.09

(e) Time 0.14

(f) Time 0.19

Figure 4.10: The area of interest is represented by pink box, while the gray box represents the entire domain.

Bibliography

[1] B. Chen, "Eulerian-lagrangian modeling of cavitation," *Proceedings of CFD with OpenSource Software, 2013, Edited by Nilsson H.* [Online]. Available: https://dx.doi.org/10.17196/OS_CFD#YEAR_2013

[2] E. Ghahramani, "Numerical simulation and analysis of multi-scale cavitating flows using a hybrid mixture-bubble model PhD thesis," Ph.D. dissertation, Chalmers University of Technology, 2020.

[3] A. Vallier, "Simulations of cavitation from the large vapour structures to the small bubble dynamics PhD thesis," Ph.D. dissertation, Lund University, 2013.

[4] D. Vaca-Revelo and A. Gnanaskandan, "Numerical assessment of the condensation shock mechanism in sheet to cloud cavitation transition," *International Journal of Multiphase Flow*, vol. 169, p. 104616, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0301932223002367

[5] E. Ghahramani, U. Svennberg, and R. E. Bensow, "Numerical simulation of tip vortex cavitation inception," *Proceedings of the 11th International Symposium on Cavitation, Daejeon, Korea.*

[6] K. A. Ghasemi, "Implementing a non-isothermal interPhaseChangeFoam solver with a thermodynamic cavitation model," *Proceedings of CFD with OpenSource Software, 2023, Edited by Nilsson H.* [Online]. Available: https://dx.doi.org/10.17196/OS_CFD#YEAR_2023

[7] A. Asnagi, "interPhaseChangeFoam tutorial and pans turbulence model," *Proceedings of CFD with OpenSource Software, 2013, Edited by Nilsson H.* [Online]. Available: https://dx.doi.org/10.17196/OS_CFD#YEAR_2013

[8] N. LU, "Solve cavitating flow around a 2D hydrofoil using a user modified version of interPhaseChangeFoam," *Proceedings of CFD with OpenSource Software, 2008, Edited by Nilsson H.* [Online]. Available: https://dx.doi.org/10.17196/OS_CFD#YEAR_2008

[9] M. Andersen, "A interPhaseChangeFoam tutorial," *Proceedings of CFD with OpenSource Software, 2011, Edited by Nilsson H.* [Online]. Available: https://dx.doi.org/10.17196/OS_CFD#YEAR_2011

[10] M. Lavari, D. Vaca-Revelo, and A. Gnanaskandan, "Multiscale modelling of sheet to cloud cavitation transition," *Proceedings of the 12th International Cavitation Symposium – CAV2024, MCh Conference Centre in Chania, Greece.*

[11] S. Elghobashi, "On predicting particle-laden turbulent flows," *Applied Scientific Research, 52(4), 309–329.*

[12] A. Lopez, "LPT for erosion modeling in OpenFOAM," *Proceedings of CFD with OpenSource Software, 2013, Edited by Nilsson H.* [Online]. Available: https://dx.doi.org/10.17196/OS_CFD#YEAR_2013

[13] J. Andric, "Lagrangian particle tracking," *Proceedings of CFD with OpenSource Software, 2009, Edited by Nilsson H.* [Online]. Available: https://dx.doi.org/10.17196/OS_CFD#YEAR_2009

[14] Örjan Fjällborg, "Implementation of a new heat transfer model in OpenFOAM for lagrangian particle tracking solvers for use in porous media," *Proceedings of CFD with OpenSource Software, 2022, Edited by Nilsson H.* [Online]. Available: https://dx.doi.org/10.17196/OS_CFD#YEAR_2022

[15] M. Kampili, "Implementation of decay heat model as a submodel in lagrangian library for reactingParcelFoam solver," *Proceedings of CFD with OpenSource Software, 2017, Edited by Nilsson H.* [Online]. Available: https://dx.doi.org/10.17196/OS_CFD#YEAR_2017

