ROYAL INSTITUTE
OF TECHNOLOGY

Master of Science Thesis

Development and validation of a two-phase CFD model using OpenFOAM

Alberto Ghione

Supervisors:
Henryk Anglart
Kristian Angele
Nicolas Edh

Division of Nuclear Reactor Technology
Royal Institute of Technology
Stockholm, Sweden, December 2012

TRITA-FYS 2012:61 ISSN 0280-316X ISRN KTH/FYS/–12:61–SE

ABSTRACT

This master thesis deals with the development and validation of an existing two-phase CFD code designed by Kai Fu [1] in OpenFoam.

The code is used to predict the radial and axial distribution of two-phase bubbly flow parameters, such as the void fraction, the bubble diameter, the gas and liquid velocities.

In this work a two-fluid model is used and both phases are modeled with Eulerian conservation equations (*Euler-Euler model*) with the Reynolds-averaged treatment of the turbulent stress in the momentum equation. Six conservation equations for the liquid and the gas phase are solved and closure laws are introduced for the modeling of the interfacial momentum transfer term in the momentum equations. A standard $k - \epsilon$ turbulence model consisting of two transport equations for the turbulent kinetic energy and the turbulent dissipation is used.

The code is able to deal with subcooled boiling, however the present study focuses on a comprehensive revision and development of the adiabatic part of the solver.

A detailed description of the mathematical models used, and of their implementation in OpenFoam, is provided including comparisons with other two-phase solvers and a discussion on possible future improvements of the code.

New correlations and models for the interfacial momentum transfer term in the momentum equation were implemented. The implementation of the Ishii-Zuber drag model for fluid particles improved significantly gas velocity prediction, since it is taking into account the effect of the deformation of the gas bubbles on the drag coefficient.

Many different sensitivity tests have been performed in order to eliminate numerical instabilities, highlight the deficiencies of the solver and to determine the effects of different models and parameters on the results.

The Rhie-Chow like treatment of the interfacial momentum transfer term in the momentum equation proved to cancel out radial oscillations in the void fraction and velocity profile when a coarse mesh is used ($y^+ > 20$).

The validation of the solver has been done against a set of experimental data of vertical upward water-air adiabatic flow in a pipe with a diameter of 25.4 mm. Good agreement with experimental data has been obtained and the results has been proved to be grid independent.

PREFACE AND ACKNOWLEDGEMENTS

The thesis work has been done at the Nuclear Reactor Technology department of the Royal Institute of Technology (Kungliga Tekniska Högskolan, KTH) in Stockholm with the supervision of Professor Henryk Anglart.

Supervision and support was also provided by Vattenfall AB energy company thanks to the help of Kristian Angele (senior R & D engineer at Vattenfall Research and Development AB) and Nicolas Edh (thermal-hydraulic engineer at Forsmarks Kraftgrupp AB) within the framework of the DKC-TS. DKC-TS (Distribuerat Kompetenscentra - Termohydraulik och Strömning) is Vattenfall's initiative to exchange knowledge and experiences between nuclear units within the area of thermal hydraulics and fluid mechanics.

I would like to thank everyone who helped me to write this master thesis and especially my supervisors (Henryk Anglart, Kristian Angele and Nicolas Forsberg) for their constant support and guide throughout the development of the code. I would like to thank them also for the respectful relationship that we have had and for the fruitful discussions during our frequent meeting.

Special thanks go also to Kai Fu, who taught me how to use the two-phase solver in OpenFoam in the first weeks of my work and to the whole Nuclear Reactor Technology department for their availability and suggestions.

I have also to thank Professor Cristina Bertani and Professor Mario Malandrone from the Energy department at Politecnico di Torino for their availability and supervision of my work.

I would like to thank many other people who shared with me this five-years long experience at the university, but for the sake of brevity, special thanks go to my good friends: Luca, Alberto, Mattia and Giuseppe.

Last but not least, I would like to thank my family who has been supporting me during my whole life. I am very indebted to you, thank you!

CONTENTS

1 Introduction **1**

1.1 CFD methodology for two-phase flows 3

1.1.1 OpenFoam CFD software 4

1.2 Objectives of the work 6

2 Mathematical model and implementation **7**

2.1 The solution procedure and the main program 8

2.2 The mass conservation equation 10

2.3 The momentum conservation equation 12

2.4 The pressure equation and the PISO algorithm 16

2.5 The interfacial momentum transfer closure laws 21

2.5.1 Drag force 23

2.5.2 Lift force 25

2.5.3 Virtual mass force 27

2.5.4 Wall lubrication force 27

2.5.5 Turbulent dispersion force 29

2.6 Interfacial area concentration transport equation 30

2.6.1 The Hibiki-Ishii breakup and coalescence source terms 31

2.7 Turbulence modeling 32

2.7.1 The turbulence of the liquid phase 32

2.7.2 The turbulence of the vapor phase 36

2.7.3 Wall functions 37

3 Description of the test cases **39**

4 Results and discussion **47**
4.1 Checker-board instabilities . . . 47
4.1.1 Influence of the plotting setting . . . 49
4.1.2 Influence of numerical schemes . . . 49
4.1.3 Influence of the compressibility terms . . . 50
4.1.4 Effect of the IAC equation . . . 50
4.1.5 Influence of turbulence models . . . 50
4.1.6 Influence of near-wall damping . . . 50
4.1.7 Influence of the pipe diameter . . . 51
4.1.8 Influence of the interfacial momentum transfer term treatment . . . 51
4.1.9 Influence of the virtual mass force . . . 55
4.1.10 Mesh influence . . . 56
4.2 Sensitivity tests of the turbulence modeling . . . 58
4.3 Sensitivity tests of the interfacial momentum transfer closure laws . . . 60
4.3.1 Turbulent dispersion force . . . 60
4.3.2 Drag force . . . 61
4.3.3 Lift and wall lubrication force . . . 64
4.4 Sensitivity tests on the influence of the IAC equation . . . 67
4.5 Test cases with higher Reynolds number . . . 69
4.6 Sensitivity tests on the mesh refinement . . . 71
4.7 Error estimates . . . 73

5 Conclusions **79**

Appendix A The implemented main program **87**

Appendix B The implemented continuity equation **91**

Appendix C The implemented momentum equation **95**

Appendix D The implemented pressure equation **99**

Appendix E The implemented Interfacial Area Concentration equation **105**

Appendix F The implemented $k-\epsilon$ turbulence models **107**

Appendix G The implemented interfacial momentum transfer coefficients **115**

Appendix H The implemented drag models **119**

Appendix I The implemented lift models **127**

Appendix J The implemented virtual mass models **131**

Appendix K The implemented wall lubrication models **133**

Appendix L The implemented Turbulence Dispersion force models **137**

Appendix M The Hibiki-Ishii breakup and coalescence source term **141**

Appendix N Numerical schemes **145**

NOMENCLATURE

Symbol	Dimensions	Description
		Latin Symbols
C_d	-	Drag coefficient
C_l	-	Lift coefficient
C_t	-	Turbulence response coefficient
c_p	$J/kg/K$	Specific heat capacity
C_w	-	Wall lubrication coefficient
d_h	m	Wellek horizontal bubble diameter
D_{pipe}	m	Pipe diameter
D_S	m	Bubble Sauter mean diameter
Eo	-	Eötvös number $Eo = \frac{(\rho_b - \rho_a) g D_S^2}{\sigma}$
$\mathbf{g}$	m/s^2	Gravity vector
$\mathbf{I}$	-	Identity tensor
IAC	m^{-1}	Interfacial area concentration
k	m^2/s^2	Turbulent kinetic energy
L/D	-	Length to pipe diameter ratio
$\mathbf{M}$	$N/m^3 = kg/m^2/s^2$	Interfacial forces per unit volume
$\mathbf{n}_w$	-	Wall normal vector
N''	m^{-2}	Active nucleation site density
p	Pa	Pressure
Pr	-	Prandtl number $Pr = \frac{c_p \mu}{\lambda}$
$\mathbf{R}$	$N/m^2 = kg/m/s^2$	Stress tensor
Re_b	-	Bubble Reynolds number $Re_b = \frac{\lvert \mathbf{U}_a - \mathbf{U}_b \rvert D_S}{\nu_b}$
t	s	Time
T	K or $°C$	Temperature
$\mathbf{U}$	m/s	Velocity
y	m	Near wall distance
y^+	m	Dimensionless distance from the wall

Symbol	Dimensions	Description
	Greek Symbols	
α	-	Phase volume fraction
Γ_{vl}	$kg/m^3/s$	Evaporation rate per unit volume
Γ_{lv}	$kg/m^3/s$	Condensation rate per unit volume
ϵ	m^2/s^3	Turbulent energy dissipation
κ	-	Von Karman constant $\kappa = 0.42$
λ	$W/m/K = kg \cdot m/s^3/K$	Thermal conductivity
μ	$Pa \cdot s = kg/m/s$	Dynamic viscosity
μ^*	-	Dimensionless dynamic viscosity
ν	m^2/s	Kinematic viscosity
ρ	kg/m^3	Density
σ	kg/s^2	Surface tension
τ	$N/m^2 = kg/m/s^2$	Shear stress
Φ	m^{-3}	Source/sink in the IAC equation
$\bar{\psi}$	s^2/m^2	Compressibility coefficient

	Subscripts
$_a$	Dispersed phase
$_b$	Continuous phase
$_{BB}$	Bubble Breakup
$_{BC}$	Bubble Coalescence
$_l$	Liquid phase
$_m$	Mixture
$_{max}$	Maximum
$_{NUC}$	Nucleation
$_{sat}$	Saturation
$_v$	Vapor phase
$_w$	Wall
	Superscripts
^{bulk}	Bulk
^{d}	Drag
^{eff}	Effective
^{l}	Lift
^{nw}	Near wall
^{t}	Turbulent
^{T}	Transpose
^{td}	Turbulent dispersion
^{vm}	Virtual mass
^{wl}	Wall lubrication
$^\perp$	Perpendicular to the wall

LIST OF FIGURES

1.1 Flow patterns and heat transfer regimes in a vertical pipe with upward flow, taken from [2]. . . . 2
1.2 Overall OpenFOAM structure [3]. . . . 5

2.1 Schematic representation of the implemented solution procedure for each time-step. 8
2.2 Influence of the mean Sauter bubble diameter on the Tomiyama lift coefficient C_l. 26
2.3 Influence of the mean Sauter bubble diameter on the wall lubrication force coefficient C_{wl}. . . . 29

3.1 Taitel et al. flow pattern map with red points representing the experimental conditions. . . . 41
3.2 Example of mesh geometry in the radial direction of the pipe. . . . 42
3.3 Example of mesh geometry of the pipe (the scales are not respected for presentation purposes). . . . 42
3.4 Not-uniform mesh in the radial direction. . . . 43

4.1 Effect of a coarse mesh on the results. . . . 48
4.2 Influence of different approaches on the treatment of the momentum equation. . . . 49
4.3 Influence of Michta's treatment of the momentum equation with a fine mesh. . . . 52
4.4 Influence of different treatments of the interfacial momentum transfer term. . . . 53
4.5 Influence of different treatments of the interfacial momentum transfer term (the removal of the virtual mass force can produce unstable results). . . . 54
4.6 Mesh influence with the removal of the virtual mass force. . . . 55
4.7 Influence of mesh refinement with the mesh setting in Figure 3.4 and no Rhie-Chow like treatment of the interfacial momentum transfer term. . . . 56
4.8 Influence of the mesh setting in Figure 3.4 with different treatment of the interfacial momentum transfer term. . . . 57
4.9 Influence of different standard $k-\epsilon$ turbulence models ($j_l = 0.3$ m/s $j_g = 0.09$ m/s). 58

4.10 Influence of different standard $k-\epsilon$ turbulence models ($j_l = 0.64$ m/s $j_g = 0.09$ m/s). . . . 59
4.11 Influence of the turbulence response coefficient C_t ($j_l = 0.64$ m/s $j_g = 0.09$ m/s). . 59
4.12 Sensitivity study of different turbulent dispersion force models ($j_l = 0.64$ m/s $j_g = 0.09$ m/s). . . . 60
4.13 Sensitivity study of different turbulent dispersion force models ($j_l = 0.3$ m/s $j_g = 0.09$ m/s). . . . 61
4.14 Radial and axial component of the interfacial forces ($j_l = 0.3$ m/s $j_g = 0.09$ m/s). 61
4.15 Radial and axial component of the interfacial forces ($j_l = 0.64$ m/s $j_g = 0.09$ m/s). 62
4.16 Sensitivity study on different drag force models ($j_l = 0.64$ m/s $j_g = 0.09$ m/s, z/D = 62). . . . 62
4.17 Sensitivity study on different drag force models ($j_l = 0.64$ m/s $j_g = 0.09$ m/s, z/D = 112). . . . 63
4.18 Influence of different models for the wall lubrication and lift force (z/D = 62). . . 64
4.19 Influence of different models for the wall lubrication and lift force (z/D = 112). . . 65
4.20 Simulation results for $j_l = 0.3$ m/s $j_g = 0.09$ m/s, z/D = 112 with Tomiyama lift model. . . . 66
4.21 Influence of the Interfacial Area Concentration equation on the results. 68
4.22 Comparison of simulation results with experiments at different location (z/D = 62, 112) for $j_l = 1.1$ m/s $j_g = 0.16$ m/s. . . . 69
4.23 Comparison of simulation results with experiments at different location (z/D = 62, 112) for $j_l = 2.0$ m/s $j_g = 0.16$ m/s. . . . 70
4.24 Influence of the radial mesh refinement with Kai's $k-\epsilon$ model (z/D = 62). 71
4.25 Influence of the axial mesh refinement with Kai's $k-\epsilon$ model (z/D = 62). 72
4.26 Influence of the radial mesh refinement with Kai's $k-\epsilon$ model (z/D = 112). . . . 72
4.27 Convergence as a function of CVs: mean void fraction and gas velocity. 73
4.28 Error estimate of the void fraction and gas velocity ($j_l = 0.3$ m/s $j_g = 0.09$ m/s, z/D=62) 73
4.29 Error estimate of the void fraction and gas velocity ($j_l = 0.3$ m/s $j_g = 0.09$ m/s, z/D=112) 74
4.30 Error estimate of the void fraction and gas velocity ($j_l = 0.64$ m/s $j_g = 0.09$ m/s, z/D=62) 74
4.31 Error estimate of the void fraction and gas velocity ($j_l = 0.64$ m/s $j_g = 0.09$ m/s, z/D=112) 74
4.32 Error estimate on the void fraction and gas velocity ($j_l = 1.1$ m/s $j_g = 0.16$ m/s, z/D=62) 75
4.33 Error estimate on the void fraction and gas velocity ($j_l = 1.1$ m/s $j_g = 0.16$ m/s, z/D=112) 75
4.34 Error estimate on the void fraction and gas velocity ($j_l = 2.0$ m/s $j_g = 0.16$ m/s, z/D=62) 75

4.35 Error estimate on the void fraction and gas velocity ($j_l = 2.0$ m/s $j_g = 0.16$ m/s, z/D=112) 76

LIST OF TABLES

3.1 Transport properties of water and air in Leung experiments [4] 40
3.2 Boundary conditions used in the simulations. 43
3.3 Constant inlet boundary conditions used in some simulations 45

4.1 Estimate of the standard deviations and the relative errors. 76

CHAPTER 1

INTRODUCTION

In the context of increasing energy demand, fossil fuel shortage and global warming, nuclear power can represent part of the solution of the energy and environmental problem. The major advantages of nuclear power are the absence of greenhouse gas emissions into the atmosphere, the low electricity production cost per kWh and the high energy density of the nuclear fuel that allows large production of electricity in relatively small power plants. Furthermore nuclear power is a very mature and reliable technology (over 14500 cumulative reactor-years of commercial nuclear power operation in 32 countries [5]) especially compared with renewable energies that lack in reliability of supply and need further development in order to be used on larger scales. Of course, there are still problems to solve regarding the nuclear waste management, the public acceptance of nuclear power and the safety of the plant during accidents, as proved by the recent Fukushima accident. Therefore the interest in nuclear energy research and development is still strong and huge efforts to improve the safety, the reliability of nuclear power plants and reduce the risks are still ongoing nowadays.

In this context, the master thesis deals with the development and validation of a two-phase and subcooled boiling CFD code in OpenFoam. The subcooled boiling consists in the evaporation of liquid at the heated walls while the bulk of the liquid is still subcooled (i.e. before the liquid bulk reaches saturation conditions). The interest in a better understanding of subcooled boiling in the nuclear industry lies in the fact that the water in a Light Water Reactor (LWR) enters the core subcooled (i.e. below the saturation temperature) and the first stage in the boiling process is then subcooled boiling, as shown in Figure 1.1. Therefore the determination of the parameters in subcooled boiling (such as void fraction, bubble diameter, liquid and vapor velocity) with computational codes is important in the perspective of improved coupled neutronic thermo-hydraulic analysis and to determine the distribution and deposition of impurities along the axis of the core. The development of codes for adiabatic and diabatic two-phase flows are of interest also for many other industrial applications in the chemical, power and aerospace industry in order to safely and optimally design processes involving two-phase flows.

As shown in Figure 1.1, the heat transfer and boiling process in a vertical pipe (or analogously in a nuclear reactor sub-channel) can be divided in different regimes. When the liquid is subcooled, the heat transfer is governed by single-phase forced convection but, at a certain point along the pipe, subcooled boiling will start with evaporation of liquid in micro-cavities (or nucleation sites) at the heated walls. The generated bubbles will grow and then detach when a critical size is reached. However, since the bulk of the liquid is still subcooled, the bubbles will condensate,

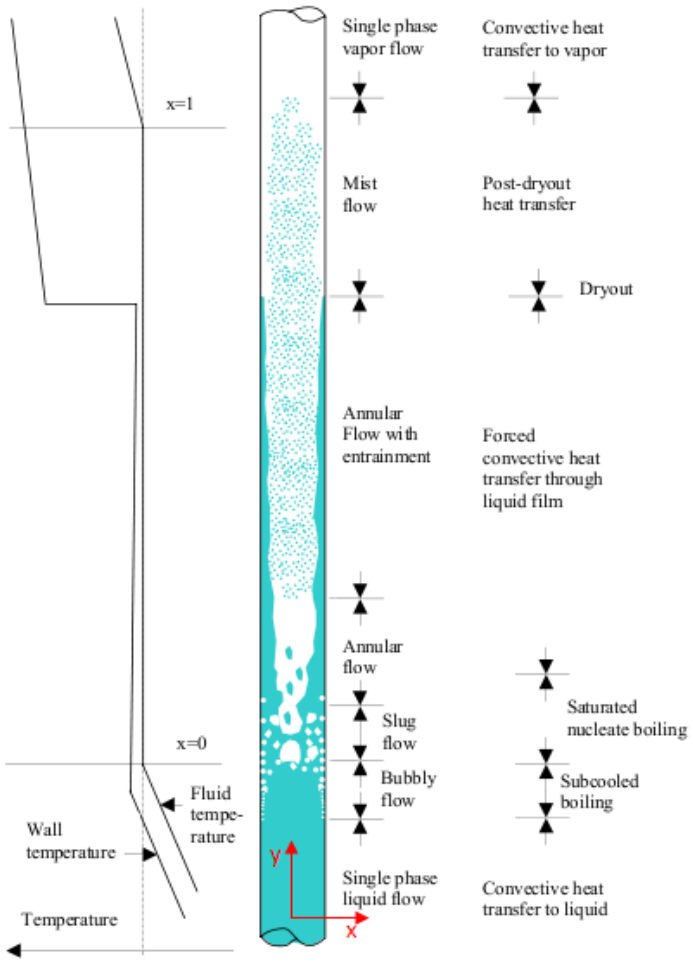

Figure 1.1: Flow patterns and heat transfer regimes in a vertical pipe with upward flow, taken from [2].

heating up the liquid phase and leading to a high heat transfer coefficient. When the bulk of the liquid reaches the saturation temperature, the saturated nucleate boiling regime starts, during which many other different flow patterns can be observed (slug, churn and anular flow). When the liquid film thickness in the anular flow goes to zero due to evaporation, the so-called *Dry-Out* occurs with a high increase in wall temperature due to the deterioration of the heat transfer coefficient due to the direct contact between the vapor and the heated walls. A different and more dangerous thermal crisis typology, the so-called *Departure from Nucleate Boiling (DNB)*, can occur with large heat fluxes or low mass flow rates so that the heat flux at the wall is larger than the *Critical Heat Flux (CHF)*. During DNB the boiling mechanism changes from nucleate boiling (very high heat transfer coefficient) to film boiling where a vapor layer prevents liquid from reaching the heated walls leading to a sudden deterioration of the heat transfer coefficient which can eventually cause fuel rods damage and, in worst case scenario, core melt-down. DNB can occur both in Pressurized Water Reactors (PWR) and in the lower part of Boiling Water Reactors (BWR) during accidents and therefore a clear understanding of two-phase flows and of the boiling mechanism is of huge interest in the nuclear industry. In fact, a correct modeling of subcooled boiling can be considered as a first step towards better CHF predictions in nuclear reactors during accidents. Furthermore, it can improve the prediction of impurities deposition along the axis of the core that can cause the so-called *Axial Offset Anomaly (AOA)* in PWR cores and improve the coupled neutronic thermo-hydraulic calculations since the radial and axial

distribution of void fraction influences the reactivity of the core providing non-negligible reactivity feedbacks. The AOA in PWR refers to deviations of the measured neutron flux in the top half of the core from the predicted values due to the enhanced corrosion products and boron deposition on the cladding surfaces caused by subcooled boiling [6]. The solver described in the thesis is designed for dealing with subcooled boiling and adiabatic bubbly flows. However, on the present thesis the focus will be on adiabatic bubbly flow in order to obtain a reliable code which can be trusted and further developed with the introduction of subcooled boiling.

1.1 CFD methodology for two-phase flows

In order to predict the parameters in two-phase flows, a Computational Fluid Dynamics (CFD) methodology is used in this study. The CFD methodology allows to obtain numerical approximate solutions of fluid flows through *discretization procedures* that approximate the set of coupled differential equations describing the flow by a set of algebraic equations that can be easily solved by a computer.

The CFD methodology became more and more important with the increase of computational computer power and now it represents an important engineering tool that allows to avoid the usage of experimental studies and empirical correlations, substituting them with more generally applicable and cheaper way of solving engineering problems. However CFD models need to be validate against experimental data before their usage and this will be the focus on the present thesis.

CFD, as any other numerical methodology, consists of a model (i.e. a mathematical representation of a physical phenomena neglecting less important features) and a solution procedure in order to obtain an approximate numerical solution from the model.

The modeling in CFD starts from the Navier-Stokes equations for fluid flows which are valid for every flow regime (e.g. laminar or turbulent) but very difficult to solve numerically especially for high Reynolds number flows which are of interest in this study. Therefore, simplifications of the equations and modeling assumptions are required to reduce the complexity of the mathematical model and the computational cost of the simulations. However, in some applications, the Navier-Stokes equations are also solved without any manipulation in so-called Direct Numerical Simulation (DNS). That requires very high resolution in order to resolve all the temporal and spatial scales and a huge computational effort. Therefore the use of DNS is limited to the simulations of low Reynolds number flows and, regarding two-phase flows, to the simulations of one (or few) Dispersed Phase Element (DPE).

Since turbulent flows are of interest in this study, the mathematical model uses conservation equations with mean properties of the flow obtained from a suitable averaging procedure of the the microscopic Navier-Stokes equations as described more in detail in [7] and [8]. The averaging procedure simplifies the complexity of the set of equations (greatly reducing the required computational effort) but introduces additional terms in the averaged equations that needs closure laws (e.g. the Reynolds stress in the momentum conservation equation appearing also in single-phase flow equations). While the Navier-Stokes equations are well established, the closure laws especially for two-phase flows are not generally applicable, needs further development and are still an object of debate among different scientists. Therefore validation of such models against experimental data is still necessary.

In this work a two-fluid model is used and both phases (continuous and dispersed phase) are modeled with Eulerian conservation equations (*Euler-Euler model*).

Therefore each phase is treated as a continuum with the use of averaged Navier-Stokes equations with the introduction of the volume phase fractions in the equations. As mentioned above, the averaging procedure introduces new terms in the momentum equations such as the Reynolds stress which requires a two-phase turbulence model. The interfacial momentum transfer term which models the transfer of momentum between the phases also needs to be improved since is not well established yet. These closure laws are the weak points of the two-fluid model approach and therefore will be the focus of attention in this master thesis.

1.1.1 OpenFoam CFD software

OpenFOAM ® (OF) is a free and open-source CFD software package produced by OpenCFD Ltd [9]. The development was initiated at the Imperial College in London in the 1980's and the first commercial version was released in 2004.

Since 2004, OpenFoam has been further developed and the number of users has increased significantly in the last few years due to three main advantages:

- *free of charge*;
- *open-source*, i.e. the code can easily be modified in order to improve existing solvers and peer reviewed;
- *object-oriented* through which the users can introduce new models and solvers (user-selectable) without changing the main code independently from the discretization scheme used, providing great flexibility and simplicity of use.

The programming language is C++ and the software runs on Linux systems. OpenFoam uses Finite Volume Methodology in order to discretize and solve complex fluid dynamics problems. First of all, the 3-D volume of interest is created and divided into small volumes or cells, obtaining the so-called *mesh*. Then the initial and boundary conditions necessary for solving the conservation equations are defined and applied to the geometry. Then OpenFoam discretizes the modeled equations using the previously built mesh. There are three main discretization approach used in CFD (more details can be found in [10]):

- *the Finite Difference Method* which uses conservation equations in differential form discretized on a mesh, resulting in one algebraic equation for each grid node;
- *the Finite Volume Method* which uses the integral form of the conservation equations, dividing the domain into small control volumes (CVs) and applying the conservation equations to each CVs. Volume and surface integrals are approximated with adequate quadrature formulas considering the center of the CV as the computational node and obtaining values at the CV faces through different interpolation schemes using the nodal values. This procedure results in the definition of one algebraic equation for each CV and leads to a conservative method by construction (i.e. the obtained solution satisfies the conservation equations if evaluated in the whole domain). This is the commonly used method in CFD codes.
- *the Finite Element Method* which is similar to the Finite Volume Method but weight functions are used before integrating the equations.

OpenFoam uses the Finite Volume Method over a *Colocated grid arrangement*. The colocated arrangement stores all dependent variables at the cell center and the same CVs are used for all variables, so that the computational effort is minimized. A different approach is used in staggered arrangement where different variables can be defined on different grids. The collocated arrangement provides advantages compared with other grid arrangement (e.g. staggered) and therefore most CFD codes have adopted this arrangement. For example, the difficulties linked to the pressure-velocity coupling and the consequent *checker-board instability* (i.e. oscillations) in the pressure fields were solved through the Rhie and Chow cure [8]. The main advantages are a minimization of the computational effort since all variables are stored using the same CV and an effective treatment of complex domains, especially with discontinuous boundary conditions (more details in [8, page 79]).

The structure of OpenFoam is presented in Figure 1.2. As any other CFD software it consists of a *pre-processing* tool where the user can define the mesh, the initial and boundary conditions and the fluid properties; a *solver* where the set of equations is specified and discretized and the *post-processing* tool (external to OpenFoam) used to visualize and plot the results (in the master thesis, the ParaView ® plotting tool is used). The *sample* utility is used to obtain the raw set of results in the region of interest, so that they can be easily plotted and compared with experiments.

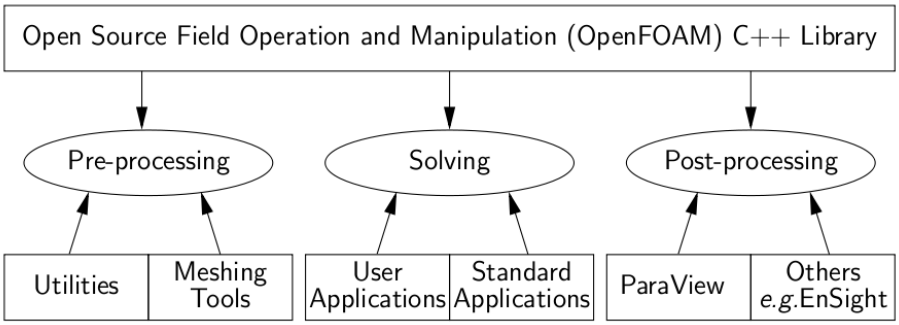

Figure 1.2: Overall OpenFOAM structure [3].

Another strength of OpenFoam is the simplicity of adding governing equations to the solvers, since the equations' syntax in the code is very similar to the mathematical one. For example, the equation with the generic unknown x:

$$\frac{\partial x}{\partial t} + \nabla \cdot (x\mathbf{U}) - \nabla \cdot (\nu \nabla x) = 0 \tag{1.1}$$

is implemented in the code as:

```
solve
(
    fvm::ddt(x)
  + fvm::div(U, x)
  - fvm::div(nu, div(x))
);
```

Then the main code along with all the solvers and utilities (i.e. applications dedicated to data manipulations) should be compiled through terminal with the command *./Allwmake*. Once the code is compiled, the user can run simulations of his cases of interest specifying the main features of the flow and geometry in the folder *case*.

The *case* folder contains all the information about the geometry, physical parameters and flow condition that can be set from the user but also information on the computational schemes and

the time-step used in the simulation. It contains different sub-folders:

- The *system* folder contains information about the computational schemes, the time-step, the duration of the simulation and the plotting settings (in the *sampleDict* file). The numerical discretization schemes (e.g. upwind, linear, QUICK, etc.) are set in the file *fvSchemes* and the linear algebraic solvers definition, relaxation and tolerances in the file *fvSolution*. The time-step, the duration of the simulation and the results' writing interval are set in the *controlDict* file.
- The *0* folder is used to set up the boundary and initial conditions.
- The *constant* folder contains informations about the mesh (in the sub-folder *polyMesh*),the physical fluid properties and the definition of the solvers and models used in the solution procedure.
  - The file *transportProperties* specifies the physical properties of the two phases which are to be assumed constant in the solution procedure, such as the viscosity, density, etc.;
  - The file *g* defines the gravity field *g*;
  - The file *interfacialProperties* allows one to choose and set up the models for the interfacial forces in the inter-phase momentum transfer term;
  - The file *turbulenceProperties* defines the turbulence model used and its parameters;
  - The file *moduleControls* specifies the models used during the simulation;
  - The files *wallBoilingProperties*, *phaseChangeProperties* and *ppProperties* defines subcooled boiling parameters and wall properties such as the wall heat flux.

More details on OpenFoam can be found in the OF User Guide [3] and the OF Programmers Guide [11].

In the present study, two versions of OpenFoam were utilized: OpenFOAM 1.7.1 and OpenFOAM 2.0.1 which gave similar results and behaves in the same way.

1.2 Objectives of the work

The aim of this master thesis is to further develop and validate an existing two-phase solver for subcooled nucleate boiling designed by Kai Fu [1] in OpenFoam.

In particular the present study focuses on a comprehensive revision and on the development of the adiabatic part of the solver introducing new correlations and models in the code.

A detailed description of the mathematical models used and of the implementation in the code is provided together with considerations, comparisons with previous solvers and discussions on other possible approaches and future improvements.

The solver was validated against a set of experimental data of vertical upward two-phase flows in cylindrical pipes and sensitivity tests were performed in order to highlight the deficiencies of the solver.

CHAPTER 2

MATHEMATICAL MODEL AND IMPLEMENTATION

This chapter deals with a detailed description of the models used in the adiabatic part of the solver and how they are actually implemented. The solver is meant for compressible, two-phase flow with the possibility of dealing with subcooled boiling and heat transfer.

The implemented code solves a set of coupled differential equations (the Navier-Stokes equations) with an iterative *segregated approach*. In the segregated approach each equation is solved sequentially as if decoupled from the other equations and iterations are performed in order to obtain good approximations of the results.

An Euler-Euler model is used, which means that each phase is treated as a continuum and represented by averaged conservation equations.

This approach is different from DNS (Direct Numerical Simulation) where the Navier-Stokes equations are solved without any further manipulation, leading to computationally expensive problems due to the very high resolution required for solving all spatial and temporal scales.

The averaging process introduces new variables and terms and therefore closure laws are required. In the momentum equations, the Reynolds stresses (analogous to single-phase flow) and the averaged interfacial momentum transfer appear and are modeled in the code. The various terms in the implemented equations are discretized mainly implicitly but some particular terms are also discretized explicitly or semi-implicitly for numerical reasons deduced from [7].

The properties of the liquid and vapor phase are considered constant everywhere and they are set by the user. This approximation can be acceptable if the variations of the properties are small in the considered temperature and pressure range, that is the case of sub-cooled boiling and adiabatic flows in vertical pipes discussed in this study. This hypothesis is not valid if large variations in thermo-physical properties are expected.

In this document the dispersed phase is indicated with the subscript a or v indicating the vapor phase (useful notation when dealing with boiling) while the continuous phase is indicated with b or l.

2.1 The solution procedure and the main program

The solution procedure was implemented according mainly to [8] and [7]. The main program code used can be found in Appendix A.

Before starting the iteration loop the Courant number $Co = \frac{U\Delta t}{\Delta L}$ is calculated, where Δt is the discretization time-step, ΔL is the mesh size and U the different fluid velocities. The Courant number should be less than 1 to avoid numerical instabilities during transients when an explicit discretization method is used, but small Courant number are required also with implicit scheme in order to accurately resolve transient details [12]. Therefore a fine mesh requires an adequately small time-step in order to accurately solve the transient. No adaptive scheme for the time-step or the mesh size are used, conversely they are set by the user depending on the performed simulation.

The solution procedure includes two iterative corrector loops for each time-step of the transient calculation. The default number of iterations of the outer corrector loop is equal to five, but it can be modified by the user in the file *fvSolution*.

$$\text{Outer corrector loop} \begin{cases} heatTransfer.H \\ alphaEqn.H \\ IACEqn.H \\ kEpsilon.H \\ HEqn.H \\ liftDragCoeffs.H \\ UEqn.H \\ \text{PISO loop} \begin{cases} pEqn.H \\ alphaEqn.H \\ IACEqn.H \end{cases} \end{cases}$$

Figure 2.1: Schematic representation of the implemented solution procedure for each time-step.

The nested PISO (Pressure-Implicit with Splitting of Operators) loop (default number of iterations equal to two) solves the pressure equation iteratively. This solution algorithm is used to handle the pressure-velocity coupling with a momentum predictor and a pressure correction loop. The PISO loop is used to obtain a velocity and pressure field satisfying both the momentum and continuity equation, therefore the pressure equation is derived combining the momentum and the continuity equation in a Poisson-like equation for the pressure (more detail are presented in Section 2.4).

First the velocity flux fields are computed from the momentum equations without satisfying the continuity equation, then this fluxes are corrected with the new computed pressure calculated from the pressure equation that satisfies the continuity equation. At this stage, the obtained velocity fields do not satisfy the momentum equations. Iterations are therefore required in order to obtain a solution that satisfies both the continuity and the momentum equations [10].

The solver solves the following sequence of equations:

- `"heatTransfer.H"` contains closure laws to apply in case of boiling (for example, the calculation of the evaporation and condensation terms Γ_{vl} and Γ_{lv} necessary to close the continuity equation);
- `"alphaEqn.H"` contains the continuity equations for both phases;

- `"IACEqn.H"` contains the Interfacial Area Concentration equation used to calculate the mean Sauter diameter of the bubble;

- `"kEpsilon_std_Kai.H"`, `"kEpsilon_std_Ghione.H"`, `"kEpsilon_std_KaiEff.H"` and `"kEpsilon_YaoMorel.H"` contain four different implementations of the standard k-ϵ models derived with different hypothesis but starting from the formulations in [8] and [13] . These models consist of two transport equations for calculating the turbulent kinetic energy k and the turbulent dissipation ϵ which are used to calculate the Reynolds stress tensor in the momentum equation;

- `"HEqns.H"` contains one energy conservation equation for the liquid phase, conversely the enthalpy of the vapor phase is supposed at saturation condition (i.e. no energy conservation equation for the vapor phase [1]). This equation is switched off when adiabatic cases are considered;

- `"liftDragCoeffs.H"` calculates the forces acting on the bubbles for the interfacial momentum transfer term in the momentum conservation equation;

- `"UEqns_Michta.H"`, `"UEqns_Rusche.H"` and `"UEqns_Standard.H"` contain the discretized momentum equations for both phases implemented with different hypothesis that will be discussed in the following chapters. The user can select which model will be used in the simulation in the input file: *moduleControls*. The momentum equations are not solved at this stage but the matrices of coefficients are set up and they will be used to obtain an approximation of the velocity field needed to solve the following pressure equation;

- Associated with the different momentum equations implemented, a corresponding PISO loop is defined. Once the PISO loop is started, the pressure equation is solved and the velocity fluxes corrected accordingly. Also the continuity and the IAC equations are updated and solved except for the last cycle of the PISO loop in order to avoid to repeat the solution of these two equations twice because the same equations are also solved at the beginning of the outer loop;

- `"DpDt.H"` and `"DDtU.H"` calculate the total derivatives of the pressure and velocity field respectively ;

The solution procedure used in the present solver is slightly different from the one implemented in OpenFoam by Michta [14]. In particular the sequence of solved equations is different, even if the structure of the code remains intact. The main difference consists in the fact that the k-ϵ model is solved at the end of the solution procedure as well as the enthalpy equation in agreement with [7] and [8], while in our code both of them are solved before the momentum equation and the PISO loop. This discrepancy should not affect the results and at most it could affect the convergence performances of the code but no studies on this possibility were done or found in the literature. This approach can be justified observing that the sequence of equations is the same as the one used in the *compressibleTwoPhaseEulerFoam* solver which is one of the default solvers for multi-phase flows in OpenFoam 2.1.1 [2]. This default solver can be regarded as a simplified version of our code where most of the interfacial forces are not implemented or simplified, the IAC equation and subcooled boiling are not considered.

[1] The enthalpy equation for the vapor phase is implemented in the solver but switched off.

[2] OpenFoam 2.1.1 is the newly released OF version on 31st May 2012.

2.2 The mass conservation equation

The mass conservation equation for both phases k can be written as:

$$\frac{\partial(\alpha_k \rho_k)}{\partial t} + \nabla \cdot (\alpha_k \rho_k \mathbf{U}_k) = \Gamma_k \tag{2.1}$$

where Γ_k stands for the mass gained by phase k ($k = a, b$) per unit volume and time.

This equation for compressible flow is manipulated before being implemented in the current solver, according to Weller [7, page 22]. The manipulation turned out to be necessary because the direct discretization of the previous equation has proved to be very unstable in two-phase flows with large density ratios between the two phases and furthermore this manipulation guarantees the boundedness of the phase fraction [7].

Eqn. (2.1) is re-written as:

$$\rho_k \frac{\partial(\alpha_k)}{\partial t} + \alpha_k \frac{\partial(\rho_k)}{\partial t} + \rho_k \nabla \cdot (\alpha_k \mathbf{U}_k) + \alpha_k \mathbf{U}_k \cdot \nabla(\rho_k) = \Gamma_k \tag{2.2}$$

dividing by ρ_k and re-arranging using the definition of total derivative $\frac{\mathrm{D}(\rho_k)}{\mathrm{d}t} = \frac{\partial(\rho_k)}{\partial t} + \mathbf{U}_k \cdot \nabla(\rho_k)$:

$$\frac{\partial(\alpha_k)}{\partial t} + \nabla \cdot (\alpha_k \mathbf{U}_k) + \frac{\alpha_k}{\rho_k} \frac{\mathrm{D}(\rho_\mathrm{k})}{\mathrm{dt}} = \frac{\Gamma_k}{\rho_k} \tag{2.3}$$

Introducing $\mathbf{U}_r = \mathbf{U}_a - \mathbf{U}_b$ and $\mathbf{U} = \alpha_a \mathbf{U}_a + \alpha_b \mathbf{U}_b$, the following expression is obtained:

$$\mathbf{U}_a = \mathbf{U} + \alpha_b \mathbf{U}_r \tag{2.4}$$

Substituting Eqn. (2.4) in Eqn. (2.3) gives:

$$\frac{\partial(\alpha_a)}{\partial t} + \nabla \cdot (\alpha_a \mathbf{U}) + \nabla \cdot (\alpha_a \alpha_b \mathbf{U}_r) + \frac{\alpha_a}{\rho_a} \frac{\mathrm{D}(\rho_\mathrm{a})}{\mathrm{dt}} = \frac{\Gamma_a}{\rho_a} \tag{2.5}$$

$$\frac{\partial(\alpha_b)}{\partial t} + \nabla \cdot (\alpha_b \mathbf{U}) - \nabla \cdot (\alpha_a \alpha_b \mathbf{U}_r) + \frac{\alpha_b}{\rho_b} \frac{\mathrm{D}(\rho_\mathrm{b})}{\mathrm{dt}} = -\frac{\Gamma_a}{\rho_b} \tag{2.6}$$

where Γ_a stands for the mass gained by the vapor phase.

In order to guarantee the boundedness of the void fraction [7], Eqn. (2.5) and (2.6) are summed:

$$\nabla \cdot \mathbf{U} = -\frac{\alpha_a}{\rho_a} \frac{\mathrm{D}(\rho_\mathrm{a})}{\mathrm{dt}} - \frac{\alpha_b}{\rho_b} \frac{\mathrm{D}(\rho_\mathrm{b})}{\mathrm{dt}} + \frac{\Gamma_a}{\rho_a} - \frac{\Gamma_a}{\rho_b} \tag{2.7}$$

where $\alpha_a + \alpha_b = 1$.

Eqn. (2.7) looks very similar to the incompressibility constraint $\nabla \cdot \mathbf{U} = 0$ for incompressible flows, where the r.h.s. takes into account the compressibility of both phases and the phase change.

Substituting Eqn. (2.7) in Eqn. (2.5) and (2.6) and rearranging using $\nabla \cdot (\alpha_k \mathbf{U}) = \alpha_k \nabla \cdot (\mathbf{U}) + \mathbf{U} \cdot \nabla \alpha_k$ [7]:

$$\frac{\partial(\alpha_a)}{\partial t} + \mathbf{U} \cdot \nabla \alpha_a + \nabla \cdot (\alpha_a \alpha_b \mathbf{U}_r) = \alpha_a \alpha_b \left(\frac{1}{\rho_b} \frac{\mathrm{D}(\rho_\mathrm{b})}{\mathrm{dt}} - \frac{1}{\rho_a} \frac{\mathrm{D}(\rho_\mathrm{a})}{\mathrm{dt}} \right) + \alpha_a \left(\frac{\Gamma_a}{\rho_b} - \frac{\Gamma_a}{\rho_a} \right) + \frac{\Gamma_a}{\rho_a} \tag{2.8}$$

$$\frac{\partial(\alpha_b)}{\partial t} + \mathbf{U}\cdot\nabla\alpha_b - \nabla\cdot(\alpha_a\alpha_b\mathbf{U}_r) = \alpha_a\alpha_b\left(\frac{1}{\rho_a}\frac{\mathrm{D}(\rho_\mathrm{a})}{\mathrm{dt}} - \frac{1}{\rho_b}\frac{\mathrm{D}(\rho_\mathrm{b})}{\mathrm{dt}}\right) + \alpha_b\left(\frac{\Gamma_a}{\rho_b} - \frac{\Gamma_a}{\rho_a}\right) - \frac{\Gamma_a}{\rho_b} \quad (2.9)$$

Finally the following equations are implemented in the solver:

$$\begin{aligned}\frac{\partial(\alpha_a)}{\partial t} + \nabla\cdot(\alpha_a\mathbf{U}) - \alpha_a\nabla\cdot(\mathbf{U}) \;&+\; \nabla\cdot(\alpha_a\alpha_b\mathbf{U}_r) = \alpha_a\alpha_b\left(\frac{1}{\rho_b}\frac{\mathrm{D}(\rho_\mathrm{b})}{\mathrm{dt}} - \frac{1}{\rho_a}\frac{\mathrm{D}(\rho_\mathrm{a})}{\mathrm{dt}}\right) + \frac{\Gamma_{ab}}{\rho_a} \\ &- \frac{\Gamma_{ba}}{\rho_a} - \alpha_a\left(\frac{\Gamma_{ab}}{\rho_a} - \frac{\Gamma_{ab}}{\rho_b} + \frac{\Gamma_{ba}}{\rho_b} - \frac{\Gamma_{ba}}{\rho_a}\right)\end{aligned} \quad (2.10)$$

$$\begin{aligned}\frac{\partial(\alpha_b)}{\partial t} + \nabla\cdot(\alpha_b\mathbf{U}) - \alpha_b\nabla\cdot(\mathbf{U}) \;&-\; \nabla\cdot(\alpha_a\alpha_b\mathbf{U}_r) = \alpha_a\alpha_b\left(\frac{1}{\rho_a}\frac{\mathrm{D}(\rho_\mathrm{a})}{\mathrm{dt}} - \frac{1}{\rho_b}\frac{\mathrm{D}(\rho_\mathrm{b})}{\mathrm{dt}}\right) - \frac{\Gamma_{ab}}{\rho_b} \\ &+ \frac{\Gamma_{ba}}{\rho_b} - \alpha_b\left(\frac{\Gamma_{ab}}{\rho_a} - \frac{\Gamma_{ab}}{\rho_b} + \frac{\Gamma_{ba}}{\rho_b} - \frac{\Gamma_{ba}}{\rho_a}\right)\end{aligned} \quad (2.11)$$

where the following notation (consistent with [14]) is used:

- Γ_{ba} = condensation rate (i.e. mass transferred from vapor to liquid phase);
- Γ_{ab} = evaporation rate (i.e. mass transferred from liquid to vapor phase);
- $\Gamma_a = \Gamma_{ab} - \Gamma_{ba}$ (i.e. $\Gamma_a > 0$ in case of evaporation and $\Gamma_a = 0$ for adiabatic cases)

Although only one of the previous phase fraction equations need to be solved, Weller [7] suggested to solve both equations together with a recombination scheme in order to maintain the boundedness of the void fraction and accelerate the convergence of the iterative procedure. This is done in the code, solving an alpha correction loop (default number of iteration equal to one) where both phase conservation equations are solved and the results recombined according to the following recombination scheme:

$$\alpha_a = \frac{1}{2}\left(1 - (1-\alpha_a)^2 + (1-\alpha_b)^2\right) \quad (2.12)$$

The obtained value of the void fraction in the alpha correction loop is used as initial condition when solving the void fraction equation for the dispersed phase α_a, as shown in Appendix B. Then the phase fraction α_b is calculated as $\alpha_b = 1 - \alpha_a$.

In the code, the quantities phia, phib, phi and phir are defined and indicate the face volumetric fluxes (i.e. the interpolation of the velocity at the cell face) required by the discretization arising from the Finite Volume approach to the problem:

$$\phi_k = \mathbf{S}\cdot\mathbf{U}_{kf} \quad (2.13)$$

where $\mathbf{S}$ stands for the face area vector and $\mathbf{U}_{kf}$ for the velocity interpolated on the cell face (usually it is defined at the cell center).

These quantities are defined in OpenFoam as:

```
phia == (fvc::interpolate(Ua) & mesh.Sf());
phib == (fvc::interpolate(Ub) & mesh.Sf());
phi = alphaf*phia + betaf*phib;
surfaceScalarField phir("phir", phia - phib);
```

Finally the implemented discretized equation for the void fraction reads:

$$\begin{aligned}\frac{\partial(\alpha_a)}{\partial t} + \nabla\cdot(\alpha_a\phi) - \alpha_a\nabla\cdot(\phi) \quad &+ \quad \nabla\cdot(\alpha_a\alpha_b\phi_r) = \alpha_a\alpha_b\left(\frac{1}{\rho_b}\frac{\mathrm{D}(\rho_\mathrm{b})}{\mathrm{dt}} - \frac{1}{\rho_a}\frac{\mathrm{D}(\rho_\mathrm{a})}{\mathrm{dt}}\right) + \frac{\Gamma_{ab}}{\rho_a} \\ &- \quad \frac{\Gamma_{ba}}{\rho_a} - \alpha_a\left(\frac{\Gamma_{ab}}{\rho_a} - \frac{\Gamma_{ab}}{\rho_b} + \frac{\Gamma_{ba}}{\rho_b} - \frac{\Gamma_{ba}}{\rho_a}\right)\end{aligned} \tag{2.14}$$

where all terms are treated implicitly in time because the "fvm::" discretization option is used, while the discretization schemes in space are user defined. The standard discretization schemes in space used in this thesis are the *Gauss upwind* for the divergence terms $\nabla\cdot(\alpha_a\phi)$ and $\nabla\cdot(\alpha_a\alpha_b\phi_r)$ and the *Gauss linear* for the divergence term $\nabla\cdot(\phi)$ with a linear interpolation scheme. More details on the discretization schemes and the equation discretization in OpenFoam can be found in the OF Programmers Guide[11, page 33].

