Cite as: Tsiapkinis, I.: Free surface shape calculation using the interfaceTrackingFvMesh class and considering external pressure and fixed contact angles. In Proceedings of CFD with OpenSource Software, 2022, Edited by Nilsson. H., http://dx.doi.org/10.17196/OS_CFD#YEAR_2022

CFD with OpenSource software

A course at Chalmers University of Technology
Taught by Håkan Nilsson

Free surface shape calculation using the interfaceTrackingFvMesh class and considering external pressure and fixed contact angles

Developed for OpenFOAM-v2112
Requires: —

Author:
Iason Tsiapkinis
Leibniz-Institute for Crystal Growth
iason.tsiapkinis@ikz-berlin.de

Peer reviewed by:
Arvid Åkerblom
Mohammad Hossein Arabnejad
Kaspars Dadzis

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

January 24, 2023

Learning outcomes

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

The reader will learn:

How to use it:

- How to use the `interfaceTrackingFvMesh` class and the accompanying `freeSurface` boundary conditions.
- How to set up a test case including free surface mesh deformation using `interfaceTrackingFvMesh`.
- How to set up contact conditions at the liquid-gas-solid interface.

The theory of it:

- The theory behind the interface tracking method for finite volumes.
- The theory behind the methods of the Finite-Area-Method utilized in this class.
- The numerical technique for a contact angle constrained by an adapted pressure boundary condition.

How it is implemented:

- How the interface tracking class `interfaceTrackingFvMesh` is designed and implemented in OpenFOAM.
- How the `freeSurfaceVelocity` and `freeSurfacePressure` boundary conditions are implemented and applied.
- The differences between the `interTrackFoam` solver in foam-extend and the `interfaceTrackingFvMesh` class.

How to modify it:

- How to modify the class to calculate, write and use additional variables
- How to write out additional surface data.
- How to modify the implementation of the contact angle condition to make it more strict
- How to implement the contact angle condition inside the pressure boundary condition

Prerequisites

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

- How to run standard document tutorials like the `damBreak` tutorial with OpenFOAM
- A basic understanding of the Finite-Volume-Method
- An understanding of the PIMPLE method

Contents

Nomenclature

Acronyms

CFD	Computational Fluid Dynamics
ETP	External Triple Point
FAM	Finite Area Method
FAM	Finite Area
FVM	Finite Volume Method
FZ	Floating Zone
ITP	Internal Triple Point
ITT	Interface Tracking Technique
PIMPLE	Pressure Implicit Method for Pressure Linked Equations
PISO	Pressure Implicit with Splitting of Operator
VOF	Volume Of Fluids

English symbols

$\boldsymbol{d}$	Motion direction unit vector	
$\boldsymbol{e}_r$	Rotational axis unit vector	
$\boldsymbol{g}$	Gravitational acceleration	$\mathrm{m/s^2}$
$\boldsymbol{m}$	Binormal unit vector	
$\boldsymbol{n}$	Normal unit vector	
$\boldsymbol{t}$	Tangential unit vector	
$\boldsymbol{u}$	Fluid velocity	$\mathrm{m/s}$
$\boldsymbol{v}$	Mesh velocity	$\mathrm{m/s}$
$\boldsymbol{x}$	Position vector	m
$\boldsymbol{x}_f$	Position vector of face f	m
$\dot{V}$	Volume flux	$\mathrm{m^3/s}$
L	Edge length	m
m	Mass flux	$\mathrm{kg/s}$
p	Fluid pressure	$\mathrm{kg/s^2m}$
p_{EM}	Electromagnetic pressure	$\mathrm{kg/s^2m}$
R'	Curvature radius	m
S	Face area	$\mathrm{m^2}$
t	Time	s

Greek symbols

α	Current face angle	$^\circ$
$\boldsymbol{\delta}$	Vector pointing from edge to cell centre	m
$\boldsymbol{\tau}$	Stress tensor	$\mathrm{kg/s^2m}$
η	Fluid dynamic viscosity	$\mathrm{kg \cdot s/m}$
Γ_f	Face surface region	
κ	Surface mean curvature	$\mathrm{m^2}$
Λ_f	Edge region	