[16] J. Xu, "Modification of stochastic model in lagrangian tracking method," *Proceedings of CFD with OpenSource Software, 2016, Edited by Nilsson H.* [Online]. Available: https://dx.doi.org/10.17196/OS_CFD#YEAR_2016

[17] M. Nobile, "Improvement of lagrangian approach for multiphase flow," *Proceedings of CFD with OpenSource Software, 2014, Edited by Nilsson H.* [Online]. Available: https://dx.doi.org/10.17196/OS_CFD#YEAR_2014

[18] E. Ghahramani, "Improvement of the VOF-LPT solver for bubbles," *Proceedings of CFD with OpenSource Software, 2016, Edited by Nilsson H.* [Online]. Available: https://dx.doi.org/10.17196/OS_CFD#YEAR_2016

[19] A. Vallier, "Descriptions and modifications of icoLagrangianFoam," *Proceedings of CFD with OpenSource Software, 2009, Edited by Nilsson H.* [Online]. Available: https://dx.doi.org/10.17196/OS_CFD#YEAR_2009

[20] A.Vallier, "LPT and VOF with OpenFOAM," 2011. [Online]. Available: https://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2011/OF_kurs_LPT_120911.pdf

[21] E. Ghahramani, "Coupling of VOF-based solver with LPT for simulation of cavitating flows." [Online]. Available: https://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2017/OSCFD2017_Ghahramani/OF_Course_Ghahramani.pdf

[22] M. Heinrich and R. Schwarze, "3D-coupling of volume-of-fluid and lagrangian particle tracking for spray atomization simulation in OpenFOAM," *SoftwareX*, vol. 11, p. 100483, 2020. [Online]. Available: https://doi.org/10.1016/j.softx.2020.100483

Study questions

- How is the Reynolds transport equation for liquid volume fraction defined in `interPhaseChangeFoam`?
- How do Lagrangian forces affect the velocity of parcels in OpenFOAM?
- How is the effective viscosity calculated for the momentum equation in OpenFOAM?
- What is the rate of mass change when the bubble transitions from the Lagrangian framework?
- How is a parcel injected into the parcel cloud?
- What criterion is utilized to model mass transfer in the Schnerr-Sauer model?

Appendix A

Implemented Transition Algorithms

A.1 Lagrangian to Eulerian Algorithm

Algorithm for transition from the Lagrangian framework to the Eulerian framework.