The code uses an implementation similar to the one used by Michta [14], but Michta considered both phases incompressible obtaining a much simpler formulation:

$$\frac{\partial(\alpha_a)}{\partial t} + \nabla\cdot(\alpha_a\mathbf{U}) + \nabla\cdot(\alpha_a\alpha_b\mathbf{U}_r) = \frac{\Gamma_{ab}}{\rho_a} - \frac{\Gamma_{ba}}{\rho_a} \tag{2.15}$$

Furthermore Michta did not implement the alpha correction loop, solving iteratively only the void fraction equation.

2.3 The momentum conservation equation

The momentum conservation equations are written for both phases. According to [1] and [14] the momentum conservation equation can be written as:

$$\begin{aligned}\frac{\partial(\alpha_k\rho_k\mathbf{U}_k)}{\partial t} + \nabla\cdot(\alpha_k\rho_k\mathbf{U}_k\mathbf{U}_k) \quad &= \quad -\alpha_k\nabla p + \nabla\cdot\left[\alpha_k(\tau_k + \tau_k^\mathrm{t})\right] \\ &+ \quad \alpha_k\rho_k\mathbf{g} + \Gamma_{ki}\mathbf{U}_i - \Gamma_{ki}\mathbf{U}_k + \mathbf{M}_{ki}\end{aligned} \tag{2.16}$$

where the subscripts k and i indicate the two phases (i.e. the dispersed and continuous phase or vice versa), $\tau_k + \tau_k^\mathrm{t}$ are the combined Reynolds viscous and turbulent stress and $\mathbf{M}_{ki}$ is the averaged interfacial momentum transfer term, which needs accurate modeling.

In order to model the Reynolds stress, the Boussinesq hypothesis for turbulent stress-strain relation is used. This hypothesis is valid for a Newtonian fluid and reads:

$$\tau_k^\mathrm{eff} = \tau_k + \tau_k^\mathrm{t} = \rho_k\nu_k^\mathrm{eff}\left(\nabla\mathbf{U}_k + (\nabla\mathbf{U}_k)^T - \frac{2}{3}\mathbf{I}\nabla\cdot\mathbf{U}_k\right) - \frac{2}{3}\mathbf{I}\rho_k k_k \tag{2.17}$$

where the identity tensor is identified with $\mathbf{I}$, the turbulent kinetic energy is k_k, the effective kinematic viscosity is $\nu_k^{\text{eff}} = \nu_k + \nu_k^{\text{t}}$, ν_k^{t} is the turbulent kinematic viscosity and ν_k is the physical kinematic viscosity of phase k.

The momentum conservation equation is implemented differently from the previously shown formulation, following the suggestions in [8, page 108-111] and [7]. The momentum equation is implemented in a *phase-intensive* version, i.e. the equation is divided by the phase fraction α_k. Rusche [8] stated that this modification prevents the momentum equation from becoming singular when the phase fraction approaches zero but the main reason is probably that this formulation allows to implement the l.h.s. of the equation as if it is formulated for a mono-phase flow, simplifying the implementation of the model. Dividing also for the density ρ_k, it is obtained:

$$\frac{\partial(\mathbf{U}_k)}{\partial t} + \nabla \cdot (\mathbf{U}_k \mathbf{U}_k) = -\frac{\nabla p}{\rho_k} + \frac{\nabla \cdot \left[\tau_k^{\text{eff}}\right]}{\rho_k} + \mathbf{g} + \frac{\Gamma_{ki}\mathbf{U}_i}{\alpha_k \rho_k} - \frac{\Gamma_{ki}\mathbf{U}_k}{\alpha_k \rho_k} + \frac{\mathbf{M}_{ki}}{\alpha_k \rho_k} \tag{2.18}$$

According to Rusche [8], the momentum equation can be re-written as:

$$\frac{\partial(\mathbf{U}_k)}{\partial t} + \mathbf{U}_k \cdot \nabla \mathbf{U}_k + \nabla \cdot R_k^{\text{eff}} + \frac{\nabla \alpha_k}{\alpha_k} \cdot R_k^{\text{eff}} = -\frac{\nabla p}{\rho_k} + \mathbf{g} + \frac{\Gamma_{ki}\mathbf{U}_i}{\alpha_k \rho_k} - \frac{\Gamma_{ki}\mathbf{U}_k}{\alpha_k \rho_k} + \frac{\mathbf{M}_{ki}}{\alpha_k \rho_k} \tag{2.19}$$

where R_k^{eff} can be decomposed for numerical reasons in a diffusive component and a correction [8]:

$$R_k^{\text{eff}} = -\frac{\tau_k^{\text{eff}}}{\rho_k} = R_k^{\text{eff,D}} + R_k^{\text{eff,corr}}$$

$$R_k^{\text{eff,D}} = -\nu_k^{\text{eff}} \nabla \mathbf{U}_k$$

$$R_k^{\text{eff,corr}} = R_k^{\text{eff}} + \nu_k^{\text{eff}} \nabla \mathbf{U}_k = -\nu_k^{\text{eff}} \left[(\nabla \mathbf{U}_k)^T - \frac{2}{3}\mathbf{I} \nabla \cdot \mathbf{U}_k \right] + \frac{2}{3}\mathbf{I} k_k \tag{2.20}$$

It can be defined a total phase velocity as:

$$\mathbf{U}_k^{\tau} = \mathbf{U}_k - \nu_k^{\text{eff}} \frac{\nabla \alpha_k}{\alpha_k} \tag{2.21}$$

Therefore substituting Eqn. (2.20) and (2.21) in Eqn. (2.19):

$$\begin{aligned} \frac{\partial(\mathbf{U}_k)}{\partial t} + \mathbf{U}_k^{\tau} \cdot \nabla \mathbf{U}_k \;&-\; \nabla \cdot \left(\nu_k^{\text{eff}} \nabla \mathbf{U}_k\right) + \nabla \cdot R_k^{\text{eff,corr}} + \frac{\nabla \alpha_k}{\alpha_k} \cdot R_k^{\text{eff,corr}} \\ &= -\frac{\nabla p}{\rho_k} + \mathbf{g} + \frac{\Gamma_{ki}\mathbf{U}_i}{\alpha_k \rho_k} - \frac{\Gamma_{ki}\mathbf{U}_k}{\alpha_k \rho_k} + \frac{\mathbf{M}_{ki}}{\alpha_k \rho_k} \end{aligned} \tag{2.22}$$

Substituting the definition of total phase velocity (Eqn. (2.21)) in the equation Eqn. (2.22):

$$\begin{aligned} \frac{\partial(\mathbf{U}_k)}{\partial t} + \mathbf{U}_k \cdot \nabla \mathbf{U}_k \;&-\; \nu_k^{\text{eff}} \frac{\nabla \alpha_k}{\alpha_k} \cdot \nabla \mathbf{U}_k - \nabla \cdot \left(\nu_k^{\text{eff}} \nabla \mathbf{U}_k\right) + \nabla \cdot R_k^{\text{eff,corr}} + \frac{\nabla \alpha_k}{\alpha_k} \cdot R_k^{\text{eff,corr}} \\ &= -\frac{\nabla p}{\rho_k} + \mathbf{g} + \frac{\Gamma_{ki}\mathbf{U}_i}{\alpha_k \rho_k} - \frac{\Gamma_{ki}\mathbf{U}_k}{\alpha_k \rho_k} + \frac{\mathbf{M}_{ki}}{\alpha_k \rho_k} \end{aligned} \tag{2.23}$$

The discretization of the momentum equation Eqn. (2.23) in finite volume gives:

$$\begin{aligned}\frac{\partial(\mathbf{U}_k)}{\partial t} &+ \nabla\cdot(\phi_k\mathbf{U}) - \mathbf{U}\cdot\nabla\phi_k + \nabla\cdot(\phi_k^R\mathbf{U}) - \mathbf{U}\cdot\nabla\phi_k^R \\ &- \nabla\cdot\left(\nu_k^{\text{eff}}\nabla\mathbf{U}_k\right) + \nabla\cdot R_k^{\text{eff,corr}} + \frac{\nabla\alpha_k\rho_k}{(\alpha_k+\delta)\rho_k}\cdot R_k^{\text{eff,corr}} = \\ &- \frac{\nabla p}{\rho_k} + \mathbf{g} + \frac{\Gamma_{ki}\mathbf{U}_i}{(\alpha_k+\delta)\rho_k} - \frac{\Gamma_{ki}\mathbf{U}_k}{(\alpha_k+\delta)\rho_k} + \frac{\mathbf{M}_{ki}}{(\alpha_k+\delta)\rho_k}\end{aligned} \tag{2.24}$$

where the formula $\nabla\cdot(\phi_k^\tau\mathbf{U}) = \phi_k^\tau\nabla\cdot(\mathbf{U}) + \mathbf{U}\cdot\nabla\phi_k^\tau$ is used and the total phase flux is defined as:

$$\phi_k^\tau = \phi_k + \phi_k^R = \phi_k - \nu_{k_f}^{\text{eff}} S\frac{\nabla_f^\perp\alpha_k}{\alpha_{k_f}+\delta} \tag{2.25}$$

The factor $\delta = 10^{-12}$ is added in the denominator in order to avoid problems with the division by α_k when the void fraction approaches zero. This unphysical term is added to all terms in the r.h.s. of the momentum equation. Regarding the division by α_k for the other terms in the l.h.s., it was demonstrated by Weller that this will not lead to divergent terms because the ratio of $\nabla\alpha_k$ and α_k approaches zero as the phase fraction vanishes [8].

The phase-intensive version of the momentum equation has the big disadvantage of the division by the void fraction that can approach zero, therefore this kind of implementation is starting to be abandoned in favor of a more safe formulation where this division is avoided. This statement can be confirmed, observing how the momentum equation is implemented in the *compressibleTwoPhaseEulerFoam* solver in OpenFoam 2.1.1, where the phase intensive formulation of the momentum equation is abandoned. The conversion of this code into a compatible form with OpenFoam 2.1.1 will be done during a later phase of the development of the code that will start in December 2012.

Furthermore the use of the turbulent response C_t in the code is an attempt to take into consideration the turbulence of the vapor phase (see more detail in Section 2.7.2).

According to [8] and [7], almost all terms are treated implicitly in time, while the Reynolds stress correction containing the term $\nabla\alpha_k$ and the buoyancy term are treated explicitly. The discretization of the interfacial momentum transfer term requires special treatment and it will be discussed in Chapter 2.5.

Again the discretization schemes in space are user defined, but the standard ones used in this thesis are the *Gauss upwind* for the divergence term $\nabla\cdot(\phi_k\mathbf{U})$, the *Gauss linear* for all other divergence terms and the gradient terms, the *Gauss linear corrected* for the term $\nabla\cdot\left(\nu_k^{\text{eff}}\nabla\mathbf{U}_k\right)$ that is written in OpenFoam as:

fvm::laplacian(**nuEffa**,**Ua**,"laplacian(nuEffa,Ua)")

even it is not solving for a laplacian, indeed this is how such a term should be written in OpenFoam as specified in [11, page 33]. More details on the used discretization schemes in [7] and [11, page 33].

The equations (2.23) and (2.24) are not used exactly in this form, but some of the terms in the r.h.s. are moved to the pressure equation and not discretized at this stage. In order to avoid pressure-velocity decoupling and pressure oscillations (checker-boarding), the gravity and pressure gradient terms in the momentum equation are moved to the pressure equation, as shown in the implemented code in Appendix C.

This is necessary because a colocated grid arrangement is used [10], and therefore the cure proposed by Rhie and Chow for the treatment of the pressure gradient [8] was used.

In this thesis, four different approaches (user-selectable in the file *moduleControls*) were considered in the treatment of the pressure-velocity coupling:

1. The first approach was used by Fu [1] and Weller [7] and only the gravity and pressure gradient terms in the momentum equation are moved to the pressure equation. Therefore the finally implemented momentum equation reads:

$$\begin{aligned} \frac{\partial(\mathbf{U}_k)}{\partial t} &+ \nabla\cdot(\phi_k\mathbf{U}) - \mathbf{U}\cdot\nabla\phi_k + \nabla\cdot(\phi_k^R\mathbf{U}) - \mathbf{U}\cdot\nabla\phi_k^R \\ &- \nabla\cdot\left(\nu_k^{\text{eff}}\nabla\mathbf{U}_k\right) + \nabla\cdot R_k^{\text{eff,corr}} + \frac{\nabla\alpha_k\rho_k}{(\alpha_k+\delta)\rho_k}\cdot R_k^{\text{eff,corr}} \\ &= \frac{\Gamma_{ki}\mathbf{U}_i}{(\alpha_k+\delta)\rho_k} - \frac{\Gamma_{ki}\mathbf{U}_k}{(\alpha_k+\delta)\rho_k} + \frac{\mathbf{M}_{ki}}{(\alpha_k+\delta)\rho_k} \end{aligned} \tag{2.26}$$

   This is considered as the reference model in the thesis.

2. The approach used by Michta in [14] is also implemented and it can be selected switching on the option *MomentumEqn_Michta* in the file *moduleControls*. In this approach also the explicit drag term (see more details on the drag force in Section 2.5.1) is moved to the pressure equation. The movement of the explicit drag term was suggested in [8, page 125] and such approach was justified referring to a previous paper written by Weller in 2002 [3] and explaining that the movement of the explicit drag and of the gravity term was done *"so that it mimics the operation of a solution procedure devised for a staggered variable arrangement, but keeping the colocated variables"*. However, few differences compared to Michta's code were introduced. First of all, Michta dealt with non-compressible phases and conversely this code is designed for compressible phases. Furthermore the virtual mass force is not neglected, but this gives a small contribution to the steady state fully developed flow condition. The virtual mass force is more important during the transient evolution (as also tested with few simulations). However the addition of the virtual mass force seemed to be beneficial for the stability of the code, as discussed in Chapter 4.1.

   The results obtained with this configuration of the equations gives the same results as the previous approach and no visible improvement in the stability performances of the code was observed.

3. The approach suggested by Rusche in [8] was implemented and it can be selected switching on the option *MomentumEqn_Rusche* in the file *moduleControls*. Here both the explicit drag term and the turbulent dispersion force term are moved to the pressure equation. The movement of the turbulent dispersion force term to the pressure equation is justified by Rusche in [8, page 118]. The turbulent dispersion force is proportional to the gradient of the dispersed phase void fraction $\nabla\alpha_a$, as explained in detail in Section 2.5.5 and since the gradient of the void fraction has a diffusive effect on the phase fraction distribution, it can lead to momentum and continuity equation decoupling and consequently instabilities (checker-boarding). Checker-boarding in the radial direction was actually observed during simulations with coarse mesh, but this behavior usually disappears with the usage of a finer mesh in the radial direction of the pipe. However it was tried to implement this Rhie-Chow-like cure for the turbulent dispersion force suggested by Rusche [8] in order to see

[3] This paper is a previous version written in 2002 and with the same title of [7] that was published in 2005.

the improvement in terms of stability performance. The complete reconstruction procedure performed after solving the pressure equation is explained in detail in Section 2.4.

4. A Rhie-Chow-like treatment for the lift force in the interfacial momentum transfer term was also implemented. It can be selected switching on the option *MomentumEqn_RhieChowLift* in the file *moduleControls*. Here both the explicit drag, the turbulent dispersion and the lift force are moved to the pressure equation. Indeed the non-linearity of the interfacial momentum transfer term requires special treatment in order to avoid instabilities. Therefore the implementation of the lift force through Rhie-Chow was done and results compared in Section 4.1.8. The complete reconstruction procedure performed after solving the pressure equation is explained in detail in Section 2.4.

As previously stated, the momentum equation is constructed outside the PISO loop but not solved because it was found out that this can lead to instabilities [8]. The approximate solution of the momentum equation without the gravity and pressure term is performed within the PISO loop and gives an approximation of the velocity field that does not obey the continuity equation. The velocity field is corrected in the PISO loop using an updated pressure field that satisfies the continuity equation. Then the iterative procedure is started in order to obtain a better approximation of the velocity field that satisfies both the momentum and continuity equation.

The derivation of the pressure equation will be performed in Section 2.4, however few considerations on the solving procedure for the momentum equation will be also useful for a better understanding of that derivation.

An approximation of the velocity field is obtained using the Jacobi iteration scheme as stated by [1]. The Jacobi iteration scheme for solving linear set of coupled equations states that if $\mathbf{A}x = \mathbf{R}$, where $\mathbf{A}$ is the matrix of the coefficients and $\mathbf{R}$ the source vector, then the approximate solution to this system can be found iterating:

$$x \approx \mathbf{A}_D^{-1}\mathbf{A}_H \tag{2.27}$$

where $\mathbf{A}_D$ is the matrix with only the diagonal components of $\mathbf{A}$ (it is very easy to compute the inverse of a diagonal matrix), and $\mathbf{A}_H$ is defined as:

$$\mathbf{A}_H = \mathbf{R} - \mathbf{A}_N x \tag{2.28}$$

where $\mathbf{A}_N$ refers to the matrix with only the off-diagonal components [8].

The Jacobi solution procedure can be summarized as:

1. initial guess x_0 taken from the previous time-step;
2. calculation of $\mathbf{A}_H = \mathbf{R} - \mathbf{A}_N x_0$;
3. evaluation of the approximate solution $x_j \approx \mathbf{A}_D^{-1}\mathbf{A}_H$;
4. repetition of step 2 and 3 until $x_j - x_{j-1} <= tollerance$.

2.4 The pressure equation and the PISO algorithm

The pressure equation deals with the pressure-velocity coupling and corrects the fluxes and velocities with an updated pressure field so that continuity is obeyed.

As already said, different approaches for dealing with the pressure-velocity coupling are implemented. The procedure for the reference standard model used in this thesis (implemented in the file `"pEqn_Standard.H"`, as shown in Appendix D) and in [1] and [7] will be outlined and the differences introduced by the other models will be discussed afterwards.

First of all a velocity prediction $\mathbf{U}_k^*$ and consequently a flux prediction ϕ_k^* are obtained before entering the PISO loop from the Jacobi iteration scheme:

$$\mathbf{U}_k^* = \mathbf{A}_{D,k}^{-1}\mathbf{A}_{H,k}$$

$$\phi_k^* = \mathbf{S}\cdot\mathbf{U}_{kf}^* = \mathbf{S}\cdot(\mathbf{A}_{D,k}^{-1}\mathbf{A}_{H,k})_f \tag{2.29}$$

where $\mathbf{A}$ is the system of linear algebraic equations generated from the discretization of the phase momentum equation without the pressure gradient and the gravity term.

Then in the solver, the flux is corrected for including the gravity term:

$$\phi_k^{**} = \phi_k^* + (\mathbf{g}\cdot\mathbf{S})\ (\mathbf{A}_{D,k}^{-1})_f \tag{2.30}$$

The mixture flux is defined as:

$$\phi = \alpha\phi_a + \beta\phi_b \tag{2.31}$$

and analogously the mixture velocity is:

$$\mathbf{U} = \alpha\mathbf{U}_a + \beta\mathbf{U}_b \tag{2.32}$$

Now, the solver constructs and solves the pressure equation. The pressure equation is obtained from the momentum and the continuity equation, trying to define a unique pressure field even if two phases are present. In order to do so, the mixture flux in Eqn. (2.31) is used.

According to Weller [7] the continuity constraint in Eqn. (2.7) is used in terms of the mixture flux:

$$\nabla\cdot\phi = -\frac{\alpha_a}{\rho_a}\frac{\mathrm{D}(\rho_\mathrm{a})}{\mathrm{Dt}} - \frac{\alpha_b}{\rho_b}\frac{\mathrm{D}(\rho_\mathrm{b})}{\mathrm{dt}} + \frac{\Gamma_a}{\rho_a} - \frac{\Gamma_a}{\rho_b} \tag{2.33}$$

The total flux including the pressure gradient correction is written as:

$$\phi_k = \phi_k^{**} - \left(\frac{1}{\bar{\rho}_k\mathbf{A}_{D,k}}\right)_f S_f\nabla_f^\perp\bar{p} \tag{2.34}$$

So substituting Eqn. (2.34) and (2.31) in (2.33):

$$\begin{aligned}\nabla\cdot\phi^{**} \quad - \quad & \nabla\cdot\left(\left(\alpha_{af}\left(\frac{\mathbf{A}_{D,a}{}^{-1}}{\rho_a}\right)_f + \alpha_{bf}\left(\frac{\mathbf{A}_{D,b}{}^{-1}}{\rho_b}\right)_f\right)\nabla p\right) \\ = \quad & -\frac{\alpha_a}{\rho_a}\frac{\mathrm{D}(\rho_\mathrm{a})}{\mathrm{dt}} - \frac{\alpha_b}{\rho_b}\frac{\mathrm{D}(\rho_\mathrm{b})}{\mathrm{dt}} + \frac{\Gamma_a}{\rho_a} - \frac{\Gamma_a}{\rho_b}\end{aligned} \tag{2.35}$$

where $\phi^{**} = \alpha\phi_a^{**} + \beta\phi_b^{**}$.

In the solver the pressure equation is constructed using the flux prediction corrected with gravity ϕ_k^{**}, while the calculation of the total derivatives of the densities in the compressibility terms are performed using the fluxes evaluated at the previous iteration (i.e. before the flux prediction previously described), according to Weller [7].

Furthermore the density is substituted with the following constitutive law that linearly correlate the pressure and the density:

$$\rho_k = \bar{\psi}_k p \tag{2.36}$$

where $\bar{\psi}_k$ stands for the compressibility constant (in the simulations, the liquid compressibility $\bar{\psi}_b$ is equal to zero, i.e. incompressible liquid phase).

Therefore Weller [7] derived the following pressure equation:

$$\begin{aligned}
&\left[\nabla \cdot \left(\left(\alpha_{af}\left(\frac{\mathbf{A}_{D,a}{}^{-1}}{\rho_a}\right)_f + \alpha_{bf}\left(\frac{\mathbf{A}_{D,b}{}^{-1}}{\rho_b}\right)_f\right)\nabla p\right)\right] \\
=&\nabla \cdot \phi_k^{**} + \frac{\alpha_a}{\rho_a}\left(\left[\frac{\partial[\bar{\psi}_a]}{\partial t}\right]p + \nabla \cdot \left(\phi_a^{prevIter}\rho_{af}\right) - \rho_a \nabla \cdot \phi_a^{prevIter}\right) \\
&+ \frac{\alpha_b}{\rho_b}\left(\left[\frac{\partial[\bar{\psi}_b]}{\partial t}\right]p + \nabla \cdot \left(\phi_b^{prevIter}\rho_{bf}\right) - \rho_b \nabla \cdot \phi_b^{prevIter}\right) - \Gamma_a\left(\frac{1}{\rho_a} - \frac{1}{\rho_b}\right)
\end{aligned} \tag{2.37}$$

The implemented pressure equation is slightly different from the one suggested by Weller and reads:

$$\begin{aligned}
&\left[\nabla \cdot \left(\left(\alpha_{af}\left(\frac{\mathbf{A}_{D,a}{}^{-1}}{\rho_a}\right)_f + \alpha_{bf}\left(\frac{\mathbf{A}_{D,b}{}^{-1}}{\rho_b}\right)_f\right)\nabla p\right)\right] \\
=&\nabla \cdot \phi_k^{**} + \frac{\alpha_a}{\rho_a}\left(\frac{\partial[\bar{\psi}_a p]}{\partial t} + \nabla \cdot \left(\phi_a^{prevIter}\rho_{af}\right) - \rho_a \nabla \cdot \phi_a^{prevIter}\right) \\
&+ \frac{\alpha_b}{\rho_b}\left(\frac{\partial[\bar{\psi}_b p]}{\partial t} + \nabla \cdot \left(\phi_b^{prevIter}\rho_{bf}\right) - \rho_b \nabla \cdot \phi_b^{prevIter}\right) - \Gamma_a\left(\frac{1}{\rho_a} - \frac{1}{\rho_b}\right)
\end{aligned} \tag{2.38}$$

where the pressure is taken inside the temporal derivative in the compressibility term. This modification is necessary because otherwise the time derivative would be always zero since a constant user-defined compressibility coefficient is assumed in our code.

Once the pressure equation is solved, the velocity fields (and also the fluxes) are corrected including the gravity and pressure gradient term which are reconstructed from the fluxes computed with the updated pressure field:

$$\mathbf{U}_k = \mathbf{U}_k^* + reconstruct\left((\mathbf{g}\cdot\mathbf{S})\ (\mathbf{A}_{D,k}^{-1})_f - \left(\frac{1}{\bar{\rho}_k \mathbf{A}_{D,k}}\right)_f S_f \nabla_f^{\perp}\bar{p}\right) \tag{2.39}$$

where the OpenFoam operator *reconstruct* produces an approximate velocity vector with a reconstruction procedure from the volumetric flow rate [15].

Once the corrected velocities are evaluated, an equation for recalculating the densities is recalled with `#include "rhoEqns.H`. The implemented density equation consists in the mass conservation equation in Eqn. (2.1) and reads:

$$\frac{\partial(\alpha_k \rho_k)}{\partial t} + \nabla \cdot (\alpha_{kf}\rho_k \phi_k) = \Gamma_k \tag{2.40}$$

where the only differences are the introduction of the flux ϕ_k and the use of the interpolated value of the phase fraction α_{kf} in the second term of the l.h.s. due to the discretization. Eqn. (2.40) is computed by keeping the previously evaluated phase fractions constant.

This is needed because compressible phases are considered. These equations for the evaluation of the new densities were not used in Michta's code because incompressible phases were assumed.

Finally the PISO loop can be summarized as a series of consecutive steps:

1. Predict the fluxes using Eqn. (2.29) and correct them with Eqn. (2.30);
2. Construct and solve the pressure equation in Eqn. (2.38);
3. Correct the fluxes with Eqn. (2.34) and the velocities with Eqn. (2.39) using the updated pressure field;
4. Update the phase densities, solving Eqn. (2.40).

Now, the differences between the basic model and the modified versions will be outlined.

The PISO loop in the file `"pEqn_Michta.H"` is slightly different because the fluxes and the velocities are corrected also with the contribution of the explicit drag term removed from the correspondent momentum equation:

$$\phi_k^{**} = \phi_k^{*} + (\mathbf{g}\cdot\mathbf{S})\ (\mathbf{A}_{D,k}^{-1})_f + \left(\frac{K(\mathbf{A}_{D,k}^{-1})}{\rho_k\alpha_k}\right)_f \phi_k \tag{2.41}$$

$$\begin{aligned}\mathbf{U}_k = \mathbf{U}_k^{*} \quad &+ \quad reconstruct\left((\mathbf{g}\cdot\mathbf{S})\ (\mathbf{A}_{D,k}^{-1})_f - \left(\frac{1}{\bar{\rho}_k\mathbf{A}_{D,k}}\right)_f S_f\nabla_f^{\perp}\bar{p}\right)\\ &+ \quad reconstruct\left(\left(\frac{K(\mathbf{A}_{D,k}^{-1})}{\rho_k\alpha_k}\right)_f \phi_k\right)\end{aligned} \tag{2.42}$$

where K is the drag term coefficient defined as the drag force divided by the relative velocity $\mathbf{U}_r$, i.e. $K = -\frac{3}{4}\frac{C_d}{D_S}\rho_b\alpha_a|\mathbf{U}_r|$, see more details on the drag force in Section 2.5.1.

Therefore the PISO loop with Michta's approach [14] can be summarized as:

1. Predict the fluxes using Eqn. (2.29) and correct them with Eqn. (2.41);
2. Construct and solve the pressure equation in Eqn. (2.38);
3. Correct the fluxes with Eqn. (2.34) and the velocities with Eqn. (2.42) using the updated pressure field;
4. Update the phase densities, solving Eqn. (2.40).

The PISO loop in the file `"pEqn_Rusche.H"` corrects the fluxes and the velocities with the contribution of both the explicit drag term and the turbulent dispersion force:

$$\phi_a^{**} = \phi_a^{*} + (\mathbf{g}\cdot\mathbf{S})\ (\mathbf{A}_{D,a}^{-1})_f + \left(\frac{K(\mathbf{A}_{D,a}^{-1})}{\rho_a\alpha_a}\right)_f \phi_a + \left(\frac{(\mathbf{A}_{D,a}^{-1})}{\rho_a}\right)_f \frac{K^{td}|\mathbf{S}|\nabla_f^{\perp}\alpha_a}{(\alpha_a)_f} \tag{2.43}$$

$$\phi_b^{**} = \phi_b^{*} + (\mathbf{g}\cdot\mathbf{S})\ (\mathbf{A}_{D,b}^{-1})_f + \left(\frac{K(\mathbf{A}_{D,b}^{-1})}{\rho_b\alpha_b}\right)_f \phi_b - \left(\frac{(\mathbf{A}_{D,b}^{-1})}{\rho_b}\right)_f \frac{K^{td}|\mathbf{S}|\nabla_f^{\perp}\alpha_a}{(\alpha_b)_f} \tag{2.44}$$

$$\mathbf{U}_k = \mathbf{U}_k^* + reconstruct\left(\phi_k^{**} - \left(\frac{1}{\bar{\rho}_k \mathbf{A}_{D,k}}\right)_f S_f \nabla_f^{\perp} \bar{p}\right) \tag{2.45}$$

where $K^{td} = \frac{\mathbf{M}_a^{td}}{\nabla \alpha_a}$ is the turbulent dispersion force coefficient and $\mathbf{M}_a^{td}$ is the turbulent dispersion force defined in Section 2.5.5.

Therefore the PISO loop with Rusche‘s approach [8] can be summarized as:

1. Predict the fluxes using Eqn. (2.29) and correct them with Eqns. (2.43) and (2.44);
2. Construct and solve the pressure equation in Eqn. (2.38);
3. Correct the fluxes with Eqn. (2.34) and the velocities with Eqn. (2.45) using the updated pressure field;
4. Update the phase densities, solving Eqn. (2.40).

The PISO loop in the file `"pEqn_RhieChowLift.H"` corrects the fluxes and the velocities with the contribution of both the explicit drag, the turbulent dispersion and the lift force:

$$\begin{aligned}\phi_a^{**} = \phi_a^* \;+\; & (\mathbf{g}\cdot\mathbf{S})\,(\mathbf{A}_{D,a}^{-1})_f + \left(\frac{K(\mathbf{A}_{D,a}^{-1})}{\rho_a \alpha_a}\right)_f \phi_a \\ & + \left(\frac{(\mathbf{A}_{D,a}^{-1})}{\rho_a}\right)_f \frac{K^{td}|\mathbf{S}|\nabla_f^{\perp}\alpha_a}{(\alpha_a)_f} - \left(\frac{(\mathbf{A}_{D,a}^{-1})}{\rho_a}\right)_f \left(\left(K^{Lift}\right)_f \cdot \mathbf{S}\right)\end{aligned} \tag{2.46}$$

$$\begin{aligned}\phi_b^{**} = \phi_b^* \;+\; & (\mathbf{g}\cdot\mathbf{S})\,(\mathbf{A}_{D,b}^{-1})_f + \left(\frac{K(\mathbf{A}_{D,b}^{-1})}{\rho_b \alpha_b}\right)_f \phi_b \\ & - \left(\frac{(\mathbf{A}_{D,b}^{-1})}{\rho_b}\right)_f \frac{K^{td}|\mathbf{S}|\nabla_f^{\perp}\alpha_a}{(\alpha_b)_f} + \left(\frac{(\mathbf{A}_{D,b}^{-1})}{\rho_b}\right)_f \frac{(\alpha_a)_f \left(\left(K^{Lift}\right)_f \cdot \mathbf{S}\right)}{(\alpha_b)_f}\end{aligned} \tag{2.47}$$

where $K^{Lift} = C_l \rho_b(\mathbf{U}_r) \times \nabla \times \mathbf{U}_b$ is the lift force coefficient as also defined in Section 2.5.2.

Therefore the PISO loop in the file `"pEqn_RhieChowLift.H"` can be summarized as:

1. Predict the fluxes using Eqn. (2.29) and correct them with Eqns. (2.46) and (2.47);
2. Construct and solve the pressure equation in Eqn. (2.38);
3. Correct the fluxes with Eqn. (2.34) and the velocities with Eqn. (2.45) using the updated pressure field;
4. Update the phase densities, solving Eqn. (2.40).

2.5 The interfacial momentum transfer closure laws

In this section the closure laws for the interfacial momentum transfer in the momentum equation are discussed. The interfacial momentum transfer is caused by the forces acting at the interface between the two phases (i.e. the forces acting on a bubble are caused by the liquid which surrounds it). Therefore according to the Newton‘s third law of motion:

$$\mathbf{M}_a + \mathbf{M}_b = 0 \tag{2.48}$$

The bubble is subjected to different kinds of forces that need specific modeling. The implemented forces in this code are the drag force, the lift force, the wall lubrication force, the turbulent dispersion force and the virtual mass force, which were summarized by Michta in [14] as:

$$\mathbf{M}_a = \mathbf{M}_a^d + \mathbf{M}_a^l + \mathbf{M}_a^{wl} + \mathbf{M}_a^{td} + \mathbf{M}_a^{vm} \tag{2.49}$$

Under fully developed flow conditions, the drag force determines the terminal velocity of the bubble through a balance with the buoyancy force, conversely the balance of the lateral forces acting on the bubble influences the void fraction profile and acts as follows:

- the wall force drives the bubbles away from the walls;
- the lift force pushes bubbles towards the wall or the centerline depending on the bubble size;
- the turbulent dispersion force evens out gradients in the void fraction distribution.

Weller [7] suggested the usage of the simple mixture model for the modeling of the drag, lift and virtual mass force:

$$\begin{aligned}
\mathbf{M}_a = \; & - \frac{3}{4}\alpha_a\alpha_b\left(f_a\frac{C_{da}\rho_b}{d_a} + f_b\frac{C_{db}\rho_a}{d_b}\right)|\mathbf{U}_a - \mathbf{U}_b|(\mathbf{U}_a - \mathbf{U}_b) \\
& - \alpha_a\alpha_b f_a\left(C_{la}\rho_b(\mathbf{U}_a - \mathbf{U}_b) \times (\nabla \times \mathbf{U}_b)\right) \\
& - \alpha_a\alpha_b f_b\left(C_{lb}\rho_a(\mathbf{U}_a - \mathbf{U}_b) \times (\nabla \times \mathbf{U}_a)\right) \\
& + \alpha_a\alpha_b(f_a C_{vm_a}\rho_b + f_b C_{vm_b}\rho_a)\left(\frac{D\mathbf{U}_b}{Dt} - \frac{D\mathbf{U}_a}{Dt}\right)
\end{aligned} \tag{2.50}$$

where C_{da}, C_{la} and C_{vm_a} are respectively the drag, lift and virtual mass coefficient for a bubble (i.e. gas phase), while C_{db}, C_{lb} and C_{vm_b} are respectively the drag, lift and virtual mass coefficient for a droplet (i.e. liquid phase). The modifier functions f_a and f_b are defined as functions of the phase fraction with $f_a + f_b = 1$ and $f_a \to 1$ for $\alpha_a \to 0$. Different models for this functions were proposed: Weller used $f_a = \alpha_b$ and $f_b = \alpha_a$ while more complicated models are explained in detail in [8, page 103].

This model represents a numerical trick useful to cope with phase separation and annular flow with entrained droplets (i.e. the liquid phase becomes the dispersed phase), however these kinds of situation are far away from our range of study (sub-cooled boiling) where the only possible dispersed phase is the gas phase.

Therefore this approach was abandoned and the following standard interfacial forces model designed for situation with gas phase a dispersed in the liquid phase b is used in the solver:

$$\begin{aligned}\mathbf{M}_a = & - \frac{3}{4}\alpha_a \frac{C_{da}\rho_b}{d_a}|\mathbf{U}_a - \mathbf{U}_b|(\mathbf{U}_a - \mathbf{U}_b) \\ & - \alpha_a \left(C_{la}\rho_b(\mathbf{U}_a - \mathbf{U}_b) \times (\nabla \times \mathbf{U}_b)\right) \\ & + \alpha_a C_{vm_a}\rho_b \left(\frac{D\mathbf{U}_b}{Dt} - \frac{D\mathbf{U}_a}{Dt}\right)\end{aligned} \tag{2.51}$$

Prior the start of the masters thesis, in the solver it was assumed that $f_a = 1$ and $f_b = 0$, leading to the following formulation of the interfacial forces:

$$\begin{aligned}\mathbf{M}_a = & - \frac{3}{4}\alpha_a\alpha_b \frac{C_{da}\rho_b}{d_a}|\mathbf{U}_a - \mathbf{U}_b|(\mathbf{U}_a - \mathbf{U}_b) \\ & - \alpha_a\alpha_b \left(C_{la}\rho_b(\mathbf{U}_a - \mathbf{U}_b) \times (\nabla \times \mathbf{U}_b)\right) \\ & + \alpha_a\alpha_b C_{vm_a}\rho_b \left(\frac{D\mathbf{U}_b}{Dt} - \frac{D\mathbf{U}_a}{Dt}\right)\end{aligned} \tag{2.52}$$

This kind of implementation was not consistent because the simple mixture model was modified with the hypothesis that the only possible dispersed phase was the gas phase. The usage of the mixture model was then useless and also counterproductive because it reduced the forces acting on the bubble of a factor equal to the liquid phase fraction α_b that during the case studies can reach values up to 0.75. Therefore in order to avoid this inconsistency and reduction in acting forces, the standard interfacial forces model of Eqn. (2.51) was implemented in the solver.

Finally the implemented interfacial momentum transfer term with the addition of the wall lubrication force and the turbulent dispersion force reads:

$$\begin{aligned}\mathbf{M}_a = & - \frac{3}{4}\alpha_a \frac{C_{da}\rho_b}{d_a}|\mathbf{U}_a - \mathbf{U}_b|(\mathbf{U}_a - \mathbf{U}_b) \\ & - \alpha_a \left(C_{la}\rho_b(\mathbf{U}_a - \mathbf{U}_b) \times (\nabla \times \mathbf{U}_b)\right) \\ & + \alpha_a C_{vm_a}\rho_b \left(\frac{D\mathbf{U}_b}{Dt} - \frac{D\mathbf{U}_a}{Dt}\right) \\ & + \mathbf{M}_a^{wl} \\ & + \mathbf{M}_a^{td}\end{aligned} \tag{2.53}$$

Now a short description of the discretization of the interfacial momentum transfer term is presented. The lift, the wall lubrication and the turbulent dispersion forces are treated explicitly because an implicit treatment is *extremely difficult* [8], even if this can cause instabilities. A semi-implicit treatment is used for the virtual mass and the drag term as suggested in [8] and [7] due to proved stability in both transient and steady state simulations. The semi-implicit treatment implies that one part of the term is treated implicitly and the other part explicitly. For example, in the drag term of the momentum equation for the gas phase the relative velocity is re-written in terms of the liquid and gas velocity and the term with the liquid velocity is treated explicitly while the term with the gas velocity is treated implicitly. Analogous treatment is used for the virtual mass force, while in the momentum equation for the liquid phase the treatment is reversed (i.e. the liquid velocity is treated implicitly and the gas velocity explicitly), as shown in Appendix C.

In the following subsections, the models for the different forces acting on the bubble are introduced and discussed.

2.5.1 Drag force

This force represents the resistance opposed to the bubble motion in the fluid (or, more generally, the resistance of the relative motion between two phases). The drag force clearly depends on the bubble's size (i.e. a larger bubble experiences a larger drag force) and on the relative velocity between the two phases $\mathbf{U}_r = \mathbf{U}_a - \mathbf{U}_b$:

$$\mathbf{M}_a^d = -\frac{3}{4}\frac{C_d}{D_S}\rho_b\alpha_a|\mathbf{U}_r|\mathbf{U}_r \tag{2.54}$$

where D_S is the mean Sauter bubble diameter (defined as the diameter of a sphere that has the same volume/surface ratio of the bubble) and C_d is the drag coefficient that needs careful modeling.

The drag coefficient is usually deduced from experiments and many models were developed in order to fit different experimental data-set. An interesting and complete literature study on the topic can be found in [8].

In the solver many different drag coefficient models are implemented:

- *The Schiller-Naumann model (1935)* [16]

$$C_d = \frac{24}{Re_b}(1 + 0.15Re_b^{0.687}) \tag{2.55}$$

  where the bubble Reynolds number is defined as: $Re_b = \frac{|\mathbf{U}_a - \mathbf{U}_b|D_S}{\nu_b}$. This formulation of the drag coefficient is valid only for solid spherical particles with $Re_b < 1000$. For $Re_b > 1000$ the drag coefficient for solid particles is approximately constant and equal to 0.44 [8], therefore the following model is implemented:

$$C_d = \max\left(\frac{24}{Re_b}(1 + 0.15Re_b^{0.687}), 0.44\right) \tag{2.56}$$

- *The Ishii-Zuber model (1979)* [17]

$$C_d = \max\left(\frac{24}{Re_{bm}}(1 + 0.15Re_{bm}^{0.687}), 0.44\right) \tag{2.57}$$

  where the Reynolds number is defined as:

$$\begin{aligned} Re_{bm} &= \frac{\rho_b|\mathbf{U}_a - \mathbf{U}_b|D_S}{\mu_m} \\ \mu_m &= \mu_b\left(1 - \frac{\alpha_a}{\alpha_{\max}}\right)^{-2.5\alpha_{\max}\mu^*} \\ \mu^* &= \frac{\mu_a + 0.4\mu_b}{\mu_a + \mu_b} \end{aligned} \tag{2.58}$$

  This model is valid only for solid particles in the undistorted regime [8]. $\alpha_{\max}$ is the maximum phase fraction and depends on the maximum packing value considered. It is user-defined and the standard value in this thesis is taken equal to 0.74048 equal to the maximum possible packing for solid spheres as used in [1], although Rusche [8] suggested a value of 0.62 for solid particles and 1.0 for fluid particles (if the Ishii-Zuber model valid for bubbles is used).

- *The Wen-Yu model (1966)* [18]

  The implemented model in the solver is:

$$C_d = \max\left(\frac{24}{Re_b}(1 + 0.15Re_b^{0.687}), 0.44\right)\alpha_b^{-2.65} \tag{2.59}$$

  This model is valid again only for solid particles. Different implementation of the same model can be found on the literature. In particular both in [8] and in the CFX manual [12] the improved model by Gidaspow is used with small discrepancies on the exponent of the liquid phase fraction α_b (-1.65 in [12] and -1.7 in [8]).

All the models described so far are valid only for solid spherical particles and their usage in the simulation of liquid-vapor flows introduces approximations that can lead to wrong gas velocity prediction especially at high Reynolds number where distorted bubble regime is observed. Therefore two models valid for deformable gas bubbles taken for CFX manual [12] were implemented. These models take into account that the bubble can be distorted and deformed (see [8, page 42-50] and [19, page 15-25]). Indeed the drag force depends strongly on the bubble‘s shape (sphere, ellipse or cap) and, for example, a cap-shaped bubble will experience a larger drag force compared with a spherical bubble.

- *The Ishii-Zuber model for densely distributed fluid particles* [12]

  This model distinguishes three bubble regime with different drag modeling:

  – The spherical bubble regime with a drag coefficient defined by the Ishii-Zuber formula for spherical solid particles:

$$C_d(sphere) = \frac{24}{Re_{bm}}(1 + 0.15Re_{bm}^{0.687}) \tag{2.60}$$

  – The ellipse distorted bubble regime:

$$C_d(ellipse) = E(\alpha_a)C_{d\infty} \tag{2.61}$$

  where:

$$C_{d\infty} = \frac{2}{3}Eo^{1/2}$$

$$E(\alpha_a) = \frac{1 + 17.67f(\alpha_a)^{6/7}}{18.67f(\alpha_a)}$$

$$f(\alpha_a) = \frac{\mu_b}{\mu_m}(1 - \alpha_a)^{1/2} \tag{2.62}$$

  and the Eötvös number is defined as the ratio between the gravitational and surface tension forces:

$$Eo = \frac{(\rho_b - \rho_a)gD_S^2}{\sigma} \tag{2.63}$$

  – The cap distorted bubble regime

$$C_d(cap) = C_{d\infty}(1 - \alpha_a)^2 \tag{2.64}$$

  where:

$$C_{d\infty} = \frac{8}{3} \tag{2.65}$$

Finally an automatic regime selection is performed in the code according to the following algorithm derived from CFX manual [12]:

$$\begin{aligned} C_d &= C_d(sphere) && \text{if} \quad C_d(sphere) \geq C_d(ellipse) \\ C_d &= min(C_d(ellipse), C_d(cap)) && \text{if} \quad C_d(sphere) < C_d(ellipse) \end{aligned} \tag{2.66}$$

- *The Ishii-Zuber model for sparsely distributed fluid particles* [12]

  This model distinguishes also three bubble regime with different drag modeling:

  - The spherical bubble regime with a drag coefficient defined by the Ishii-Zuber formula for spherical solid particles:

$$C_d(sphere) = \frac{24}{Re_{bm}}(1 + 0.15Re_{bm}^{0.687}) \tag{2.67}$$

  - The ellipse distorted bubble regime:

$$C_d(ellipse) = C_{d\infty} = \frac{2}{3}Eo^{1/2} \tag{2.68}$$

  - The cap distorted bubble regime

$$C_d(cap) = C_{d\infty} = \frac{8}{3} \tag{2.69}$$

  Finally an automatic regime selection is performed in the code according to the following algorithm derived from CFX manual [12]:

$$\begin{aligned} C_d(dist) &= min(C_d(ellipse), C_d(cap)) \\ C_d &= max(C_d(sphere), C_d(dist)) \end{aligned} \tag{2.70}$$

These last two models for fluid particles are quite similar for low values of the void fraction because the modifying functions $E(\alpha_a)$ and $(1 - \alpha_a)^2$ goes to 1, but appreciable differences can probably be observed at relatively high void fraction. Since this study focused on bubbly flow and consequently low void fraction, then almost no difference between the two models were observed during simulations as expected.

2.5.2 Lift force

The lift force plays an important role on the radial distribution of the void fraction and it consists of a lateral force acting on bubbles which can push bubbles towards the wall or towards the centerline of the pipe depending on the bubble size (i.e. mean Sauter diameter).

$$\mathbf{M}_a^l = -C_l \rho_b \alpha_a (\mathbf{U}_r) \times \nabla \times \mathbf{U}_b \tag{2.71}$$

For a spherical bubble, the lift coefficient C_l is always positive so that the lift force acts towards the pipe wall. That phenomena generates due to a non-constant pressure distribution over the bubble surface [20]. In fact assuming a spherical solid particle rising in a pipe, the relative velocity between the two phases is larger at the side with the lower liquid velocity (i.e. at the side closer to the wall considering a turbulent liquid velocity profile) causing a lower pressure at that side that pushes the bubbles towards the pipe wall.

For deformed larger bubbles more complicated phenomena arises and an inversion of sign for the lift coefficient C_l is observed in experiments. Studies were performed in order to explain this phenomenon and explanations were related to the interaction between the deformed bubble wakes and the corresponding vorticity generated at the bubble surface [20]. However this phenomena is still not fully understood and research is ongoing on the topic.