ν	Fluid kinematic viscosity	m^2/s
$\partial\Gamma_f$	Face boundary	
∂'_f	Face displacement	m
ψ	Flow quantity	$-$
ρ	Fluid density	kg/m^3
σ	Contact angle	$^\circ$
θ	Surface tension	kg/s^2

Superscripts

eC	edge centre
PMP	Patch Mirror Point
rot	Rotated

Subscripts

c	cell
e	edge
f	face
p	pressure
+	Outer fluid
-	Inner fluid
a	ambient
BP	Boundary Point
N	Neighbor
O	Owner
r	rotating
s	Surface

Chapter 1

Introduction

Materials for various technological applications, e.g., power devices, detector applications, and laser materials, are produced in high-temperature crystal growth furnaces involving multiple materials in different phases. For example, in the Floating Zone (FZ) crystal growth process, a silicon feed rod is melted by high-frequency induction heating and solidifies as a single crystal. Figure 1.1 (left) shows a sketch of this process, where the liquid melt is held between the feed rod and the growing crystal and has only contact with the surrounding gas [1]. The free surface between the melt and the gas plays a decisive role in this technique [2]. As shown in Figure 1.1 (right), the free surface is deformed by an electromagnetic pressure p_{EM}. Furthermore, the electrically conducting liquid melt is influenced by Lorenz forces, adding momentum to the melt flow. The contact angles θ between the solid and the liquid are fixed, and the internal triple point (ITP) can move depending on the location of the solid-liquid phase boundary. This process involves high-density ratios between the solid and the liquid ($\sim 10^4$) and high surface tension (0.88 N/m). Existing publications use the `interFoam` solver to predict the 3D free surface shape in FZ silicon growth in Han et al. 2020 [3], while in a PhD thesis by Beckstein 2018 [4] the `interTrackFoam` solver is modified and applied to simulate the surface shape in a similar process.

This report aims to investigate the interface tracking method as it is implemented in OpenFOAM and its possible application to the above-described problem. Additional modifications to the library are done in order to satisfy the boundary conditions in Figure 1.1 (right). In Chapter 2 the theory behind the interface tracking method is described. Chapter 4 shows a tutorial using the `interfaceTrackingFvMesh` library together with `pimpleFoam`. The files, classes, and functions contained in this library are explained in Chapter 3. In Chapter 5 it is shown how to implement modifications to the library, like calculating a surface field or writing out the surface data. Additionally, the contact angle boundary condition is restructured for more functionality, and a different approach to this boundary condition is implemented via a pressure boundary condition.

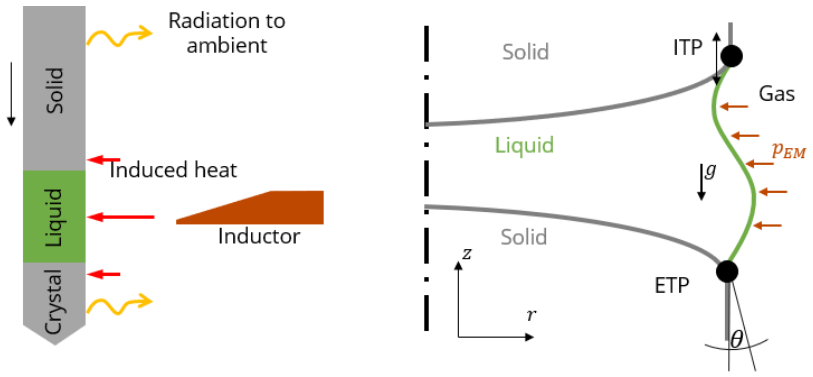


Figure 1.1: Left: Sketch of FZ process. Right: Free surface deformation due to electromagnetic forces on the liquid with a fixed contact angle and a moving internal triple point (ITP).

Chapter 2

Interface Tracking Method

Interface tracking methods use a boundary-fitted moving mesh, resulting in a sharp interface, as opposed to other multiphase methods, like the Volume-of-Fluids (VOF) method, where the interface is diffusive. Interface tracking methods can therefore provide the most precise results as the boundary between phases is represented by the computational boundary.