Listing A.1: `LagrangianToEulerian.H`

```
//-ML: Set data from bubble
vector posB = this->position();
scalar radB = this->d() / 2;
label startCell = p.cell();
scalar totalBubbleVolume = (p.nParticle()*p.volume());

//-ML: DynamicList to store cells and their distances
DynamicList<std::pair<label, scalar>> cellsWithDistances;

//-ML: Add the start cell to the list
vector cellCenter = this->mesh().cellCentres()[startCell];
cellsWithDistances.append(std::make_pair(startCell, mag(cellCenter - posB)));  //-ML: Add the first
    cell with distance

//-ML: Create a set to avoid re-visiting cells
labelHashSet visitedCells;
visitedCells.insert(startCell);

//-ML: Dynamic list to store all neighbors for further exploration
DynamicList<label> cells(0);
cells.append(this->mesh().cellCells()[startCell]);

//-ML: Start outer loop to explore neighbors
while (cells.size() > 0)
{
    //-ML: New dynamic list to store newly found neighbors within the radius
    DynamicList<label> newNeighbours;

    //-ML: Loop over all current neighbors in the cells list
    forAll(cells, i)
    {
        label cellI = cells[i];
        if (!visitedCells.found(cellI))  //-ML: Check if cellI is already visited
        {
            cellCenter = this->mesh().cellCentres()[cellI];

            //-ML: Check if this neighbor cell's center is within the radius
            scalar distance = mag(cellCenter - posB);  //-ML: Calculate distance to posB

            if (distance <= (2.0 * radB))
```

```
            {
                cellsWithDistances.append(std::make_pair(cellI, distance));  //-ML: Store the cell and
     its distance
                newNeighbours.append(this->mesh().cellCells()[cellI]);  //-ML: Add neighbors to the
     list
            }

            visitedCells.insert(cellI);  //-ML: Mark cell as visited
        }
    }

    //-ML: Sort the cells based on their distance to posB
    std::sort(cellsWithDistances.begin(), cellsWithDistances.end(),
              [](const std::pair<label, scalar>& a, const std::pair<label, scalar>& b)
              {
                  return a.second < b.second;  //-ML: Compare distances
              });

    //-ML: Continue to the next iteration with new Neighbours
    cells.clear();
    cells.transfer(newNeighbours);
}

//-ML: Initialize the boolean flag for proximity to the interface
bool nearInterface = false;

scalar alphaThreshold = this->LE_alphaThreshold(); //-ML: Set threshold for interface proximity


//-ML: Check if cellsWithDistances contains only one cell
if (cellsWithDistances.size() == 1)
{
    //-ML: Initialize a new DynamicList to hold the single cell and its neighbors
    DynamicList<label> singleCellAndNeighbors;

    //-ML: Get the single cell and add it to the new list
    label onlyCell = cellsWithDistances[0].first;
    singleCellAndNeighbors.append(onlyCell);

    //-ML: Add the neighbors of this single cell to the list
    singleCellAndNeighbors.append(this->mesh().cellCells()[onlyCell]);

    //-ML: Loop over the single cell and its neighbors to check alpha values
    forAll(singleCellAndNeighbors, j)
    {
        label cellJ = singleCellAndNeighbors[j];
        scalar alphaValue = cloud.alpha1()[cellJ];  //-ML: Retrieve the alpha value for cellJ

        //-ML: Check if alpha value exceeds the threshold
        if (alphaValue < alphaThreshold)
        {
            nearInterface = true;
            break;  //-ML: Exit loop early if the condition is met
        }
    }
}
else
{
    //-ML: loop over cellsWithDistances and check alpha values
    forAll(cellsWithDistances, j)
    {