Two different models for the lift coefficient C_l are currently included in the solver:

- *The Tomiyama lift model [21]*

$$C_l = \begin{cases} \min(0.288\tanh(0.121Re_b), f(Eo_d)) & Eo_d < 4 \\ f(Eo_d) & 4 < Eo_d < 10 \\ -0.27 & Eo_d > 10 \end{cases} \tag{2.72}$$

$$f(Eo_d) = 0.00105Eo_d^3 - 0.0159Eo_d^2 - 0.0204Eo_d + 0.474 \tag{2.73}$$

Here the Eotvos number is based on the Wellek horizontal bubble diameter d_h:

$$Eo_d = \frac{(\rho_b - \rho_a)gd_h^2}{\sigma} \tag{2.74}$$

$$d_h = D_S(1 + 0.163Eo^{0.757})^{1/3} \tag{2.75}$$

With this formulation the lift coefficient changes sign at $D_S > 5.8$ mm as shown in Figure 2.2, so that large bubbles are moved towards the centerline of the pipe as shown by experiments.

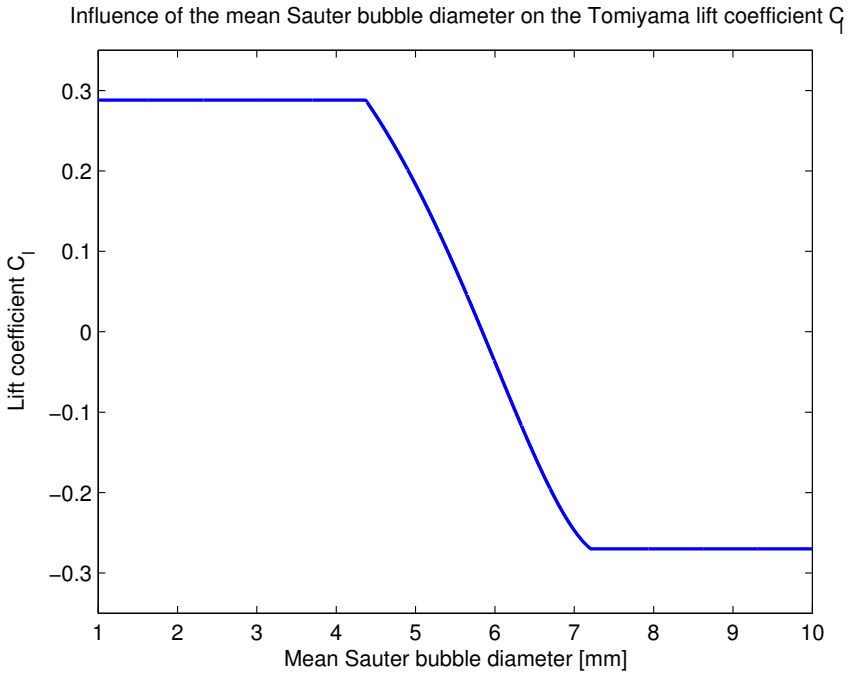

Figure 2.2: Influence of the mean Sauter bubble diameter on the Tomiyama lift coefficient C_l.

The Tomiyama lift model was obtained fitting experimental data on single bubbles in a laminar shear flow with $3.7 < Re < 210$ (i.e. low Reynolds number) and using a solution of glycerol and water (i.e. an high viscosity system) [20].

The simulation cases in this thesis deal with high Reynolds number water-air flows and therefore the validity of this model is not guaranteed. However Lucas [20] stated that this correlation leads to satisfactory and good results if compared with experiments and therefore it was used as the default model in the current solver.

- *The Rusche lift model [8, page 278]*

$$C_l = 6.51 \times 10^{-4} \alpha_a^{-1.2} \tag{2.76}$$

This formulation depends directly on the void fraction but it is also deduced from a small data-set so that the extension and usage for many different flow conditions is of doubtful validity. Furthermore this formulation leads to an always positive lift coefficient C_l contrary to experimental findings for large bubbles.

The lift force coefficient C_l can be also set constant by the user in order to perform sensitivity studies. It should be noted that, as default (but this option can be turned off by the user), the lift force is turned off in the cells adjacent to the walls in order to avoid possible unexpected fluctuation of void fraction in those cells during the numerical simulation.

2.5.3 Virtual mass force

The virtual mass (or added mass) force is the inertia added to the system because the accelerating or decelerating bubble must move the surrounding fluid as it moves through it. Currently in the solver two different models for the virtual mass force are implemented:

- The following model is used in [1], [12], [7] and [8] and it is set as default in the solver:

$$\mathbf{M}_a^{vm} = -C_{vm} \rho_b \alpha_a \left(\frac{D\mathbf{U}_a}{Dt} - \frac{D\mathbf{U}_b}{Dt} \right) \tag{2.77}$$

where it is assumed that $C_{vm} = 0.5$ but it is also user-modifiable.

- The following model was used in Michta's master thesis [14] and reads:

$$\mathbf{M}_a^{vm} = -\frac{1}{2} \rho_b \alpha_a \frac{1 + 2\alpha_a}{1 - \alpha_a} \left(\frac{D\mathbf{U}_a}{Dt} - \frac{D\mathbf{U}_b}{Dt} \right) \tag{2.78}$$

However it was proved that the virtual mass force has little influence when calculating the steady state conditions (fully developed flows) because the acceleration term goes almost to zero. Therefore the virtual mass force is useful only to predict more exactly the transient evolution of the flow. That explains why the virtual mass force is finally neglected in Michta's master thesis [14]. However it was found in our simulations that the virtual mass force has a stabilizing effect on the solution procedure even if it goes almost to zero at steady state.

2.5.4 Wall lubrication force

This force was first proposed by Antal in [22] to predict the near wall peak void fraction. Experimentally it was found that the void fraction is often concentrated close to the wall but not touching it (wall-peaked void fraction distribution).

The use of the wall force allows a good prediction of the experimental results avoiding the concentration of all the bubbles right on the wall as highlighted by Rzehak et al. in [23].

According to Frank [24], the implemented model in the solver reads:

$$\mathbf{M}_a^{wl} = C_w \rho_b \alpha_a |\mathbf{U}_r - (\mathbf{U}_r \cdot \mathbf{n}_w)\mathbf{n}_w|^2 (-\mathbf{n}_w) \tag{2.79}$$

where $\mathbf{n}_w$ is the vector normal to the wall and $\mathbf{U}_r - (\mathbf{U}_r \cdot \mathbf{n}_w)\mathbf{n}_w$ is the projection of the relative velocity parallel to the wall as also required in [23]. The wall lubrication coefficient C_w depends on the distance to the wall and should be always positive so that the bubbles are pushed away from the wall.

Currently three models for the wall lubrication coefficient C_w are implemented in the solver:

- *The Tomiyama wall lubrication model* [21]

$$C_w = \frac{1}{2} C_{wl} D_S \left(\frac{1}{y_w^2} - \frac{1}{(D_{\text{pipe}} - y_w)^2} \right) \tag{2.80}$$

$$C_{wl} = \begin{cases} 0.47 & Eo < 1 \\ \exp(-0.933 Eo + 0.179) & 1 < Eo < 5 \\ 0.00599 Eo - 0.0187 & 5 < Eo < 33 \\ 0.179 & Eo > 33 \end{cases} \tag{2.81}$$

  This formulation is based on experiments on single bubbles in laminar flow of a glycerol-water solution [23]. The Morton number[4] in the experiments was equal to $\log_{10} Mo = -2.8$, while the range of interest for water-air system is around the value $\log_{10} Mo = -10.6$. Therefore the applicability of this formulation cannot be guaranteed since it is out of the range of validity. However it should be highlighted that this is one of the most used model in two-phase flow CFD simulations. It proved to be efficient in many different works (e.g. [23] and [24]), and therefore this was used as the reference model in this thesis.

- *The Hosokawa wall lubrication model* [23]

  Hosokawa [25] investigated a larger range of Morton numbers always with glycerol-water solutions, deducing the following expression for C_{wl}:

$$C_{wl} = 0.0217 Eo \tag{2.82}$$

  According to Rzehak et al. [23] this model shows better performances in modeling air-water flows if compared with the Tomiyama wall lubrication model.

  A comparison between the Tomiyama and Hosokawa model for the wall lubrication coefficient is shown in Figure 2.3.

- *The Frank wall lubrication model* [24]

$$C_w = C_{wl} \max\left(0, \frac{1}{C_{wd}} \frac{1 - y_w/C_{wc}/D_S}{y_w (y_w/C_{wc}/D_S)^{p-1}}\right) \tag{2.83}$$

  with $C_{wc} = 10.0$, $C_{wd} = 6.8$, $p = 1.7$ and C_{wl} equal to the one calculated in the Tomiyama model.

[4] $Mo = \frac{g \mu_b^4 \Delta\rho}{\rho_b^2 \sigma^3}$ where $\Delta\rho = \rho_b - \rho_a$

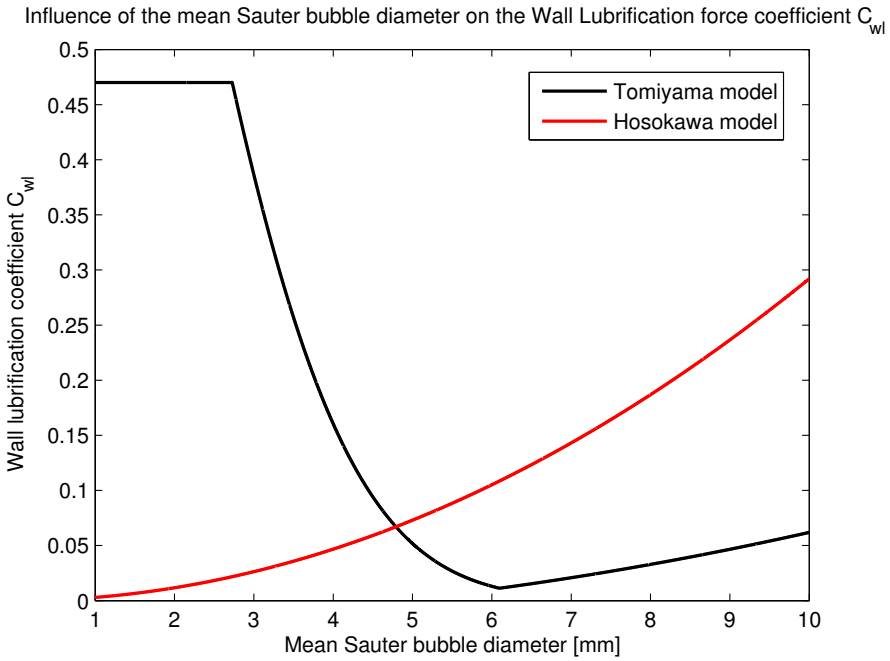

Figure 2.3: Influence of the mean Sauter bubble diameter on the wall lubrication force coefficient C_{wl}.

The implemented model in (2.79) is slightly different from the one used in Michta's master thesis [14] and [23] that reads:

$$\mathbf{M}_a^{wl} = C_w \rho_b \alpha_a |\mathbf{U}_r|^2 (-\mathbf{n}_w) \tag{2.84}$$

where the parallel component of the relative velocity is approximated with the relative velocity itself. According to [23] this approximation generates negligible errors since the bubble motion in vertical pipes is predominantly parallel to the wall so that the difference between the two formulation is small. However the formulation used in the solver is more general and it can be applied also to geometries different from vertical pipes.

2.5.5 Turbulent dispersion force

The turbulent dispersion force accounts for the turbulent fluctuations of liquid velocity and the effect that has on the gas bubbles. This force influences the sharpness of the wall peak of the void fraction, i.e. reduction in turbulent dispersion force leads to a sharper wall peak.

The characterizing feature of this force is the fact that it is proportional to the void-fraction gradient and therefore this could theoretically generate unstable results as previously discussed in Section 2.3 and 2.4.

Three models for the turbulent dispersion force are implemented in the solver:

- The Gosman model [26]

$$\mathbf{M}_a^{td} = -\frac{3}{4} C_d \frac{\rho_b}{D_S} \frac{\nu_b^t}{\sigma_t} |\mathbf{U}_r| \nabla \alpha_a \tag{2.85}$$

  where in our solver $\sigma_t = 0.9$ and C_d stands for the Schiller-Naumann drag force coefficient in (2.55).

- The Lopez de Bertodano model [27]

$$\mathbf{M}_a^{td} = -C_{td}\rho_b k_b \nabla \alpha_a \tag{2.86}$$

where C_{td} is set as default equal to 1.0, but it can be modified by the user. This is one of the simplest available turbulent dispersion force model. It was used as the reference model in this thesis for its simplicity and stability performances.

- The Burns model [28]

Burns [28] derived the following expression for the turbulent dispersion force through an Favre averaging of the drag force

$$\mathbf{M}_a^{td} = -\frac{3}{4} C_d \frac{\rho_b}{D_S} \frac{\nu_b^t}{\sigma_t} |\mathbf{U}_r| \alpha_a \left(\frac{1}{\alpha_a} + \frac{1}{\alpha_b} \right) \nabla \alpha_a \tag{2.87}$$

where $\sigma_t = 0.9$ and C_d stands for the drag force coefficient that is used during the simulations. The implementation of this model in the solver was simplified in order to avoid unnecessary division by the void fraction and reads:

$$\mathbf{M}_a^{td} = -\frac{3}{4} C_d \frac{\rho_b}{D_S} \frac{\nu_b^t}{\sigma_t} |\mathbf{U}_r| \frac{1}{1-\alpha_a} \nabla \alpha_a \tag{2.88}$$

2.6 Interfacial area concentration transport equation

The interfacial area concentration corresponds to the area of the gas bubbles per unit volume and for spherical bubbles is defined as:

$$IAC = \frac{6\alpha_a}{D_S} \tag{2.89}$$

where D_S is the mean Sauter bubble diameter.

The IAC equation is used to evaluate the mean Sauter bubble diameter which is used in the calculation of the drag, lift and turbulent dispersion force.

The IAC equation according to [13] reads:

$$\frac{\partial (IAC)}{\partial t} + \nabla \cdot ((IAC)\mathbf{U}_a) = \frac{2}{3} \frac{IAC}{\alpha_a} \left(\frac{\partial \alpha_a}{\partial t} + \nabla \cdot (\alpha_a \mathbf{U}_a) \right) + \Phi_{\mathrm{BB}} + \Phi_{\mathrm{BC}} + \Phi_{\mathrm{NUC}} \tag{2.90}$$

substituting the continuity equation Eqn. (2.3) for α_a, the final implemented IAC equation is:

$$\frac{\partial (IAC)}{\partial t} + \nabla \cdot ((IAC)\mathbf{U}_a) = \frac{2}{3} \frac{IAC}{\alpha_a} \left(\frac{\Gamma_a}{\rho_a} - \frac{\alpha_a}{\rho_a} \frac{\mathrm{D}(\rho_a)}{\mathrm{dt}} \right) + \Phi_{\mathrm{BB}} + \Phi_{\mathrm{BC}} + \Phi_{\mathrm{NUC}} \tag{2.91}$$

This equation was developed by Hibiki and Ishii [29] for 1-D geometry and 1-group bubble diameter, and it was then extended to 3-D geometry. The first term on the r.h.s. of Eqn. (2.91) refers to the contribution of phase change and expansion due to pressure-density change. Φ_{BB} and Φ_{BC} stands for the source and sink term induced by the breakup and coalescence phenomena, respectively. Φ_{NUC} stands for the nucleation source term at the heated walls and it is active only in near wall cells in diabatic cases.

The code includes the IAC equation and also the possibility to choose a constant bubble diameter through the option *haveIACeqn* in the file *moduleControls* as shown in Appendix E.

Three models are implemented for the breakup and coalescence source terms: *Hibiki-Ishii* [29], *Yao-Morel* [13] and *Lo-Zhang* [1].

Hibiki and Ishii model coupled with the standard $k-\epsilon$ turbulence model is the default model used for adiabatic cases with bubbly flow since it was derived with adiabatic air-water experimental results with $\alpha_a < 0.25$. On the contrary the other two implemented models for the breakup and coalescence source terms are specifically designed for sub-cooled boiling and therefore they are not taken into consideration in this master thesis. The Yao-Morel source terms require the usage of the Yao-Morel $k-\epsilon$ model, while the Lo-Zhang source terms require the usage of the standard $k-\epsilon$ model.

2.6.1 The Hibiki-Ishii breakup and coalescence source terms

Hibiki and Ishii [29] modeled sink and source terms of the interfacial area concentration based on mechanisms of bubble-bubble and bubble-turbulent eddy random collisions:

$$\Phi_{\mathrm{BC}} = -\Gamma_C \frac{\alpha_a^2 \epsilon_b^{1/3}}{D_S^{5/3}(\alpha_{\max} - \alpha_a)} \exp\left(-K_C \frac{D_S^{5/6} \rho_b^{1/2} \epsilon_b^{1/3}}{\sigma^{1/2}}\right) \tag{2.92}$$

with $\Gamma_C = 0.0314$ and $K_C = 1.29, \alpha_{\max} = 0.74$, and

$$\Phi_{\mathrm{BB}} = \Gamma_B \frac{\alpha_a(1-\alpha_a)\epsilon_b^{1/3}}{D_S^{5/3}(\alpha_{\max} - \alpha_a)} \exp\left(-K_B \frac{\sigma}{D_S^{5/3} \rho_b \epsilon_b^{2/3}}\right) \tag{2.93}$$

with $\Gamma_B = 0.0209$ and $K_B = 1.59$ (standard values suggested in [29]).

Φ_{NUC} was modeled in [30] as:

$$\Phi_{\mathrm{NUC}} = \pi d_{lo}^2 \cdot N'' f a_w \tag{2.94}$$

where N'' is the active nucleation site density, and f is the bubble departure frequency.

The adjustable variables Γ_B and Γ_C have been modeled so far with constant values but correlations fitted over experiments were developed by Hibiki-Ishii [29] and therefore implemented in the solver. In order to use the varying adjustable variables, the user should turn on the switch *varyBreakCoal* in the file *interfacialProperties* in the case folder.

According to Hibiki-Ishii [29], the adjustable variables are modeled as:

$$\Gamma_C = 1.82 \times 10^{-8} \alpha_a \tilde{\epsilon} \tag{2.95}$$

$$\Gamma_B = 5.02 \times 10^{-8} \alpha_a \tilde{\epsilon} \tag{2.96}$$

The dimensionless energy dissipation rate per unit mass $\tilde{\epsilon}$ is defined as:

$$\tilde{\epsilon} = \frac{\epsilon Lo^4}{\nu_b^3} \tag{2.97}$$

where $Lo = \sqrt{\frac{\sigma}{g\Delta\rho}}$ is the Laplace length that characterizes the bubble diameter length scale [29] and $\Delta\rho$ is the density difference between the two phases.

2.7 Turbulence modeling

2.7.1 The turbulence of the liquid phase

In order to obtain the value of the turbulent kinematic viscosity, a $k-\epsilon$ turbulence model is used. The turbulent kinematic viscosity is then used to model the effect of the turbulence on the Reynolds stresses in the momentum conservation equation.

Four different $k-\epsilon$ models are implemented and user-selectable. The first three versions of the turbulence model are the implementation of a standard $k-\epsilon$ model modified for compressible two-phase flows. The user can select each of them in the file *moduleControls* selecting the options *stdkEpsilonKai*, *stdkEpsilonKaiEff* or *stdkEpsilonGhione*. If none of these models are selected, then the modified Yao-Morel model designed for sub-cooled boiling is used.

The $k-\epsilon$ model solves two differential transport equation in order to determine the turbulent kinetic energy k_b and the turbulent dissipation ϵ_b for the liquid phase. Then the turbulent kinematic viscosity is computed as:

$$\nu_b^{\mathrm{t}} = C_\mu \frac{k_b^2}{\epsilon_b} \tag{2.98}$$

or according to Sato and Sekoguchi [31] (default model in the solver):

$$\nu_b^{\mathrm{t}} = C_\mu \frac{k_b^2}{\epsilon_b} + \frac{1}{2} C_{\mu b} D_S \alpha_a |\mathbf{U}_a - \mathbf{U}_b| \tag{2.99}$$

where $C_\mu = 0.09$ and $C_{\mu b} = 1.2$. Finally the turbulent effective liquid kinematic viscosity is defined as:

$$\nu_b^{\mathrm{eff}} = \nu_b + \nu_b^{\mathrm{t}} \tag{2.100}$$

and the thermal diffusivity as:

$$\kappa_b{}^{\mathrm{eff}} = \frac{\lambda_b}{\rho_b c_{p_b}} + \frac{\nu_b^{\mathrm{t}}}{Pr_b^{\mathrm{t}}} \tag{2.101}$$

where Pr_b^{t} is the turbulent Prandtl number assumed equal to 0.9 [1].

The different $k-\epsilon$ models will be now outlined and discussed.

The standard $k-\epsilon$ model in two fluid flow (Kai's version)

This model consists of the standard $k-\epsilon$ model for the liquid phase originally developed in [32] with additional terms S_ϵ and S_k to take into account the effect of the dispersed gas phase on the turbulence according to [8]:

$$\frac{\partial(k_b)}{\partial t} + (\mathbf{U}_b \cdot \nabla) k_b = \nabla \cdot \left[\frac{\nu_b^{\mathrm{eff}}}{\sigma_k} \nabla k_b \right] + G - \epsilon_b + S_k \tag{2.102}$$

$$\frac{\partial(\epsilon_b)}{\partial t} + (\mathbf{U}_b \cdot \nabla) \epsilon_b = \nabla \cdot \left[\frac{\nu_b^{\mathrm{eff}}}{\sigma_\epsilon} \nabla \epsilon_b \right] + \frac{\epsilon_b}{k_b} (C_{\epsilon 1} G - C_{\epsilon 2} \epsilon_b) + S_\epsilon \tag{2.103}$$

where G stands for the production of turbulent kinetic energy and is defined as:

$$G = \nu_b^{\mathrm{t}} \left(\nabla \mathbf{U}_b : \mathrm{dev}(\nabla \mathbf{U}_b + (\nabla \mathbf{U}_b)^T) \right) \tag{2.104}$$

where : stands for the double inner product and the operator *dev* takes the deviatoric component of a rank 2 tensor T with the property of being traceless (i.e. trace = 0). The operator *dev* is available in OpenFoam and it is defined as:

$$dev(T) = T - \frac{1}{3}tr(T)I \tag{2.105}$$

I is the identity matrix and the other coefficient are $C_{\epsilon 1} = 1.44$, $C_{\epsilon 2} = 1.92$, $\sigma_\epsilon = 1.3$, $\sigma_k = 1.0$.

The formulation of the production of turbulent kinetic energy is not in agreement with the formulation in Rusche [8] because it is missing a multiplying factor equal to 2 and ν_b^{t} is used instead of ν_b^{eff}.

However Rusche's formulation is of doubtful validity, in fact according to [33], the standard $k-\epsilon$ model contain a turbulent kinetic energy production term that can be expressed in tensorial notation as:

$$G = - < U_i U_j > \frac{\partial(U_i)}{\partial x_j} \tag{2.106}$$

where $< U_i U_j >$ is the liquid Reynolds turbulent stress tensor that can be expressed according to the Boussinesq hypothesis in [34]:

$$\tau_b = - < U_i U_j > = \nu_b^{\mathrm{t}} \left(\nabla \mathbf{U}_b + (\nabla \mathbf{U}_b)^T\right) - \frac{2}{3}\left(\nu_b^{\mathrm{t}} \nabla \cdot \mathbf{U}_b\right)\mathbf{I} \tag{2.107}$$

Since it is valid:

$$\nu_b^{\mathrm{t}} dev\left(\nabla \mathbf{U}_b + (\nabla \mathbf{U}_b)^T\right) = \nu_b^{\mathrm{t}}\left(\nabla \mathbf{U}_b + (\nabla \mathbf{U}_b)^T\right) - \frac{2}{3}\nu_b^{\mathrm{t}} \mathbf{I} \nabla \cdot \mathbf{U}_b \tag{2.108}$$

therefore Eqn. (2.121) is re-written as:

$$\tau_b = \nu_b^{\mathrm{t}} dev\left(\nabla \mathbf{U}_b + (\nabla \mathbf{U}_b)^T\right) \tag{2.109}$$

The turbulent kinetic energy production term can be re-written according to the definition in Eqn. (2.106):

$$G = \nu_b^{\mathrm{t}}\left(\nabla \mathbf{U}_b : \mathrm{dev}(\nabla \mathbf{U}_b + (\nabla \mathbf{U}_b)^T)\right) \tag{2.110}$$

therefore the implementation results correct if the Boussinesq hypothesis in Eqn. (2.107) is used.

In order to calculate $\nabla \mathbf{U}_b + (\nabla \mathbf{U}_b)^T$, the OpenFoam function *twoSymm* is used. It returns twice the symmetric part of a tensor and therefore it returns $2symm(T) = 2\left(\frac{1}{2}(T + T^T)\right) = (T + T^T)$.

The final implemented model in OpenFoam reads (see implemented code in Appendix F):

$$\begin{aligned} \frac{\partial(k_b)}{\partial t} + \nabla \cdot (\mathbf{U}_b k_b) \;&-\; k_b \nabla \cdot (\mathbf{U}_b) - \nabla \cdot \left[\alpha_k \nu_b^{\mathrm{eff}} \nabla k_b\right] + \nabla \cdot (\mathbf{U}_{Vb} k_b) \\ &-\; k_b \nabla \cdot (\mathbf{U}_{Vb}) = G - \frac{\epsilon_b}{k_b} k_b + \frac{k_b}{\alpha_b \rho_b}\left(\Gamma_{VL} - \Gamma_{LV}\right) \end{aligned} \tag{2.111}$$

$$\begin{aligned} \frac{\partial(\epsilon_b)}{\partial t} + \nabla \cdot (\mathbf{U}_b \epsilon_b) \;&-\; \epsilon_b \nabla \cdot (\mathbf{U}_b) - \nabla \cdot \left[\alpha_\epsilon \nu_b^{\mathrm{eff}} \nabla \epsilon_b\right] + \nabla \cdot (\mathbf{U}_{Eb} \epsilon_b) \\ &-\; \epsilon_b \nabla \cdot (\mathbf{U}_{Eb}) = \frac{\epsilon_b}{k_b}(C_{\epsilon 1} G - C_{\epsilon 2} \epsilon_b) + \frac{\epsilon_b}{\alpha_b \rho_b}\left(\Gamma_{VL} - \Gamma_{LV}\right) \end{aligned} \tag{2.112}$$

where $\alpha_k = \frac{1}{\sigma_k}$ and $\alpha_\epsilon = \frac{1}{\sigma_\epsilon}$ and:

$$\mathbf{U}_{Vb} = -\alpha_k (\nu_b^{\text{eff}})_f \frac{\nabla^\perp (\alpha_b \rho_b)}{(\rho_b)_f (\alpha_b + \delta)_f} \tag{2.113}$$

$$\mathbf{U}_{Eb} = -\alpha_\epsilon (\nu_b^{\text{eff}})_f \frac{\nabla^\perp (\alpha_b \rho_b)}{(\rho_b)_f (\alpha_b + \delta)_f} \tag{2.114}$$

Therefore in this turbulence model the compressibility of the liquid phase is considered and the definition of the additional source term takes into account the evaporation-condensation fluxes due to phase change. This formulation of the compressibility term and of the additional source was the one used to derive the results in [1].

Other formulations of the additional source terms can be found in the literature (e.g. in [8, page 106 and 120]) but they are not included in the solver.

The standard $k-\epsilon$ model in two fluid flow (Kai's version with effective liquid viscosity)

This model can be selected by the user with the option *stdkEpsilonKaiEff*. It consists in the standard $k-\epsilon$ model (Kai's version) with the usage of the effective liquid viscosity ν_b^{eff} in the production of turbulent kinetic energy G instead of ν_b^{t}, as suggested by Rusche in [8]. Therefore the production of turbulent kinetic energy G reads:

$$G = \nu_b^{\text{eff}} \left(\nabla \mathbf{U}_b : \text{dev}(\nabla \mathbf{U}_b + (\nabla \mathbf{U}_b)^T) \right) \tag{2.115}$$

The standard $k-\epsilon$ model in two fluid flow (Ghione's version)

This version of the standard $k-\epsilon$ model is derived starting from the implementation for the mono-phase $k-\epsilon$ model in the CFX manual [12, page 111] that reads:

$$\frac{\partial (k_b)}{\partial t} + \nabla \cdot (\mathbf{U}_b \epsilon_b) = \nabla \cdot \left[\left(\nu_b + \frac{\nu_b^{\text{eff}}}{\sigma_k} \right) \nabla k_b \right] + P_k - \epsilon_b + P_{kb} \tag{2.116}$$

$$\frac{\partial (\epsilon_b)}{\partial t} + \nabla \cdot (\mathbf{U}_b \epsilon_b) = \nabla \cdot \left[\left(\nu_b + \frac{\nu_b^{\text{eff}}}{\sigma_\epsilon} \right) \nabla \epsilon_b \right] + \frac{\epsilon_b}{k_b} (C_{\epsilon 1} P_k - C_{\epsilon 2} \epsilon_b + C_{\epsilon 3} P_{\epsilon_b}) \tag{2.117}$$

where the dissipation coefficient $C_{\epsilon 3} = 1.0$, P_{ϵ_b} and P_{kb} represents the influence of the buoyancy forces and reads in the hypothesis of full buoyancy model [12]:

$$P_{kb} = -\alpha_k \nu_b^{\text{t}} \frac{\mathbf{g} \cdot \nabla (\rho_b)}{\rho_b} \tag{2.118}$$

$$P_{\epsilon_b} = C_{\epsilon 3} max(0, P_{kb}) \tag{2.119}$$

This terms are not implemented in our solver, since the gradient of the liquid density is small, even if the liquid phase would be considered compressible (in our future simulation the liquid will be actually considered incompressible and therefore $\nabla \rho_b = 0$).

P_k stands for the turbulence production due to viscous forces:

$$P_k = \nu_b^{\text{t}} \left(\nabla \mathbf{U}_b + (\nabla \mathbf{U}_b)^T \right) \nabla \mathbf{U}_b - \frac{2}{3} \nabla \mathbf{U}_b \left(k_b + 3 \nu_b^{\text{t}} \nabla \cdot \mathbf{U}_b \right) \mathbf{I} \tag{2.120}$$

The multiplying factor equal to 3 in the last term of the r.h.s. is derived from the "frozen stress" assumption [12], but since the Boussinesq hypothesis is used, a factor equal to 1 is preferable.

Therefore this $k-\epsilon$ model uses a more consistent Boussinesq hypothesis according also to [35] and [13]:

$$\tau_b = -<U_i U_j> = \nu_b^{\mathrm{t}}\left(\nabla \mathbf{U}_b + (\nabla \mathbf{U}_b)^T\right) - \frac{2}{3}\left(k_b + \nu_b^{\mathrm{t}} \nabla \cdot \mathbf{U}_b\right)\mathbf{I} \tag{2.121}$$

This hypothesis is the one also used to derive the momentum equation (see Eqn. (2.17)) and therefore it is more consistent with the whole structure of the code. Then using Eqn. (2.108), it is obtained:

$$\tau_b = \nu_b^{\mathrm{t}} dev\left(\nabla \mathbf{U}_b + (\nabla \mathbf{U}_b)^T\right) - \frac{2}{3} k_b \mathbf{I} \tag{2.122}$$

The turbulent kinetic energy production term can be re-written according to the definition in (2.106):

$$G = \nu_b^{\mathrm{t}}\left(\nabla \mathbf{U}_b : \mathrm{dev}(\nabla \mathbf{U}_b + (\nabla \mathbf{U}_b)^T)\right) - \frac{2}{3} k_b \nabla \cdot \mathbf{U}_b \tag{2.123}$$

Therefore this model includes one more term in the definition of the turbulent production term, which is proportional to the divergence of the liquid velocity that in case of incompressible liquid phase should become zero.

The other difference compared to Kai's version is in the usage of the expression $(\nu_b + \alpha_k \nu_b^{\mathrm{t}})$ instead of $\alpha_k \nu_b^{\mathrm{eff}}$ according to CFX manual [12]

The finally deduced model for compressible liquid phase reads:

$$\begin{aligned} \frac{\partial(k_b)}{\partial t} + \nabla \cdot (\mathbf{U}_b k_b) &- k_b \nabla \cdot (\mathbf{U}_b) - \nabla \cdot \left[\left(\nu_b + \alpha_k \nu_b^{\mathrm{t}}\right) \nabla k_b\right] + \nabla \cdot (\mathbf{U}_{Vb} k_b) \\ &- k_b \nabla \cdot (\mathbf{U}_{Vb}) = G - \frac{\epsilon_b}{k_b} k_b + \frac{k_b}{\alpha_b \rho_b}\left(\Gamma_{VL} - \Gamma_{LV}\right) - \frac{2}{3} k_b \nabla \cdot \mathbf{U}_b \end{aligned} \tag{2.124}$$

$$\begin{aligned} \frac{\partial(\epsilon_b)}{\partial t} + \nabla \cdot (\mathbf{U}_b \epsilon_b) &- \epsilon_b \nabla \cdot (\mathbf{U}_b) - \nabla \cdot \left[\left(\nu_b + \alpha_\epsilon \nu_b^{\mathrm{t}}\right) \nabla \epsilon_b\right] + \nabla \cdot (\mathbf{U}_{Eb} \epsilon_b) \\ &- \epsilon_b \nabla \cdot (\mathbf{U}_{Eb}) = \frac{\epsilon_b}{k_b}(C_{\epsilon 1} G - C_{\epsilon 2} \epsilon_b) - \frac{2}{3} C_{\epsilon 1} \epsilon_b \nabla \cdot \mathbf{U}_b \\ &+ \frac{\epsilon_b}{\alpha_b \rho_b}\left(\Gamma_{VL} - \Gamma_{LV}\right) \end{aligned} \tag{2.125}$$

The Yao-Morel Modified $k-\epsilon$ model [13]

The Yao-Morel was developed in the frame of sub-cooled boiling modeling and additional source terms were added to incorporate the effects of the dispersed phase on the liquid turbulence:

$$\begin{aligned} \frac{\partial(\alpha_b \rho_b k_b)}{\partial t} + \nabla \cdot (\alpha_b \rho_b \mathbf{U}_b k_b) &= \nabla \cdot \left[\alpha_b \left(\frac{\mu_b^{\mathrm{t}}}{\sigma_k}\right) \nabla k_b\right] - \alpha_b \rho_b \epsilon_b + \alpha_b \tau_b : \nabla \mathbf{U}_b \\ &- (\mathbf{M}_a^d + \mathbf{M}_a^{vm}) \cdot (\mathbf{U}_a - \mathbf{U}_b) - \sigma(\Phi_{\mathrm{BC}} + \Phi_{\mathrm{BB}}) + k_{li}\Gamma_l \end{aligned} \tag{2.126}$$

$$\begin{aligned}\frac{\partial(\alpha_b\rho_b\epsilon_b)}{\partial t}+\nabla\cdot(\alpha_b\rho_b\mathbf{U}_b\epsilon_b) &= \nabla\cdot\left[\alpha_b\left(\frac{\mu_b^{\mathrm{t}}}{\sigma_\epsilon}\right)\nabla\epsilon_b\right]-C_{\epsilon2}\alpha_b\rho_b\frac{\epsilon_b^2}{k_b}\\ &+ C_{\epsilon1}\alpha_b\frac{\epsilon_b}{k_b}\tau_b:\nabla\mathbf{U}_b-\frac{2}{3}\alpha_b\rho_b\epsilon_b\nabla\cdot\mathbf{U}_b\\ &- C_{\epsilon3}(\mathbf{M}_a^d+\mathbf{M}_a^{vm})\cdot(\mathbf{U}_a-\mathbf{U}_b)\left(\frac{\epsilon_b}{D_S^2}\right)^{1/3}+\epsilon_{li}\Gamma_l\end{aligned} \tag{2.127}$$

where the default value for the user-modifiable coefficient $C_{\epsilon3}$ is 1.0 in our solver as in [13].

The following $k-\epsilon$ model is finally implemented in OpenFoam (see the implemented code in Appendix F):

$$\begin{aligned}\frac{\partial(k_b)}{\partial t}+\nabla\cdot(\mathbf{U}_bk_b) &- k_b\nabla\cdot(\mathbf{U}_b)-\nabla\left[\left(\nu_b+\alpha_k\nu_b^{\mathrm{t}}\right)\nabla k_b\right]+\nabla\cdot(\mathbf{U}_{Vb}k_b)\\ &- k_b\nabla\cdot(\mathbf{U}_{Vb})=G-\frac{\epsilon_b}{k_b}k_b-\frac{\sigma}{(\alpha_b+\delta)\rho_b}(\Phi_{\mathrm{BC}}+\Phi_{\mathrm{BB}})\\ &- \frac{2}{3}k_b\nabla\cdot\mathbf{U}_b-\frac{1}{(\alpha_b+\delta)\rho_b}(\mathbf{M}_a^d+\mathbf{M}_a^{vm})\cdot(\mathbf{U}_a-\mathbf{U}_b)\end{aligned} \tag{2.128}$$

$$\begin{aligned}\frac{\partial(\epsilon_b)}{\partial t}+\nabla\cdot(\mathbf{U}_b\epsilon_b) &- \epsilon_b\nabla\cdot(\mathbf{U}_b)-\nabla\left[\left(\nu_b+\alpha_\epsilon\nu_b^{\mathrm{t}}\right)\nabla\epsilon_b\right]+\nabla\cdot(\mathbf{U}_{Eb}\epsilon_b)\\ &- \epsilon_b\nabla\cdot(\mathbf{U}_{Eb})=\frac{\epsilon_b}{k_b}(C_{\epsilon1}G-C_{\epsilon2}\epsilon_b)-\frac{2}{3}C_{\epsilon1}\epsilon_b\nabla\cdot\mathbf{U}_b\\ &- \frac{C_{\epsilon3}}{(\alpha_b+\delta)\rho_b}(\mathbf{M}_a^d+\mathbf{M}_a^{vm})\cdot(\mathbf{U}_a-\mathbf{U}_b)\left(\frac{\epsilon_b}{D_S^2}\right)^{1/3}\end{aligned} \tag{2.129}$$

In the implemented equation $k_{li}\Gamma_l$ and $\epsilon_{li}\Gamma_l$ are neglected and compressibility terms appears (because the incompressibility assumption was used in the derivation in [13]).

For the sake of comparison, Michta [14] implemented a simplified version of the Yao-Morel $k-\epsilon$ model in (2.126) and (2.127), that is valid for incompressible liquid phase and where the additional source terms due to the turbulence of the gas phase are neglected:

$$\frac{\partial(\alpha_bk_b)}{\partial t}+\nabla\cdot(\mathbf{U}_bk_b) = \nabla\left[\alpha_k\nu_b^{\mathrm{t}}\nabla k_b\right]-\alpha_b\frac{\epsilon_b}{k_b}k_b+\alpha_bG \tag{2.130}$$

$$\frac{\partial(\alpha_b\epsilon_b)}{\partial t}+\nabla\cdot(\mathbf{U}_b\epsilon_b) = \nabla\left[\alpha_\epsilon\nu_b^{\mathrm{t}}\nabla\epsilon_b\right]-C_{\epsilon2}\alpha_b\frac{\epsilon_b^2}{k_b}+C_{\epsilon1}\alpha_b\frac{\epsilon_b}{k_b}G \tag{2.131}$$

It is interesting to note that by mistake it is missing the void fraction in the divergence term.

2.7.2 The turbulence of the vapor phase

The turbulence of vapor phase is assumed to be dependent on that of the liquid phase through a turbulence response coefficient C_t, defined as the ratio of the root mean square velocity fluctuations of the dispersed phase velocity $\mathbf{U}_a^{'}$ and of the continuous phase velocity $\mathbf{U}_b^{'}$ [8].

$$C_t=\frac{\mathbf{U}_a^{'}}{\mathbf{U}_b^{'}} \tag{2.132}$$

According to experimental results, Rusche [8, page 180] found out that the turbulence response coefficient depends on the void fraction α_a. In particular C_t is larger than one for low void fraction but reach an almost constant value equal to 1.0 for void fraction larger than 6 %. Therefore the suggested value for the turbulence response coefficient is equal to 1.0, even if, until now, the turbulence of the vapor phase has been neglected and a turbulence response coefficient equal to 0.0 has been used as default.

The turbulence response coefficient is used to calculate the effective viscosity of the vapor phase. This is simply expressed in [1] and [8] as:

$$\nu_a^{\text{eff}} = \nu_a + C_{\text{t}}^2 \nu_b^{\text{t}} \tag{2.133}$$

$$k_a = C_{\text{t}}^2 k_b \tag{2.134}$$

and the thermal diffusivity as:

$$K_a{}^{\text{eff}} = \frac{\lambda_a}{\rho_a c_{p_a}} + \frac{\nu_a^{\text{t}}}{Pr_a^{\text{t}}} \tag{2.135}$$

where Pr_a^{t} is the turbulent Prandtl number assumed equal to 0.9.

2.7.3 Wall functions

In order to properly predict the velocity profile close to the wall where in turbulent flows a viscous sub-layer and a log-layer profile can be observed according to the law of the wall, wall functions within the frame of the $k - \epsilon$ model are used.

The k equation is solved in the whole domain (also in near-wall cells) but that is not the case for the ϵ equation where special values are set in the near-wall cells as for the liquid turbulent viscosity. The procedure followed in the solver will be briefly outlined here.

Before setting the ϵ equation in "wallFunction.H", the value of ϵ in near-wall cells is set equal to:

$$\epsilon^{nw} = \frac{C_\mu^{0.75} k_b^{1.5}}{\kappa y} \tag{2.136}$$

where κ is the Von-Karman constant equal to 0.42.

Then the field y^+ is computed:

$$y^+ = \frac{C_\mu^{0.25} k_b^{0.5}}{\nu_b} y \tag{2.137}$$

where y stands for the distance from the nearest wall.

Depending on the value of y^+, the function G in near-wall cells is set up:

$$\begin{aligned} G^{nw} &= 0 && \text{if} \quad y^+ < 11.6 \quad \text{(viscous sub-layer)} \\ G^{nw} &= \nu_b^{\text{t}} \frac{C_\mu^{0.25} k_b^{0.5}}{Ky} \nabla^{\perp} \mathbf{U_b} && \text{if} \quad y^+ > 11.6 \quad \text{(turbulent layer)} \end{aligned} \tag{2.138}$$

Once the ϵ equation is set (but before solving it), the solver enters the file: "wallDissipation.H" where the values of ϵ in near-wall cells are set up and the corresponding terms in the ϵ equation matrix are eliminated.

Solved the ϵ equation and the k equation, the turbulent liquid viscosity at the wall is adjusted in "wallViscosity.H":

$$\begin{aligned} \nu_b^{tnw} &= 0 && \text{if} \quad y^+ < 11.6 \quad \text{(viscous sub-layer)} \\ \nu_b^{tnw} &= \nu_b \left[\frac{Ky^+}{ln(Ey^+)} - 1 \right] && \text{if} \quad y^+ > 11.6 \quad \text{(turbulent layer)} \end{aligned} \tag{2.139}$$

where $E = 9.793$.

This implementation is the same as in Michta's master thesis [14].

CHAPTER 3

DESCRIPTION OF THE TEST CASES

The two-phase solver was tested against a wide set of experimental results of adiabatic water-air upward bubbly flow in vertical pipes. The experimental data for the radial distribution of gas velocity, void fraction, IAC and bubble diameter were taken from Leung's PhD thesis [4], whose aim was to improve models for interfacial area concentration and inter-phase momentum transfer. The experiments consists of adiabatic upward flow of a mixture water-air at atmospheric pressure (at the outlet) and a temperature of 25°C (considered constant along the pipe).

The vertical pipe was made of Lucite (also known as Polymethyl methacrylate (PMMA) or Plexiglas) due to its transparency. It was 3.75 m long with an inner diameter of 25.4 mm (equal to 1 inch). The deionized and demineralized water was constantly filtered in order to remove impurities bigger than 5 μm and air was injected through a porous tube located at the bottom of the experimental setup. More details on the experimental facility can be found in Leung's PhD thesis [4].

The interest in experiments with small diameters of 25.4 mm is justified by the fact that standard hydraulic diameters in fuel bundles of BWR and PWR are around this value, if not even smaller. The selected test cases were not simulated previously with two-phase CFD codes in the reviewed literature. Most of the literature papers use experiments in pipes with larger diameter ($d_{pipe} > 38mm$, but usually $> 50mm$) and the models were adjusted accordingly in order to fit these experiments. Therefore it was interesting to test the solvers and models with experiments in a pipe with a relatively small diameter.

The measurements of the radial void fraction, bubble diameter, IAC and gas velocity were taken at three different axial locations $z/D = 12$, 62, 112. The flow conditions at $z/D = 12$ are considered as the inlet ones for our simulations and therefore boundary conditions were set according to the experimental values.

In order to evaluate the radial distribution of the void fraction, interfacial area concentration and gas velocity, a *double-sensor probe* was used by Leung [4]. The probe could be moved in the radial direction with an accuracy of 25.4 μm, so that the radial distributions of the properties was mapped out by successive measurements. This method was calibrated with a non-intrusive gamma densitometer measurement and since the double-sensor probe method is intrusive, the measurements were corrected to take into account of the influence of the probe on the measured results (more details in [4] and [36]).

The sensor of the probe consists of a small wire with a diameter of 0.12 mm made of platinum

and 13% rhodium. The two sensors are immersed in the fluid flow at a distance between the upstream and downstream sensors varying from 2 to 4 mm and an electrical current is driven through it. The probe measures the resistivity of the medium in which it is immersed and it is called *resistivity probe*. The liquid phase will close the electrical circuit thanks to the low resistivity and therefore the measured voltage output will be low. Otherwise if a gas bubble hits a sensor, the measured voltage will be high since the electrical circuit will be broken.

The fraction of time that the upstream sensor spends in the gas phase over the total time of the measurement (average sampling time of 75 s in order to sample a statistically significant number of bubbles) gives the time averaged local void fraction. The downstream sensor can not be used to measure the void fraction because it would measure an inaccurate and lower value since the upstream sensor prevents some bubbles from reaching the downstream sensor.

The axial gas velocity was evaluated using the distance between the two sensors which are placed vertically in the direction of the flow and measuring accurately when the bubble hits the upstream and downstream sensors. The interfacial area concentration is then evaluated using the measured bubble number frequency and the axial interfacial velocity as described in [4] and more in detail in [36].

The transport properties for the simulation of these test cases are considered constant along the pipe and are shown in Table 3.1.

Table 3.1: Transport properties of water and air in Leung experiments [4]

Property	**Value**
p	1.013 bar
ρ_b	998.21 kg/m^3
λ_b	0.5995 W/m/K
c_{pb}	4184.8 J/kg/K
ν_b	$1.003{\cdot}10^{-6}$ m^2/s
λ_a	0.0257 W/m/K
c_{pa}	1004.7 J/kg/K
ν_a	$1.511{\cdot}10^{-5}$ m^2/s
σ	0.0727 N/m

The only transport property that can actually vary during the simulation is the gas density that is calculated as a linear function of the pressure:

$$\rho_a = \bar{\psi}_a p \tag{3.1}$$

where $\bar{\psi}_a$ stands for the gas compressibility constant (the liquid compressibility constant is assumed equal to zero, i.e. incompressible liquid phase). The compressibility constant is evaluated from the ideal gas law that reads:

$$pV = nRT \tag{3.2}$$

it can be rewritten as:

$$\rho = \frac{M}{RT} p \tag{3.3}$$

where $M = 28.97\ g/mol$ is the dry air molar mass and $R = 8.314\ \frac{J}{molK}$ is the ideal gas constant. Therefore the compressibility constant can be evaluated as:

$$\bar{\psi}_a = \frac{M}{RT} = \frac{28.97 \cdot 10^{-3}}{8.314 \cdot 298} = 1.1693 \cdot 10^{-5} \tag{3.4}$$

Many experiments were performed by Leung with different combinations of liquid and gas superficial velocities and different flow regimes.

However, the solver is meant for the modeling of bubbly flow since the correlations used are valid only in this flow regime. Therefore only a few experiments with bubbly flow were selected in order to run sensitivity tests and validate the code. Four experiments were selected with maximum void fraction smaller than 25 % in bubbly flow regime. The choice of the value 25 % can be justified considering that the usual range of variation for dispersed bubbly flow is from 0 to 25 %, as clearly stated in [37], even if bubbly flow can be obtained with a void fraction up to 48 % [37].