The mathematical model and numerical methods for a fluid flow with a sharp free surface are presented, as it is implemented in the solver `interTrackFoam` in foam-extend and the library `interfaceTrackingFvMesh` in OpenFOAM. This chapter follows the description of the interface tracking method by Tukovic and Jasak 2012 [5] and by Muzaferija and Perić 1997 [6] primarily.

2.1 Mathematical model

This section describes the mathematical model for a two-phase fluid flow with a sharp interface. With this model, the governing equations of a flow can be solved separately by the Finite-Volume-Method on each region, defined by a combined mesh. Coupling of the fluid velocities and pressure is accomplished by applying proper boundary conditions at the interface. In the following sections, the term *free surface* describes an interface between two fluids, where only one fluid is simulated. This assumption is valid for a large difference in dynamic viscosity as is shown in Section 2.1.3.

2.1.1 Navier-Stokes equations

The governing equations for each phase are given by the incompressible and isothermal Navier-Stokes equations [7] for a Newtonian fluid as

$$\nabla \cdot \boldsymbol{u} = 0 \tag{2.1}$$

$$\rho \left(\frac{\partial \boldsymbol{u}}{\partial t} + \boldsymbol{u} \cdot \nabla \boldsymbol{u} \right) = \nabla \cdot \boldsymbol{\tau}, \tag{2.2}$$

where ρ is the fluid density and $\boldsymbol{u}$ is the fluid velocity. The stress tensor is defined in terms of the local fluid pressure and velocity fields as

$$\boldsymbol{\tau} = \eta \left(\nabla \boldsymbol{u} + (\nabla \boldsymbol{u})^T \right) - p\boldsymbol{I}, \tag{2.3}$$

where η is the dynamic viscosity of the fluid and p is the dynamic pressure obtained by subtracting the hydrostatic pressure $\rho \boldsymbol{g} \cdot \boldsymbol{x}$ from the absolute pressure. Here, $\boldsymbol{g}$ is the gravitational acceleration vector, and $\boldsymbol{x}$ is the position vector.

2.1.2 Kinematic and dynamic conditions on the free surface

The conditions on a free surface for immiscible fluids can be split into *kinematic* and *dynamic conditions* [8]. To derive these conditions the jump $[\![\cdot]\!]$ of a quantity ψ is defined as

$$[\![\psi]\!] = \psi^+ - \psi^-. \tag{2.4}$$

The suffixes $^+$ and $^-$ describe one of the fluids on each side of the interface respectively. The surface normal vector $\boldsymbol{n}$ however is always pointing from - to +, and being therefore associated with the fluid "-".

$$\boldsymbol{n} = \boldsymbol{n}^- = -\boldsymbol{n}^+ \tag{2.5}$$

The *kinematic condition* requires the free surface to not "break" and relates the fluid velocities on the two sides of the interface. Since, in the case of two immiscible fluids, there is no mass flux through the phase boundaries, the velocity must be continuous across the interface and therefore

$$[\![\boldsymbol{u}]\!] = 0. \tag{2.6}$$

The *dynamic condition* is derived from the momentum conservation law on the surface and states that forces acting on the fluid from the surface are in equilibrium. The general form at the interface, which gives the fundamental relationship between the jump of the stress tensor $\boldsymbol{\tau}$ across an interface and the surface tension force, is given by

$$[\![\boldsymbol{n}\boldsymbol{\tau}]\!] = \bar{\nabla}\sigma - \sigma\kappa\boldsymbol{n}, \tag{2.7}$$

where σ is the surface tension and $\bar{\nabla}$ is surface gradient operator (or tangential gradient operator) given by

$$\bar{\nabla} = (\boldsymbol{I} - \boldsymbol{n}\boldsymbol{n}) \cdot \nabla = \nabla - \boldsymbol{n}\frac{\partial}{\partial n}, \tag{2.8}$$

and κ is twice the mean curvature of the interface given by

$$\kappa = -\bar{\nabla} \cdot \boldsymbol{n}. \tag{2.9}$$

The gradient of the surface tension coefficient $\bar{\nabla}\sigma$ captures changes in surface tension due to a temperature gradient or non-uniform distribution of surfactants at the interface.