        label cellI = cellsWithDistances[j].first;  //-ML: Extract cell label
        scalar alphaValue = cloud.alpha1()[cellI];  //-ML: Retrieve the alpha value for cellI

        //-ML: Check if alpha value exceeds threshold
        if (alphaValue < alphaThreshold)
        {
            nearInterface = true;
```

```
            break;  //-ML: Exit loop early if the condition is met
        }
    }
}

//-ML: Gaussian standard deviation
scalar sigma = radB / this->LE_deviation();

scalar totalWeightWithinRadius = 0.0;

scalar minCellOccupancy = this->LE_minCellOccupancy();

//-ML: Maximum fraction of volume for any single cell
scalar maxFraction = 0.0;

//-ML: Accumulate excess volume to redistribute
scalar excessVolume = 0.0;

//-ML: Set a threshold for the number of cells within the radius
label cellThreshold = this->LE_cellThreshold();

//-ML: Minimum bubble radius threshold
scalar bubbleSizeThreshold = this->LE_bubbleSizeThreshold();

//-ML: Calculate the Gaussian weight for each cell and accumulate total weight within radius
List<scalar> gaussianWeights(cellsWithDistances.size());

forAll(cellsWithDistances, j) {

    scalar gaussFactor;

    if (cellsWithDistances.size() == 1) {
        gaussFactor = 1.0;  //-ML: Set gaussFactor to 1 if there is only one cell
    } else {
        scalar distance = cellsWithDistances[j].second;  //-ML: Get distance from pair
        //-ML: Calculate Gaussian weight for this cell
        gaussFactor = exp(-0.5 * pow(distance / sigma, 2.0));
    }
    gaussianWeights[j] = gaussFactor;

    //-ML: Accumulate total weight
    totalWeightWithinRadius += gaussFactor;
}

//-ML: Create a list to store cell IDs and distances with excess volume
DynamicList<label> excessVolumeCells;

//-ML: Distribute the bubble volume using normalized Gaussian weights
forAll(cellsWithDistances, j) {

    label cellJ = cellsWithDistances[j].first;
    //-ML:scalar distance = cellsWithDistances[j].second;  //-ML: Get distance from pair
    scalar cellVolume = this->mesh().cellVolumes()[cellJ];

    //-ML: Normalize the Gaussian weight and compute `vi_j` for this cell
    scalar normalizedWeight = gaussianWeights[j] / totalWeightWithinRadius;

    //-ML:Info << "normalizedWeight" << normalizedWeight << endl;
    scalar vi_j = normalizedWeight * totalBubbleVolume / cellVolume;

    maxFraction = (cloud.betal()[cellJ] - minCellOccupancy);
    //-ML: Apply the limit on maximum fraction of volume
    scalar cappedVi_j = min(vi_j, maxFraction);

    //-ML: Calculate any excess volume that needs redistribution
    if (vi_j > maxFraction) {
        excessVolume += (vi_j - maxFraction) * cellVolume;
    } else if (vi_j < maxFraction) {
```

```
        excessVolumeCells.append(cellJ);  //-ML: Store the cell
    }

    //-ML: Update beta value with limit check
    cloud.betal()[cellJ] -= cappedVi_j;

    //-ML: Check if the updated beta value goes below 1; if so, set alpha to 1
    if (cloud.betal()[cellJ] < 1 && !nearInterface)
    {
        cloud.alphal()[cellJ] = 1.0;
    }
}

//-ML: Check if cellsWithDistances has only one cell and
// there is still excessVolume, so there is no excessVolumeCells
if (cellsWithDistances.size() == 1 && excessVolume > 0) {

    label cellJ = cellsWithDistances[0].first;
    //-ML: Add the neighbors of this single cell to the list
    excessVolumeCells.append(this->mesh().cellCells()[cellJ]);

}

//-ML: Redistribute any excess volume among cells that still have available capacity
forAll(excessVolumeCells, j) {

    label cellJ = excessVolumeCells[j];  //-ML: Get the cell ID

    scalar cellVolume = this->mesh().cellVolumes()[cellJ];