The experiments are classified according to the different liquid and gas superficial velocities j_l and j_g. The four selected test cases are:

- $j_l = 0.3$ m/s and $j_g = 0.09$ m/s with maximum void fraction equal to 25 %;
- $j_l = 0.64$ m/s and $j_g = 0.09$ m/s with maximum void fraction equal to 15 %;
- $j_l = 1.1$ m/s and $j_g = 0.16$ m/s with maximum void fraction equal to 24 %;
- $j_l = 2.0$ m/s and $j_g = 0.16$ m/s with maximum void fraction equal to 16 %;

The effective flow pattern of the test cases was checked on flow pattern maps.

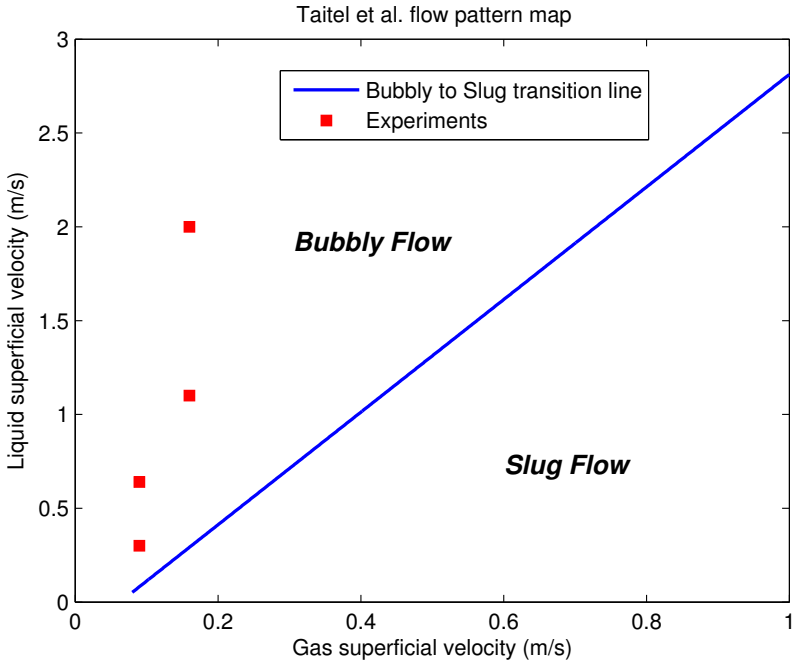

Figure 3.1: Taitel et al. flow pattern map with red points representing the experimental conditions.

According to Taitel et al. [38] the transition line from bubbly flow to slug flow can be represented with the following expression:

$$j_l = 3.0 j_g - 1.15 \left[\frac{g\left(\rho_l - \rho_g\right)}{\rho_l^2} \sigma \right]^{0.25} \tag{3.5}$$

The transition line described by Eqn. (3.5) and the experimental conditions are shown in Figure 3.1. All the four cases are located to the left of the transition line and therefore bubbly flow regime is expected.

The tube was modeled as a quasi-2D cylindrical geometry, i.e. using an equivalent 2D-geometry instead of a real 3D-geometry.

The use of such a geometry was already validated in [1] and [23] and it reduces the computational cost and the simulation time. Only a slice of the pipe is modeled within an angle of 5 °and with symmetry boundary conditions, as shown in Figure 3.2 and 3.3.

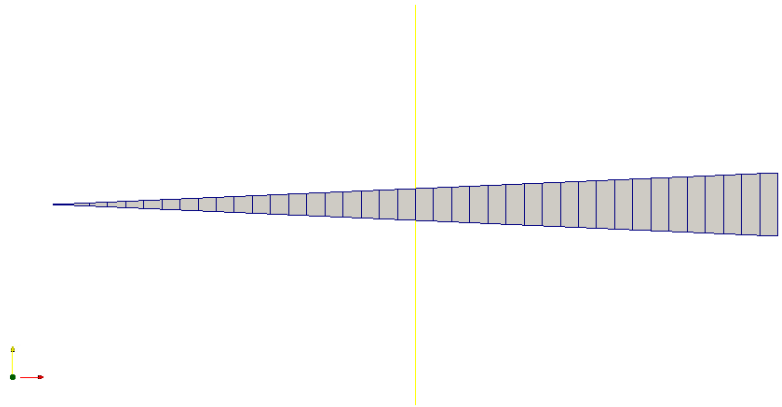

Figure 3.2: Example of mesh geometry in the radial direction of the pipe.

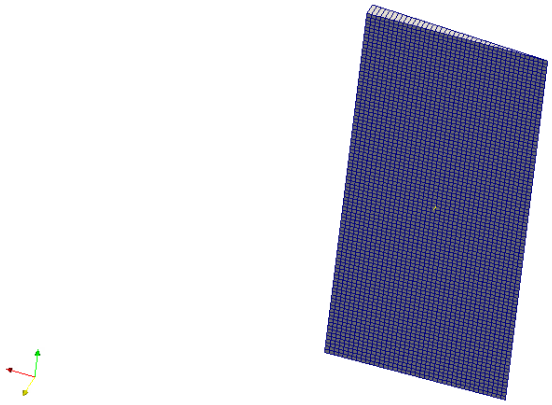

Figure 3.3: Example of mesh geometry of the pipe (the scales are not respected for presentation purposes).

The length of the tube was taken equal to 2800 mm which corresponds to 110 % of the minimum needed length as suggested in [23], so that there is no influence of the outlet boundary conditions on the results at $z/D = 112$, i.e. $z = 2.54$ m from the inlet in the simulations.

The utilized mesh is homogeneous (unless otherwise specified) and the number of cells varies according to the need of each simulation. However, the radial number of cells ranges between 10 and 70 and the axial number of cells between 100 and 200.

The solver requires the boundary and initial conditions for the void fraction, the interfacial area concentration, the pressure, the gas and liquid real velocities (not the superficial velocities), the turbulent kinetic energy k, the turbulent dissipation ϵ and the phase enthalpy for diabatic cases. The initial conditions are set uniform in the pipe.

Due to limitation on the validity of the standard $k - \epsilon$ turbulence model in near-wall cells ($y^+ > 20$), another mesh setting was also used in some simulations. This mesh setting for the test case

with $j_l = 0.64$ m/s and $j_g = 0.09$ m/s is not homogeneous and the near-wall cell has a constant radial size equal to 1.7 mm, as shown in Figure 3.4.

The use of a constant size close to the wall allows to maintain the $y^+ > 20$, avoiding the limitation on the validity of the turbulence model. Different refinement and grading of the mesh in the bulk was also tested, as discussed in Section 4.1.

Figure 3.4: Not-uniform mesh in the radial direction.

The typology of the boundary conditions used in the majority of the simulations is shown in Table 3.2.

Table 3.2: Boundary conditions used in the simulations.

Field	Inlet	Outlet	Pipe Wall
U_l [m/s]	constant fixedValue	pressureInletOutletVelocity	fixedValue = 0
U_g [m/s]	groovyBC	pressureInletOutletVelocity	slip
α [-]	groovyBC	inletOutlet	zeroGradient
p [kg/(ms^2)]	buoyantPressure	constant fixedValue = 1 atm	zeroGradient
IAC [m^{-1}]	groovyBC	inletOutlet	zeroGradient
k [m^2/s^2]	constant fixedValue	inletOutlet	zeroGradient
ϵ [m^2/s^3]	constant fixedValue	inletOutlet	zeroGradient

The lateral surfaces and the central axis use respectively the wedge and empty boundary conditions in order to define a quasi 2D geometry (see more details in [3]).

The pressureInletOutletVelocity option is usually used at the outlet of the pipe for the velocity fields. This option allows to deal with back-flows, in fact when the direction of the velocity at the outlet is reversed, then the boundary condition is switched from zeroGradient to pressureInletVelocity, which calculates the velocity from the flux normal to the patch if the pressure at the outlet is known. Analogously the inletOutlet boundary condition switches from zeroGradient to fixedValue (which is taken equal to zero, since no back-flow is expected) depending on the direction of the velocity. These different boundary conditions were tested and compared. It was shown that the pressureInletOutletVelocity boundary condition behaves exactly as an inletOutlet and a zeroGradient boundary condition in the simulations, as expected since no back-flow occurs.

The slip condition sets the normal component to the wall of the gas velocity equal to zero, while the boundary condition for the tangential velocity component is set to zeroGradient. The liquid velocity at the wall is fixed to zero since no slip is expected.

Regarding the pressure, a constant fixed value equal to the atmospheric pressure is set at the outlet, while the inlet pressure is evaluated with the buoyantPressure option which takes into account the hydrostatic pressure variation. The buoyantPressure option sets the gradient of the pressure equal to $\rho\vec{g}\cdot\hat{n}_w$. Other boundary conditions for the pressure field were tested (in particular

buoyantPressure for the wall and zeroGradient for the inlet) but no significant differences on the results were observed.

This is consistent because the calculated gradient for a vertical wall is equal to zero because $\vec{g} \cdot \hat{n}_w = 0$. The use of a buoyantPressure boundary condition can be important for non-vertical walls.

The groovyBC option allows to input non-uniform boundary conditions in OpenFoam without programming and it is very useful to insert non-uniform boundary conditions at the inlet of the pipe. The use of this option requires the installation of additional libraries to OpenFoam as explained in [39]. The use of such option in OpenFoam 2.0.1 requires the installation of the whole software package called: *swak4Foam* (i.e. *SWiss Army Knife for Foam*).

The experimental profiles of the required quantities at the inlet (i.e. $z/D = 12$) are taken from Appendix C in Leung's thesis [4].

The only missing data at the inlet is the liquid velocity profile, which therefore was deduced from the superficial liquid velocity and taken constant in most of our simulations at the inlet.

The constant value is evaluated using the definition of superficial velocity:

$$j_l = \frac{\int_A U_b \alpha_b dA}{A} \tag{3.6}$$

which can be rewritten using only the radial coordinate since a 2D geometry is considered. Therefore the constant equivalent liquid velocity reads:

$$\bar{U_b} = j_l \frac{R}{\int_0^R \alpha_b dr} \tag{3.7}$$

The constant value for the turbulent kinetic energy and the turbulent energy dissipation are calculated with approximate formulation for round pipes taken from [40]. The inlet value for the turbulent kinetic energy reads:

$$k = \frac{3}{2} (U_b I)^2 \tag{3.8}$$

where the turbulence intensity $I = 0.16 Re_{d_h}^{-\frac{1}{8}}$ and $Re_{d_h} = \frac{U_b d_h}{\nu_b}$ where the hydraulic diameter is equal to the pipe diameter. The inlet value for the turbulent energy dissipation is set equal to:

$$\epsilon = C_\mu \frac{k^{\frac{3}{2}}}{L} \tag{3.9}$$

where the turbulent length scale $L = 0.038 d_h$ for fully developed flow in a pipe.

The influence of the inlet boundary conditions on the final results was also tested. In particular, simulations were performed with all the inlet boundary conditions defined as constant fixedValue, where the constant values are evaluated as the average of the experimental results weighted over the void fraction, as it was done in [14] and [23].

The calculated values for the simulation with constant inlet boundary conditions are shown in Table 3.3. As already said, the experimental values used with the groovyBC option in most of the performed simulations can be found in Appendix C of Leung's thesis [4].

A turbulent profile for the inlet liquid velocity was also constructed and tested. The turbulent profile was constructed starting from the universal velocity turbulent profile in a pipe, so that the condition in Eqn. (3.6) was respected.

Table 3.3: Constant inlet boundary conditions used in some simulations

Properties	**Test Cases**			
Liquid superficial velocity j_l [m/s]	0.3	0.64	1.1	2.0
Gas superficial velocity j_g [m/s]	0.09	0.09	0.16	0.16
Real liquid velocity U_b [m/s]	0.357	0.698	1.170	2.095
Real gas velocity U_a [m/s]	0.471	0.806	1.312	2.193
Void fraction α_a	0.159	0.084	0.06	0.045
IAC [m^{-1}]	250	121	100	74
Turbulent kinetic energy k_b [m^2/s^2]	0.00050	0.00162	0.0040	0.0111
Turbulent dissipation ϵ_b [m^2/s^3]	0.00105	0.00610	0.02366	0.1092

The conclusion of these tests is that the radial distribution of the inlet boundary conditions are not affecting the fully developed radial distribution of the void fraction and of the velocities, as expected.

The usage of the boundary conditions in the thesis is consistent with Michta‘s thesis [14] and Rzehak‘s paper [23]. The biggest difference consists in the use of non-uniform boundary conditions thanks to the usage of the *groovyBC* libraries. The use of groovyBC proved to be stable and efficient, thanks to the relative simplicity of use.

CHAPTER 4

RESULTS AND DISCUSSION

This chapter shows the results obtained from the simulations performed. The simulation results are compared with the experimental data at different axial locations z/D = 62, 112. As already written in Chapter 3, four different adiabatic air-water cases were simulated with different liquid and gas superficial velocities. The goal is to identify deficiencies in the code that need to be addressed in its future development and to validate it.

The test case with $j_l = 0.64$ m/s $j_g = 0.09$ m/s is considered as the reference case because it has intermediate values of the void fraction, gas and liquid superficial velocities. Furthermore it shows a void fraction evolution along the axis going from a wall-peaked to a core-peaked void distribution. Therefore most of the sensitivity tests will be performed using this test case.

The utilized mesh for the test case with $j_l = 0.64$ m/s $j_g = 0.09$ m/s consisted of 40 radial cells and 100 axial cells without grading (if not specified differently). A small time-step (order of magnitude 10^{-4} s) was used in order to keep the Courant number smaller than unity during the transient simulation. In particular the Courant number was always kept smaller than 0.2 in all simulations in order to avoid numerical problems.

4.1 Checker-board instabilities

The vast number of performed simulations showed that checker-board instabilities in the radial direction can be found if the mesh is coarse in the radial direction.

Mesh refinement proved to be beneficial for oscillations, which disappear for sufficiently fine meshes. The mesh refinement increases the computational effort, especially if 3D geometry simulations are required, and that is the main reason why a vast amount of time and work was spent on the understanding of the nature of the oscillations observed in the results for simulations with coarse meshes.

It should also be said that the validity of some models in near wall cells is limited and more studies are required to assess the effect of this limitation on the validity. In particular, the limitation is linked to the y^+ parameter which was less than 20 in most of the simulations. Indeed the wall functions in the turbulence model are valid only for $y^+ > 30$ (a limitation equal to 20 is considered in the thesis, even if the flow starts to be in the buffer layer). This limitation implies that a coarse mesh (with around 8 radial uniform cells) should be used, because the pipe diameter is small.

However the use of coarse meshes produce checker-board instabilities in the radial direction and this is the main focus of this chapter. The effect of a coarse mesh on checker-board instabilities in the radial direction is shown in Figure 4.1.

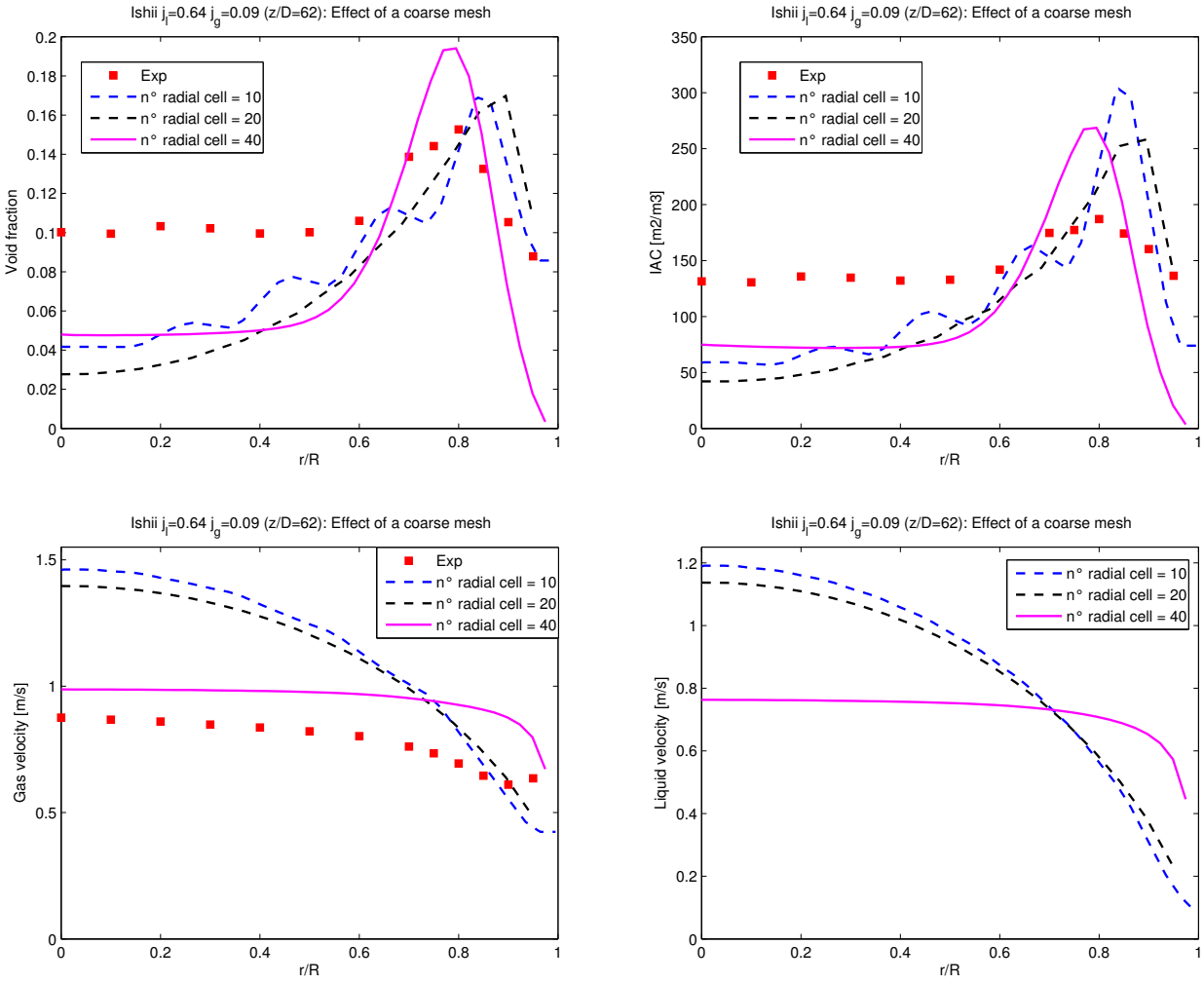

Figure 4.1: Effect of a coarse mesh on the results.

Oscillations from cell to cell could be observed in the void fraction, the IAC, mean Sauter bubble diameter and the radial component of the gas velocity. In particular oscillations around the value zero in the radial component of the gas velocity were observed, as shown in Figure 4.4f. No visible oscillations were observed in the axial component of the gas and liquid velocity, as shown in Figure 4.1.

Furthermore, the use of a fine mesh causes a change in the axial gas and liquid velocity profiles, as shown in Figure 4.1. The coarse mesh predicts a laminar velocity profile, while a finer mesh predicts a turbulent profile which shows a better agreement with experimental data. This change in velocity profile is probably connected to the limitation of the validity of the turbulence model in near-wall cells, but further studies will be required in order to have a clear understanding of the phenomena.

This behavior was never observed and reported in previous studies with previous version of this code (e.g. [1] and [14]). The problem has remained hidden until now, probably due to the fact that few adiabatic simulations were performed and results were plotted with inappropriate plot settings, as discussed in Chapter 4.1.1.

Many sensitivity studies and tests were performed in order to determine the nature of these instabilities and to eliminate them, since no clear solution could be found in the literature. The results and a short discussion of these sensitivity tests is shown in the following sub-chapters.

4.1.1 Influence of the plotting setting

The plot setting was previously wrong and results were smoothed through interpolation in the plotting procedure.

In this master thesis, the results were obtained through the *sample* OpenFoam utility and one value (the value of the CV center) for each control volume was plotted. Conversely the results in [1] were obtained with an oversampling, i.e. more than one value per cell was computed and therefore the values were calculated with the usage of interpolation between different cell values. This helped to cancel out oscillations with coarse meshes, as verified in a few cases.

The use of the correct plot setting with a mesh of ten radial cells reveals the presence of checker-board instabilities in the radial direction (not in the axial direction), as shown in Figure 4.2. In this figure, the results obtained with the solver used in Kai Fu's article [1] (called in the plot: α_b *old*) and the current solver are compared. It is clear that the instabilities were not caused by the change in modeling implemented during the thesis work, but they were preexistent in the previous versions of the solver.

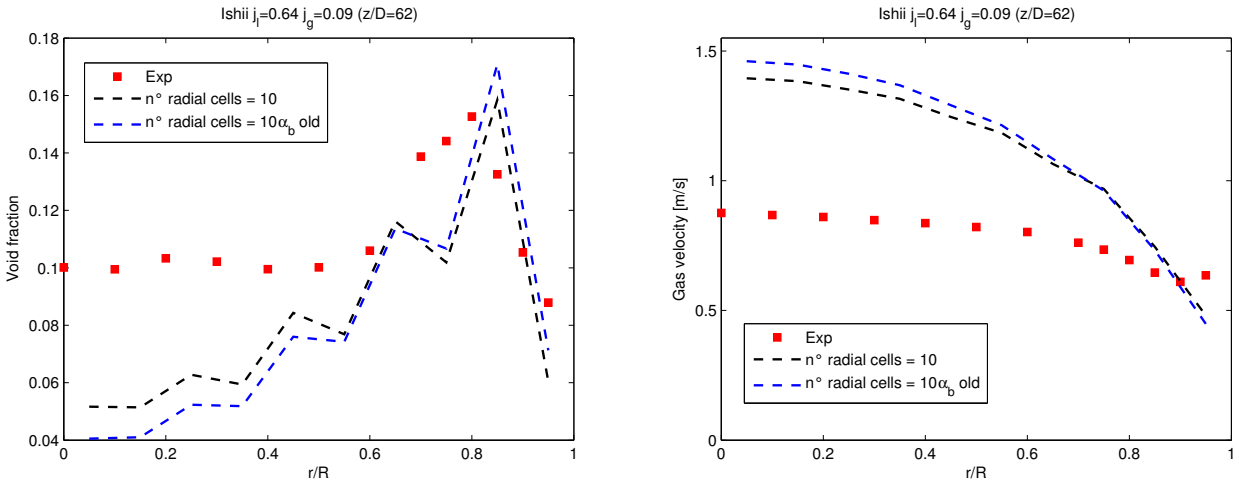

Figure 4.2: Influence of different approaches on the treatment of the momentum equation.

However, the radial oscillations were slightly amplified when the treatment of the interfacial momentum transfer term was modified, as shown in Figure 4.2. This modification (i.e. the removal of the liquid void fraction α_b from the interfacial momentum transfer term) was already discussed in Chapter 2.5. It is necessary in order to have a consistent formulation of the interfacial momentum transfer term and in order to be in line with most of the literature (e.g. [14], [8], [7]). This oscillation amplification shows how strong the coupling between the void fraction and the velocity field is. This strong coupling will be also discussed in Chapter 4.1.8.

4.1.2 Influence of numerical schemes

The use of different numerical schemes from the ones suggested in the literature [7] and implemented in the previous solver [1] was tested. The default numerical schemes used in the

simulations are shown in Appendix N.

Upwind numerical schemes were used for all divergence terms in the code, in order to introduce more numerical diffusion which could help to cancel out oscillations. However the results were not affected. The use of a different scheme for the gradient of the void fraction (equal to *cellLimited Gauss linear 1* instead of *Gauss linear*), as suggested on a CFD forum [41], was also tested but this approach was not successful.

Therefore the conclusion was that oscillations were not caused by an inappropriate setting of the numerical schemes in the file *fvSchemes*.

4.1.3 Influence of the compressibility terms

Regarding checker-board instabilities, it should also be noted that such instabilities were not reported in most of the analyzed literature papers (except for [8]) and no countermeasure was considered (e.g. in Weller [7]). Then it is interesting to observe that basically all the two-phase CFD simulations published so far in the literature were obtained using incompressible two-phase solvers which do not take into consideration the compressibility of the gas phase. Therefore these instabilities could be enhanced by the compressibility terms included in the equations, but further studies are required in order to exclude or confirm this hypothesis. However, the compressibility terms are necessary in order to validate the code in long vertical pipes, as the one considered in this master thesis. That is because the expansion of the gas bubbles along the height of the pipe has to be taken into account.

4.1.4 Effect of the IAC equation

Tests were also performed switching off the interfacial area concentration equation and setting a constant bubble diameter along the whole pipe (the value for the constant bubble diameter was taken from experimental data at z/D = 62 and also at z/D = 112). The results showed that the interfacial area concentration equation is not the cause for oscillations, since very small changes were observed if compared with previously done simulations (see also Figure 4.21). Therefore the cause for these oscillations lies in the implementation of the momentum and mass conservation equation.

4.1.5 Influence of turbulence models

In order to exclude the influence of the turbulence model on the oscillations, different approaches for the standard $k - \epsilon$ model were tested. These tests proved that all the different turbulence models behave similarly both in terms of oscillations with coarse mesh and in terms of convergence performances. Therefore the influence of the turbulence models on the oscillations was excluded. However, a further improvement of the code would be to change the current standard $k-\epsilon$ model into a turbulence model which is valid also in cells very close to the wall, such as the SST $k-\omega$ turbulence model used in [23].

4.1.6 Influence of near-wall damping

In the solver developed by Kai Fu [1], different terms in the implemented equations were damped (i.e. the term is set to zero) in near-wall cells for stability reasons. The damping options have

been done user-selectable in the file *moduleControls* in order to be easily removed and to test the influence on instabilities.

Damping options in near-wall cells for the lift force (as suggested in [42]), the turbulent dispersion force and the coalescence and break-up terms in the IAC equation were implemented.

The damping of the lift force affects significantly the stability of the code. If no damping for the lift force is used, then very unstable results (especially close to the wall) are observed both in space and time. Therefore the damping of the lift force proved to improve the stability of the code.

Conversely the damping of the turbulent dispersion force and of the coalescence and break-up terms in the IAC equation do not affect the results and stability performances. Therefore the results shown in this thesis are obtained with no damping in near-wall cells for the turbulent dispersion force and the coalescence and break-up terms in the IAC equation.

For a future development of the code, it would interesting to test the influence of a damping for the drag force term in near-wall cells as also suggested in [42].

4.1.7 Influence of the pipe diameter

The selected test cases were not simulated previously in the reviewed literature. More in general, no CFD simulations of two-phase flows in pipe with small diameters (= 25.4 mm) could be found in the literature. Most of the literature papers used experiments in larger pipes ($d_{pipe} > 38mm$, but usually $> 50mm$) and the models were adjusted accordingly in order to fit these experiments. Also, the solver used here was previously tested and validated against an adiabatic experiment with larger diameter (the DEDALE experiments $d_{pipe} = 38.1mm$) in [14] and [1].

Therefore the influence of the pipe diameter on instabilities was assessed. Few simulations were performed with a pipe diameter equal to 50 mm and constant inlet boundary conditions. These simulations were not performed in order to validate the code but only to check the presence of oscillations keeping the $y^+ > 20$ with an uniformly distributed mesh (as in Figure 3.2). Radial oscillations were observed, as shown in Figure 4.4.

Few simulations of the DEDALE case (described in [1] and [14]) were performed and oscillations were observed with the plotting setting described in 4.1.1.

Therefore oscillations are also present when larger pipe diameter are used and they are not caused by the use of a too small diameter.

4.1.8 Influence of the interfacial momentum transfer term treatment

As already discussed in Chapter 4.1.1, a strong coupling between the void fraction and the velocity field can be observed. Therefore different treatment of the interfacial momentum transfer term were implemented and tested in order to see the influence on instabilities. The different approaches for the momentum and pressure equation are explained in detail in Chapter 2.3.

The use of Michta's approach for the momentum equation produced no improvements concerning oscillations in the radial direction with coarse mesh. Similar results were consistently observed when a finer mesh was used, as shown in Figure 4.3. The same behavior was also observed for the other test cases with different gas and liquid superficial velocities.

The implementation and use of Rusche's approach for the momentum equation was more prob-

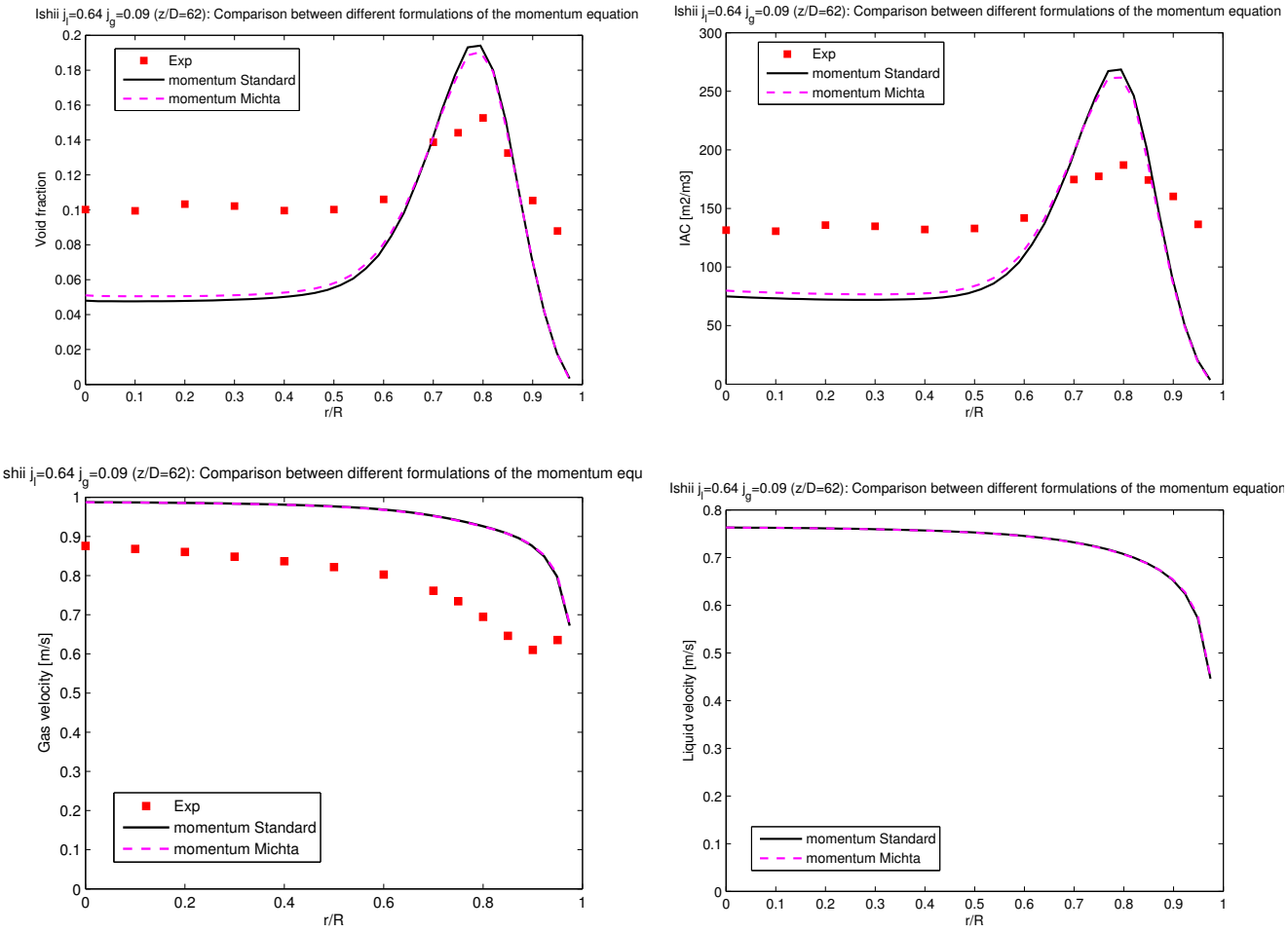

Figure 4.3: Influence of Michta's treatment of the momentum equation with a fine mesh.

lematic and it was not working properly initially. However these implementation problems were fixed and such an approach seems to considerably reduce the amplitude of the oscillations with a coarse mesh maintaining the same void fraction and velocity radial profiles as the previous approaches, as shown in Figure 4.4.

The results in Figure 4.4 were obtained for pipe diameter of 50 mm and 10 radial cells, which implies a $y^+ > 20$. Simulations with $y^+ > 20$ of Leung's experiments were also performed in order to test different approaches for the momentum equation and gave similar results, as shown in Figure 4.5. Different mesh setting were tested for these cases and they will be discussed in Chapter 4.1.10.

This phenomena suggests that one of the main source of oscillations is the turbulent dispersion force which should be treated very carefully in order to avoid momentum and continuity equation decoupling as already suggested by Rusche in [8] (more details in Chapter 2.3).

The influence of the turbulent dispersion force on oscillations was also observed with other sensitivity tests. Indeed the reduction in magnitude of the force causes reduction in oscillation amplitude. The reduction in magnitude of the turbulent dispersion force was obtained modifying the C_{td} parameter in the Lopez de Bertodano model from 1.0 to 0.5. Therefore the conclusion is that the turbulent dispersion force is one of the main term affecting oscillations with a coarse mesh.

Unfortunately oscillations did not disappear completely, especially close to the wall. Therefore

a Rhie-Chow like treatment of the lift force was also implemented and tested, as described in Chapter 2.3. This approach proved to reduce even further the amplitude of oscillations, which basically disappear in the radial distribution of the void fraction, IAC and mean Sauter bubble diameter, as shown in Figure 4.4 and 4.5. However, oscillations can still be observed in the radial component of the gas velocity around the zero value, as shown in Figure 4.4f. These oscillations are quite significantly reduced in amplitude if compared with previous approaches.

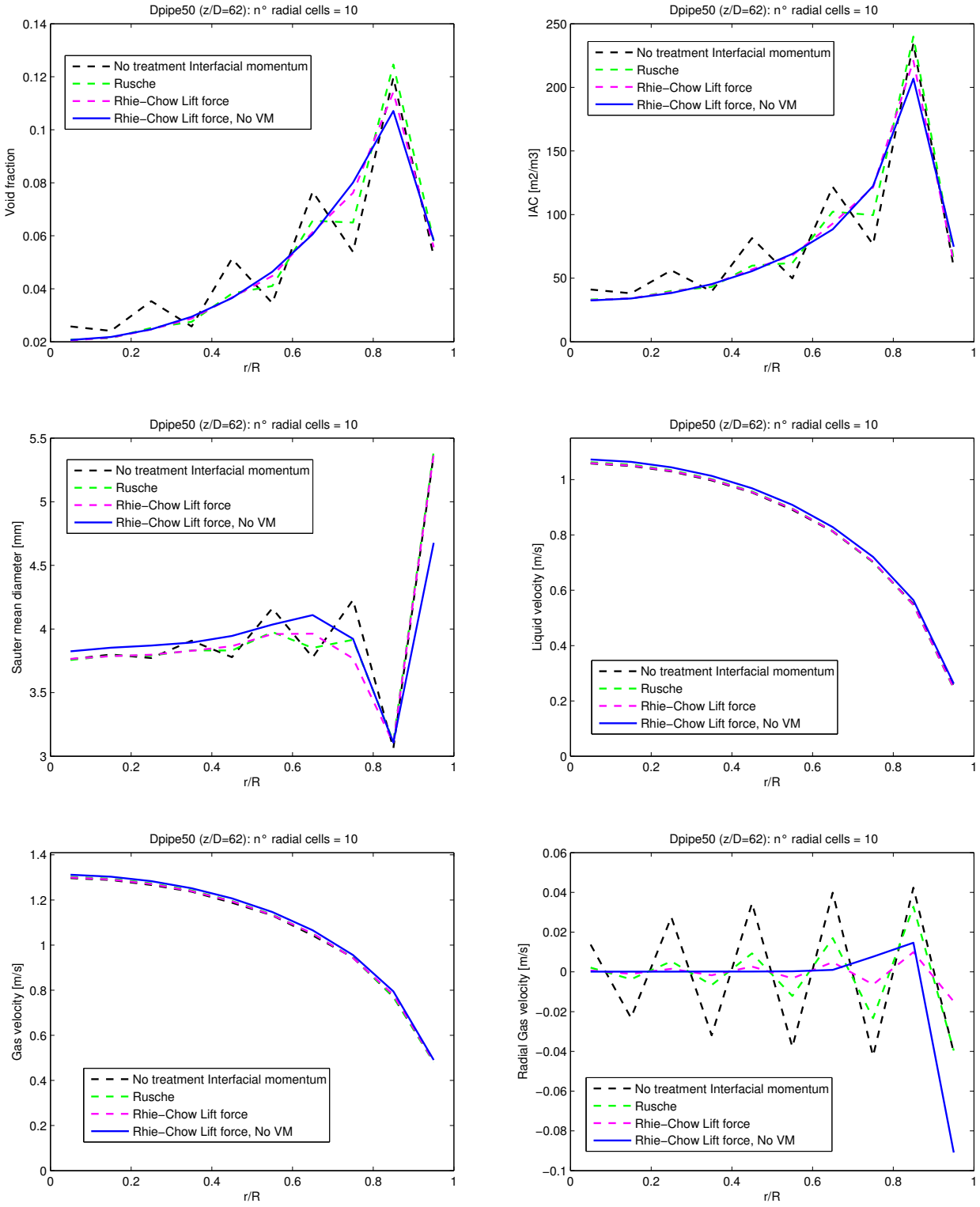

Figure 4.4: Influence of different treatments of the interfacial momentum transfer term.

The Rhie-Chow like treatment of the interfacial momentum transfer term proved to be stable and robust when $y^+ > 20$. However it should be pointed out that when this limitation is violated, unstable results (especially in near-wall cells) can be generated in some cases. This behavior seems to be consistent considering the fact that the turbulence model is not valid in near-wall cells when $y^+ < 20$. However, the stability of the Rhie-Chow like treatment of the interfacial momentum transfer term will need to be tested more accurately if the use of a turbulence model without limitation on the y^+ parameter will be used. Furthermore a change in velocity profile from laminar to turbulent was observed (as in Figure 4.1) when $y^+ < 20$. This shows that the solver is behaving very similarly to the previous solver without the Rhie-Chow like treatment of the interfacial momentum transfer term when $y^+ < 20$.

In summary, a Rhie-Chow like treatment of interfacial momentum transfer term (explicit drag, turbulent dispersion and lift force) in the momentum equation proved to be beneficial for reducing the numerical instabilities with coarse meshes and $y^+ > 20$. The appropriate treatment (Rhie-Chow like) of the interfacial momentum transfer term proved to cancel out radial checker-boarding with coarse meshes, however further studies will be required in order to improve and validate the approach.

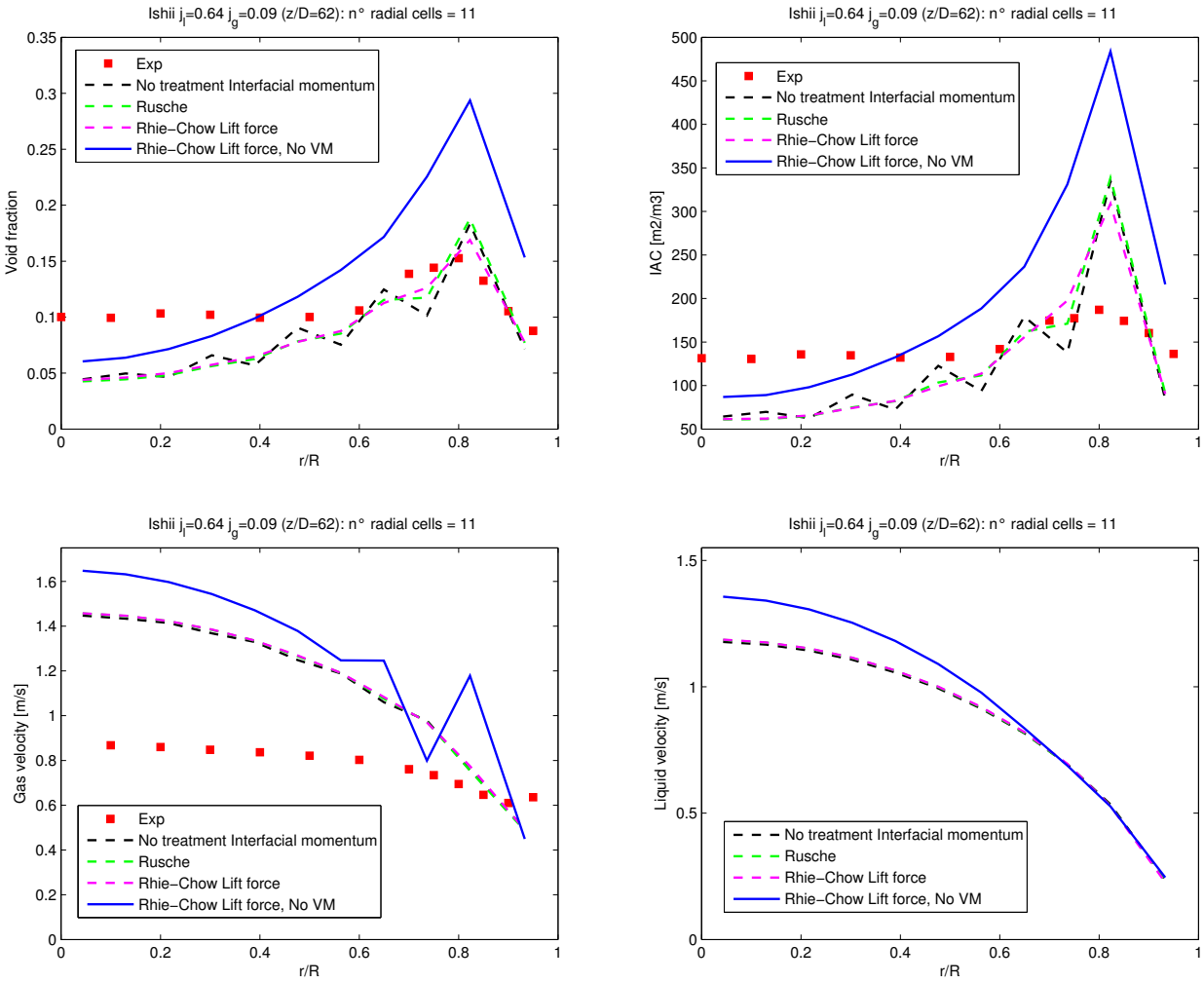

Figure 4.5: Influence of different treatments of the interfacial momentum transfer term (the removal of the virtual mass force can produce unstable results).

Also the virtual mass force will probably need an improved treatment in order to eliminate the residual oscillations in the radial component of the gas velocity, as shown in Figure 4.4f.

Indeed the removal of the virtual mass force (i.e. $C_{vm} = 0.0$) proved in some cases to be beneficial for the elimination of oscillations together with a Rhie-Chow like treatment of the interfacial momentum transfer term. However in some tests (e.g. see Figure 4.5), the removal of the virtual mass force has generated instabilities at the inlet which propagates along the pipe. The effect of the virtual mass force will be discussed in Chapter 4.1.9.

4.1.9 Influence of the virtual mass force

As shown in Figure 4.4, the removal of the virtual mass force helps to cancel out oscillations. However it was observed during the work that the virtual mass force is important for the stability of the code. The removal of the virtual mass force can generate very unstable results during some simulations, as shown in Figure 4.5 and 4.6. The instabilities are generated at the inlet of the pipe and propagate along the pipe. In particular, large values for the forces acting on the bubbles are observed in few cells at the inlet. These large values for the forces cause high gas velocities in few cells and these high velocities generate instabilities both in the spatial and the temporal scale.

The influence of the mesh setting on the instabilities associated with the removal of the virtual mass force was assessed, as shown in Figure 4.6.

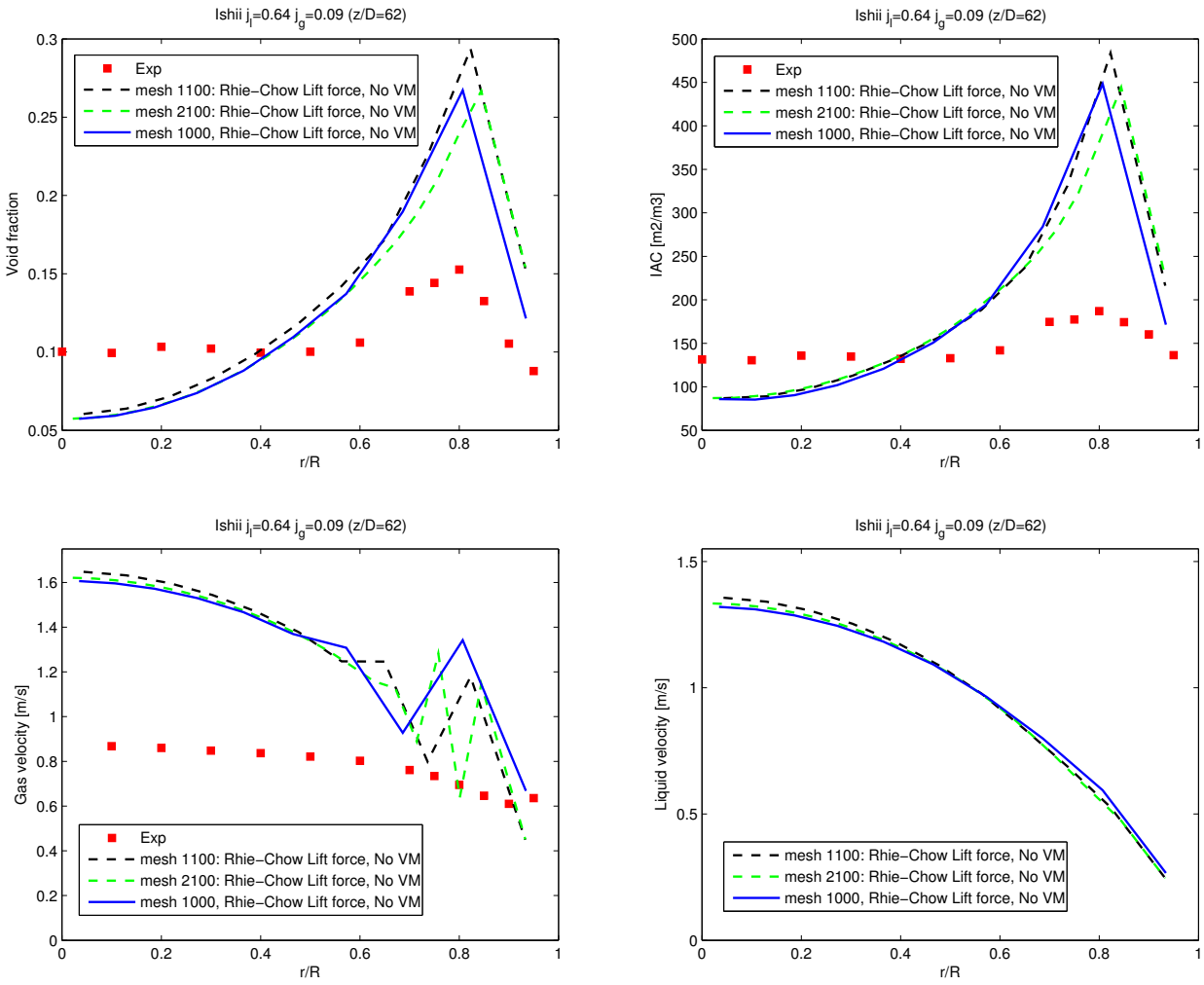

Figure 4.6: Mesh influence with the removal of the virtual mass force.

In Figure 4.6, all the simulated cases have $y^+ > 20$, but different mesh setting were used. The cases with the denomination *mesh 1100* and *mesh 2100* use the mesh setting in Figure 3.4 with respectively 11 and 21 radial cells. Case *mesh 1000* uses an uniform mesh with 10 radial cells, as in Figure 3.2. Unstable results were observed for all the tested mesh settings and it was therefore proved that instabilities were not generated by the use of a inappropriate mesh.

The influence of the virtual mass force on stability performances was also tested with other cases, $y^+ < 20$ and the different treatments of the interfacial momentum transfer term in the momentum equation. The conclusion is that the virtual mass force is important in order to obtain stable results with the current solver.

4.1.10 Mesh influence

As already stated, mesh refinement proved to be beneficial for oscillations, which disappear for sufficiently fine meshes. However, the limitation of validity of the turbulence model due to the y^+ parameter requires the use of a sufficiently large radial cell size close to the wall. Therefore different mesh setting were tested, maintaining $y^+ > 20$.

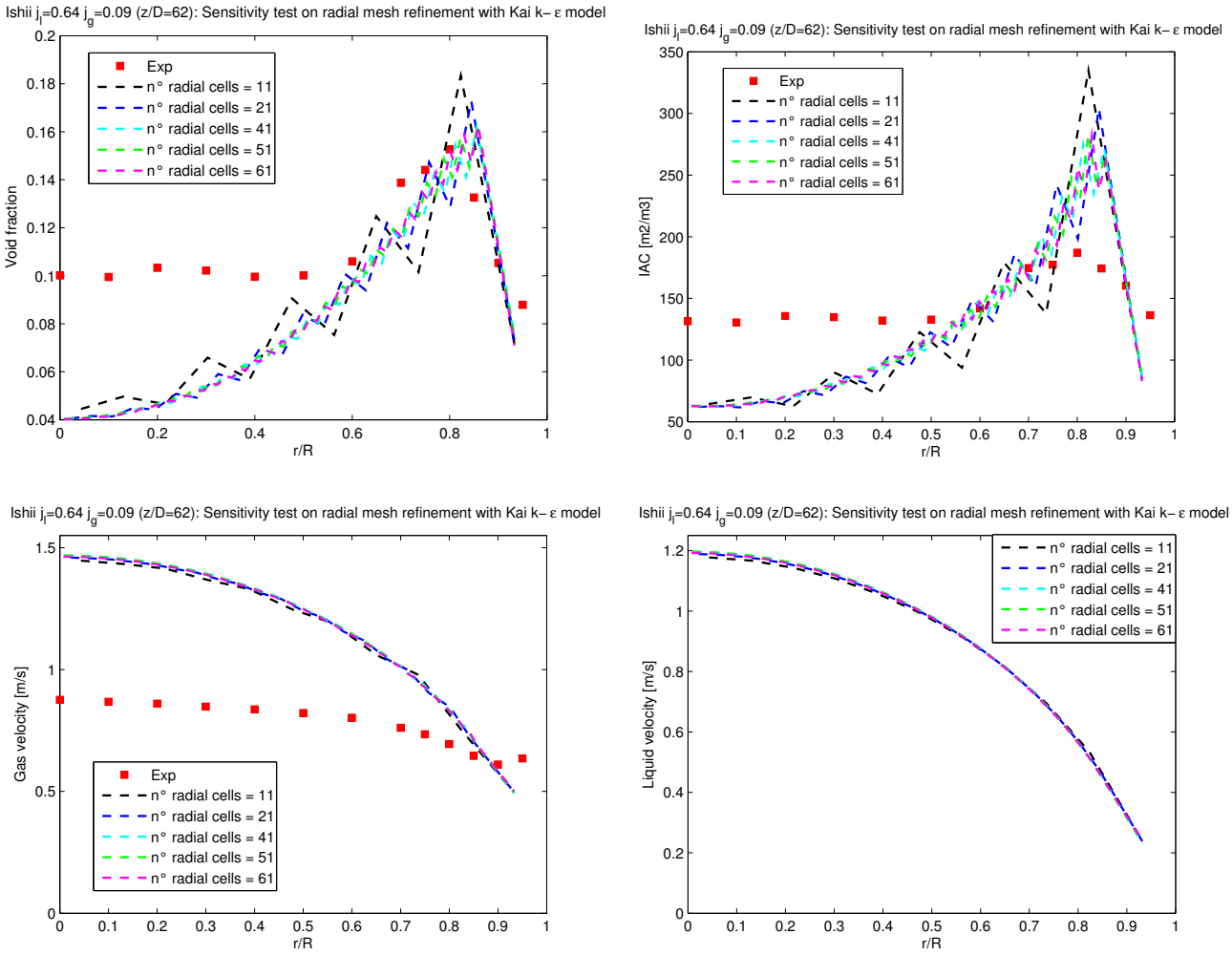

Figure 4.7: Influence of mesh refinement with the mesh setting in Figure 3.4 and no Rhie-Chow like treatment of the interfacial momentum transfer term.

The mesh setting in Figure 3.4 was tested with different radial mesh refinement in the bulk. Tests with this mesh setting were performed in order to see if reduction in oscillations could be obtain

with a mesh refinement in the bulk, but keeping the $y^+ > 20$. These tests were performed with no Rhie-Chow like treatment of the interfacial momentum transfer term. As shown in Figure 4.7, a reduction in oscillation amplitude could be observed with mesh refinement in the bulk region. However, oscillations are still there but with reduced amplitude. Therefore oscillations can not be removed keeping $y^+ > 20$ and using no Rhie-Chow like treatment of the interfacial momentum transfer term.

Tests with the same mesh setting were also performed with different treatment of the interfacial momentum transfer term, as shown in Figure 4.8. In particular, 21 radial and 100 axial cells were used.

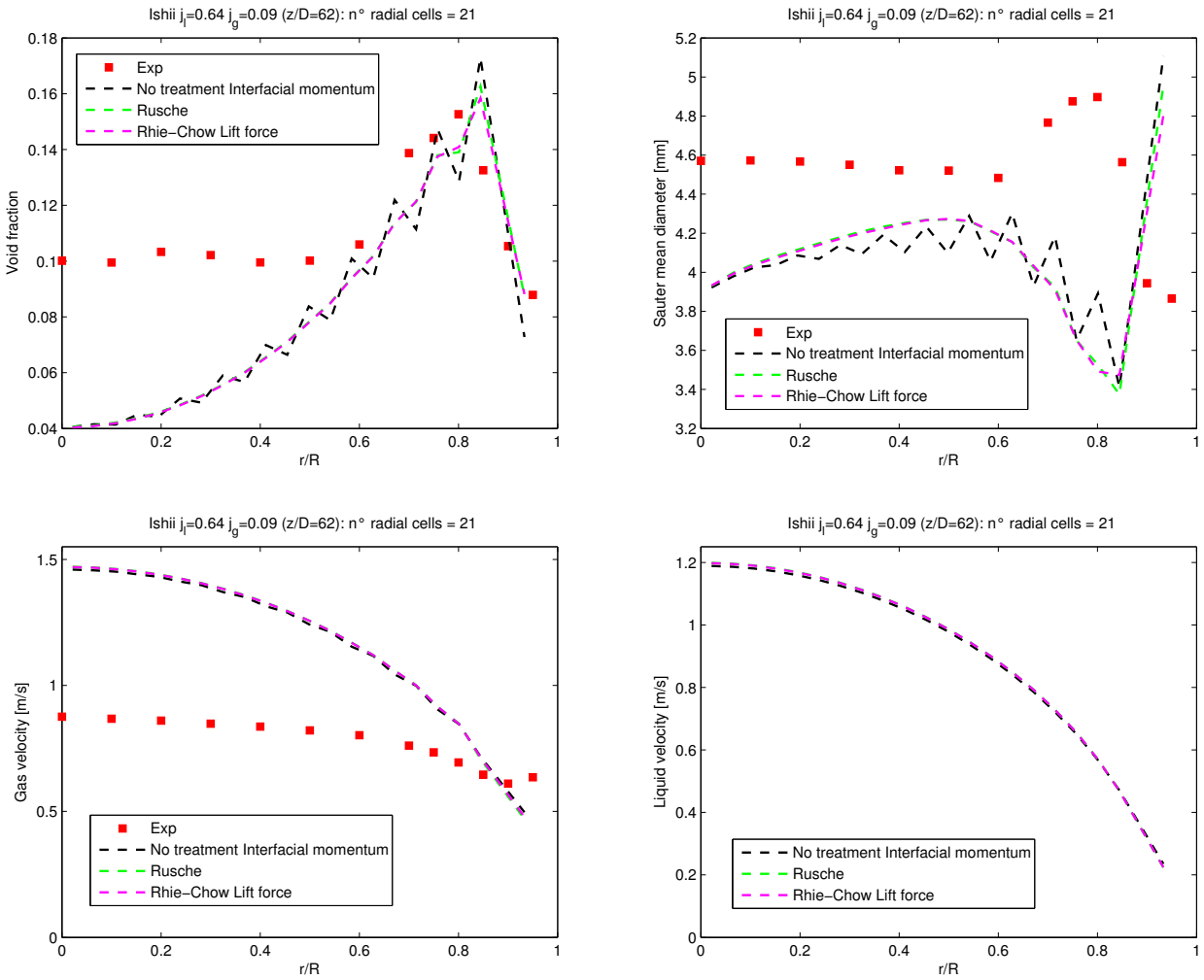

Figure 4.8: Influence of the mesh setting in Figure 3.4 with different treatment of the interfacial momentum transfer term.

As already shown in Chapter 4.1.8, an appropriate treatment of the interfacial momentum transfer term can reduce significantly oscillations. Instabilities can be observed where there is the transition between a large cell close to the wall to the small cell in the bulk. The use of this mesh setting is therefore not adequate, as expected. However, Figure 4.8 shows that the Rhie-Chow like treatment of the interfacial momentum transfer term produces stable results also with a fine mesh if $y^+ > 20$.

In the following chapters, the results obtained with fine uniform meshes (mainly with 40 radial and 100 axial cells) and no Rhie-Chow like treatment of the interfacial momentum transfer term

are presented. These results were obtained violating the y^+ limitation, but they can be useful in order to make comments and considerations on the implemented models.

4.2 Sensitivity tests of the turbulence modeling

Three different standard $k-\epsilon$ turbulence models were tested and similar results were obtained. The comparison is shown in Figure 4.9 and 4.10.

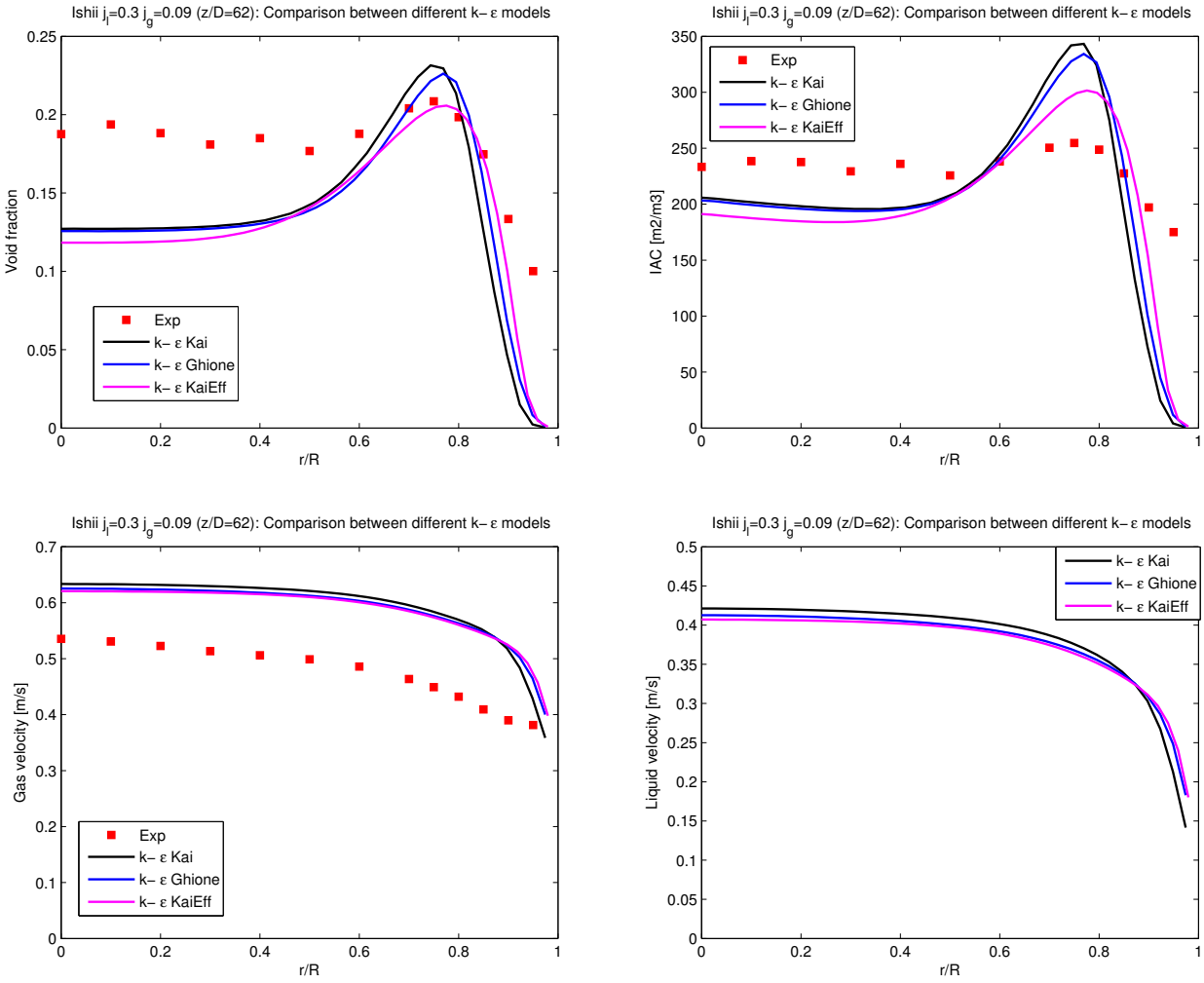

Figure 4.9: Influence of different standard $k-\epsilon$ turbulence models ($j_l = 0.3$ m/s $j_g = 0.09$ m/s).

The results were obtained with 60 radial and 120 axial cells (for $j_l = 0.64$ m/s) and with 50 radial and 100 axial cells respectively(for $j_l = 0.3$ m/s). The simulations differ only in the choice of the turbulence model and a similar behavior can be observed if the results at z/D = 112 are compared.

No conclusion can be drawn concerning which model is the best. Therefore a more-extensive study is needed in the future.

The following sensitivity studies were simulated using Kai‘s version of the standard $k-\epsilon$ model because this model was the one originally implemented in the solver and it was consequently more widely tested. However, it should be underlined that all the three approaches behave similarly with a coarse mesh and show similar convergence performances when the mesh is refined.

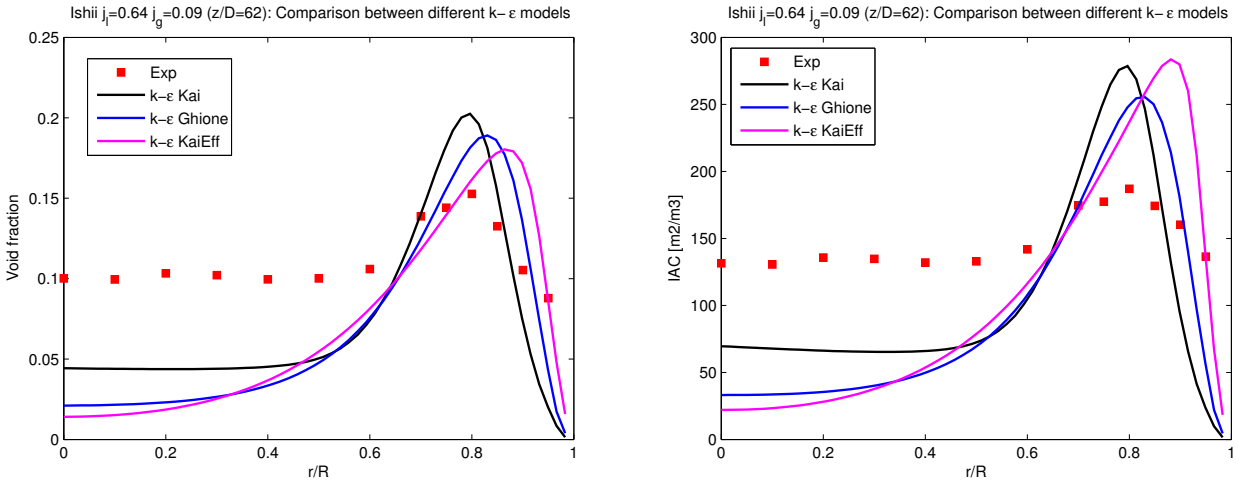

Figure 4.10: Influence of different standard $k-\epsilon$ turbulence models ($j_l = 0.64$ m/s $j_g = 0.09$ m/s).

The influence of the turbulence response coefficient C_t was also studied and simulations were performed both neglecting the turbulence of the gas phase (i.e. $C_t = 0.0$) and assuming a constant $C_t = 1.0$ as suggested by Rusche in [8]. As expected, almost no differences between the two cases were observed as shown in Figure 4.11. Indeed, although the effective kinematic gas viscosity in the Reynolds averaged turbulent stress term can be increased in average up to one order of magnitude, the small ratio of gas and liquid densities ($\propto 10^{-3}$) makes this increase in viscosity negligible, as also stated in [23]. This consideration is valid when applied to water-air flows with small ratio of gas and liquid densities but this could be false if other combination of fluids are considered.

In the following simulations, the turbulence of the gas phase will be neglected, i.e. $C_t = 0.0$.

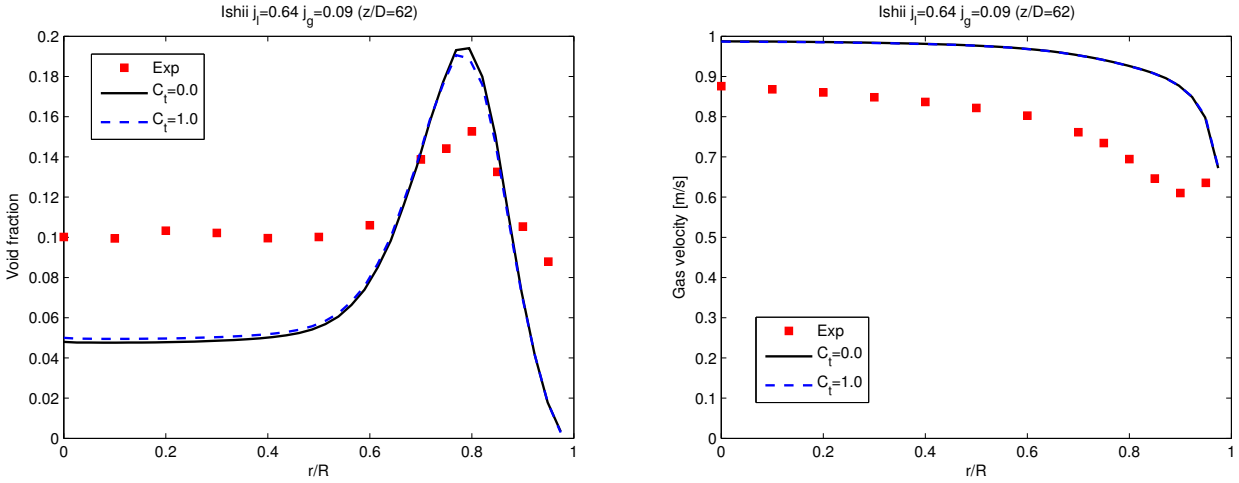

Figure 4.11: Influence of the turbulence response coefficient C_t ($j_l = 0.64$ m/s $j_g = 0.09$ m/s).

4.3 Sensitivity tests of the interfacial momentum transfer closure laws

Different models for the forces in the interfacial momentum transfer term in the momentum equation were implemented and tested. The influence of the different models will be discussed in the following sub-chapters.

4.3.1 Turbulent dispersion force

As already discussed in Chapter 4.1.8, the turbulent dispersion force has a huge impact on the stability performance of the code. Different models were implemented and tested, as shown in Figure 4.12 and 4.13.

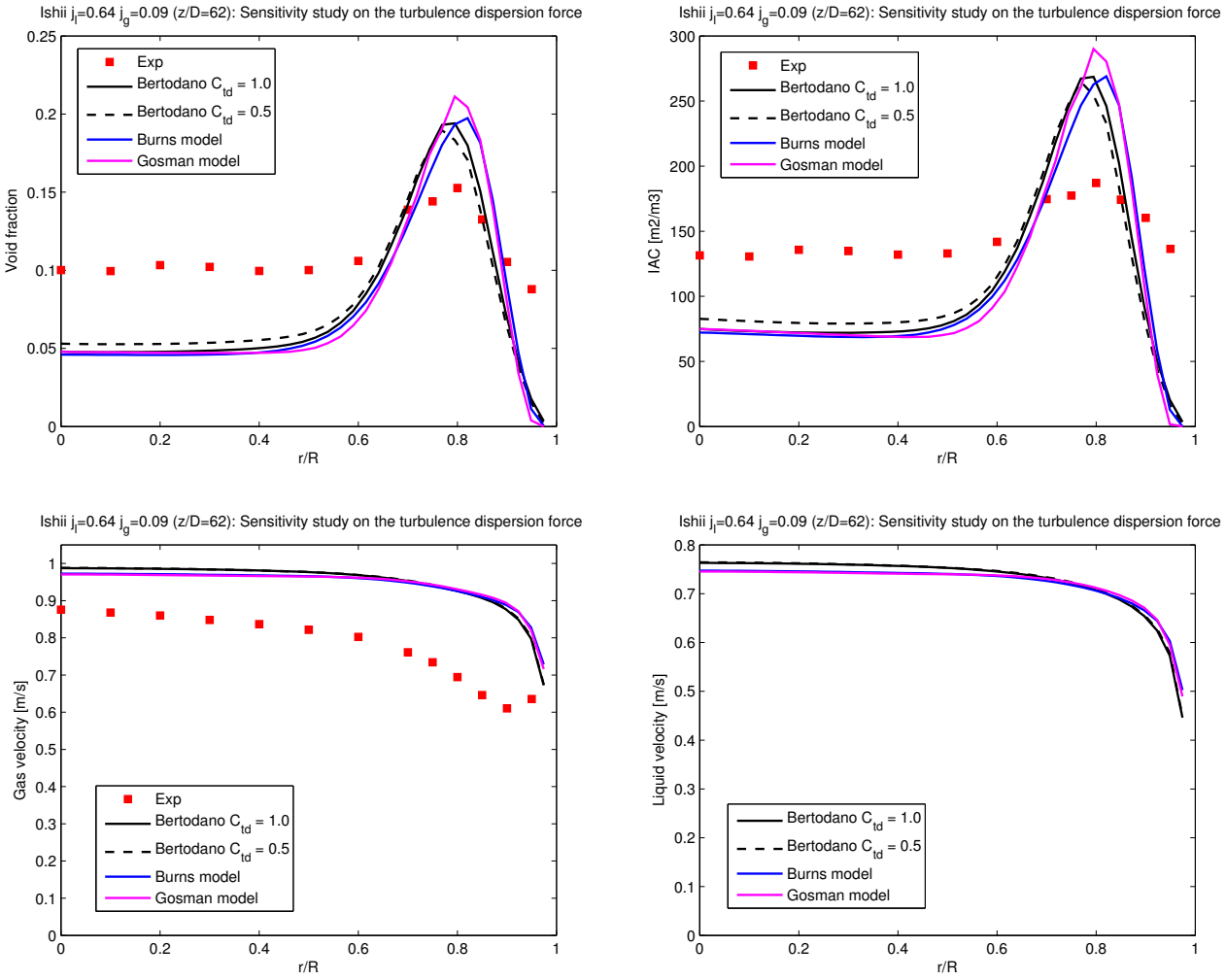

Figure 4.12: Sensitivity study of different turbulent dispersion force models ($j_l = 0.64$ m/s $j_g = 0.09$ m/s).

It is shown that the different modeling of the turbulent dispersion force provides very similar results. Consistent small differences between the different models can be observed, indeed an increase in turbulent dispersion force generates smoother solutions in the void fraction radial distribution and conversely for a decrease in turbulent dispersion force. However, it should be said that the Gosman and Burns model resulted in more difficulties to converge and also required

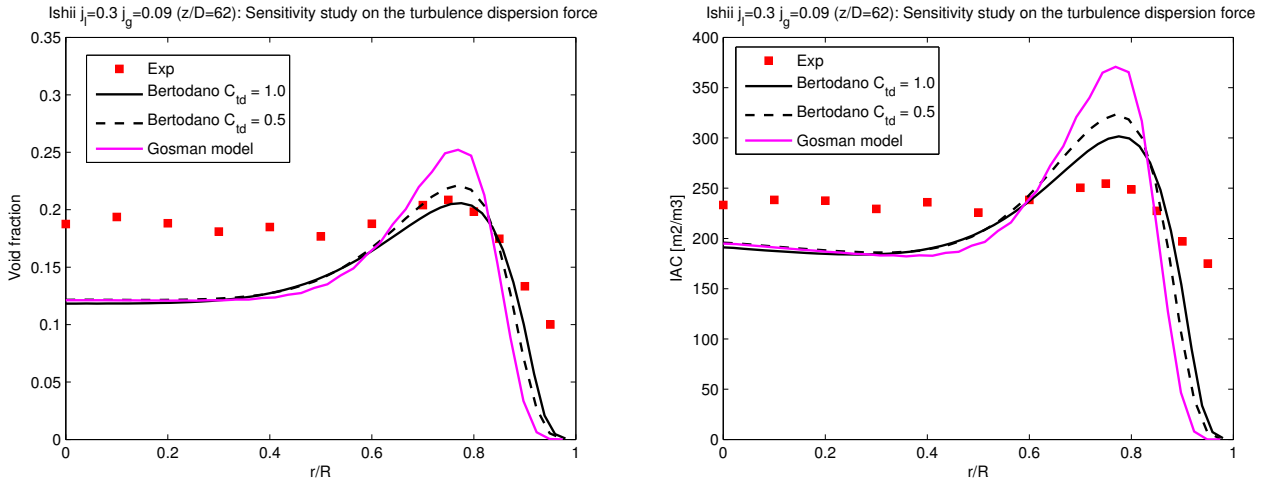

Figure 4.13: Sensitivity study of different turbulent dispersion force models ($j_l = 0.3$ m/s $j_g = 0.09$ m/s).

finer meshes. Therefore the Lopez de Bertodano model was used as the standard model in the following simulations.

The reason for the small influence of the turbulent dispersion force on the results is shown in Figure 4.14 and 4.15. Here the radial and axial components of the different forces acting on the gas bubbles are plotted along the radius.

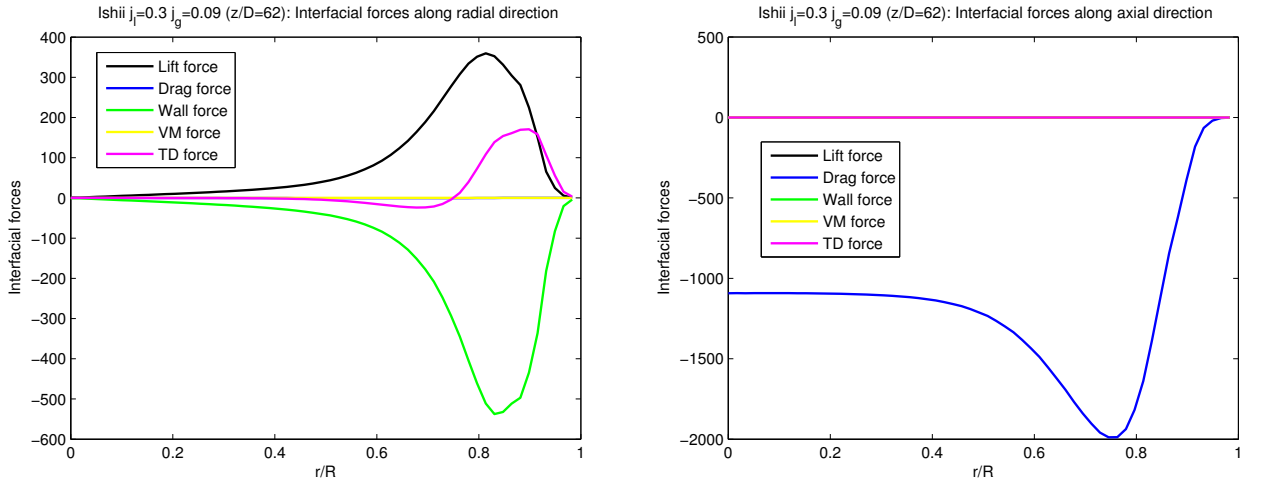

Figure 4.14: Radial and axial component of the interfacial forces ($j_l = 0.3$ m/s $j_g = 0.09$ m/s).

In the radial direction, the lift and wall lubrication forces are the dominant terms and then determine the radial distribution of the void fraction. The magnitude of the turbulent dispersion force is small compared to the lift and wall lubrication forces, but not negligible especially where strong void fraction gradients occur (i.e. in the wall peak region).

4.3.2 Drag force

In the axial direction, the only relevant force is the drag force which is counterbalanced by the gravity-driven buoyancy term causing a constant terminal velocity for the rising bubbles.

The axial component of the drag force is negative because it is acting in the opposite direction of the z direction (i.e. in the direction opposite to the gas flow). As shown in Figure 4.14 and 4.15, the only force which determines the gas velocity profile is the drag force.

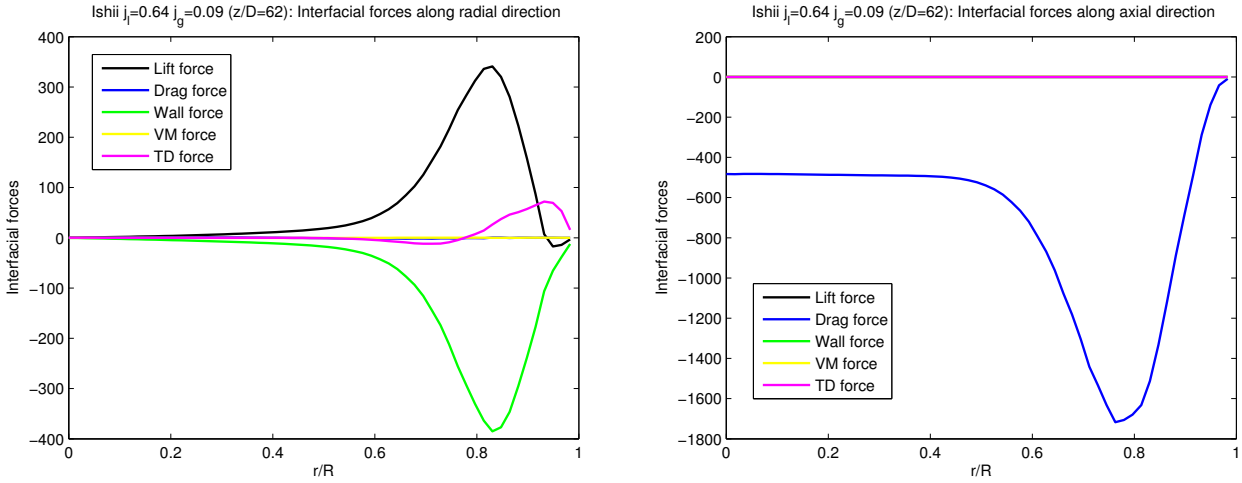

Figure 4.15: Radial and axial component of the interfacial forces ($j_l = 0.64$ m/s $j_g = 0.09$ m/s).

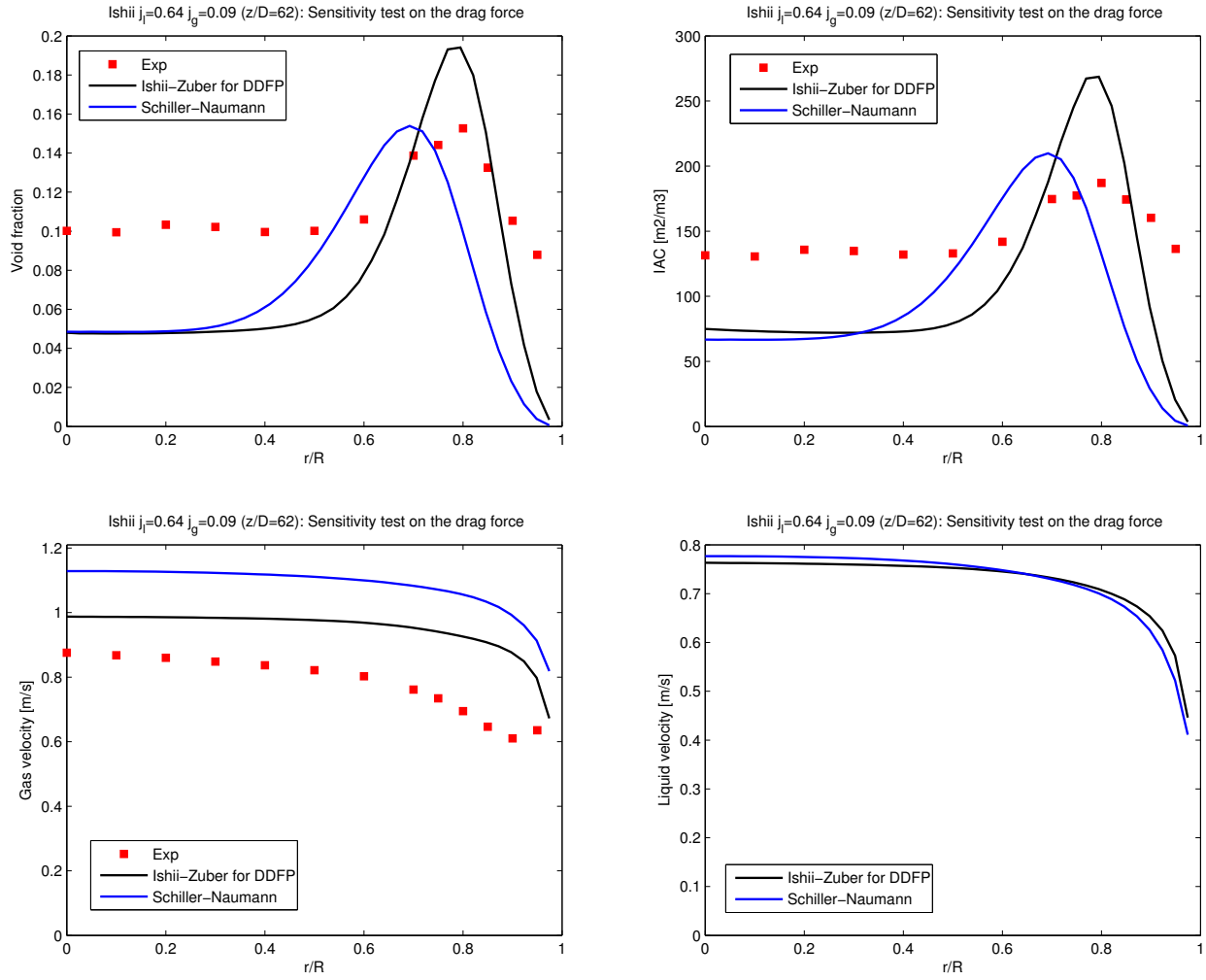

Figure 4.16: Sensitivity study on different drag force models ($j_l = 0.64$ m/s $j_g = 0.09$ m/s, z/D = 62).

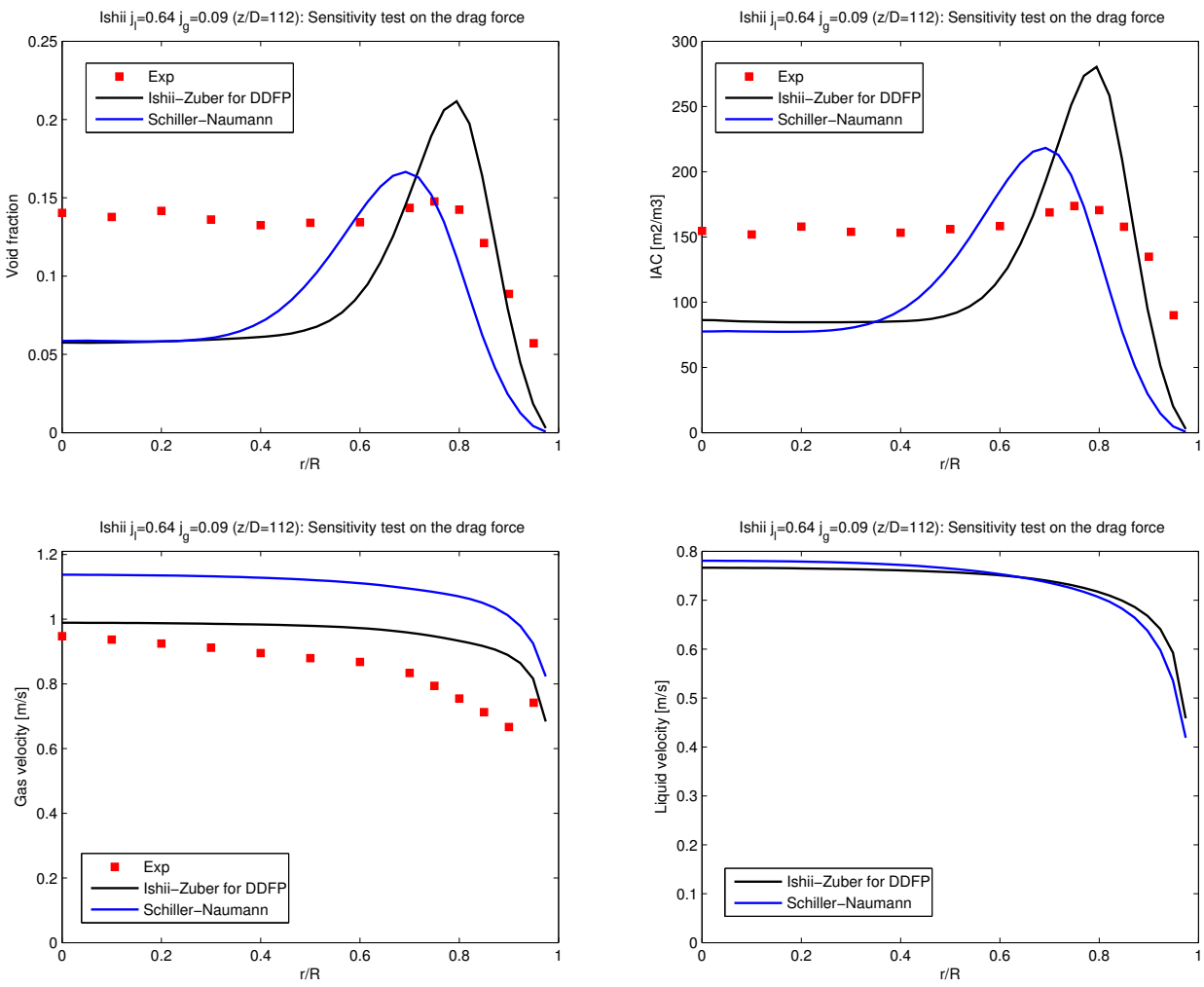

Figure 4.17: Sensitivity study on different drag force models (j_l = 0.64 m/s j_g = 0.09 m/s, z/D = 112).

Initially a significant over-prediction of the gas velocity was observed during simulations. Therefore a literature study was performed and the validity of the implemented drag models was assessed. The only implemented models were the Schiller-Naumann, the Wen-Yu and the Ishii-Zuber model which were developed and are valid only for solid particles. Correlations valid for fluid particles were then implemented in order to see the improvements in gas velocity prediction. The obtained improvements are shown in Figure 4.16 and 4.17. A significant improvement in gas velocity prediction can be observed comparing the Schiller-Naumann and Ishii-Zuber model for densely distributed fluid particles (Ishii-Zuber for DDFP). The explanation for such improvement lies in the fact that the Ishii-Zuber model for densely distributed fluid particles predicts a drag coefficient C_d much bigger than the Schiller-Naumann model, since the Ishii-Zuber model takes into account the effect on the drag coefficient of the deformation of the gas bubbles from spherical to an ellipsoidal or cap shape (see more details in Chapter 2.5.1). The gas velocity is still over-predicted probably due to deficiencies in the modeling of the other forces and the under-prediction of the mean Sauter bubble diameter, but the gas velocity prediction is significantly improved.

Even if no experimental data for the liquid velocity distribution are available, it is interesting to notice that the liquid turbulent velocity profile is much less affected by the change in drag model as expected. However, with the improved drag model a stronger coupling between the liquid and gas velocity was observed due to the larger value of the drag coefficient that lead to a smaller

value of the relative velocity between the two phases (reduction of the order of 10 - 20 %).

The change in gas velocity and relative velocity between the two phases also affects the other terms in the interfacial momentum transfer causing a change in the radial distribution of the void fraction and of the interfacial area concentration. It seems like the new drag model predicts the radial position of the wall peak better, but at the same time it leads to an over-prediction of the magnitude of the void fraction at the position of the wall peak. However these considerations are misleading, because the radial distribution of the void fraction is determined by the lift and wall lubrication force and not by the drag force. Therefore once a good prediction of the gas velocity (and consequently of the relative velocity) can be obtained, the different lift and wall lubrication models can be tested to see the influence on the radial void fraction distribution.

In all the simulations shown in this master thesis, the Ishii-Zuber model for densely distributed fluid particles was used. Similar results could be obtained with the Ishii-Zuber model for sparsely distributed fluid particles as already discussed in Chapter 2.5.1 and tested in some cases.

4.3.3 Lift and wall lubrication force

Different models for the lift and wall lubrication force have been implemented and tested as shown in Figure 4.18 and 4.19.

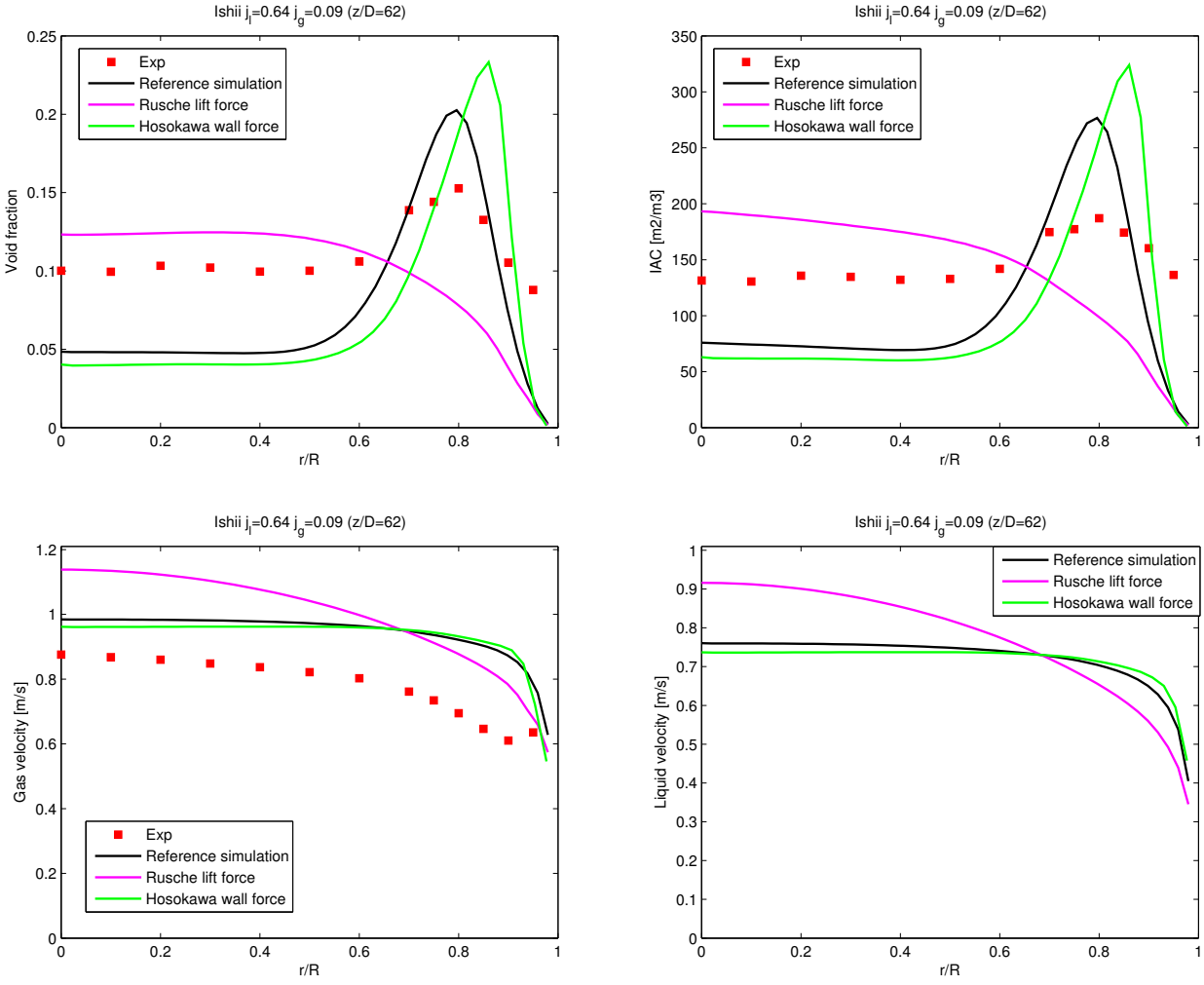

Figure 4.18: Influence of different models for the wall lubrication and lift force (z/D = 62).

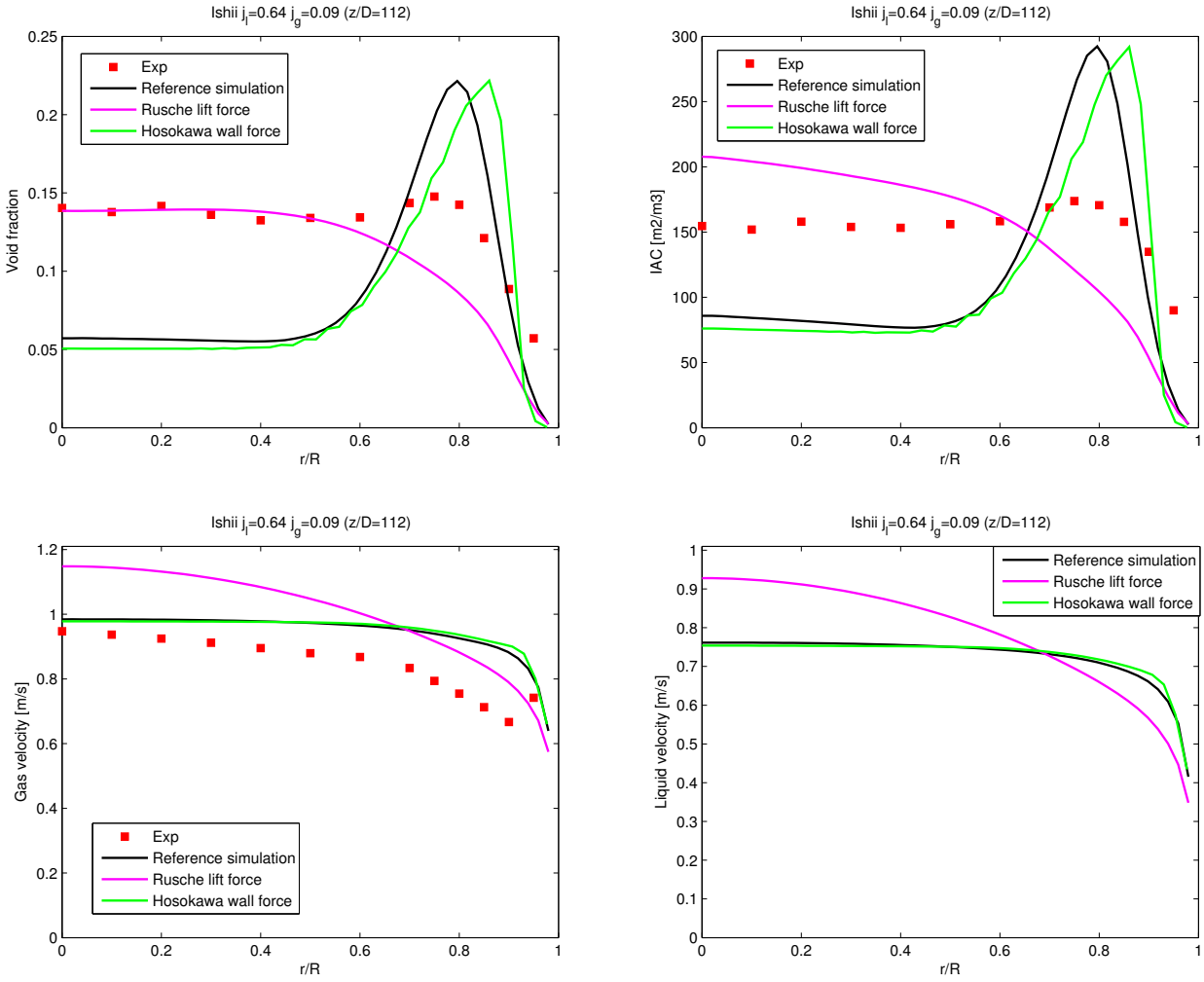

Figure 4.19: Influence of different models for the wall lubrication and lift force (z/D = 112).