    //-ML: Log the initial beta value before redistribution
    scalar initialBeta = cloud.betal()[cellJ];

    //-ML: Determine remaining capacity for this cell
    scalar remainingCapacity = initialBeta - minCellOccupancy;

    //-ML: Only distribute to cells that still have capacity available
    if (remainingCapacity > 0 && excessVolume > 0) {
        //-ML: Calculate the portion of excess volume this cell can take
        scalar additionalVolume = min(excessVolume, remainingCapacity * cellVolume);

        //-ML: Apply additional volume to beta
        cloud.betal()[cellJ] -= additionalVolume / cellVolume;

        //-ML: Reduce excess volume by the amount distributed
        excessVolume -= additionalVolume;

        if (cloud.betal()[cellJ] < 1 && !nearInterface)
        {
            cloud.alphal()[cellJ] = 1.0;
        }

    }
}

//-ML: Log the information
Info << "This bubble with ID: " << p.origId()
<< "\n with the processor ID: " << p.origProc()
<< "\n in position: " << posB
<< "\n with radius: " << radB
<< "\n within center cellID: " << startCell
<< "\n with bubble volume: " << totalBubbleVolume
<< "\n final excessVolume: " << excessVolume
<< "\n the number of cells within 2xdiameter: " <<  cellsWithDistances.size() << endl;

//-ML: Apply the condition for number of cells within the radius or proximity to interface
if ((cellsWithDistances.size() > cellThreshold || nearInterface) && radB >= bubbleSizeThreshold)
{
```

```
    forAll(cellsWithDistances, j) {
        label cellJ = cellsWithDistances[j].first;  //-ML: Get the cell ID
        //-ML: Decrease alphal by betal, ensuring alphal doesn't go below zero
        cloud.alphal()[cellJ] = max(cloud.alphal()[cellJ] - (1.0 - cloud.betal()[cellJ]), 0.0);
        //-ML: Set betal to 1
        cloud.betal()[cellJ] = 1.0;
    }

    //-ML: Stop tracing the bubble by setting it as inactive
    td.keepParticle = false;
    Info << "This bubble tracing deactivated for particle at position: " << posB << endl;

    //-ML: Log the reason for deactivating bubble tracing
    if (cellsWithDistances.size() > cellThreshold) {
        Info << "\n Reason: Number of cells within the radius exceeded the threshold." << endl;
    }
    else if (nearInterface) {
        Info << "\n Reason: Proximity to interface detected." << endl;
    }
}
else if (radB < bubbleSizeThreshold)
{
    //-ML: If bubble size is below the threshold, deactivate tracking
    td.keepParticle = false;
    Info << "This bubble tracing deactivated for particle at position: " << posB << endl;
    Info << "\n Reason: Bubble radius (" << radB << ") is below the size threshold (" <<
     bubbleSizeThreshold << ")." << endl;
}

//-ML: Activation control for box boundary check
bool boxCheckEnabled = this->LE_boxCheckEnabled();

//-ML: Define the box boundaries
vector boxTopLeftCorner = this->LE_boxTopLeftCorner();  //-ML: the top-left corner coordinates
vector boxBottomRightCorner = this->LE_boxBottomRightCorner();  //-ML: the bottom-right corner
     coordinates

//-ML: Check if the bubble parcel is within the box boundaries
bool withinBox = (posB.x() >= boxTopLeftCorner.x() && posB.x() <= boxBottomRightCorner.x() &&
                  posB.y() <= boxTopLeftCorner.y() && posB.y() >= boxBottomRightCorner.y() &&
                  posB.z() >= boxTopLeftCorner.z() && posB.z() <= boxBottomRightCorner.z());

//-ML: If bubble parcel is outside the box and box check is enabled, deactivate it
if (!withinBox && boxCheckEnabled) {
    td.keepParticle = false;
    Info << "This bubble at position " << posB << " is outside the box and removed." << endl;
}
```