The reference simulation in Figure 4.18 and 4.19 uses the Tomiyama lift and wall lubrication forces, which are the commonly used models in CFD two-phase codes.

The modeling of the lift and wall lubrication force are the weak points of the current solver, since no well-established formulation for these forces are available in the literature. In particular the coefficients of these forces (which are very important for the radial distribution of the void fraction) were fitted to experiments for single bubbles in laminar flows and are now applied to the simulation of turbulent two-phase flows with many bubbles. Therefore the prediction of the radial void fraction profile provided by these models are not fully satisfactory. After an extensive literature study two new models were implemented in the solver and tested: the Rusche model for the lift coefficient and the Hosokawa model for the wall lubrication force coefficient.

The tests performed with the Hosokawa wall lubrication force showed similar results if compared with the Tomiyama model. However the Hosokawa model computes in average a smaller wall lubrication force coefficient C_{wl} in the range of interest of the performed simulations (bubble diameter range of 4-5 mm) as shown in Figure 2.3. Therefore the Hosokawa model computes a smaller wall lubrication force in amplitude which causes, consistently, a shift of the wall peak to a position closer to the wall.

It should be also noted that the usage of the Hosokawa wall lubrication model requires a finer mesh in order to remove the radial oscillations. This is probably caused by the fact that the

Hosokawa model predicts a smaller wall lubrication force and therefore the turbulent dispersion force has a larger influence on the radial void fraction distribution.

A more detailed discussion on the lift force is needed, because it was found that a change in models can lead to large changes in the void fraction distribution as shown in Figure 4.18 and 4.19. The newly implemented Rusche lift force proved to predict the radial distribution of the void fraction better when the experimental data show a core peak instead of a wall peak, as shown in Figure 4.20. Indeed the Tomiyama lift force is not able to accurately predict a core peaked void fraction distribution, as already observed in [14], [1] and [23] and shown in Figure 4.20.

It is interesting to notice that the Rusche lift model predicts laminar velocity profiles even if the liquid Reynolds number $Re_l = \frac{U_b D_{pipe}}{\nu_l} \approx 1.8\ 10^4$ which means that the flow is turbulent.

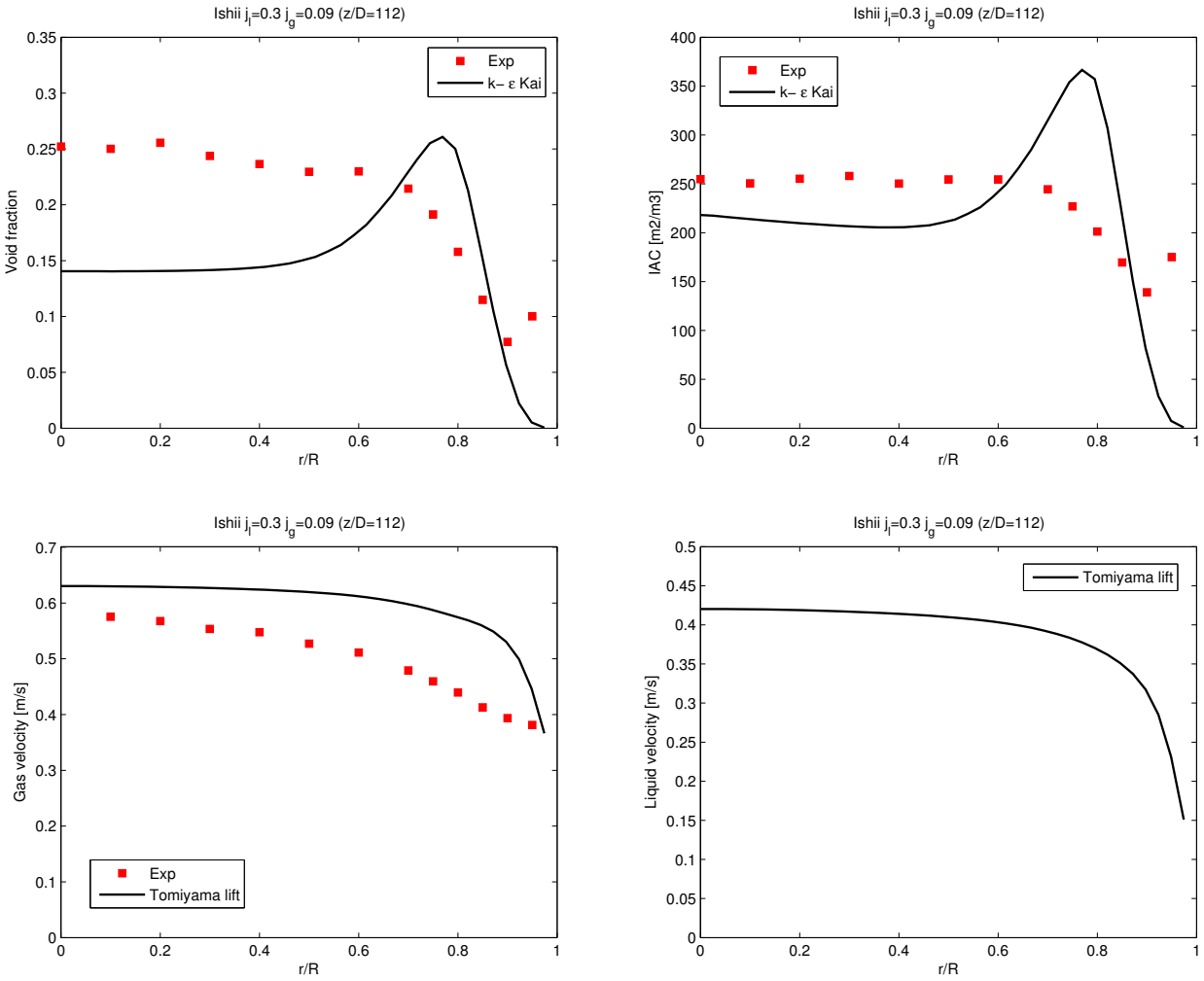

Figure 4.20: Simulation results for $j_l = 0.3$ m/s $j_g = 0.09$ m/s, z/D = 112 with Tomiyama lift model.

Simulations showed that the Tomiyama lift model used together with the Tomiyama (or Hosokawa) wall lubrication model can only predict a wall-peaked void distribution and the Rusche model seems to predict only core peaked distributions. Therefore further development of the lift models is necessary.

Sensitivity tests with a constant lift coefficient C_l were also performed and they showed that a wall peaked void distribution can be obtained if C_l is positive and ranges between 0.1 and 0.3 (in particular, the void fraction profile obtained with the Tomiyama lift model can be approximately be reproduced if $C_l = 0.25$ is used). Conversely core-peaked distributions can be obtained if C_l is

negative or null and ranging between -0.2 and 0.0. The solution obtained with the Rusche model can be approximatively reproduced if $C_l = 0.0$ is used and a negative coefficient leads to even more core-peaked solutions.

In the literature most of the authors uses the Tomiyama lift model as the reference model for their simulations and therefore all the simulations in this master thesis have been obtained with this model.

These deficiencies have already been pointed out by other authors (e.g. [43], [44]) which stated that the only way to predict the transition from wall-peaked to core-peaked distribution was the usage of a Multiple Bubble SIze Group (MUSIG) approach. According to these authors the use of a mono-disperse bubble size for the modeling of the gas phase (as the one used in this master thesis) is not sufficient to predict the void fraction distribution for all flow conditions and therefore a MUSIG approach is needed.

The MUSIG approach divides the gaseous dispersed phase into a certain number of groups characterized by their own bubble size, velocities and therefore lift coefficient. In particular the bubble diameter range is divided into many different classes governed by conservation equations. Closure laws for bubble breakup and coalescence are required for connecting the equations in the different groups. The conceptual idea of this approach is very similar to the Multi-Group theory for neutron transport and more details can be found in [44]. This approach is very promising and good agreement with experiments were shown in [44], especially regarding the radial separation of large and small bubbles which has been observed during experiments. This phenomena (i.e. large bubbles, $D_s > 5.8mm$, in the bulk and small bubbles close to the wall) can not be predicted by a mono-disperse bubble size approach. Furthermore the MUSIG approach seems to be the only possible solution in order to predict the radial void fraction distributions showing both a wall and core peak.

Therefore the implementation of a MUSIG approach is suggested in order to further develop the code, even if this modification will significantly increase the complexity of the model.

4.4 Sensitivity tests on the influence of the IAC equation

The interfacial area concentration equation plays an important role in the calculation of the mean Sauter bubble diameter, which is used in many closure laws. As shown in the previous figures, the prediction of the interfacial area concentration are in general good agreement with experimental data. However problems are generated when the bubble diameter is evaluated close to the wall. Large values of the bubble diameter are predicted as shown in Figure 4.21b. In order to remove this behavior in the near wall cells, different options for the boundary conditions at the wall were tested but no improvements were observed.

Firstly the wall boundary condition for the void fraction was modified from *zeroGradient* to *fixedValue* equal to zero but it generated unstable results. A *fixedGradient* option was also tested with a fixed gradient for the void fraction at the wall derived from experiments but no improvements in bubble diameter at the wall were observed. The conclusion is that this diverging bubble diameter at the wall is not caused by an inappropriate wall boundary condition.

The problem is most certainly caused by the definition of the interfacial area concentration for spherical bubbles $IAC = \frac{6\alpha}{D_s}$. The code evaluates both the void fraction and the interfacial area concentration and then computes the mean Sauter bubble diameter using the following definition: $D_s = \frac{6\alpha}{IAC}$. This definition is not appropriate in the near wall cells since both the void fraction and the IAC goes to zero and therefore unphysical results for the bubble diameter are generated.

Few cases with constant user-defined bubble diameter were simulated, in order to determine the influence of this phenomena and, in general terms, of the interfacial area concentration equation on the results. These cases were simulated switching off the IAC equation and assuming a constant bubble diameter (deduced from experimental data) in the whole domain. The obtained results showed that the IAC equation has little influence on the void fraction and velocities distribution, as shown in Figure 4.21.

Furthermore, the newly implemented Hibiki-Ishii model for the adjustable variables Γ_B and Γ_C in Eqn. (2.95) and (2.96) was tested. As shown in Figure 4.21, no significant differences in results were observed if compared with the simulation with constant adjustable variables, confirming the small influence of the interfacial area concentration equation on the results.

It should also be said that in all the performed simulations, the mean Sauter bubble diameter in the bulk is under-predicted. Therefore the interfacial area concentration equation needs to be further developed. In particular, the implementation of the MUSIG approach is suggested for future development of the solver.

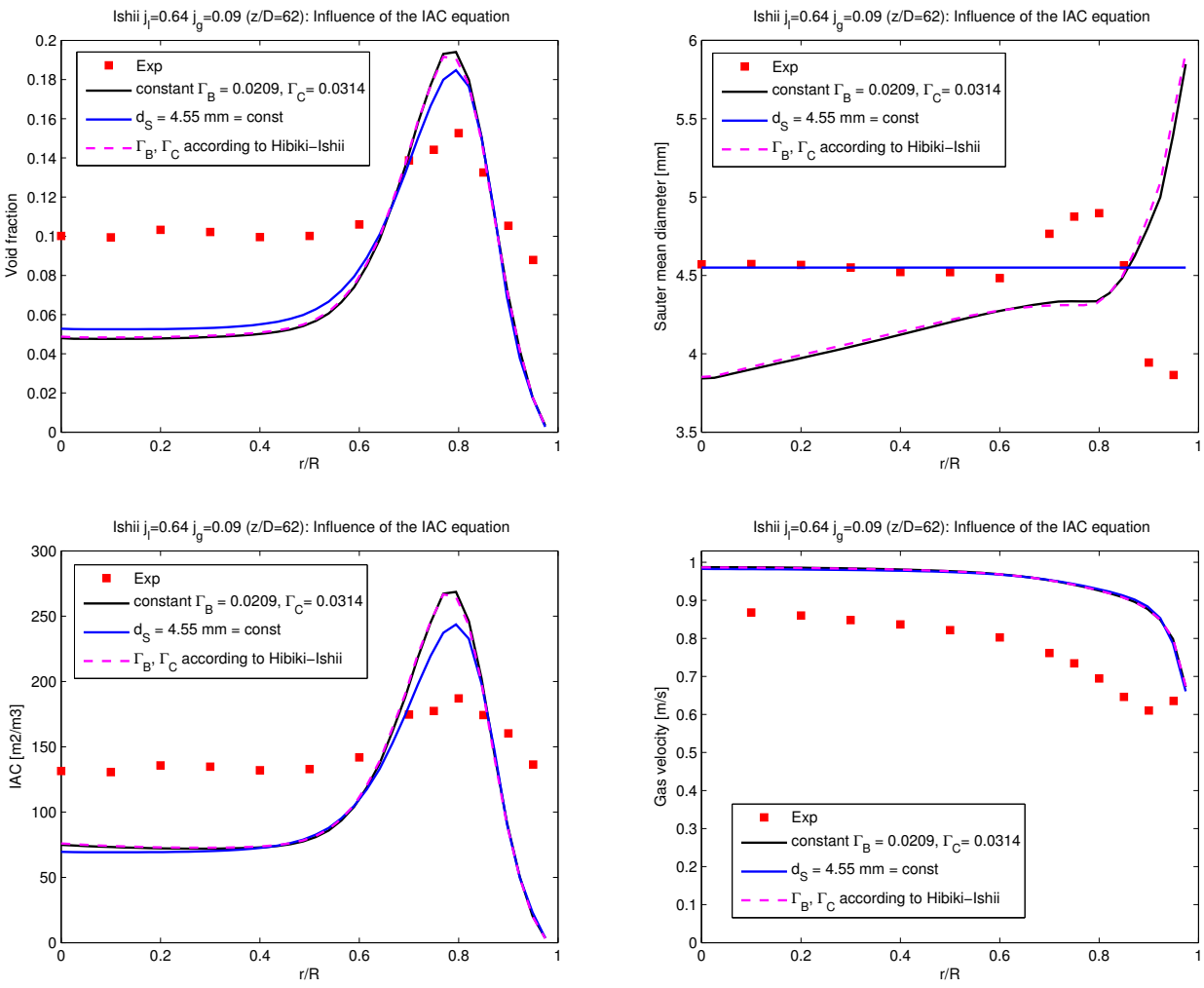

Figure 4.21: Influence of the Interfacial Area Concentration equation on the results.

4.5 Test cases with higher Reynolds number

The simulation of test cases with higher Reynolds number requires a finer mesh in order to produce reasonable and stable results.

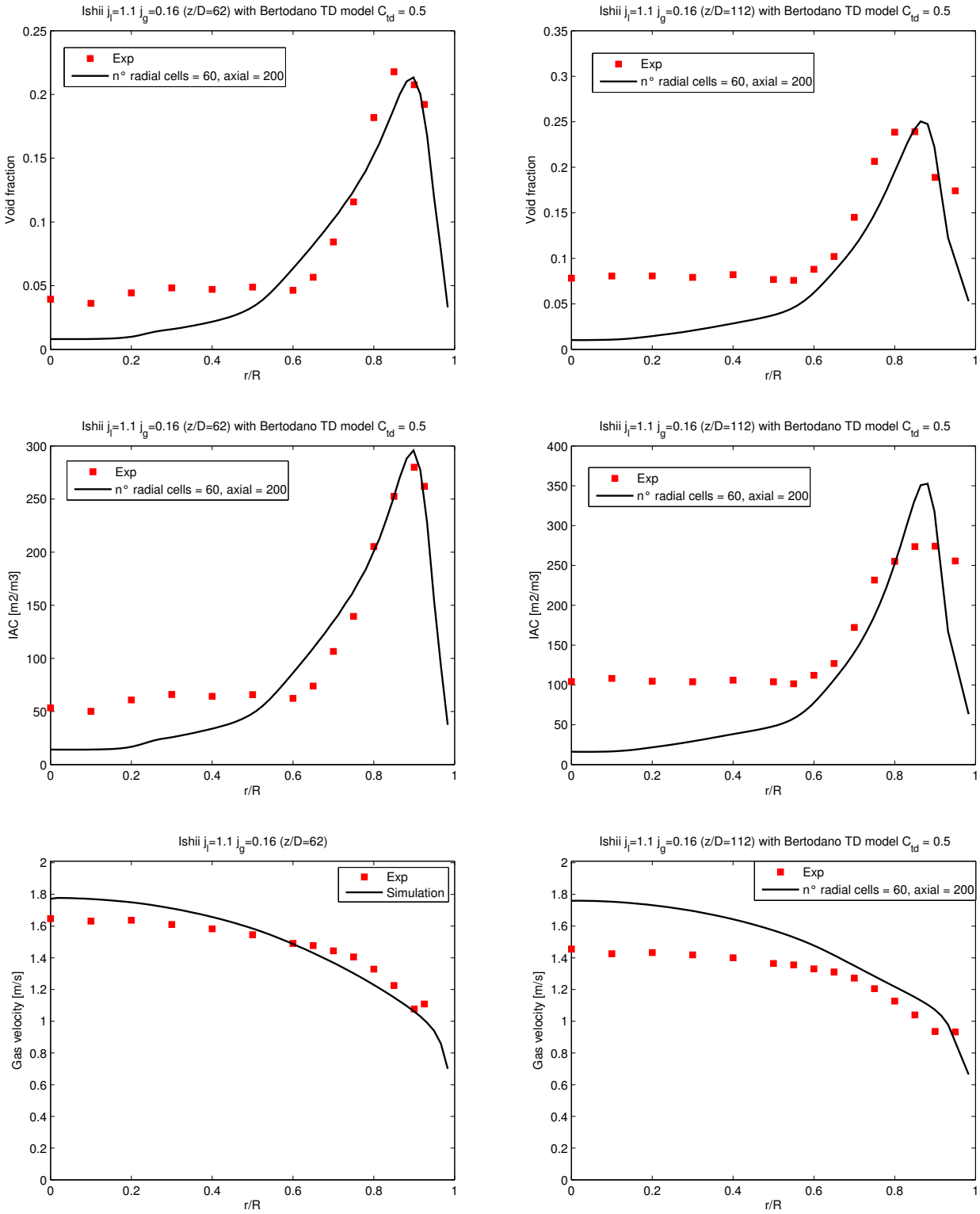

Figure 4.22: Comparison of simulation results with experiments at different location (z/D = 62, 112) for $j_l = 1.1$ m/s $j_g = 0.16$ m/s.

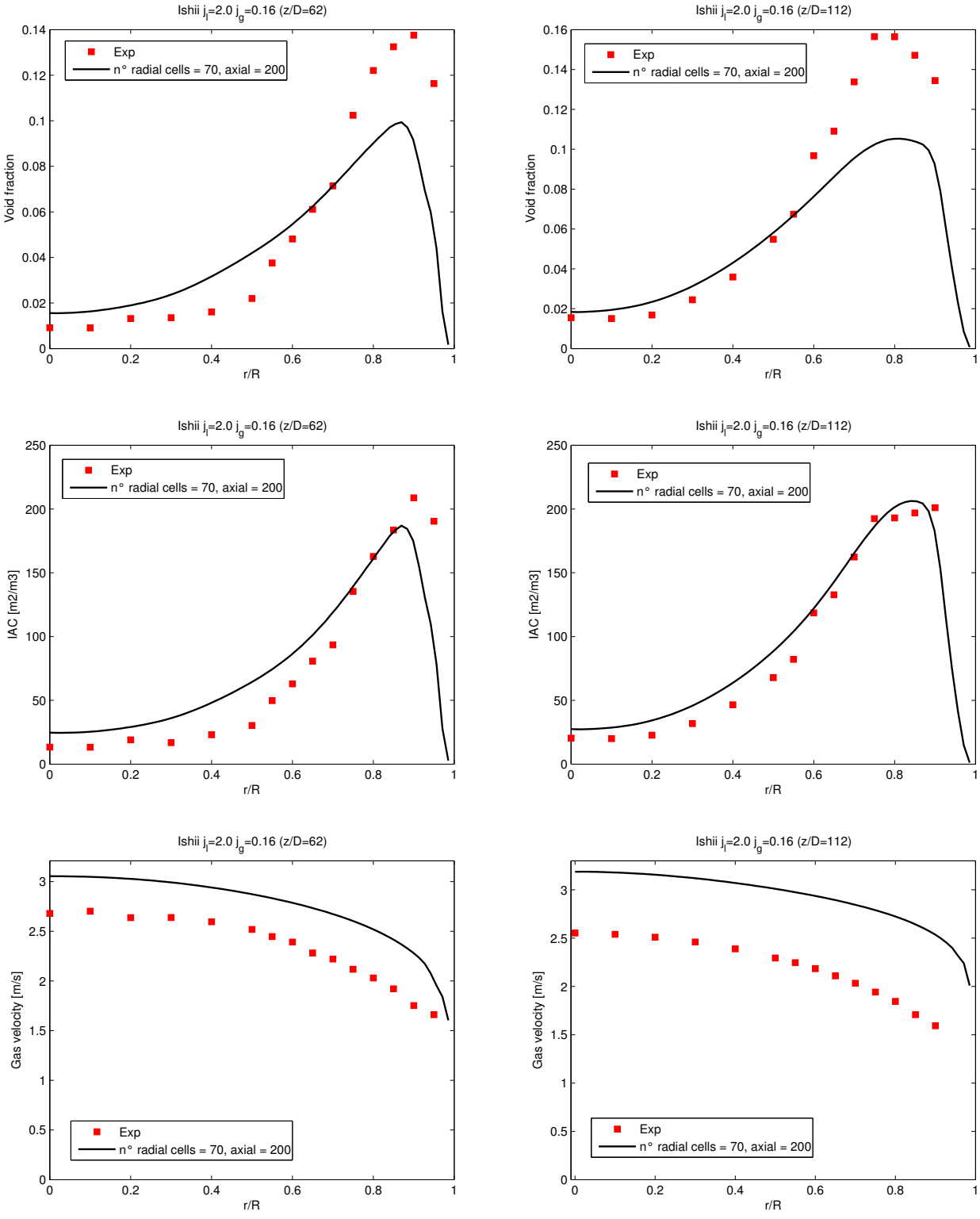

Figure 4.23: Comparison of simulation results with experiments at different location (z/D = 62, 112) for $j_l = 2.0$ m/s $j_g = 0.16$ m/s.

The use of a fine mesh poses the problem of the validity of the correlation in the near wall region and increases the required computational effort. Indeed a mesh refinement implies also a decrease in time-step due to the Courant number limitations, as discussed previously.

The test cases with $j_l = 1.1$ m/s $j_g = 0.16$ m/s and $j_l = 2.0$ m/s $j_g = 0.16$ m/s required a more

refined grid in order to give reasonable results if compared with previous simulations. For the previously shown sensitivity studies, a grid of 40 radial cells and 100 axial cells was used, while the following test cases required respectively 60 and 70 radial cells and 200 axial cells. Due to the required computational effort, no sensitivity studies were performed using these test cases.

In these simulations, the use of a reduced turbulent dispersion force coefficient $C_{td} = 0.5$ in the Lopez de Bertodano model proved to be beneficial for stability and agreement with experimental data and it was therefore used.

Good agreement between simulation results and experimental data were obtained as shown in Figure 4.22 and 4.23. These flows can be properly described by the implemented models in the solver, because they have low values for the void fraction and show a well-defined wall peak in the void fraction distribution, as already discussed previously.

4.6 Sensitivity tests on the mesh refinement

The results should not depend on the utilized mesh, i.e. *grid independent* results are desired. The effect of the use of a coarse mesh on the results was already discussed in Chapter 4.1, however oscillations disappear when fine meshes are used together with no treatment of the interfacial momentum transfer term in the momentum equation.

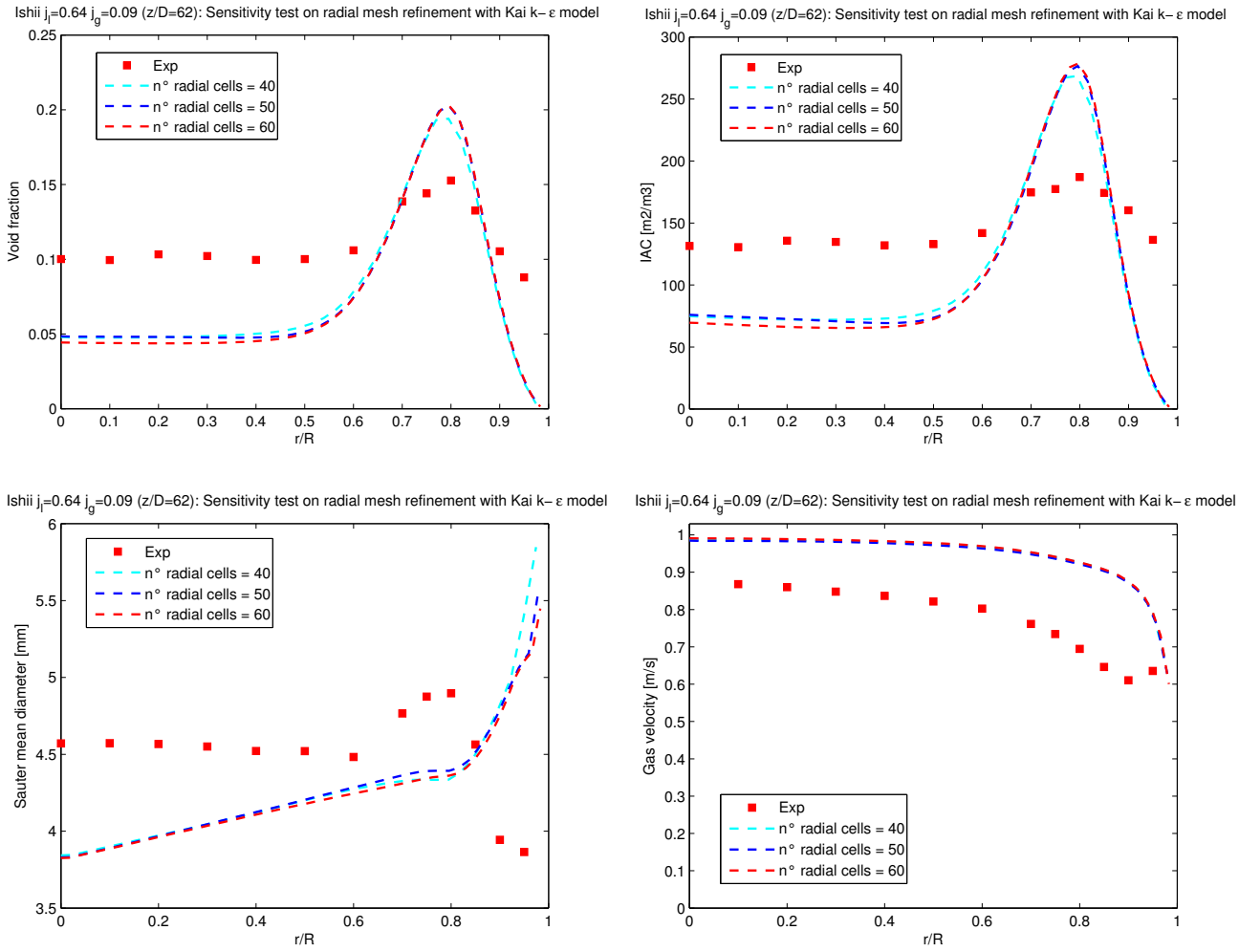

Figure 4.24: Influence of the radial mesh refinement with Kai's $k - \epsilon$ model (z/D = 62).

The simulation results converge increasing the number of cells both in the radial and axial direction, as shown in Figure 4.24, 4.25, 4.26 and 4.27. Therefore the results proved to be *grid independent*, as desirable.

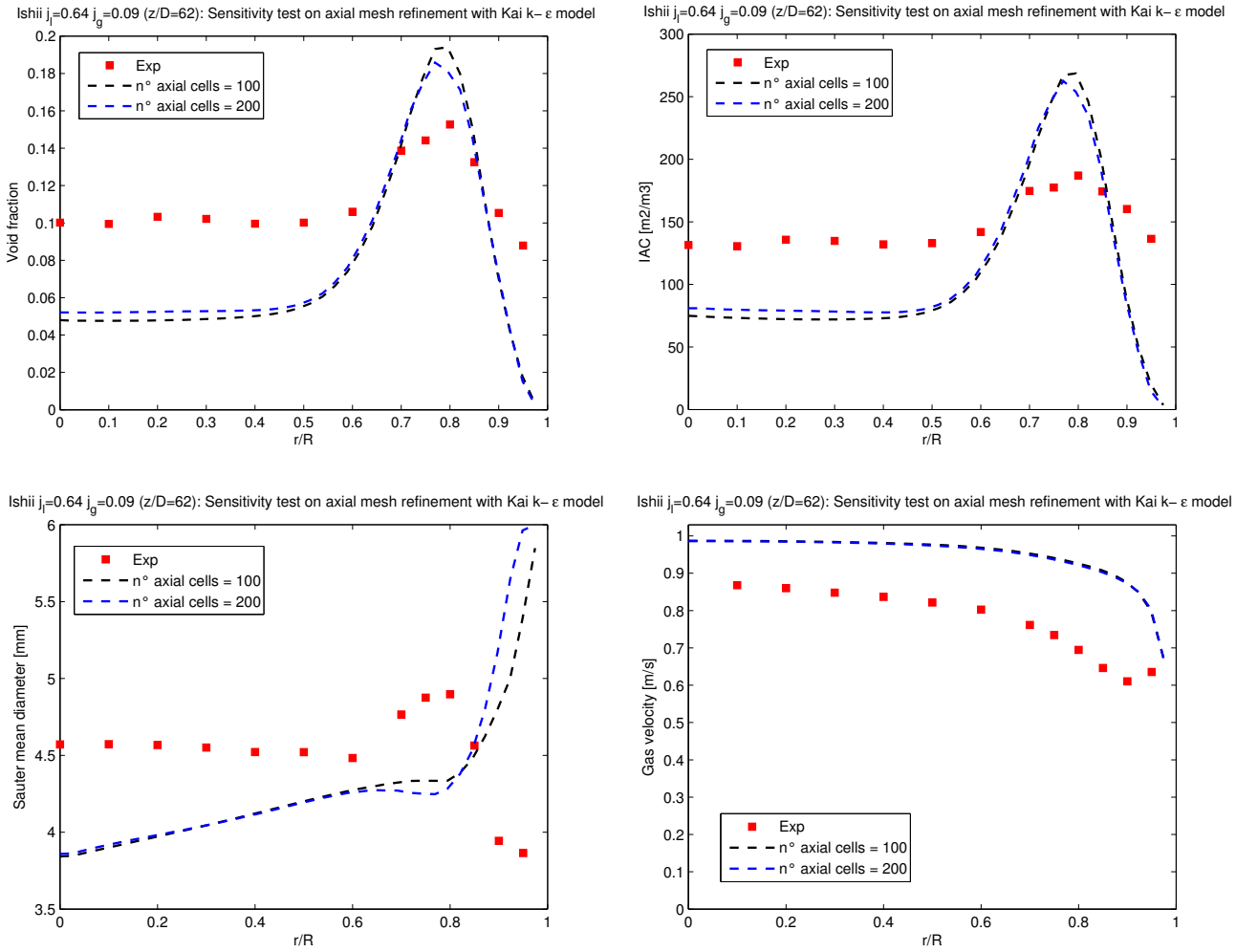

Figure 4.25: Influence of the axial mesh refinement with Kai's $k-\epsilon$ model (z/D = 62).

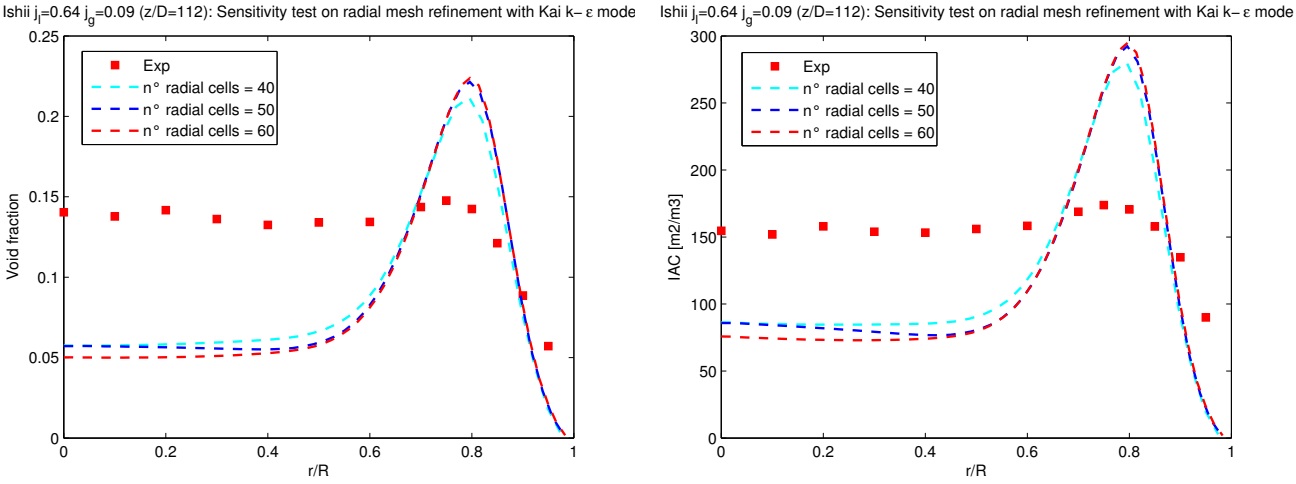

Figure 4.26: Influence of the radial mesh refinement with Kai's $k-\epsilon$ model (z/D = 112).

The plots in Figure 4.27 were obtained subtracting the results in a given grid from the reference solution (i.e. the solution with the finest mesh). The errors on the mean void fraction and gas velocity are then plotted against the number of cells, as suggested in [10, page 210].

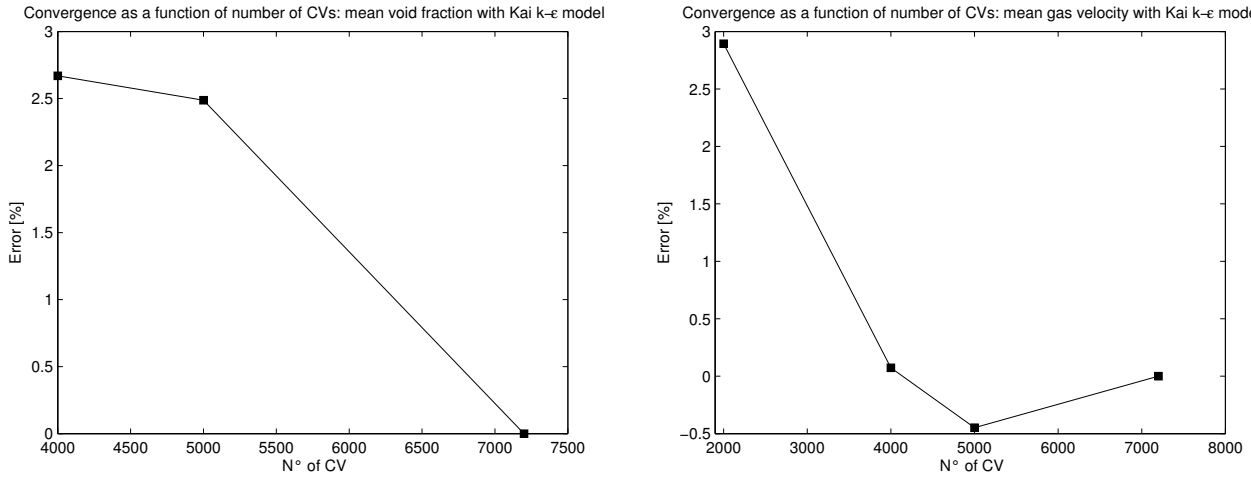

Figure 4.27: Convergence as a function of CVs: mean void fraction and gas velocity.

Similar convergence studies were also performed with the other two implemented standard $k-\epsilon$ turbulence models and with the case $j_l = 0.3$ $j_g = 0.09$. The obtained results showed similar performances and therefore are not presented in this thesis.

4.7 Error estimates

An error estimate analysis was performed on the simulation results in order to estimate the magnitude of the discrepancies between simulation results and experimental data. In the following figures, the experimental data for the void fraction and the gas velocities along the radial direction are plotted against the simulation results in order to evaluate the maximum relative error and the standard deviation.

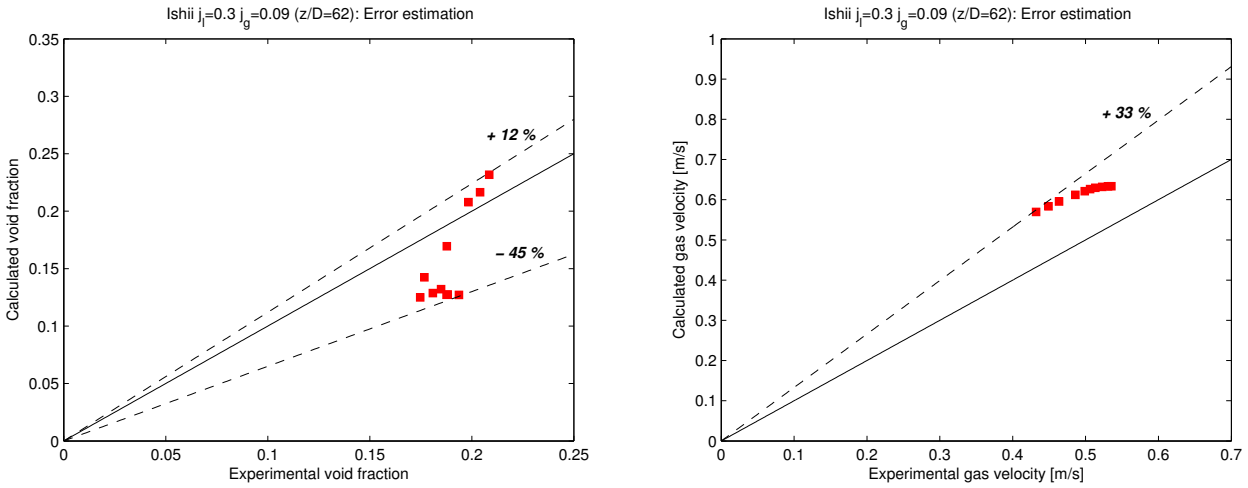

Figure 4.28: Error estimate of the void fraction and gas velocity ($j_l = 0.3$ m/s $j_g = 0.09$ m/s, z/D=62)

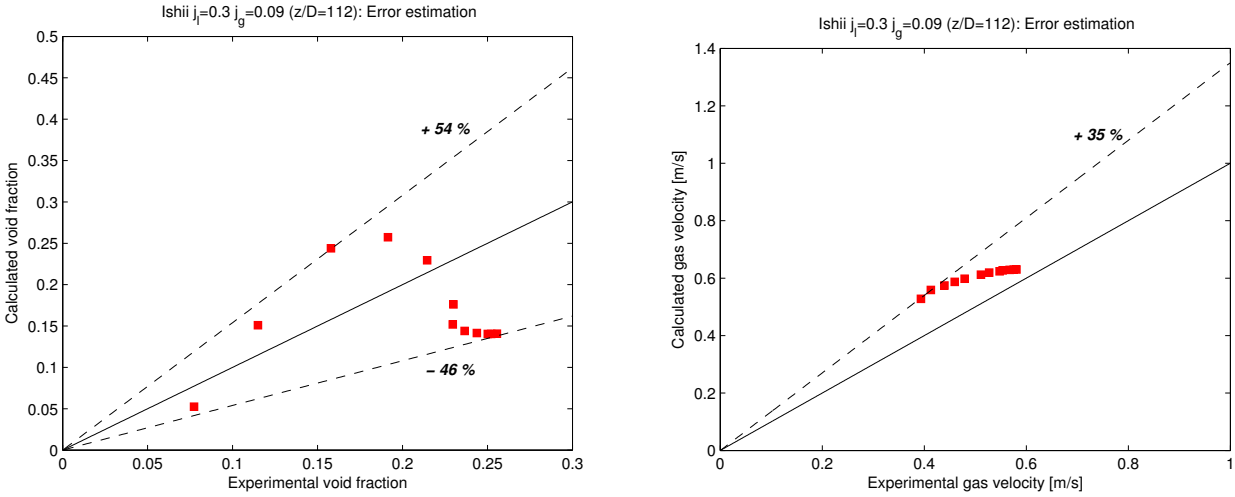

Figure 4.29: Error estimate of the void fraction and gas velocity ($j_l = 0.3$ m/s $j_g = 0.09$ m/s, z/D=112)

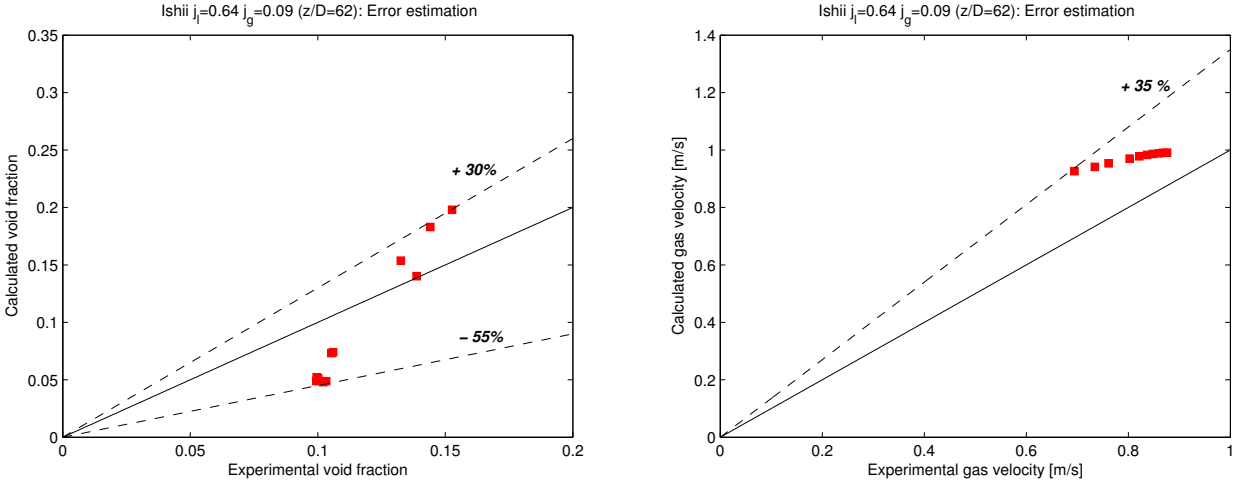

Figure 4.30: Error estimate of the void fraction and gas velocity ($j_l = 0.64$ m/s $j_g = 0.09$ m/s, z/D=62)

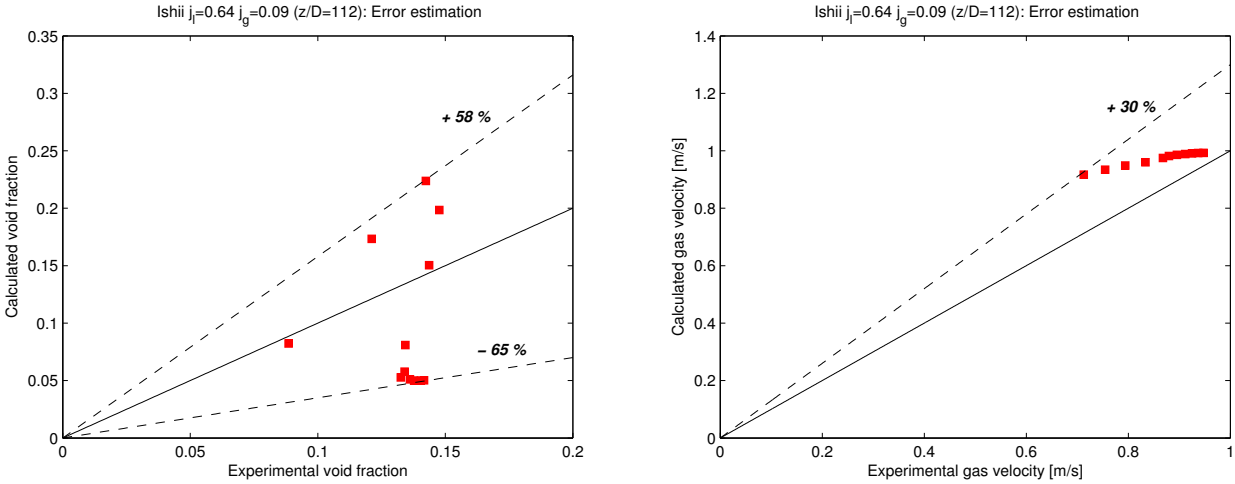

Figure 4.31: Error estimate of the void fraction and gas velocity ($j_l = 0.64$ m/s $j_g = 0.09$ m/s, z/D=112)

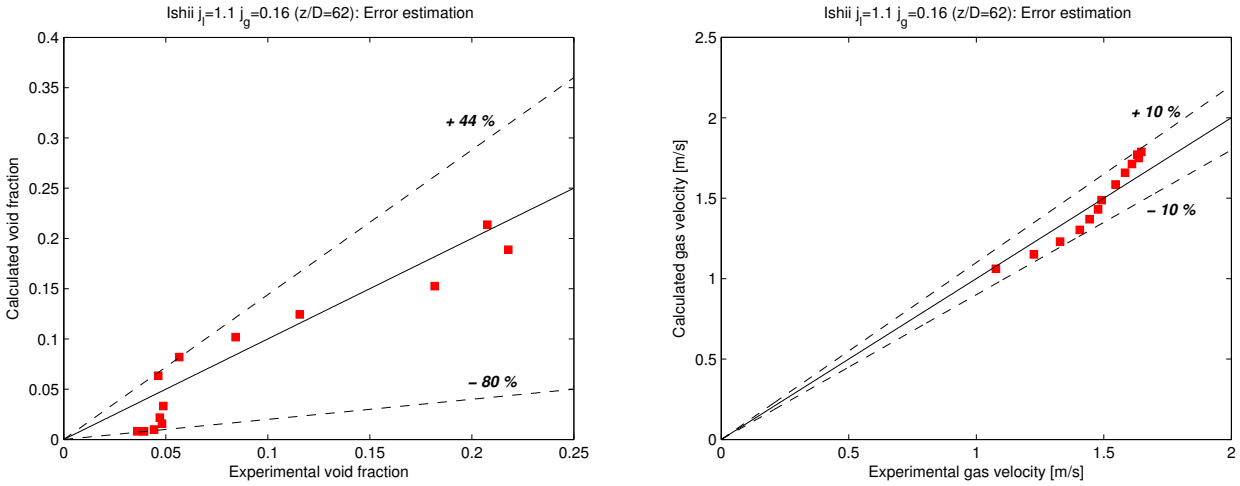

Figure 4.32: Error estimate on the void fraction and gas velocity ($j_l = 1.1$ m/s $j_g = 0.16$ m/s, z/D=62)

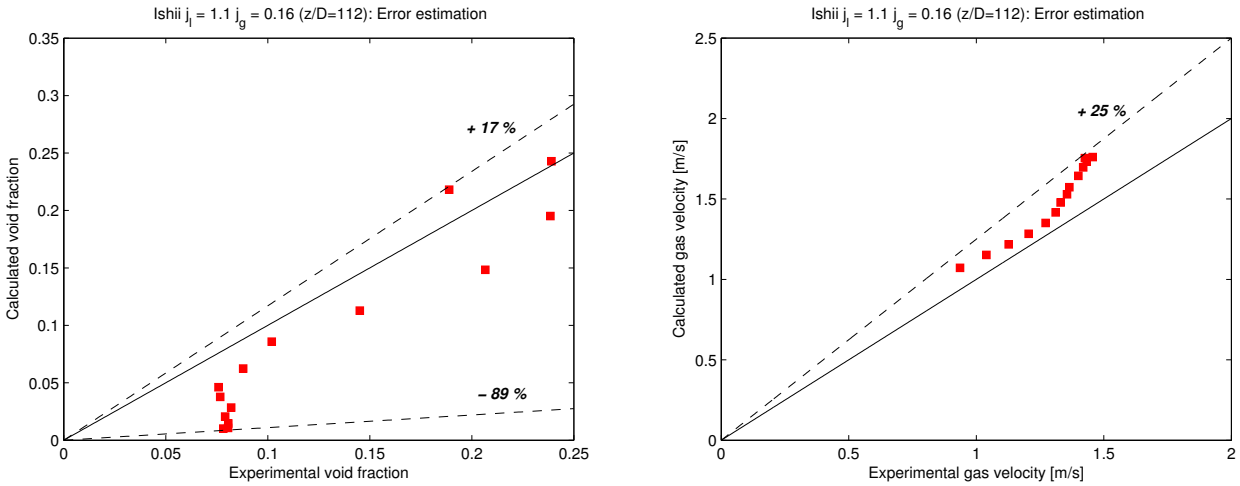

Figure 4.33: Error estimate on the void fraction and gas velocity ($j_l = 1.1$ m/s $j_g = 0.16$ m/s, z/D=112)

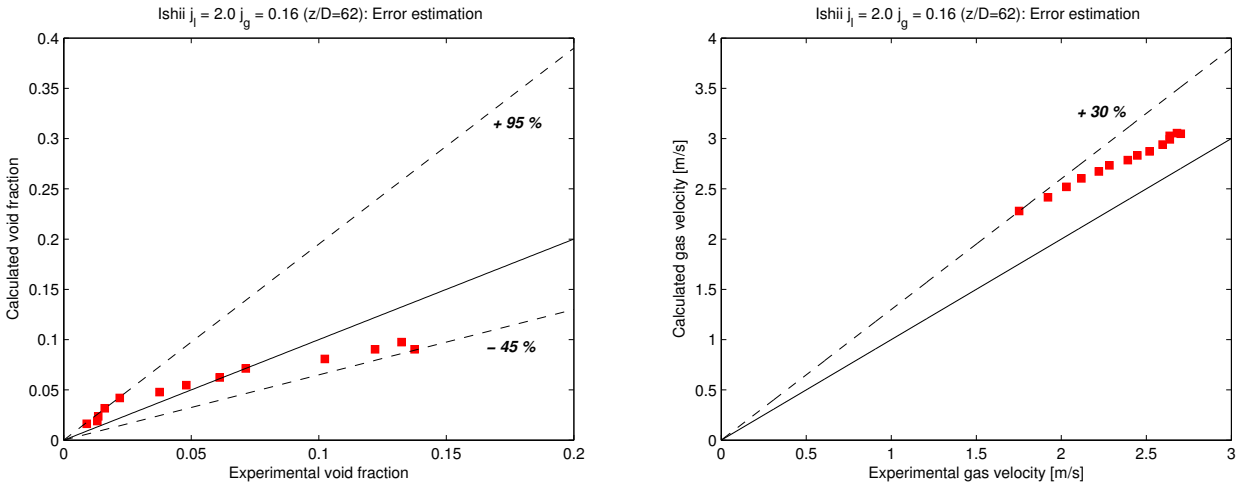

Figure 4.34: Error estimate on the void fraction and gas velocity ($j_l = 2.0$ m/s $j_g = 0.16$ m/s, z/D=62)

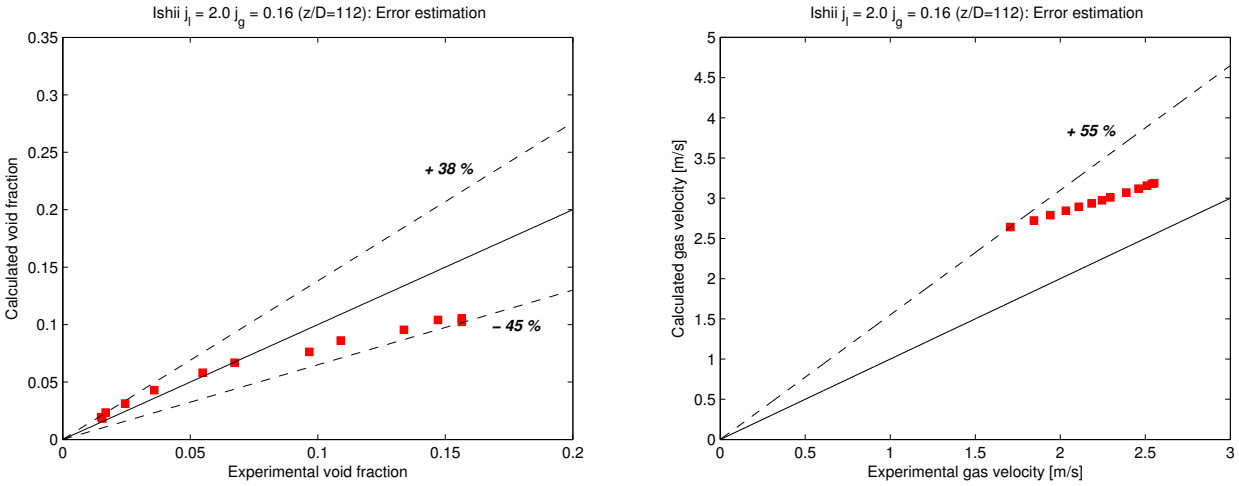

Figure 4.35: Error estimate on the void fraction and gas velocity (j_l = 2.0 m/s j_g = 0.16 m/s, z/D=112)

The standard deviations and the average errors in the void fraction and the gas velocity are shown in Table 4.1. The standard deviation for the void fraction was evaluated with the following formulation:

$$\sigma_\alpha = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(\alpha_{calc,i} - \alpha_{exp,i}\right)^2} \tag{4.1}$$

where N is the number of experimental data along the radius. Analogous calculations were done for the gas velocity.

Table 4.1: Estimate of the standard deviations and the relative errors.

Test cases	σ_α	σ_α[%]	Error range α	σ_{U_a}	σ_{U_a}[%]	Error range U_a
$j_l = 0.3$ $j_g = 0.09$, z/D = 62	0.050	28.0	-45% - +12%	0.119	24.7	0% - +33%
$j_l = 0.3$ $j_g = 0.09$, z/D = 112	0.081	39.2	-46% - +54%	0.100	19.8	0% - +35%
$j_l = 0.64$ $j_g = 0.09$, z/D = 62	0.046	41.5	-55% - +30%	0.183	23.3	0% - +35%
$j_l = 0.64$ $j_g = 0.09$, z/D = 112	0.070	54.2	-65% - +58%	0.128	15.0	0% - +30%
$j_l = 1.1$ $j_g = 0.16$, z/D = 62	0.025	29.0	-80% - +44%	0.09	6.1	-10% - +10%
$j_l = 1.1$ $j_g = 0.16$, z/D = 112	0.047	38.5	-89% - +17%	0.198	15.4	0% - +25%
$j_l = 2.0$ $j_g = 0.16$, z/D = 62	0.021	38.1	-45% - +95%	0.417	17.8	0% - +30%
$j_l = 2.0$ $j_g = 0.16$, z/D = 112	0.029	38.1	-45% - +38%	0.768	38.1	0% - +55%

The error estimates show that the agreement of the computed gas velocities with the experimental data is generally good.

The gas velocity is generally over-predicted but the average relative errors are of the order of 20 %, value deduced from Table 4.1. In particular the larger discrepancies between the computed and experimental gas velocities are observed near the wall, where the validity of the code is not guaranteed and furthermore the measurement from experiments are less reliable.

Larger discrepancies are observed between the computed void fraction and the experimental data. However the average relative errors are acceptable and of the order of 35-40 %. High values for the relative error in some points are found for the high Reynolds cases.

This can be explained by the fact that these cases have relatively small values of the void fraction and therefore the relative errors are large even if the standard deviations are smaller than the other cases.

Observing the standard deviations and relative errors in Table 4.1 the high Reynolds cases show the best performances in predicting the void fraction distribution, because these cases have relatively low void fraction values and a wall-peaked radial distribution. Therefore the Tomiyama lift force model can predict accurately the void fraction distribution.

Analogously in the test cases $j_l = 0.3$ $j_g = 0.09$ and $j_l = 0.64$ $j_g = 0.09$, the results for the void fraction show better agreement with experimental data at the location z/D = 62 if compared with z/D = 112.

Indeed the void fraction distribution at z/D = 62 shows a wall peak, while at the axial location z/D = 112 a pronounced core peak is observed. This confirm even further that the solver is not able to predict accurately core peaked void fraction distributions and that further development of the solver is needed.

CHAPTER 5

CONCLUSIONS

An existing two-phase CFD code in OpenFoam has been further developed and validated against experimental data of adiabatic water-air bubbly flow in a vertical pipe with a diameter of 25.4 mm.

The implemented mathematical model uses a two-fluid model with Eulerian conservation equations for both phases (*Euler-Euler model*), which are treated as compressible (in particular, the gas phase). Closure laws for the interfacial momentum transfer term and the Reynolds-averaged turbulent stress term are also implemented.

The simulations are performed using a quasi-2D cylindrical geometry in order to reduce the required computational cost. The non-uniform inlet boundary conditions are implemented using the GroovyBC libraries, that proved to be stable, efficient and relatively user-friendly.

The obtained results proved to be grid-independent but oscillations in the radial direction were observed with coarse meshes. The use of a coarse mesh is required by the limitation of the validity of the standard $k-\epsilon$ turbulence model in near wall cells (i.e. $y^+ > 20$). A vast amount of work and time has been spent on the solution of this problem and many sensitivity tests have been performed. The implementation of a Rhie-Chow like treatment of the interfacial momentum transfer term in the momentum equation proved to be beneficial. Oscillations have been significantly reduced and almost completely removed. The cause for these instabilities proved to be an inappropriate treatment of the non-linear interfacial momentum transfer term in the momentum equation (in particular, the turbulent dispersion and the lift force). Indeed the gradient of the void fraction in the turbulent dispersion force and the non-linearities introduced by the lift force requires special treatment in order to avoid the decoupling of the momentum and mass conservation equation. The implemented Rhie-Chow like treatment proved to work properly when $y^+ > 20$ and oscillations could be completely eliminated with the removal of the virtual mass force. However, the virtual mass force proved to be important for the stability of the code and its removal produced unstable results in some simulations. Therefore a future development of the code will imply the implementation of a more appropriate treatment of the virtual mass force.

Another solution for the removal of the decoupling between the conservation equations (and therefore of the oscillations) could be the use of a coupled solver. It means that the equations will not be solved with a segregated approach (see more details in Chapter 2.1). The coupled equations will therefore be solved simultaneously with the definition of a block-coupled matrix

which can be constructed and solved in OpenFoam. This approach can prevent the decoupling between equations improving the solver. However the choice of which variables (pressure, velocity, void fraction, etc...) should be coupled is not straightforward and will require a detailed study. Furthermore a large coefficient matrix could be obtained which might eventually be ill-conditioned and difficult to solve.

The test of different approaches for the standard $k-\epsilon$ turbulence model produced similar results in terms of convergence performances and agreement with experimental data. The implementation of a different turbulence model (e.g. the SST $k-\omega$ model) is suggested in order to remove the limitation of the validity of the wall functions in the near wall cells, so that results with $y^+ < 20$ can become acceptable.

The use of a fine mesh (with $y^+ < 20$) have produced results in general good agreement with experimental data, but further development of the models is still required in order to improve the agreement between simulations and experiments. The average relative errors in gas velocity and void fraction are of the order of 20 % and 35 % respectively.

In order to improve the agreement between simulations and experimental data, new correlations for the interfacial momentum transfer term in the momentum equation have been implemented and tested. The newly implemented drag force model significantly improved the gas velocity prediction, since it is taking into account the effect of the deformation of the gas bubbles (from a spherical to an ellipsoidal or cap shape) on the drag coefficient. Therefore the Ishii-Zuber drag model for densely distributed fluid particles is the suggested model for future simulations.

More studies and improved correlations for the lift force coefficient will be required in order to accurately predict the void fraction radial distribution. The performed simulations showed that the Tomiyama lift model is able to predict only wall-peaked void fraction distributions, while the Rusche lift model can predict only core-peaked distributions. This consideration was also confirmed by the fact that the solver (using the Tomiyama lift force model) can predict quite accurately the cases with higher Reynolds number which have a well-defined wall peak and low values of the void fraction. Therefore the implementation of a Multiple Bubble SIze Group (MUSIG) approach is suggested in order to predict the radial separation of large and small bubbles. That should lead to better void fraction prediction, even if this modification will significantly increase the complexity of the code. The implementation of the MUSIG approach can also improve the prediction of the mean Sauter bubble diameter that was slightly under-predicted during the performed simulations.

Large values of the bubble diameter in near-wall cells were observed. This was probably caused by the definition of the interfacial area concentration for spherical bubbles, as discussed in Chapter 4.4. A possible solution for this problem can be the implementation of an appropriate wall function for the interfacial area concentration equation. The wall function can set the interfacial area concentration equation at the wall, so that the bubble diameter goes to the lift-off size (avoiding to divide the void fraction and the interfacial area concentration in near-wall cells).

Further improvement of the solver would include the removal of the phase-intensive formulation of the momentum equation (as in the new two-phase solver in OpenFoam 2.1.1.), so that the division by the void fraction in some terms of the momentum equation can be avoided.

Furthermore the current solver needs the use of the absolute pressure, but this can lead to large cancellation errors. Therefore the use of the relative pressure (as in OpenFoam 2.1.1.) is suggested for future development of the code.

In conclusion, mesh independent results in general good agreement with experimental data have been obtained and instabilities with coarse meshes were removed with a Rhie-Chow like treatment of the interfacial momentum transfer term in the momentum equation. However, further development of the code will be necessary in order to deal with the highlighted deficiencies and to improve the agreement with experimental data.

BIBLIOGRAPHY

[1] K. Fu. Implementation and validation of two-phase boiling flow models in openfoam. Technical report, KTH, August 2012.

[2] H. Anglart. Thermal-hydraulics in nuclear systems, 2011.

[3] OpenCFD Limited. *OpenFoam User Guide*, 1.7.1 edition, 2010.

[4] W-H. Leung. *Modeling Of Interfacial Area Concentration And Interfacial Momentum Transfer: Theoretical And Experimental Study.* PhD thesis, Purdue University, May 1997.

[5] World Nuclear Association. Safety of nuclear reactors. http://www.world-nuclear.org/info/inf06.html Accessed on 11th Oct 2012.

[6] J. Hawkes. The simulation and study of conditions leading to axial offset anomaly in pressurized water reactors. Master's thesis, Georgia instute of Technology, 2004.

[7] H. Weller. Derivation, modelling and solution of the conditionally averaged two-phase flow equations. Technical report, OpenCFD, 2005.

[8] H. Rusche. *Computational fluid dynamics of dispersed two-phase flows at high phase fractions.* PhD thesis, Imperial College, London, 2002.

[9] Features of openfoam. http://www.openfoam.com/features/ Accessed on 20th Oct 2012.

[10] Ferziger J.H. and Perić M. *Computational Methods for Fluid Dynamics.* Springer, 3rd edition, 2002.

[11] OpenCFD Limited. *Programmers Guide*, 1.7.1 edition, 2010.

[12] Inc. ANSYS. *ANSYS CFX-Solver Theory Guide*, 12.1 edition, 2009.

[13] W. Yao and C. Morel. Volumetric interfacial area prediction in upward bubbly two-phase flow. *Int. J. Heat Mass Transfer*, 47:307, 2004.

[14] E Michta. Modeling of subcooled nucleate boiling with openfoam. Master's thesis, Royal Institute of Technology, Stockholm, 2011.

[15] fvc::reconstruct. http://www.cfd-online.com/Forums/attachments/openfoam-programming-development/10861d1326733559-fvc-reconstruct-algorithm-fvcreconstruct.pdf Accessed on 28th Sep 2012.

[16] L. Schiller and Z. Naumann. A drag coefficient correlation. *Z. Ver. Deutsch. Ing.*, 77:318, 1935.

[17] M. Ishii and N. Zuber. Drag coefficient and relative velocity in bubbly, droplet or particulate flows. *AIChE J.*, 25:843, 1979.

[18] C.Y. Wen and Y.H. Yu. Mechanics of fluidization. *Chemical Engineering Progress Symposium Series*, 62:100, 1966.

[19] P.B. Whalley. *Boiling, Condensation and Gas-Liquid Flow*. Clarendon Press, 1987.

[20] D. Lucas and A. Tomiyama. On the role of the lateral lift force in poly-dispersed bubbly flows. *International Journal of Multiphase Flow*, 2011.

[21] A. Tomiyama. Struggle with computational bubble dynamics. In *Third International Conference on Multiphase Flow*, volume 18, 1998.

[22] S.P. Antal, R. Laheyjr, and J. Flaherty. Analysis of phase distribution in fully developed laminar bubbly two-phase flow. *Int. J. Multiphase Flow*, 7:635, 1991.

[23] R. Rzehak, E. Krepper, and C. Lifante. Comparative study of wall-force models for the simulation of bubbly flows. *Nuclear Engineering and Design*, 2012.

[24] Th. Frank. Advances in computational fluid dynamics (cfd) of 3-dimensional gas-liquid multiphase flows. In *NAFEMS Seminar "Simulation of Complex Flows (CFD)"*, page 1, Wiesbaden, Germany, 2005.

[25] S. Hosokawa, A. Tomiyama, and H. Tomoyuki. Lateral migration of single bubbles due to the presence of wall. *ASME Conference Proceedings*, 2002(36150), 2002.

[26] A. D. Gosman, C. Lekakou, S. Politis, R. I. Issa, and M. K. Looney. Multidimensional modeling of turbulent two-phase flow in stirred vessels. *AIChE J.*, 38:1946, 1992.

[27] M. Lopez de Bertodano. *Turbulent bubbly two-phase flow in a triangular duct*. PhD thesis, Rensselaer Polytechnic Institute, Troy, NY, 1992.

[28] *The Favre Averaged Drag Model for Turbulent Dispersion in Eulerian Multi-Phase Flows*, 2004.

[29] T. Hibiki and M. Ishii. Development of one-group interfacial area transport equation in bubbly flow systems. *Int. J. Heat Mass Transfer*, 45:2351, 2002.

[30] B.U. Bae, H.Y. Yoon, D.J. Euh, C.H. Song, and G.C. Park. Computational analysis of a subcooled boiling flow with a one-group interfacial area transport equation. *J. Nucl. Sci. Technol.*, 45:341, 2008.

[31] Y. Sato and K. Sekoguchi. Liquid velocity distribution in two-phase bubbly flow. *Int. J. Multiphase Flow*, 2:79, 1975.

[32] B.E. Launder and D.B. Spalding. The numerical computation of turbulent flows. *Computer methods in applied mechanics and engineering*, 1974.

[33] Pope S.B. *Turbulent flows*. Cambridge press, 2010.

[34] L. Davidson. Fluid mechanics, turbulent flow and turbulence modeling, September 2012.

[35] The boussinesq hypothesis. http://www.cfd-online.com/Wiki/Boussinesq-eddy-viscosity-assumption Accessed on 2th Oct 2012.

[36] I. Kataoka. Development of research on interfacial area transport. *J. Nucl. Sci. Technol.*, 2010.

[37] A. J. Ghajar and C. C. Tang. Advances in void fraction, flow pattern maps and non- boiling heat transfer two-phase flow in pipes with various inclinations. *Advances in Multiphase Flow and Heat Transfer*, 1:1–52, 2009.

[38] R. Rozenblit, M. Gurevich, Y. Lengel, and G. Hetsroni. Flow patterns and heat transfer in vertical upward air–water flow with surfactant. *International Journal of Multiphase Flow*, 32:889–901, 2006.

[39] Contrib groovybc. http://openfoamwiki.net/index.php/Contrib_groovyBC Accessed on 6th Nov 2012.

[40] Turbulence free-stream boundary conditions. http://www.cfd-online.com/Wiki/Turbulence_free-stream_boundary_conditions#Turbulent_energy Accessed on 6th Nov 2012.

[41] Stability problem due to turbulent dispersion force in a subcooled boiling model. http://www.cfd-online.com/Forums/openfoam/83616-stability-problem-due-turbulent-dispersion-force-subcooled-boiling-model.html Accessed on 16th Nov 2012.

[42] Tomiyama wall lubrication force. http://www.cfd-online.com/Forums/openfoam/82931-tomiyama-wall-lubrication-force.html Accessed on 11th December 2012.

[43] E. Krepper, D. Lucas, and H-M. Prasser. On the modelling of bubbly flow in vertical pipes. *Nuclear Engineering and Design*, 2004.

[44] E. Krepper, D. Lucas, Th. Frank, H-M. Prasser, and P.J. Zwart. The inhomogeneous-musigmodel for the simulation of polydispersedflows. *Nuclear Engineering and Design*, 238(7):1690–1702, 2008.

[45] ddtphicorr issue. http://www.openfoam.org/mantisbt/view.php?id=169 Accessed on 27th Sep 2012.

APPENDIX A

THE IMPLEMENTED MAIN PROGRAM

```
#include "fvCFD.H"
#include "nearWallDist.H"
#include "wallFvPatch.H"
#include "Switch.H"
#include "fixedGradientFvPatchFields.H"

#include "IFstream.H"
#include "OFstream.H"

#include "wallDistReflection.H"
#include "wallDist.H"
#include "buoyantPressureFvPatchScalarField.H"

#include "phaseModel.H"
#include "dragModel.H"
#include "liftModel.H"
#include "wallLubricationModel.H"
#include "turbulentDispersionModel.H"
#include "virtualMassModel.H"
#include "breakupModel.H"
#include "coalescenceModel.H"
#include "aSDModel.H"
#include "freqModel.H"
#include "DloModel.H"
#include "h1fModel.H"
#include "hqModel.H"
#include "nucleateBoilingModel.H"
#include "heatTransferModel.H"

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

int main(int argc, char *argv[])
{
```

```
#include "setRootCase.H"

#include "createTime.H"
#include "createMesh.H"
#include "readGravitationalAcceleration.H"
#include "readWallBoilingProperties.H"
#include "readPPProperties.H"
#include "readPhaseChangeProperties.H"
#include "readModuleControls.H"
#include "createFields.H"
#include "initContinuityErrs.H"
#include "readTimeControls.H"
#include "CourantNo.H"
#include "setInitialDeltaT.H"

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

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

while (runTime.run())
{
    #include "readTwoPhaseEulerFoamControls.H"
    #include "CourantNos.H"
    #include "fieldMonitor.H"
    #include "setDeltaT.H"

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

    if (nOuterCorr != 1)
    {
        p.storePrevIter();
    }

// --- Outer-corrector loop
    for (int oCorr=0; oCorr<nOuterCorr; oCorr++)
    {
        if (detailedMonitor)
        {
            GammaLV.storePrevIter();
            GammaVL.storePrevIter();
            Ub.storePrevIter();
        }
        #include "heatTransfer.H"

        #include "alphaEqn.H"
        #include "IACEqn.H"

        if (stdkEpsilonKai)
```

```
{
    #include "kEpsilon_std_Kai.H"
}
else if (stdkEpsilonGhione)
{
    #include "kEpsilon_std_Ghione.H"
}
else
{
    #include "kEpsilon_YaoMorel.H"
}

  #include "HEqns.H"

  #include "liftDragCoeffs.H"

if (MomentumEqn_Michta)
{
    #include "UEqns_Michta.H"

    // --- PISO loop
    for (int corr=0; corr<nCorr; corr++)
    {
        #include "pEqn_Michta.H"

        if (correctAlpha && corr<nCorr-1)
        {
            #include "alphaEqn.H"
            #include "IACEqn.H"
        }
    }
}
else if (MomentumEqn_Rusche)
{
    #include "UEqns_Rusche.H"

    // --- PISO loop
    for (int corr=0; corr<nCorr; corr++)
    {

        #include "pEqn_Rusche.H"

        if (correctAlpha && corr<nCorr-1)
        {
            #include "alphaEqn.H"
            #include "IACEqn.H"
        }
    }
}
```

```
        else
        {
            #include "UEqns_Standard.H"

            // --- PISO loop
            for (int corr=0; corr<nCorr; corr++)
            {

                #include "pEqn_Standard.H"

                if (correctAlpha && corr<nCorr-1)
                {
                    #include "alphaEqn.H"
                    #include "IACEqn.H"
                }
            }
        }
        #include "DpDt.H"
        #include "DDtU.H"
    }
    #include "write.H"

    Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
        << "  ClockTime = " << runTime.elapsedClockTime() << " s"
        << nl << endl;
}
Info<< "End\n" << endl;

return 0;
}
```

APPENDIX B

THE IMPLEMENTED CONTINUITY EQUATION

```
alpha.storePrevIter();

#include "alphaIterEqns.H"
{
    word scheme("div(phi,alpha)");
    word schemer("div(phir,alpha)");

    surfaceScalarField phic("phic", phi);
    surfaceScalarField phir("phir", phia - phib);
    {
        fvScalarMatrix alphaEqn
        (
            fvm::ddt(alpha)
          + fvm::div(phic, alpha, scheme)
          - fvm::Sp(fvc::div(phic), alpha)
          + fvm::div(-fvc::flux(-phir, beta, schemer), alpha, schemer)
         ==
            GammaVL/rhoa
          - GammaLV/rhoa
          - fvm::SuSp(GammaVL*(1.0/rhoa-1.0/rhob), alpha)
          - fvm::SuSp(GammaLV*(-1.0/rhoa+1.0/rhob), alpha)
          - fvm::SuSp(-beta*DrhobDt, alpha)
          - fvm::SuSp(DrhoaDt*beta, alpha)
        );
        if (oCorr == nOuterCorr-1)
        {
            alphaEqn.relax(1);
        }
        else
        {
            alphaEqn.relax();
        }
        alphaEqn.solve();
```

```
        beta = scalar(1) - alpha;

        Info<< "Dispersed phase volume fraction = "
            << alpha.weightedAverage(mesh.V()).value()
            << "  Min(alpha) = " << min(alpha).value()
            << "  Max(alpha) = " << max(alpha).value()
            << endl;
        Info<< "Continuous phase volume fraction = "
            << beta.weightedAverage(mesh.V()).value()
            << "  Min(beta) = " << min(beta).value()
            << "  Max(beta) = " << max(beta).value()
            << endl;
    }
}
```

where the file *alphaIterEqns.H* reads:

```
for (int acorr=0; acorr<nAlphaCorr; acorr++)
{
    {
        word scheme("div(phi,alpha)");
        word schemer("div(phir,alpha)");
        surfaceScalarField phic("phic", phi);
        surfaceScalarField phir("phir", phia - phib);

        fvScalarMatrix alphaEqn
        (
            fvm::ddt(alpha)
          + fvm::div(phic, alpha, scheme)
          - fvm::Sp(fvc::div(phic), alpha)
          + fvm::div(-fvc::flux(-phir, beta, schemer), alpha, schemer)
         ==
            GammaVL/rhoa
          - GammaLV/rhoa
          - fvm::SuSp(GammaVL*(1.0/rhoa-1.0/rhob), alpha)
          - fvm::SuSp(GammaLV*(-1.0/rhoa+1.0/rhob), alpha)
          - fvm::SuSp(-beta*DrhobDt, alpha)
          - fvm::SuSp(DrhoaDt*beta, alpha)
        );
        alphaEqn.relax();
        alphaEqn.solve();
    }
    {
        word scheme("div(phi,beta)");
        word schemer("div(phir,beta)");

        surfaceScalarField phic("phic", phi);
        surfaceScalarField phir("phir", phib - phia);
```

```
        fvScalarMatrix betaEqn
        (
            fvm::ddt(beta)
          + fvm::div(phic, beta, scheme)
          - fvm::Sp(fvc::div(phic), beta)
          + fvm::div(-fvc::flux(-phir, alpha, schemer), beta, schemer)
         ==
          - (GammaVL-GammaLV)/rhob
          - fvm::SuSp(GammaVL*(1.0/rhoa-1.0/rhob), beta)
          - fvm::SuSp(GammaLV*(-1.0/rhoa+1.0/rhob), beta)
          - fvm::SuSp(-alpha*DrhoaDt, beta)
          - fvm::SuSp(DrhobDt*alpha, beta)
        );
        betaEqn.relax();
        betaEqn.solve();
    }
    alpha = 1.0/2.0*(1-sqr(1-alpha)+sqr(1-beta));
}
```

APPENDIX C

THE IMPLEMENTED MOMENTUM EQUATION

Three different formulations of the momentum equation are implemented and they are user-selectable in the file *moduleControls*.

The so-called `"UEqns_Standard.H"` is the standard model in the solver and reads:

```
fvVectorMatrix UaEqn(Ua, Ua.dimensions()*dimVol/dimTime);
fvVectorMatrix UbEqn(Ub, Ub.dimensions()*dimVol/dimTime);
{
    {
      volTensorField gradUaT = fvc::grad(Ua)().T();
      volTensorField Rca
      (
       "Rca",
       ((2.0/3.0)*I)*(sqr(Ct)*k + nuEffa*tr(gradUaT)) - nuEffa*gradUaT
      );
     surfaceScalarField phiRa =
      -fvc::interpolate(nuEffa)*mesh.magSf()*fvc::snGrad(alpha*rhoa)
      /fvc::interpolate(alpha + scalar(1e-12))/fvc::interpolate(rhoa);

      UaEqn =
      (
          (scalar(1) + AvmAlpha/rhoa)*
          (
              fvm::ddt(Ua)
            + fvm::div(phia, Ua, "div(phia,Ua)")
            - fvm::Sp(fvc::div(phia), Ua)
          )
        - fvm::laplacian(nuEffa, Ua, "laplacian(nuEffa,Ua)")
        + fvc::div(Rca)

        + fvm::div(phiRa, Ua, "div(phia,Ua)")
        - fvm::Sp(fvc::div(phiRa), Ua)
        + (fvc::grad(alpha*rhoa)/(fvc::average(alpha)
            + scalar(1e-12))/rhoa & Rca)
```

```
        ==
//          g       // Buoyancy and pressure term transfered to p-eqn
        - fvm::Sp(1.0/rhoa*K, Ua)
        + 1.0/rhoa*K*Ub
        - 1.0/rhoa*liftCoeff
        + AvmAlpha/rhoa*DDtUb
        + 1.0/rhoa*wallForceCoeff
        + 1.0/(alpha+scalar(1e-12))/rhoa*turbulentDispersionForce
        + GammaVL/(alpha+scalar(1e-12))/rhoa*Ub
        - fvm::SuSp(GammaVL/(alpha+scalar(1e-12))/rhoa, Ua)
      volTensorField gradUbT = fvc::grad(Ub)().T();
      volTensorField Rcb
      (
          "Rcb",
          ((2.0/3.0)*I)*(k + nuEffb*tr(gradUbT)) - nuEffb*gradUbT
      );
      surfaceScalarField phiRb =
        -fvc::interpolate(nuEffb)*mesh.magSf()*fvc::snGrad(beta*rhob)
        /fvc::interpolate(beta + scalar(1e-12))/fvc::interpolate(rhob);

      UbEqn =
      (
          (scalar(1) + AvmAlpha*alpha/(beta+scalar(1e-12))/rhob)*
          (
              fvm::ddt(Ub)
            + fvm::div(phib, Ub, "div(phib,Ub)")
            - fvm::Sp(fvc::div(phib), Ub)
          )
        - fvm::laplacian(nuEffb, Ub, "laplacian(nuEffb,Ub)")
        + fvc::div(Rcb)

        + fvm::div(phiRb, Ub, "div(phib,Ub)")
        - fvm::Sp(fvc::div(phiRb), Ub)

        + (fvc::grad(beta*rhob)/(fvc::average(beta)
            + scalar(1e-12))/rhob & Rcb)
        ==
//          g       // Buoyancy and pressure term transfered to p-eqn
        - fvm::Sp(alpha*K/rhob/(beta+scalar(1e-12)), Ub)
        + alpha*K/rhob/(beta+scalar(1e-12))*Ua
        + alpha/rhob/(beta+scalar(1e-12))*(liftCoeff + AvmAlpha*DDtUa)
        - alpha/(beta+scalar(1e-12))/rhob*wallForceCoeff
        - 1.0/(beta+scalar(1e-12))/rhob*turbulentDispersionForce
        + GammaLV/(beta+scalar(1e-12))/rhob*Ua
        - fvm::SuSp(GammaLV/(beta+scalar(1e-12))/rhob, Ub)
      );
    }
    if (oCorr == nOuterCorr-1)
    {
```

```
        UaEqn.relax(1);
        UbEqn.relax(1);
    }
    else
    {
        UaEqn.relax();
        UbEqn.relax();
    }
}
```

The `"UEqns_Michta.H"` uses the momentum equation as formulated in Michta's code [14], where also the explicit drag term is moved to the pressure equation. Therefore the implemented momentum equation for the dispersed phase (and analogously for the continuous phase) reads:

```
        UaEqn =
        (
            (scalar(1) + AvmAlpha/rhoa)*
            (
                fvm::ddt(Ua)
              + fvm::div(phia, Ua, "div(phia,Ua)")
              - fvm::Sp(fvc::div(phia), Ua)
            )
          - fvm::laplacian(nuEffa, Ua, "laplacian(nuEffa,Ua)")
          + fvc::div(Rca)

          + fvm::div(phiRa, Ua, "div(phia,Ua)")
          - fvm::Sp(fvc::div(phiRa), Ua)
          + (fvc::grad(alpha*rhoa)/(fvc::average(alpha)
              + scalar(1e-12))/rhoa & Rca)
         ==
//           g       // Buoyancy and pressure term transfered to p-eqn
          - fvm::Sp(1.0/rhoa*K, Ua)
//                + 1.0/rhoa*K*Ub     // Explicit drag transfered to p-eqn
          - 1.0/rhoa*liftCoeff
          + AvmAlpha/rhoa*DDtUb
          + 1.0/rhoa*wallForceCoeff
          + 1.0/(alpha+scalar(1e-12))/rhoa*turbulentDispersionForce
          + GammaVL/(alpha+scalar(1e-12))/rhoa*Ub
          - fvm::SuSp(GammaVL/(alpha+scalar(1e-12))/rhoa, Ua)
        );
```

The `"UEqns_Rusche.H"` uses the momentum equation as suggested by Rusche in [8], where both the explicit drag term and the turbulent dispersion force term are moved to the pressure equation:

```
        UaEqn =
        (
            (scalar(1) + AvmAlpha/rhoa)*
            (
                fvm::ddt(Ua)
              + fvm::div(phia, Ua, "div(phia,Ua)")
              - fvm::Sp(fvc::div(phia), Ua)
            )
```

```
        - fvm::laplacian(nuEffa, Ua, "laplacian(nuEffa,Ua)")
        + fvc::div(Rca)

        + fvm::div(phiRa, Ua, "div(phia,Ua)")
        - fvm::Sp(fvc::div(phiRa), Ua)
        + (fvc::grad(alpha*rhoa)/(fvc::average(alpha)
           + scalar(1e-6))/rhoa & Rca)
       ==
//         g      // Buoyancy and pressure term transfered to p-eqn
        - fvm::Sp(1.0/rhoa*K, Ua)
//             + 1.0/rhoa*K*Ub   // Explicit drag transfered to p-eqn
        - 1.0/rhoa*liftCoeff
        + AvmAlpha/rhoa*DDtUb
        + 1.0/rhoa*wallForceCoeff
//  + 1.0/alpha/rhoa*turbulentDispersionForce //TDforce moved to p-eqn
        + GammaVL/(alpha+scalar(1e-6))/rhoa*Ub
        - fvm::SuSp(GammaVL/(alpha+scalar(1e-6))/rhoa, Ua)
      );
```

The `"UEqns_RhieChowLift.H"` uses a Rhie-Chow like treatment of the momentum equation, where both the explicit drag, the turbulent dispersion force and the lift force are moved to the pressure equation:

```
       UaEqn =
      (
         (scalar(1) + AvmAlpha/rhoa)*
         (
            fvm::ddt(Ua)
          + fvm::div(phia, Ua, "div(phia,Ua)")
          - fvm::Sp(fvc::div(phia), Ua)
         )

        - fvm::laplacian(nuEffa, Ua, "laplacian(nuEffa,Ua)")
        + fvc::div(Rca)

        + fvm::div(phiRa, Ua, "div(phia,Ua)")
        - fvm::Sp(fvc::div(phiRa), Ua)
        + (fvc::grad(alpha*rhoa)/(fvc::average(alpha) + scalar(1e-6))/rhoa & Rca)
       ==
//         g      // Buoyancy and pressure term transfered to p-eqn
        - fvm::Sp(1.0/rhoa*K, Ua)
//             + 1.0/rhoa*K*Ub       // Explicit drag transfered to p-eqn
//       - 1.0/rhoa*liftCoeff    //Lift force transfered to p-eqn
        + AvmAlpha/rhoa*DDtUb
        + 1.0/rhoa*wallForceCoeff
//  + 1.0/alpha/rhoa*turbulentDispersionForce //TDforce moved to p-eqn
        + GammaVL/(alpha+scalar(1e-6))/rhoa*Ub
        - fvm::SuSp(GammaVL/(alpha+scalar(1e-6))/rhoa, Ua)
      );
```

APPENDIX D

THE IMPLEMENTED PRESSURE EQUATION

Three different `"pEqn.H"` files are implemented consistently with the three different formulations for the momentum equation.

The `"pEqn_Standard.H"` associated to the correspondent momentum equation reads:

```
{
    surfaceScalarField alphaf = fvc::interpolate(alpha + scalar(1e-12));
    surfaceScalarField betaf = scalar(1) - alphaf;

    volScalarField rUaA = 1.0/UaEqn.A();
    volScalarField rUbA = 1.0/UbEqn.A();

    phia.storePrevIter();
    phib.storePrevIter();

    phia == (fvc::interpolate(Ua) & mesh.Sf());
    phib == (fvc::interpolate(Ub) & mesh.Sf());

    surfaceScalarField rUaAf = fvc::interpolate(rUaA);
    surfaceScalarField rUbAf = fvc::interpolate(rUbA);

    Ua = rUaA*UaEqn.H();
    Ub = rUbA*UbEqn.H();

    surfaceScalarField rhoaf = fvc::interpolate(rhoa);
    surfaceScalarField rhobf = fvc::interpolate(rhob);

    surfaceScalarField phiDraga = rUaAf*(g & mesh.Sf());

    surfaceScalarField phiDragb = rUbAf*(g & mesh.Sf());

    // Fix for gravity on outlet boundary.
    forAll(p.boundaryField(), patchi)
    {
```

```
if (isA<zeroGradientFvPatchScalarField>(p.boundaryField()[patchi]))
{
    phiDraga.boundaryField()[patchi] = 0.0;
    phiDragb.boundaryField()[patchi] = 0.0;
}
}

phia = (fvc::interpolate(Ua) & mesh.Sf())
     + fvc::ddtPhiCorr(rUaA, rhoa*alpha, Ua, phia)/rhoaf/alphaf
     + phiDraga;

phib = (fvc::interpolate(Ub) & mesh.Sf())
     + fvc::ddtPhiCorr(rUbA, rhob*beta, Ub, phib)/rhobf/betaf
     + phiDragb;

phi = alphaf*phia + betaf*phib;

surfaceScalarField Dp("(rho*(1|A(U)))",alphaf*rUaAf/rhoaf
                                     + betaf*rUbAf/rhobf);

for(int nonOrth=0; nonOrth<=nNonOrthCorr; nonOrth++)
{
    fvScalarMatrix pEqn
    (
        fvm::laplacian(Dp, p)
      ==
        fvc::div(phi)
      + alpha/rhoa*(fvm::ddt(psia, p))
      + beta/rhob*(fvm::ddt(psib, p))
      + alpha/rhoa*(fvc::div(phia.prevIter()*rhoaf)
      - fvc::div(phia.prevIter())*rhoa)
      + beta/rhob*(fvc::div(phib.prevIter()*rhobf)
      - fvc::div(phib.prevIter())*rhob)
      - (GammaVL-GammaLV)*(1.0/rhoa-1.0/rhob)
    );
    pEqn.setReference(pRefCell, pRefValue);

    if
    (
        oCorr == nOuterCorr-1
        && corr == nCorr-1
        && nonOrth == nNonOrthCorr
    )
    {
        pEqn.solve(mesh.solver("pFinal"));
    }
    else
    {
        pEqn.solve();
```

```
        }
        if (nonOrth == nNonOrthCorr)
        {
            surfaceScalarField SfGradp = pEqn.flux()/Dp;

            phia -= rUaAf*SfGradp/rhoaf;
            phib -= rUbAf*SfGradp/rhobf;
            phi = alphaf*phia + betaf*phib;

            if (oCorr != nOuterCorr-1)
            {
                // Explicitly relax pressure for momentum corrector
                p.relax();
                SfGradp = pEqn.flux()/Dp;
            }

            Ua += fvc::reconstruct(phiDraga - rUaAf*SfGradp/rhoaf);
            Ua.correctBoundaryConditions();

            Ub += fvc::reconstruct(phiDragb - rUbAf*SfGradp/rhobf);
            Ub.correctBoundaryConditions();

            U = alpha*Ua + beta*Ub;
        }
    }
    if (detailedMonitor)
    {
        #include "mntp.H"
    }
    p = max(p, pMin);

    pa = p;
    pb = p;

    #include "rhoEqns.H"
    #include "compressibleContinuityErrs.H"

    rhoa = phasea->rho();
    rhob = phaseb->rho();
    rhoa.correctBoundaryConditions();
    rhob.correctBoundaryConditions();

    rho = alpha*rhoa + beta*rhob;

    Prandtlb = Cpb*rhob*nub/lambdab;

    #include "DrhoDt.H"
}
```

The term ddtPhiCorr introduces corrections taking into account the difference between the flux calculated with interpolated velocities at the cell face and the real flux [14], moreover it checks for the dimensional units of **U** and ϕ to decide what operation has to be performed, and if ϕ is defined in terms of mass, a division of ϕ by ρ is performed [45].

The `"pEqn_Michta.H"` corrects the fluxes also with the contribution of the explicit drag term removed from the correspondent momentum equation. Therefore the only change in the code consists in the definition of the correction fluxes *phiDraga* and *phiDragb*:

```
surfaceScalarField phiDraga = rUaAf*(g & mesh.Sf())
            + fvc::interpolate(1/rhoa*K*rUaA)*phib;
surfaceScalarField phiDragb = rUbAf*(g & mesh.Sf())
            + fvc::interpolate(alpha/rhob/beta*K*rUbA)*phia;
```

The `"pEqn_Rusche.H"` uses the same code as in `"pEqn_Michta.H"` but it corrects the fluxes also with the contribution of the turbulent dispersion force with the procedure deduced from [8].

```
surfaceScalarField phiTDf = - fvc::interpolate(ATD)
                    * mesh.magSf()*fvc::snGrad(alpha);

phia = (fvc::interpolate(Ua) & mesh.Sf())
    + fvc::ddtPhiCorr(rUaA, rhoa*alpha, Ua, phia)/rhoaf/alphaf
    + phiDraga
    + rUaAf*phiTDf/rhoaf/alphaf;

phib = (fvc::interpolate(Ub) & mesh.Sf())
    + fvc::ddtPhiCorr(rUbA, rhob*beta, Ub, phib)/rhobf/betaf
    + phiDragb
    - rUbAf*phiTDf/rhobf/betaf;
```

Furthermore this contribution from the turbulent dispersion force is also added in the *reconstruct* procedure for the velocity fields.

```
Ua += fvc::reconstruct(phiDraga -rUaAf*SfGradp/rhoaf
                    +rUaAf*phiTDf/rhoaf/alphaf);
Ub += fvc::reconstruct(phiDragb -rUbAf*SfGradp/rhobf
                   -rUbAf*phiTDf/rhobf/betaf);
```

The `"pEqn_RhieChowLift.H"` uses the same code as in `"pEqn_Rusche.H"` but it corrects the fluxes also with the contribution of the lift force.

```
surfaceScalarField phiLift = - alphaf
                * (fvc::interpolate(liftCoeff) & mesh.Sf());

phia = (fvc::interpolate(Ua) & mesh.Sf())
    + fvc::ddtPhiCorr(rUaA, rhoa*alpha, Ua, phia)/rhoaf/alphaf
    + phiDraga
    + rUaAf*phiTDf/rhoaf/alphaf
    + rUaAf*phiLift/rhoaf/alphaf;

phib = (fvc::interpolate(Ub) & mesh.Sf())
    + fvc::ddtPhiCorr(rUbA, rhob*beta, Ub, phib)/rhobf/betaf
    + phiDragb
    - rUbAf*phiTDf/rhobf/betaf
```

```
    - rUbAf*phiLift/rhobf/betaf;
```

The lift force is also added in the *reconstruct* procedure for the velocity fields:

```
Ua += fvc::reconstruct(phiDraga - rUaAf*SfGradp/rhoaf
                     + rUaAf*phiTDf/rhoaf/alphaf
                     + rUaAf*phiLift/rhoaf/alphaf);
Ub += fvc::reconstruct(phiDragb - rUbAf*SfGradp/rhobf
                     - rUbAf*phiTDf/rhobf/betaf
                     - rUbAf*phiLift/rhobf/betaf);
```

After the calculation of the pressure and of the corrected velocity field, the densities of the compressible phases are updated in the `"rhoEqns.H"` file that reads:

```
{
surfaceScalarField alphaf = fvc::interpolate(alpha);
surfaceScalarField betaf = scalar(1) - alphaf;

solve(fvm::ddt(alpha,rhoa) + fvc::div(alphaf*phia,rhoa,"div(phia,rhoa)")
    == GammaVL-GammaLV);
solve(fvm::ddt(beta, rhob) + fvc::div(betaf*phib, rhob,"div(phib,rhob)")
    == GammaLV-GammaVL);
}
```

where the continuity equations are solved in terms of the densities instead of the void fractions.