A.2 Eulerian to Lagrangian Algorithm

Algorithm for transition from the Eulerian framework to the Lagrangian framework.

Listing A.2: `EulerianToLagrangian.H`

```
//-ML:  Threshold for alpha1
scalar alpha1Lim = EL_alphaThreshold;
//-ML:  Minimum number of cells to consider as a structure
label minCells = EL_cellThreshold;

//-ML:  Reset volumeID
volScalarField vofID_
(
    IOobject
    (
        "vofID",
```

```
        mesh.time().timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::NO_WRITE
    ),
    mesh,
    dimensionedScalar("vofID", dimless, scalar(0.0)),
    zeroGradientFvPatchScalarField::typeName
);

//-ML:  Local-to-global cell reference
globalIndex globalNumbering (mesh.nCells());

//-ML:  Create the volumeID field
forAll(mesh.cells(), cellI)
{
    //-ML:  Convert local processor cellIDs to global cellIDs
    label globalCellID = globalNumbering.toGlobal(cellI);

    //-ML:  Cell is considered vapor and not yet marked
    if (alpha1[cellI] < alpha1Lim && mag(vofID_[cellI]) < 0.1)
    {

        label currentID = globalCellID+1;
        label startCell = cellI;
        //-ML:  Set volume ID
        vofID_[startCell] = currentID;

        //-ML:  Create dynamic list to store all neighbours
        DynamicList<label> cells(0);
        //-ML:  Get neighbours of startCell
        cells.append(mesh.cellCells()[startCell]);

        //-ML:  Start outer loop
        while (cells.size() > 0)
        {
            //-ML:  Create new dynamic list to store all new neighbours
            DynamicList<label> neighbours(0);

            //-ML:  Loop over all current neighbours
            forAll(cells, i)
            {
                label cellJ = cells[i];

                //-ML:  Check if neighbour cell is liquid and has no ID yet
                if (alpha1[cellJ] < alpha1Lim && mag(vofID_[cellJ] - currentID) > 0.1)
                {
                    //-ML:  Set volume ID
                    vofID_[cellJ] = currentID;

                    //-ML:  add neighbours to neighboursNeighbours list
                    neighbours.append(mesh.cellCells()[cellJ]);
                }
            }
            cells.clear();
            cells.transfer(neighbours);
        }
    }
}
vofID_.correctBoundaryConditions();


//-ML:  Correct for parallel processing
//-ML:  Number of corrections
label corr = Foam::log(scalar(Pstream::nProcs())) / Foam::log(2.0) + 1;
for(int i=0; i<=corr; i++)
{
    forAll(mesh.boundary(), patchI)
```

```
    {
        //-ML:  Current patch
        const fvPatch& pPatch = mesh.boundary()[patchI];

        //-ML:  check if patch is coupled, e.g. processor patch
        if (pPatch.coupled())
        {
            //-ML:  loop over all faces of processor patch
            forAll(pPatch, faceI)
            {
                //-ML:  get cell values
                const scalar ID_own = vofID_.boundaryField()[patchI].patchInternalField()()[faceI];
                const scalar ID_nei = vofID_.boundaryField()[patchI].patchNeighbourField()()[faceI];

                //-ML:  check, if volumeIDs on both sides are fluid
                if ( (ID_nei > 0.1) && (ID_own > 0.1) )
                {
                    //-ML:  check, if volumeIDs on both sides are different
                    if ( mag(ID_nei - ID_own) > 0.1 )
                    {
                        scalar minID = Foam::min(ID_own, ID_nei);
                        scalar maxID = Foam::max(ID_own, ID_nei);

                        scalar diff(maxID - minID);
                        volScalarField filtered = diff
                                                * pos( vofID_ - maxID + 0.1)
                                                * pos(-vofID_ + maxID + 0.1);
                        vofID_ -= filtered;
                    }
                }
            }
        }
    }
    vofID_.correctBoundaryConditions();
}

//-ML: Count number of continua and renumber vofID
{
    volScalarField tmp(vofID_);
    vofID_ = 0.0;
    scalar maxID = gMax(tmp);
    int iter = 1;
    while (maxID > 0.1)
    {
        volScalarField curVolID = pos(tmp - maxID + 0.1);
        vofID_ += iter*curVolID;
        tmp -= maxID*curVolID;
        maxID = gMax(tmp);