APPENDIX E

THE IMPLEMENTED INTERFACIAL AREA CONCENTRATION EQUATION

```
DS.storePrevIter();
IAC.storePrevIter();

if (haveIACeqn)
{
    if (haveBreakupAndCoal)
    {
        # include "breakupAndCoalescenceModel.H"
    }
    fvScalarMatrix IACEqn(IAC, IAC.dimensions()*dimVol/dimTime);
    {
        {
        pressureChangeSource = -2.0/3.0*DrhoaDt*IAC;
        bulkCondensationSource = - 2.0/3.0/(alpha+scalar(1e-12))/rhoa
                            * (GammaLV-GammaVL)*pos(-SfPerVol)*IAC;

            IACEqn =
            (
              fvm::ddt(IAC)
            + fvm::div(phia, IAC, "div(phia,IAC)")
           ==
            - fvm::SuSp(2.0/3.0/(alpha+scalar(1e-12))/rhoa
                    *(GammaLV-GammaVL)*pos(-SfPerVol), IAC)
            - fvm::SuSp(2.0/3.0*DrhoaDt, IAC)
            + coalescenceSource
            + breakupSource
            + nucleationSource
            );
        }
        if (oCorr == nOuterCorr-1)
        {
          IACEqn.relax(1);
```

```
        }
        else
        {
            IACEqn.relax();
        }
        IACEqn.solve();
    }
    dimensionedScalar smallIAC
    (
        "smallIAC",
        dimensionSet(0, -1, 0, 0, 0, 0, 0),
        scalar (1.0e-12)
    );
    DS = 6.0*min(alpha,beta)/(IAC+smallIAC);

    if (DSmin.value()>0.0)
    {
        DS = max(DS,DSmin);
    }
    if (DSmax.value()>0.0)
    {
        DS = min(DS,DSmax);
    }
}
else
{
    DS = Dsinput;
    IAC = 6.0*alpha/DS;
}
```

APPENDIX F

THE IMPLEMENTED $K - \epsilon$ TURBULENCE MODELS

The **standard $k - \epsilon$ turbulence model (Kai's version)** reads:

```
if(turbulence)
{
    if (mesh.changing())
    {
        y.correct();
    }
tmp<volTensorField> tgradUb = fvc::grad(Ub);
volScalarField G("RASModel::G",nutb*(tgradUb()&&dev(twoSymm(tgradUb()))));
tgradUb.clear();

    #include "wallFunctions.H"

    volScalarField divUb = fvc::div(phib);

surfaceScalarField phiEb =
  -alphaEps*fvc::interpolate(nuEffb)*mesh.magSf()*fvc::snGrad(beta*rhob)
   /fvc::interpolate(beta + scalar(1e-12))/fvc::interpolate(rhob);

    if (detailedMonitor)
    {
        #include "mntk.H"
    }
    fvScalarMatrix epsEqn
    (
        fvm::ddt(epsilon)
      + fvm::div(phib, epsilon, "div(phib,epsilon)")
      - fvm::Sp(divUb, epsilon)
      - fvm::laplacian
        (
            alphaEps*nuEffb, epsilon,
            "laplacian(DepsilonEff,epsilon)"
        )
```

```
      + fvm::div(phiEb, epsilon, "div(phib,epsilon)")
      - fvm::Sp(fvc::div(phiEb), epsilon)
      ==
        C1*G*epsilon/k
      - fvm::Sp(C2*epsilon/k, epsilon)
      - fvm::Sp(GammaLV/(beta+scalar(1e-12))/rhob, epsilon)
      + GammaVL/(beta+scalar(1e-12))/rhob*epsilon
    );

    #include "wallDissipation.H"

    if (oCorr == nOuterCorr-1)
    {
        epsEqn.relax(1);
    }
    else
    {
        epsEqn.relax();
    }
    epsEqn.solve();

epsilon.max(dimensionedScalar("zero", epsilon.dimensions(), 1.0e-15));

surfaceScalarField phiVb =
   -alphak*fvc::interpolate(nuEffb)*mesh.magSf()*fvc::snGrad(beta*rhob)
   fvc::interpolate(beta + scalar(1e-12))/fvc::interpolate(rhob);

    fvScalarMatrix kEqn
    (
        fvm::ddt(k)
      + fvm::div(phib, k, "div(phib,k)")
      - fvm::Sp(divUb, k)
      - fvm::laplacian
        (
            alphak*nuEffb, k,
            "laplacian(DkEff,k)"
        )
      + fvm::div(phiVb, k, "div(phib,k)")
      - fvm::Sp(fvc::div(phiVb), k)
      ==
        G
      - fvm::Sp(epsilon/k, k)
      - fvm::Sp(GammaLV/beta/rhob, k)
      + GammaVL/beta/rhob*k
    );
    if (oCorr == nOuterCorr-1)
    {
        kEqn.relax(1);
    }
```

```
    else
    {
        kEqn.relax();
    }
    kEqn.solve();

    k.max(dimensionedScalar("zero", k.dimensions(), 1.0e-8));

    //- Re-calculate turbulence viscosity
    nutb = Cmu*sqr(k)/epsilon;

    #include "wallViscosity.H"
}
{
    volVectorField Ur = Ua - Ub;
    volScalarField magUr = mag(Ur);

    nuEffb = nutb + nub;
    if(bubbleAgitation)
        nuEffb += 1.2*DS/2.0*alpha*magUr;
}
nuta = sqr(Ct)*nutb;
nuEffa = nuta + nua;

kappaEffb = nutb/PrandtlbT + lambdab/(rhob*Cpb);
kappaEffa = nuta/PrandtlbT + lambdaa/(rhoa*Cpa);
{
    wallDist yField(mesh);
    dimensionedScalar Cmu25 = pow(Cmu, 0.25);
    Ubf = sqrt(k)*Cmu25;
    if ( maxUbf.value()>0.0 )
    {
        Ubf = min (Ubf,maxUbf);
    }
    if ( minUbf.value()>0.0 )
    {
        Ubf = max (Ubf,minUbf);
    }
    yPlusField = yField.y()*Ubf/nub;
}
```

The **standard $k-\epsilon$ turbulence model (Ghione's version)** uses transport equations in the following form:

```
fvScalarMatrix epsEqn
    (
        fvm::ddt(epsilon)
      + fvm::div(phib, epsilon, "div(phib,epsilon)")
      - fvm::Sp(divUb, epsilon)
      - fvm::laplacian
```

```
    (
        alphaEps*nutb + nub, epsilon,
        "laplacian(DepsilonEff,epsilon)"
    )
  + fvm::div(phiEb, epsilon, "div(phib,epsilon)")
  - fvm::Sp(fvc::div(phiEb), epsilon)
  ==
    C1*G*epsilon/k
  - fvm::Sp(C2*epsilon/k, epsilon)
  - fvm::Sp(GammaLV/(beta+scalar(1e-12))/rhob, epsilon)
  + GammaVL/(beta+scalar(1e-12))/rhob*epsilon
  - fvm::SuSp((2.0/3.0)*C1*divUb, epsilon)
);

fvScalarMatrix kEqn
(
    fvm::ddt(k)
  + fvm::div(phib, k, "div(phib,k)")
  - fvm::Sp(divUb, k)
  - fvm::laplacian
    (
        alphak*nutb + nub, k,
        "laplacian(DkEff,k)"
    )
  + fvm::div(phiVb, k, "div(phib,k)")
  - fvm::Sp(fvc::div(phiVb), k)
  ==
    G
  - fvm::Sp(epsilon/k, k)
  - fvm::Sp(GammaLV/beta/rhob, k)
  + GammaVL/beta/rhob*k
  - fvm::SuSp(2.0/3.0*divUb, k)
);
```

The **Yao-Morel modified $k-\epsilon$ turbulence model** uses transport equations in the following form:

```
 fvScalarMatrix epsEqn
(
    fvm::ddt(epsilon)
  + fvm::div(phib, epsilon, "div(phib,epsilon)")
  - fvm::Sp(divUb, epsilon)
  - fvm::laplacian
    (
        alphaEps*nutb + nub, epsilon,
        "laplacian(DepsilonEff,epsilon)"
    )
  + fvm::div(phiEb, epsilon, "div(phib,epsilon)")
  - fvm::Sp(fvc::div(phiEb), epsilon)
  ==
```

```
        C1*G*epsilon/k
      - fvm::Sp(C2*epsilon/k, epsilon)
      - fvm::SuSp((2.0/3.0)*C1*divUb, epsilon)
      - C3 * ((dragForce+virtualMassForce) & Ur)
      * pow(epsilon/sqr(DS), 1.0/3.0)/(beta + scalar(1e-12))/rhob
    );

    fvScalarMatrix kEqn
    (
        fvm::ddt(k)
      + fvm::div(phib, k, "div(phib,k)")
      - fvm::Sp(divUb, k)
      - fvm::laplacian
        (
            alphak*nutb + nub, k,
            "laplacian(DkEff,k)"
        )
      + fvm::div(phiVb, k, "div(phib,k)")
      - fvm::Sp(fvc::div(phiVb), k)
      ==
        G
      - fvm::Sp(epsilon/k, k)
      - fvm::SuSp(2.0/3.0*divUb, k)
      - sigb*(coalescenceSource+breakupSource)/(beta+scalar(1e-12))/rhob
      - ((dragForce+virtualMassForce) & Ur)/(beta + scalar(1e-12))/rhob
    );
```

The file **wallFunction.H** reads:

```
{
    labelList cellBoundaryFaceCount(epsilon.size(), 0);
    scalar Cmu25 = ::pow(Cmu.value(), 0.25);
    scalar Cmu75 = ::pow(Cmu.value(), 0.75);
    scalar kappa_ = kappa.value();

    const fvPatchList& patches = mesh.boundary();

    //- Initialise the near-wall P field to zero
    forAll(patches, patchi)
    {
        const fvPatch& currPatch = patches[patchi];

        if (isA<wallFvPatch>(currPatch))
        {
            forAll(currPatch, facei)
            {
            label faceCelli = currPatch.faceCells()[facei];

            epsilon[faceCelli] = 0.0;
            G[faceCelli] = 0.0;
```

```
        }
    }
}
epsilon.correctBoundaryConditions();
G.correctBoundaryConditions();

//- Accumulate the wall face contributions to epsilon and G
//  Increment cellBoundaryFaceCount for each face for averaging
forAll(patches, patchi)
{
    const fvPatch& currPatch = patches[patchi];

    if (isA<wallFvPatch>(currPatch))
    {
    const scalarField& nuw = nutb.boundaryField()[patchi];

    scalarField magFaceGradU =mag(U.boundaryField()[patchi].snGrad());

        forAll(currPatch, facei)
        {
            label faceCelli = currPatch.faceCells()[facei];

            scalar yPlus =
                Cmu25*y[patchi][facei]
                *::sqrt(k[faceCelli])
                /nub.boundaryField()[patchi][facei];

            // For corner cells (with two boundary or more faces),
            // epsilon and G in the near-wall cell are calculated
            // as an average

            cellBoundaryFaceCount[faceCelli]++;

            epsilon[faceCelli] +=
                 Cmu75*::pow(k[faceCelli], 1.5)
                /(kappa_*y[patchi][facei]);

            if (yPlus > 11.6)
            {
                G[faceCelli] +=
                    nuw[facei]*magFaceGradU[facei]
                    *Cmu25*::sqrt(k[faceCelli])
                    /(kappa_*y[patchi][facei]);
            }
        }
    }
}
epsilon.correctBoundaryConditions();
G.correctBoundaryConditions();
```

```
    // perform the averaging

    forAll(patches, patchi)
    {
        const fvPatch& curPatch = patches[patchi];

        if (isA<wallFvPatch>(curPatch))
        {
            forAll(curPatch, facei)
            {
                label faceCelli = curPatch.faceCells()[facei];

                epsilon[faceCelli] /= cellBoundaryFaceCount[faceCelli];
                G[faceCelli] /= cellBoundaryFaceCount[faceCelli];
            }
        }
    }
    epsilon.correctBoundaryConditions();
    G.correctBoundaryConditions();
}
```

The file **wallDissipation.H** reads:

```
{
    const fvPatchList& patches = mesh.boundary();

    forAll(patches, patchi)
    {
        const fvPatch& p = patches[patchi];

        if (isA<wallFvPatch>(p))
        {
            epsEqn.setValues
            (
                p.faceCells(),
                epsilon.boundaryField()[patchi].patchInternalField()
            );
        }
    }
}
```

The file **wallViscosity.H** reads:

```
{
    scalar Cmu25 = ::pow(Cmu.value(), 0.25);
    scalar kappa_ = kappa.value();
    scalar E_ = E.value();

    const fvPatchList& patches = mesh.boundary();

    forAll(patches, patchi)
```

```
{
    const fvPatch& currPatch = patches[patchi];

    if (isA<wallFvPatch>(currPatch))
    {
        scalarField& nutw = nutb.boundaryField()[patchi];

        forAll(currPatch, facei)
        {
            label faceCelli = currPatch.faceCells()[facei];

            // calculate yPlus
            scalar yPlus =
                Cmu25*y[patchi][facei]
               *::sqrt(k[faceCelli])
               /nub.boundaryField()[patchi][facei];

            if (yPlus > 11.6)
            {
            nutw[facei] =
                yPlus*  nub.boundaryField()[patchi][facei]*kappa_
                /::log(E_*yPlus)
                - nub.boundaryField()[patchi][facei];
            }
            else
            {
                nutw[facei] = 0.0;
            }
        }
    }
}
nutb.correctBoundaryConditions();
```

APPENDIX G

THE IMPLEMENTED INTERFACIAL MOMENTUM TRANSFER COEFFICIENTS

```
    volVectorField Ur = Ua - Ub;
    volScalarField magUr = mag(Ur);

//************************ DRAG FORCE ******************************//

    volScalarField Ka = draga->K(magUr,DS);
    K = Ka;

    if (dragPhase == "b")
    {
        volScalarField Kb = dragb->K(magUr,DS);
        K = Kb;
    }
    else if (dragPhase == "blended")
    {
        volScalarField Kb = dragb->K(magUr,DS);
        K = (beta*Ka + alpha*Kb);
    }
    if (!haveDragForce)
    {
        K *= 0.0;
    }
    dragForce = -alpha*K*Ur;

//************************ LIFT FORCE ******************************//
    if (fixedCl)
    {
        volScalarField Cl0
        (
            IOobject
            (
                "Cl0",
```

```
            runTime.timeName(),
            mesh,
            IOobject::NO_READ,
            IOobject::NO_WRITE
        ),
        mesh,
        dimensionedScalar
        (
            transportProperties.lookup("Cl")
        )
    );
    Cl = Cl0;
}
else
{
    Cl = lifta->AL(magUr,DS);
}
volVectorField liftCoeff = Cl*rhob*(Ur ^ fvc::curl(Ub));

if (!haveLiftForce)
{
    liftCoeff *= 0.0;
}
if (liftDamping)
{
    liftCoeff *= pos(-SfPerVol);
}
liftForce = -alpha*liftCoeff;

//********************** VIRTUAL MASS FORCE ***************************//

AvmAlpha = virtualMassa->AvmAlpha();

virtualMassForce = alpha*AvmAlpha*(DDtUb-DDtUa);

//****************** TURBULENT DISPERSION FORCE ***********************//

if (haveTurbulentForce)
{
    ATD = turbulentDispersiona->ATD(magUr,nutb,DS,Ka);

    volVectorField gradAlpha = -fvc::grad(alpha);

    if (maxTurbForce.value()>0.0)
    {
        dimensionedScalar smallMagGrad
        (
            "smallMagGrad",
            dimensionSet(0, -1, 0, 0, 0, 0, 0),
```

```
                scalar (1.0e-8)
            );
        turbulentDispersionForce = min(mag(ATD*gradAlpha),maxTurbForce)
                                 * gradAlpha/(mag(gradAlpha)+smallMagGrad);
        }
        else
        {
            turbulentDispersionForce = ATD*gradAlpha;
        }
        if (TDForceDamping)
        {
            turbulentDispersionForce *= pos(-SfPerVol);
        }
    }
//****************** WALL LUBRICATION FORCE ************************//

    Cwl = wallLubricationa->AWL(DS);
    {
        const fvPatchList& patches = mesh.boundary();

        forAll(patches, patchi)
        {
            const fvPatch& currPatch = patches[patchi];

            if (isA<wallFvPatch>(currPatch))
            {
                forAll(currPatch, facei)
                {
                    Cwl.boundaryField()[patchi][facei] *= 0.0;
                }
            }
        }
        Cwl.correctBoundaryConditions();
    }
volScalarField wallForceMag =Cwl*rhob*sqr(mag(Ur-(Ur & yr.n())*yr.n()));

    if (maxWallForce.value()>0.0)
    {
        wallForceMag = min(wallForceMag, maxWallForce);
    }

    if (maxMagUr.value()>0.0)
    {
        wallForceMag = min( wallForceMag, Cwl*rhob*sqr(maxMagUr));
    }

    wallForceMag = wallForceMag * pos(alpha-alphacr)
                + wallForceMag *(alpha-alphacr/100.0)/(0.99*alphacr)
                    * neg(alpha-alphacr)*pos(alpha-alphacr/100.0);
```

```
volVectorField wallForceCoeff = wallForceMag*(-yr.n());

if (!haveWallForce)
{
    wallForceCoeff *= 0.0;
}
wallLubricationForce = wallForceCoeff*alpha;
```

APPENDIX H

THE IMPLEMENTED DRAG MODELS

The implemented **Schiller-Naumann drag model** reads:

```
#include "SchillerNaumann.H"
#include "addToRunTimeSelectionTable.H"

// * * * * * * * * * * * * * Static Data Members * * * * * * * * * * * //
namespace Foam
{
    defineTypeNameAndDebug(SchillerNaumann, 0);

    addToRunTimeSelectionTable
    (
        dragModel,
        SchillerNaumann,
        dictionary
    );
}
// * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * //

Foam::SchillerNaumann::SchillerNaumann
(
    const dictionary& interfaceDict,
    const volScalarField& alpha,
    const phaseModel& phasea,
    const phaseModel& phaseb
)
:
    dragModel(interfaceDict, alpha, phasea, phaseb)
{}
// * * * * * * * * * * * * * * * Destructor  * * * * * * * * * * * * * //

Foam::SchillerNaumann::~SchillerNaumann()
{}
// * * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * * //
```

```
Foam::tmp<Foam::volScalarField> Foam::SchillerNaumann::K
(
    const volScalarField& Ur,
    const volScalarField& DS
) const
{
volScalarField Re = max(Ur*DS/phaseb_.nu(), scalar(1.0e-3));

volScalarField Cds = neg(Re-1000)*24.0*(scalar(1)+0.15*pow(Re,0.687))/Re
                   + pos(Re-1000)*0.44;

return 0.75*Cds*phaseb_.rho()*Ur/DS;
```

The implemented **Ishii-Zuber drag model for spherical solid particles** reads:

```
#include "IshiiZuber.H"
#include "addToRunTimeSelectionTable.H"
// * * * * * * * * * * * * * Static Data Members * * * * * * * * * * //
namespace Foam
{
    defineTypeNameAndDebug(IshiiZuber, 0);

    addToRunTimeSelectionTable
    (
        dragModel,
        IshiiZuber,
        dictionary
    );
}
// * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * //
Foam::IshiiZuber::IshiiZuber
(
    const dictionary& interfaceDict,
    const volScalarField& alpha,
    const phaseModel& phasea,
    const phaseModel& phaseb
)
:
    dragModel(interfaceDict, alpha, phasea, phaseb),
    alphaMax_( readScalar(interfaceDict_.lookup("alphaMax")) )
{}
// * * * * * * * * * * * * * * * Destructor  * * * * * * * * * * * * //
Foam::IshiiZuber::~IshiiZuber()
{}
// * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * //
Foam::tmp<Foam::volScalarField> Foam::IshiiZuber::K
(
   const volScalarField& Ur,
   const volScalarField& DS
```

```
) const
{
volScalarField mua = phasea_.rho()*phasea_.nu();
volScalarField mub = phaseb_.rho()*phaseb_.nu();
volScalarField mustar = (mua+0.4*mub)/(mua+mub);
volScalarField mumix= mub *
        pow(max(1e-8,1.0-alpha_/alphaMax_),-2.5*alphaMax_*mustar);
volScalarField Remix = max(Ur*DS*phaseb_.rho()/mumix,scalar(1.0e-3));
volScalarField Cds = neg(Remix-1000)*24.0*(scalar(1)
                + 0.15*pow(Remix,0.687))/Remix
                + pos(Remix-1000)*0.44;

return 0.75*Cds*phaseb_.rho()*Ur/DS;
}
```

The implemented **Wen-Yu drag model** reads:

```
#include "WenYu.H"
#include "addToRunTimeSelectionTable.H"
// * * * * * * * * * * * * * Static Data Members * * * * * * * * * * * //
namespace Foam
{
    defineTypeNameAndDebug(WenYu, 0);


    addToRunTimeSelectionTable
    (
        dragModel,
        WenYu,
        dictionary
    );
}
// * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * //
Foam::WenYu::WenYu
(
    const dictionary& interfaceDict,
    const volScalarField& alpha,
    const phaseModel& phasea,
    const phaseModel& phaseb
)
:
    dragModel(interfaceDict, alpha, phasea, phaseb)
{}
// * * * * * * * * * * * * * * * Destructor  * * * * * * * * * * * * * //
Foam::WenYu::~WenYu()
{}
// * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * * //
Foam::tmp<Foam::volScalarField> Foam::WenYu::K
(
    const volScalarField& Ur,
```

```
    const volScalarField& DS
) const
{
    volScalarField beta(max(scalar(1) - alpha_, scalar(1.0e-6)));
    volScalarField bp(pow(beta, -2.65));
    volScalarField Re(max(Ur*DS/phaseb_.nu(), scalar(1.0e-3)));
    volScalarField Cds
    (
        neg(Re - 1000)*(24.0*(1.0 + 0.15*pow(Re, 0.687))/Re)
      + pos(Re - 1000)*0.44
    );
    return 0.75*Cds*phaseb_.rho()*Ur*bp/DS;
}
```

The implemented **Ishii-Zuber drag model for densely distributed fluid particles** reads:

```
#include "IshiiZuberExtended.H"
#include "addToRunTimeSelectionTable.H"

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

namespace Foam
{
    defineTypeNameAndDebug(IshiiZuberExtended, 0);

    addToRunTimeSelectionTable
    (
        dragModel,
        IshiiZuberExtended,
        dictionary
    );
}
// * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * //

Foam::IshiiZuberExtended::IshiiZuberExtended
(
    const dictionary& interfaceDict,
    const volScalarField& alpha,
    const phaseModel& phasea,
    const phaseModel& phaseb
)
:
    dragModel(interfaceDict, alpha, phasea, phaseb),
    alphaMax_( readScalar(interfaceDict_.lookup("alphaMax")) )
{}
// * * * * * * * * * * * * * * * Destructor  * * * * * * * * * * * //

Foam::IshiiZuberExtended::~IshiiZuberExtended()
{}
// * * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * //
```

```
Foam::tmp<Foam::volScalarField> Foam::IshiiZuberExtended::K
(
    const volScalarField& Ur,
    const volScalarField& DS

) const
{
volScalarField mua = phasea_.rho()*phasea_.nu();
volScalarField mub = phaseb_.rho()*phaseb_.nu();

volScalarField mustar = (mua+0.4*mub)/(mua+mub);
volScalarField mumix= mub
        * pow(max(1e-8,1.0-alpha_/alphaMax_),-2.5*alphaMax_*mustar);
volScalarField Remix = max(Ur*DS*phaseb_.rho()/mumix,scalar(1.0e-3));
volScalarField Cdssphere =24.0*(scalar(1)+0.15*pow(Remix,0.687))/Remix;

volScalarField Cdscap = pow(1.0-alpha_,2)*2.66667;

//------- gravity vector --------

    dimensionedScalar g
    (
        "g",
        dimensionSet(0, 1, -2, 0, 0, 0, 0),
        scalar (9.81)
    );

//-----------------------
volScalarField Eo = mag(g)*(phaseb_.rho()-phasea_.rho())
                * sqr(DS)/phaseb_.sig();
volScalarField Cds0 = pow(Eo,0.5)*0.66667;
volScalarField fmix = mub*pow(1.0-alpha_,0.5)/mumix;
volScalarField Emix = (1.0+17.67*pow(fmix,0.85714286))/(18.67*fmix);
volScalarField Cdsellipse = Emix * Cds0;

volScalarField Cds = pos(Cdssphere-Cdsellipse)*Cdssphere
                + neg(Cdssphere-Cdsellipse)*min(Cdsellipse,Cdscap);

return 0.75*Cds*phaseb_.rho()*Ur/DS;
}
```

The implemented **Ishii-Zuber drag model for sparsely distributed fluid particles** reads:

```
#include "IshiiZuberExtendSp.H"
#include "addToRunTimeSelectionTable.H"

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

namespace Foam
```

```
{
    defineTypeNameAndDebug(IshiiZuberExtendSp, 0);

    addToRunTimeSelectionTable
    (
        dragModel,
        IshiiZuberExtendSp,
        dictionary
    );
}
// * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * //

Foam::IshiiZuberExtendSp::IshiiZuberExtendSp
(
    const dictionary& interfaceDict,
    const volScalarField& alpha,
    const phaseModel& phasea,
    const phaseModel& phaseb
)
:
    dragModel(interfaceDict, alpha, phasea, phaseb),
    alphaMax_( readScalar(interfaceDict_.lookup("alphaMax")) )
{}
// * * * * * * * * * * * * * * * Destructor  * * * * * * * * * * * * * //

Foam::IshiiZuberExtendSp::~IshiiZuberExtendSp()
{}
// * * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * //

Foam::tmp<Foam::volScalarField> Foam::IshiiZuberExtendSp::K
(
    const volScalarField& Ur,
    const volScalarField& DS

) const
{
volScalarField mua = phasea_.rho()*phasea_.nu();
volScalarField mub = phaseb_.rho()*phaseb_.nu();

volScalarField mustar = (mua+0.4*mub)/(mua+mub);
volScalarField mumix= mub
        * pow(max(1e-8,1.0-alpha_/alphaMax_),-2.5*alphaMax_*mustar);
volScalarField Remix = max(Ur*DS*phaseb_.rho()/mumix,scalar(1.0e-3));
volScalarField Cdssphere =24.0*(scalar(1)+0.15*pow(Remix,0.687))/Remix;

volScalarField Cdscap = 2.66667*Remix/Remix;

//------- gravity vector --------
```

```
    dimensionedScalar g
    (
        "g",
        dimensionSet(0, 1, -2, 0, 0, 0, 0),
        scalar (9.81)
    );
//-------------------
volScalarField Eo = mag(g)*(phaseb_.rho()-phasea_.rho())
                * sqr(DS)/phaseb_.sig();
volScalarField Cdsellipse = pow(Eo,0.5)*0.66667;

volScalarField Cdsdist = min(Cdsellipse,Cdscap);

volScalarField Cds = max(Cdsdist,Cdssphere);

return 0.75*Cds*phaseb_.rho()*Ur/DS;
}
```

APPENDIX

THE IMPLEMENTED LIFT MODELS

The implemented **Tomiyama lift model** reads:

```
#include "liftTomiyama.H"
#include "addToRunTimeSelectionTable.H"
// * * * * * * * * * * * * * Static Data Members * * * * * * * * * * * //
namespace Foam
{
    defineTypeNameAndDebug(liftTomiyama, 0);
    addToRunTimeSelectionTable
    (
        liftModel,
        liftTomiyama,
        dictionary
    );
}
// * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * //
Foam::liftTomiyama::liftTomiyama
(
    const dictionary& interfaceDict,
    const volScalarField& alpha,
    const phaseModel& phasea,
    const phaseModel& phaseb
)
:
    liftModel(interfaceDict, alpha, phasea, phaseb)
{}
// * * * * * * * * * * * * * * * Destructor  * * * * * * * * * * * * * //
Foam::liftTomiyama::~liftTomiyama()
{}
// * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * * //
Foam::tmp<Foam::volScalarField> Foam::liftTomiyama::AL
( const volScalarField& Ur, const volScalarField& DS ) const
{
volScalarField Red = (Ur*DS)/phaseb_.nu();
```

```
volScalarField Eo = mag(g)*(phaseb_.rho()-phasea_.rho())
                  * sqr(DS)/phaseb_.sig();
volScalarField dH = DS
          * pow((scalar(1.0)+scalar(0.163)*pow(Eo,0.757)),0.333);
volScalarField EodH = mag(g)*(phaseb_.rho()-phasea_.rho())
                    * sqr(dH)/phaseb_.sig();
volScalarField fEodH = scalar(0.00105)*pow(EodH,3.0)
                     -scalar(0.0159)*sqr(EodH)
                     -scalar(0.0204)*EodH+scalar(0.474);
volScalarField Cl = fEodH;

Cl = pos(EodH-scalar(10.0))*scalar(-0.27)
   + pos(EodH-scalar(4))*neg(EodH-scalar(10))*fEodH
   + neg(EodH-scalar(4))
   * min(scalar(0.288)*tanh((scalar(0.121)*Red)),fEodH);

return  Cl*scalar(1);
}
```

The implemented **Rusche lift model** reads:

```
#include "liftRusche.H"
#include "addToRunTimeSelectionTable.H"

// * * * * * * * * * * * * * Static Data Members * * * * * * * * * * * //
namespace Foam
{
    defineTypeNameAndDebug(liftRusche, 0);

    addToRunTimeSelectionTable
    (
        liftModel,
        liftRusche,
        dictionary
    );
}
// * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * //

Foam::liftRusche::liftRusche
(
    const dictionary& interfaceDict,
    const volScalarField& alpha,
    const phaseModel& phasea,
    const phaseModel& phaseb
)
:
    liftModel(interfaceDict, alpha, phasea, phaseb)
{}
// * * * * * * * * * * * * * * * Destructor  * * * * * * * * * * * * * //
```

```
Foam::liftRusche::~liftRusche()
{}
// * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * //

Foam::tmp<Foam::volScalarField> Foam::liftRusche::AL
(
    const volScalarField& Ur,
    const volScalarField& DS

) const
{
    volScalarField  Cl = 0.000651*pow(alpha_,-1.2);

    return  Cl*scalar(1);
}
```

APPENDIX J

THE IMPLEMENTED VIRTUAL MASS MODELS

The implemented **virtual mass model according to [1]** reads:

```
#include "constVM.H"
#include "addToRunTimeSelectionTable.H"
// * * * * * * * * * * * * * Static Data Members * * * * * * * * * * //

namespace Foam
{
    defineTypeNameAndDebug(constVM, 0);

    addToRunTimeSelectionTable
    (
        virtualMassModel,
        constVM,
        dictionary
    );
}
// * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * //

Foam::constVM::constVM
(
    const dictionary& interfaceDict,
    const volScalarField& alpha,
    const phaseModel& phasea,
    const phaseModel& phaseb
)
:
    virtualMassModel(interfaceDict, alpha, phasea, phaseb),
    Cvm_( readScalar(interfaceDict_.lookup("Cvm")) )
{}
// * * * * * * * * * * * * * * Destructor  * * * * * * * * * * * * * //

Foam::constVM::~constVM()
{}
```

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

Foam::tmp<Foam::volScalarField> Foam::constVM::AvmAlpha() const
{
//Note here AvmAlpha=Avm/alpha. Virtual mass Mvm=Avm(DDtUb-DDtUa)
    return Cvm_*phaseb_.rho();
}
```

The implemented **virtual mass model according to [14]** reads:

```
#include "Zuber1964.H"
#include "addToRunTimeSelectionTable.H"

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

namespace Foam
{
    defineTypeNameAndDebug(Zuber1964, 0);

    addToRunTimeSelectionTable
    (
        virtualMassModel,
        Zuber1964,
        dictionary
    );
}
// * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * //
Foam::Zuber1964::Zuber1964
(
    const dictionary& interfaceDict,
    const volScalarField& alpha,
    const phaseModel& phasea,
    const phaseModel& phaseb
)
:
    virtualMassModel(interfaceDict, alpha, phasea, phaseb)
{}
// * * * * * * * * * * * * * * * Destructor  * * * * * * * * * * * * //

Foam::Zuber1964::~Zuber1964()
{}
// * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * //

Foam::tmp<Foam::volScalarField> Foam::Zuber1964::AvmAlpha() const
{
//Note here AvmAlpha=Avm/alpha. Virtual mass Mvm=Avm(DDtUb-DDtUa)
  return 0.5*phaseb_.rho()*(1.0+2.0*alpha_)/(1.0-alpha_);
}
```

APPENDIX K

THE IMPLEMENTED WALL LUBRICATION MODELS

The implemented **Tomiyama wall lubrication model** reads:

```
#include "wallLubricationTomiyama.H"
#include "addToRunTimeSelectionTable.H"
// * * * * * * * * * * * * * Static Data Members * * * * * * * * * * * //
namespace Foam
{
    defineTypeNameAndDebug(wallLubricationTomiyama, 0);

    addToRunTimeSelectionTable
    (
        wallLubricationModel,
        wallLubricationTomiyama,
        dictionary
    );
}
// * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * //
Foam::wallLubricationTomiyama::wallLubricationTomiyama
(
    const dictionary& interfaceDict,
    const phaseModel& phasea,
    const phaseModel& phaseb
)
:
    wallLubricationModel(interfaceDict, phasea, phaseb),
    Dpipe_(interfaceDict.lookup("Dpipe"))
{}
// * * * * * * * * * * * * * * * Destructor  * * * * * * * * * * * * * //
Foam::wallLubricationTomiyama::~wallLubricationTomiyama()
{}
// * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * * //
Foam::tmp<Foam::volScalarField> Foam::wallLubricationTomiyama::AWL
(
    const volScalarField& DS
```

```
) const
{
volScalarField Eo = mag(g)*(phaseb_.rho()-phasea_.rho())
                  * sqr(DS)/phaseb_.sig();
volScalarField CwlEo = scalar(0.47)*neg(Eo-scalar(1))
                + exp(scalar(-0.933)*Eo
                + scalar(0.179))*pos(Eo-scalar(1))*neg(Eo-scalar(5))
                + (scalar(0.00599)*Eo
                - scalar(0.0187))*pos(Eo-scalar(5))*neg(Eo-scalar(33))
                + scalar(0.179)*pos(Eo-scalar(33));
volScalarField yw = y_.y();

return 0.5*CwlEo*DS*(1.0/sqr(yw)-1.0/sqr(Dpipe_-yw));
}
```

The implemented **Frank wall lubrication model** reads:

```
#include "wallLubricationFrank.H"
#include "addToRunTimeSelectionTable.H"

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

namespace Foam
{
    defineTypeNameAndDebug(wallLubricationFrank, 0);

    addToRunTimeSelectionTable
    (
        wallLubricationModel,
        wallLubricationFrank,
        dictionary
    );
}
// * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * //

Foam::wallLubricationFrank::wallLubricationFrank
(
    const dictionary& interfaceDict,
    const phaseModel& phasea,
    const phaseModel& phaseb
)
:
    wallLubricationModel(interfaceDict, phasea, phaseb),
    Cwd_(readScalar(interfaceDict_.lookup("Cwd"))),
    Cwc_(readScalar(interfaceDict_.lookup("Cwc"))),
    p_(readScalar(interfaceDict_.lookup("p")))

// * * * * * * * * * * * * * * * Destructor  * * * * * * * * * * * * //
```

```
Foam::wallLubricationFrank::~wallLubricationFrank()
{}
// * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * //

Foam::tmp<Foam::volScalarField> Foam::wallLubricationFrank::AWL
(
    const volScalarField& DS
) const
{
    dimensionedScalar g
    (
        "g",
        dimensionSet(0, 1, -2, 0, 0, 0, 0),
        scalar (9.81)
    );
    dimensionedScalar zero
    ("zero", dimensionSet(0, -1, 0, 0, 0), scalar(0.0));

    dimensionedScalar smally
    ("smally", dimensionSet(0, 1, 0, 0, 0), scalar(1e-6));

volScalarField Eo = mag(g)*(phaseb_.rho()-phasea_.rho())
                  * sqr(DS)/phaseb_.sig();

volScalarField CwlEo = scalar(0.47)*neg(Eo-scalar(1))
                + exp(scalar(-0.933)*Eo
                + scalar(0.179))*pos(Eo-scalar(1))*neg(Eo-scalar(5))
                + (scalar(0.00599)*Eo
                - scalar(0.0187))*pos(Eo-scalar(5))*neg(Eo-scalar(33))
                + scalar(0.179)*pos(Eo-scalar(33));
volScalarField yw = y_.y();

return CwlEo*max(zero,scalar(1)/Cwd_*(scalar(1)-yw/Cwc_/DS)/
       (yw*pow(yw/Cwc_/DS,p_-scalar(1))+smally));
}
```

The implemented **Hosokawa wall lubrication model** reads:

```
#include "wallLubricationHosokawa.H"
#include "addToRunTimeSelectionTable.H"

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

namespace Foam
{
    defineTypeNameAndDebug(wallLubricationHosokawa, 0);
```

```
    addToRunTimeSelectionTable
    (
        wallLubricationModel,
        wallLubricationHosokawa,
        dictionary
    );
}
// * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * //

Foam::wallLubricationHosokawa::wallLubricationHosokawa
(
    const dictionary& interfaceDict,
    const phaseModel& phasea,
    const phaseModel& phaseb
)
:
    wallLubricationModel(interfaceDict, phasea, phaseb),
    Dpipe_(interfaceDict.lookup("Dpipe"))
{}
// * * * * * * * * * * * * * * * Destructor  * * * * * * * * * * * //

Foam::wallLubricationHosokawa::~wallLubricationHosokawa()
{}
// * * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * //

Foam::tmp<Foam::volScalarField> Foam::wallLubricationHosokawa::AWL
(
    const volScalarField& DS
) const
{
    dimensionedScalar g
    (
        "g",
        dimensionSet(0, 1, -2, 0, 0, 0, 0),
        scalar (9.81)
    );
volScalarField Eo = mag(g)*(phaseb_.rho()-phasea_.rho())
                * sqr(DS)/phaseb_.sig();
volScalarField CwlEo = 0.0217*Eo;

volScalarField yw = y_.y();

return 0.5*CwlEo*DS*(1.0/sqr(yw)-1.0/sqr(Dpipe_-yw));
}
```

APPENDIX L

THE IMPLEMENTED TURBULENCE DISPERSION FORCE MODELS

The implemented **Lopez de Bertodano Turbulence Dispersion force model** reads:

```
#include "Bertodano1992RPI.H"
#include "addToRunTimeSelectionTable.H"

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

namespace Foam
{
    defineTypeNameAndDebug(Bertodano1992RPI, 0);

    addToRunTimeSelectionTable
    (
        turbulentDispersionModel,
        Bertodano1992RPI,
        dictionary
    );
}
// * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * //

Foam::Bertodano1992RPI::Bertodano1992RPI
(
    const dictionary& interfaceDict,
    const phaseModel& phasea,
    const phaseModel& phaseb,
    const volScalarField& alpha,
    const volScalarField& k
)
:
  turbulentDispersionModel(interfaceDict, phasea, phaseb, alpha, k),
  Ctd_(readScalar(interfaceDict_.lookup("Ctd")))
```

```
// * * * * * * * * * * * * * * * Destructor  * * * * * * * * * * * //

Foam::Bertodano1992RPI::~Bertodano1992RPI()
{}
// * * * * * * * * * * * * * * * Member Functions  * * * * * * * * * * //

Foam::tmp<Foam::volScalarField> Foam::Bertodano1992RPI::ATD
(
    const volScalarField& Ur,
    const volScalarField& nutb,
    const volScalarField& DS,
    const volScalarField& Ka
) const
{
    return  Ctd_*phaseb_.rho()*k_;
}
```

The implemented **Gosman Turbulence Dispersion force model** reads:

```
#include "Gosman.H"
#include "addToRunTimeSelectionTable.H"

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

namespace Foam
{
    defineTypeNameAndDebug(Gosman, 0);

    addToRunTimeSelectionTable
    (
        turbulentDispersionModel,
        Gosman,
        dictionary
    );
}
// * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * //

Foam::Gosman::Gosman
(
    const dictionary& interfaceDict,
    const phaseModel& phasea,
    const phaseModel& phaseb,
    const volScalarField& alpha,
    const volScalarField& k
)
:
  turbulentDispersionModel(interfaceDict, phasea, phaseb, alpha, k)
{}
```

```
// * * * * * * * * * * * * * * * Destructor * * * * * * * * * * * * * //

Foam::Gosman::~Gosman()
{}
// * * * * * * * * * * * * * * * Member Functions * * * * * * * * * * * //

Foam::tmp<Foam::volScalarField> Foam::Gosman::ATD
(
    const volScalarField& Ur,
    const volScalarField& nutb,
    const volScalarField& DS,
    const volScalarField& Ka
) const
{
    scalar sigmat = 0.9;
    volScalarField Re = max(Ur*DS/phaseb_.nu(), scalar(1.0e-3));

volScalarField Cds = neg(Re-1000)*24.0
                * (scalar(1)+0.15*pow(Re,0.687))/Re
                + pos(Re-1000)*0.44;

    return  0.75*Cds/DS*phaseb_.rho()*nutb/sigmat*Ur;
}
```

The implemented **Burns Turbulence Dispersion force model** reads:

```
#include "Burns.H"
#include "addToRunTimeSelectionTable.H"

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

namespace Foam
{
    defineTypeNameAndDebug(Burns, 0);

    addToRunTimeSelectionTable
    (
        turbulentDispersionModel,
        Burns,
        dictionary
    );
}
// * * * * * * * * * * * * * * * Constructors * * * * * * * * * * * * //

Foam::Burns::Burns
(
    const dictionary& interfaceDict,
    const phaseModel& phasea,
```

```
    const phaseModel& phaseb,
    const volScalarField& alpha,
    const volScalarField& k
)
:
  turbulentDispersionModel(interfaceDict, phasea, phaseb, alpha, k)
{}
// * * * * * * * * * * * * * * * Destructor * * * * * * * * * * * * * //

Foam::Burns::~Burns()
{}
// * * * * * * * * * * * * * * * Member Functions * * * * * * * * * * * //

Foam::tmp<Foam::volScalarField> Foam::Burns::ATD
(
    const volScalarField& Ur,
    const volScalarField& nutb,
    const volScalarField& DS,
    const volScalarField& Ka
) const
{
    scalar sigmat = 0.9;
    return  Ka*nutb/sigmat/(1-alpha_);
}
```

APPENDIX M

THE HIBIKI-ISHII BREAKUP AND COALESCENCE SOURCE TERM

The implemented **Hibiki-Ishii breakup source term** reads:

```
#include "HibikiIshii.H"
#include "addToRunTimeSelectionTable.H"

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

namespace Foam
{
    defineTypeNameAndDebug(HibikiIshii, 0);

    addToRunTimeSelectionTable
    (
        breakupModel,
        HibikiIshii,
        dictionary
    );
}
// * * * * * * * * * * * * * * * Constructors * * * * * * * * * * * * //

Foam::HibikiIshii::HibikiIshii
(
    const dictionary& interfaceDict,
    const phaseModel& phasea,
    const phaseModel& phaseb,
    const volScalarField& alpha,
    const volScalarField& epsilon
)
:
breakupModel(interfaceDict, phasea, phaseb, alpha, epsilon),
GammaB_( readScalar(interfaceDict_.lookup("GammaB")) ),
Kb_( readScalar(interfaceDict_.lookup("Kb")) ),
```

```
varyBreakCoal_(interfaceDict_.lookupOrDefault<Switch>
            ("varyBreakCoal",true))

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

Foam::HibikiIshii::~HibikiIshii()
{}
// * * * * * * * * * * * * * * * Member Functions * * * * * * * * * * * //

Foam::tmp<Foam::volScalarField> Foam::HibikiIshii::B
(
    const volScalarField& DS,
    const volScalarField& Ur
) const
{
//------- gravity vector --------
    dimensionedScalar g
    (
        "g",
        dimensionSet(0, 1, -2, 0, 0, 0, 0),
        scalar (9.81)
    );
//--------------------
    volScalarField GammaB2 = GammaB_*phaseb_.rho()/phaseb_.rho();

    if (varyBreakCoal_)
    {
volScalarField LapLen = pow(phaseb_.sig()/g
                    / (phaseb_.rho()-phasea_.rho()),0.5);
volScalarField epsbreak = pow(LapLen,4) * epsilon_
                      / pow(phaseb_.nu(),3);
volScalarField GammaB2 = 5.02e-10 * alpha_ * epsbreak;
    }
volScalarField sourceTIa = GammaB2*alpha_*(1.0-alpha_)
                      * pow(epsilon_,1.0/3.0);
volScalarField sourceTIb = pow(DS,5.0/3.0)
                      * max(alphaMax_-alpha_,1.0e-6);
volScalarField sourceTIc = exp(-Kb_*phaseb_.sig()/phaseb_.rho()
                      / pow(DS,5.0/3.0)/pow(epsilon_,2.0/3.0));

volScalarField sourceTI = sourceTIa/sourceTIb*sourceTIc;

return sourceTI*scalar(1.0);
}
```

The implemented **Hibiki-Ishii coalescence source term** reads:

```
#include "HibikiIshiiC.H"
#include "addToRunTimeSelectionTable.H"
```

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

namespace Foam
{
    defineTypeNameAndDebug(HibikiIshiiC, 0);

    addToRunTimeSelectionTable
    (
        coalescenceModel,
        HibikiIshiiC,
        dictionary
    );
}
// * * * * * * * * * * * * * * * Constructors * * * * * * * * * * * //

Foam::HibikiIshiiC::HibikiIshiiC
(
    const dictionary& interfaceDict,
    const phaseModel& phasea,
    const phaseModel& phaseb,
    const volScalarField& alpha,
    const volScalarField& epsilon
)
:
coalescenceModel(interfaceDict, phasea, phaseb, alpha, epsilon),
GammaC_( readScalar(interfaceDict_.lookup("GammaC")) ),
Kc_( readScalar(interfaceDict_.lookup("Kc")) ),
varyBreakCoal_(interfaceDict_.lookupOrDefault<Switch>
                ("varyBreakCoal",true))
// * * * * * * * * * * * * * * * Destructor * * * * * * * * * * * * //

Foam::HibikiIshiiC::~HibikiIshiiC()
{}
// * * * * * * * * * * * * * Member Functions * * * * * * * * * * * //

Foam::tmp<Foam::volScalarField> Foam::HibikiIshiiC::C
(
    const volScalarField& DS,
    const volScalarField& Ur
) const
{
//------- gravity vector ---------
    dimensionedScalar g
    (
        "g",
        dimensionSet(0, 1, -2, 0, 0, 0, 0),
        scalar (9.81)
    );
//-------------------------------
```

```
    volScalarField GammaC2 = GammaC_*phaseb_.rho()/phaseb_.rho();

    if (varyBreakCoal_)
    {
volScalarField LapLen = pow(phaseb_.sig()/g
                    / (phaseb_.rho()-phasea_.rho()),0.5);
volScalarField epsbreak = pow(LapLen,4)
                      * epsilon_ / pow(phaseb_.nu(),3);
volScalarField GammaC2 = 1.82e-8 * alpha_ * epsbreak;
    }
volScalarField sourceRCa=-GammaC2*sqr(alpha_)*pow(epsilon_,1.0/3.0);
volScalarField sourceRCb=pow(DS,5.0/3.0)*(alphaMax_-alpha_);
volScalarField sourceRCc=exp(-Kc_*sqrt(phaseb_.rho()/phaseb_.sig())
                       * pow(DS,5.0/6.0)*pow(epsilon_,1.0/3.0));

volScalarField sourceRC = sourceRCa/sourceRCb*sourceRCc;

return sourceRC*scalar(1.0);
}
```

APPENDIX N

NUMERICAL SCHEMES

```
    ddtSchemes
{
    default Euler;
}
gradSchemes
{
    default         Gauss linear;
}
divSchemes
{
    default         none;
// UEqns
    div(phia,Ua)    Gauss upwind;
    div(phib,Ub)    Gauss upwind;

    div(Rca)        Gauss linear;
    div(Rcb)        Gauss linear;
    div(phia)       Gauss linear;
    div(phiRa)      Gauss linear;
    div(phib)       Gauss linear;
    div(phiRb)      Gauss linear;
// HEqn
    div(phib,Hb)    Gauss upwind;
    div(phia,Ha)    Gauss upwind;
    div(phiQb)      Gauss linear;
    div(phiQa)      Gauss linear;
// k-eps model
    div(phib,k)     Gauss upwind;
    div(phib,epsilon) Gauss upwind;

    div(phiVb)      Gauss linear;
    div(phiEb)      Gauss linear;
// alphaEqn
```

```
    div(phi,alpha)  Gauss upwind;
    div(phir,alpha) Gauss upwind;

    div(phi,beta)   Gauss upwind;
    div(phir,beta)  Gauss upwind;

    div(phic)       Gauss linear;
// IACEqn
    div(phia,IAC)    Gauss upwind;

// pEqn
    div(phi)        Gauss linear;
// rhoEqn
    div(phia,rhoa)  Gauss upwind;
    div(phib,rhob)  Gauss upwind;
// DpDt
    div(phib,p)     Gauss upwind;
    div(phia,p)     Gauss upwind;
}
laplacianSchemes
{
    default         none;
// UEqn
    laplacian(nuEffa,Ua)        Gauss linear corrected;
    laplacian(nuEffb,Ub)        Gauss linear corrected;

// HEqn
    laplacian(kappaEffb,Hb)     Gauss linear corrected;
    laplacian(kappaEffa,Ha)     Gauss linear corrected;

// pEqn
    laplacian((rho*(1|A(U))),p) Gauss linear corrected;

// k-eps
    laplacian(DkEff,k) Gauss linear corrected;
    laplacian(DepsilonEff,epsilon) Gauss linear corrected;
}
interpolationSchemes
{
    default         linear;
}
snGradSchemes
{
    default         corrected;
}
```