        iter++;
    }
}

//-ML: Calculate droplet volume, velocity, and position
//-ML: Create lists to store data
label maxID = floor(gMax(vofID_) + 0.5);
labelList noCells(maxID+1, label(0));
scalarList cellVolume(maxID+1, scalar(0));
scalarList volume(maxID+1, scalar(0));
vectorList position(maxID+1, vector(0,0,0));
vectorList velocity(maxID+1, vector(0,0,0));

//-ML: Loop over all cells and store corresponding data
forAll(mesh.cells(), cellI)
{
    if (vofID_[cellI] > 0.1)
    {
        //-ML: Get volume ID
        label volID = floor(vofID_[cellI] + 0.5);
```

```
        //-ML: Store data in corresponding list
        scalar oneMinusalpha1 = (1 - alpha1[cellI]);
        noCells[volID] += 1;
        cellVolume[volID] += mesh.V()[cellI];
        volume[volID] += oneMinusalpha1*mesh.V()[cellI];
        velocity[volID] += oneMinusalpha1*mesh.V()[cellI]*U[cellI];
        position[volID] += oneMinusalpha1*mesh.V()[cellI]*mesh.C()[cellI];
    }
}

//-ML: Account for parallel processing
reduce( noCells, sumOp<labelList>() );
reduce( cellVolume, sumOp<scalarList>() );
reduce( volume, sumOp<scalarList>() );
reduce( position, sumOp<vectorList>() );
reduce( velocity, sumOp<vectorList>() );

//-ML: Print out Info
for(int i=1; i<=maxID; i++)
{
    Info << "/************** Structure Number is " << i << " **************/" << endl;
    Info << "Number of Cells: " << noCells[i] << endl;
    Info << "Volume of Structure: " << volume[i] << endl;
    Info << "position of Structure: " << position[i] / volume[i] << endl;
    Info << "velocity of Structure: " << velocity[i] / volume[i] << endl;
}


//-ML: Process small structures
//-ML: Add code to inject properties (e.g., position, velocity) into the cloud
for (label i = 1; i <= maxID; i++)
{
    if (noCells[i] < minCells)
    {

        forAll(mesh.cells(), cellI)
        {
            label strcutureID = floor(vofID_[cellI] + 0.5);
            if (strcutureID == i)
            {
                alpha1[cellI] = 1.0; //-ML: Set alpha1 to 1 for small structures
            }
        }

        //-ML: Calculate center position of the structure
        vector bubblePosition = position[i] / volume[i] ;
        //-ML: Find the cell that contains this position
        label cellSample = mesh.findCell(bubblePosition);
        //-ML: Calculate average velocity of the structure
        vector bubbleVelocity = velocity[i] / volume[i] ;

        if (cellSample > -1)
        {
            //-ML: Apply corrections to position for 2-D cases
            meshTools::constrainToMeshCentre(mesh, bubblePosition);
            //-ML: Inject into Lagrangian cloud
            basicKinematicBubbleParcel* pPtr = new basicKinematicBubbleParcel(mesh, bubblePosition,
    cellSample);
            //-ML: Number of particles per parcel
            pPtr->nParticle() = 1;
            //-ML: Particle diameter
            pPtr->d() = Foam::cbrt(6.0 * volume[i] / constant::mathematical::pi);
            //-ML: Velocity
            pPtr->U() = bubbleVelocity;

            basicKinematicBubbleParcel::trackingData td(parcels);
            scalar trackTime = mesh.time().deltaTValue();
```

```
            td.part() = basicKinematicBubbleParcel::trackingData::tpLinearTrack;

            //-ML:  Check/set parcel properties
            parcels.setParcelThermoProperties(*pPtr, trackTime);
            parcels.checkParcelProperties(*pPtr, trackTime, false);

            //-ML:  Apply correction to velocity for 2-D cases
            meshTools::constrainDirection
            (
                mesh,
                mesh.solutionD(),
                pPtr->U()
            );

            if (pPtr->move(parcels, td, trackTime)) //-ML: Check the possiblity of using move function
            {
                Pout << "Injecting Structure " << i
                << " into Lagrangian cloud by processor: " << Pstream::myProcNo()
                << " with velocity of " << pPtr->U()
                << " and diamter " << pPtr->d() << endl;
                parcels.addParticle(pPtr); //-ML: Add particle to the cloud
            }
            else
            {
                delete pPtr;
            }

        }

    }
}
```

Appendix B

Developed codes

B.1 `interPhaseChangeBubbleFoam` solver

Listing B.1: `interPhaseChangeBubbleFoam.C`

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

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

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

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

Application
    interPhaseChangeBubbleFoam

Group
    grpMultiphaseSolvers

Description
    Solver for two incompressible, isothermal immiscible fluids with
    phase-change (e.g. cavitation).
    Uses VOF (volume of fluid) phase-fraction based interface capturing.

    The momentum and other fluid properties are of the "mixture" and a
    single momentum equation is solved.

    The set of phase-change models provided are designed to simulate cavitation
    but other mechanisms of phase-change are supported within this solver
    framework.

    Turbulence modelling is generic, i.e. laminar, RAS or LES may be selected.
```

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

#include "fvCFD.H"
#include "CMULES.H"
#include "subCycle.H"
#include "interfaceProperties.H"
#include "hybridPhaseChangeTwoPhaseMixture.H"
#include "turbulentTransportModel.H"
#include "pimpleControl.H"
#include "fvOptions.H"

//-ML : declarition of basicKinematicBubbleCloud
#include "basicKinematicBubbleCloud.H"

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

int main(int argc, char *argv[])
{
    argList::addNote
    (
        "Solver for two incompressible, isothermal immiscible fluids with"
        " phase-change.\n"
        "Uses VOF (volume of fluid) phase-fraction based interface capturing."
    );

    #include "postProcess.H"

    #include "addCheckCaseOptions.H"
    #include "setRootCaseLists.H"
    #include "createTime.H"
    #include "createMesh.H"
    #include "createControl.H"
    #include "createFields.H"
    #include "createTimeControls.H"
    #include "CourantNo.H"
    #include "setInitialDeltaT.H"

    turbulence->validate();

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

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

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

        ++runTime;

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

        // --- Pressure-velocity PIMPLE corrector loop
        while (pimple.loop())
        {
            #include "alphaControls.H"

            surfaceScalarField rhoPhi
            (
                IOobject
                (
                    "rhoPhi",
                    runTime.timeName(),
                    mesh
                ),
                mesh,
```

```
                dimensionedScalar(dimMass/dimTime, Zero)
            );

            //-ML: update the gamma value from multiplication of alpha1_*beta1_
            mixture->update_gamma();
            //-ML: update the mixture nu() forlaminar by using gamma
            mixture->correct();

            #include "alphaEqnSubCycle.H"
            interface.correct();

            #include "UEqn.H"

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

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

        //-ML: Transition from Eulerian to Lagrangian
        if(EulerToLagrang_activation)
        {
            #include "EulerianToLagrangian.H"
        }
        mixture->update_gamma();

        //-ML: Adding evolution of cloud
        Info<< "Evolution of Cloud: "<< parcels.name() << endl;

        parcels.evolve();
        //-ML: update the gamma value from multiplication of alpha1_*beta1_
        mixture->update_gamma();

        runTime.write();

        runTime.printExecutionTime(Info);
    }

    Info<< "End\n" << endl;

    return 0;
}


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

B.2 Make/options for interPhaseChangeBubbleFoam solver

Listing B.2: options file

```
EXE_INC = \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude \
    -I$(LIB_SRC)/sampling/lnInclude \
    -I$(LIB_SRC)/TurbulenceModels/turbulenceModels/lnInclude \
    -I$(LIB_SRC)/TurbulenceModels/incompressible/lnInclude \
    -I$(LIB_SRC)/finiteArea/lnInclude \
    -I$(LIB_SRC)/lagrangian/distributionModels/lnInclude \
    -I$(LIB_SRC)/regionModels/regionModel/lnInclude \
```

```
    -I$(LIB_SRC)/regionModels/surfaceFilmModels/lnInclude \
    -I$(LIB_SRC)/regionFaModels/lnInclude \
    -I$(LIB_SRC)/faOptions/lnInclude \
    -I$(LIB_SRC)/lagrangian/basic/lnInclude \
    -I$(LIB_SRC)/lagrangian/intermediate/lnInclude \
    -I./hybridTransportModels/hybridTwoPhaseMixture/lnInclude \
    -I$(LIB_SRC)/transportModels \
    -I./hybridTransportModels/hybridIncompressible/lnInclude \
    -I$(LIB_SRC)/transportModels/interfaceProperties/lnInclude \
    -I./hybridPhaseChangeTwoPhaseMixtures/lnInclude \
    -I./bubbleDynamics/lnInclude \
    -I$(LIB_SRC)/ODE/lnInclude

EXE_LIBS = \
    -L$(FOAM_USER_LIBBIN) \
    -lfiniteVolume \
    -lfvOptions \
    -lmeshTools \
    -lsampling \
    -lregionModels \
    -lsurfaceFilmModels \
    -lsurfaceFilmDerivedFvPatchFields \
    -lturbulenceModels \
    -lincompressibleTurbulenceModels \
    -llagrangian \
    -llagrangianIntermediate \
    -llagrangianTurbulence \
    -lfiniteArea \
    -lfaOptions \
    -lhybridTwoPhaseMixture \
    -ltwoPhaseProperties \
    -linterfaceProperties \
    -lhybridIncompressibleTransportModels \
    -lhybridPhaseChangeTwoPhaseMixtures \
    -lbubbleDynamics \
    -lODE
```

Index