The *normal force balance* of Eq. 2.7, together with the definition of the stress tensor (Eq. 2.3) yields a jump condition for the pressure given by

$$[\![p]\!] = -2[\![\eta]\!]\left(\bar{\nabla} \cdot \boldsymbol{u}\right) + \sigma\kappa. \tag{2.10}$$

This equation shows that there is a jump in dynamic pressure due to the surface divergence of the velocity $\boldsymbol{u}$ and, together with the surface tension σ, of the local curvature κ. The surface divergence $\bar{\nabla}\cdot\boldsymbol{u}$ is not zero, as supposed to the volume divergence in Eq. 2.1, because it accounts for the pressure jump and surface tension forces that create a net flow at the surface. Additionally, the flow of the fluid near the surface can cause the surface to move horizontally, resulting in non-zero divergence. From the *tangential force balance* of Eq. 2.7, a condition for the normal velocity gradient can be derived as

$$[\![\boldsymbol{n} \cdot \eta\nabla u]\!] = -[\![\eta]\!](\bar{\nabla} \cdot \boldsymbol{u})\boldsymbol{n} - [\![\eta]\!]\bar{\nabla}\boldsymbol{u} \cdot \boldsymbol{n} - \bar{\nabla}\sigma. \tag{2.11}$$

A jump in the normal gradient of the velocity $\boldsymbol{u}$ is determined by the surface gradient of its normal component $\left(\bar{\nabla}\boldsymbol{u} \cdot \boldsymbol{n}\right)$, its surface divergence $\left(\bar{\nabla} \cdot \boldsymbol{u}\right)$, and the change of surface tension along the curvature $\left(\bar{\nabla}\sigma\right)$. The jump in viscosity η has to be accounted for in both conditions.

2.1.3 Boundary Conditions for the free surface of a fluid

In the implementation of `interfaceTrackingFvMesh`, the external Fluid "+" is not included in the simulation. By assuming a large difference in dynamic viscosity $\eta^+ \ll \eta^-$ and introducing an external dynamic pressure $p_+ = p_\mathrm{a}^\mathrm{dyn} = p_\mathrm{a} - \rho \boldsymbol{g} \cdot \boldsymbol{x}_f$, the jump conditions in Eqs. 2.10 and 2.11 can be written as boundary conditions for the fluid "−":

$$\boldsymbol{n} \cdot \eta \nabla u = -\left(\bar{\nabla} \cdot \boldsymbol{u}\right) \boldsymbol{n} - \bar{\nabla} \boldsymbol{u} \cdot \boldsymbol{n} + \frac{1}{\eta} \bar{\nabla} \sigma \tag{2.12}$$

$$p = p_\mathrm{a} - \rho \boldsymbol{g} \cdot \boldsymbol{x}_f - 2\eta \left(\bar{\nabla} \cdot \boldsymbol{u}\right) - \sigma \kappa \tag{2.13}$$

It is interesting to note that for a stationary free surface and by neglecting the surface tension, Eq. 2.13 and 2.12 are reduced to a slip condition. Furthermore, by neglecting most of the flow-related terms (i.e. the ones with $\boldsymbol{u}$), Eq. 2.13 is reduced to the Young-Laplace equation describing the pressure difference Δp across the interface between two static fluids as

$$\Delta p = -\sigma \kappa. \tag{2.14}$$

2.2 Numerical methods

In this section, the numerical methods for the discretization of the governing equations described in Section 2 are presented. The final solution procedure is described in 3.1. The discretization of the governing Eqs. 2.1 and 2.2 is done according to the finite-volume discretization methods [8]. The equations are solved using the PISO algorithm after Issa 1986 [9] in the Arbitrary-Lagrangian-Euler formulation [5]. The free surface and the two boundary conditions (equations 2.13 and 2.12) are discretized with the Finite-Area-Method (FAM). Section 2.2.1 gives a brief introduction to the FAM and shows the discretization of the surface gradient operator.

Section 2.2.2 describes the interface tracking procedure of the free surface and Section 2.3 introduces a new method for modelling a constant contact angle.

2.2.1 Finite-Area-Method: calculation of surface derivatives and surface tension

The Eq.s 2.12 and 2.13 contain surface derivatives of the fluid velocity $\boldsymbol{u}$. The surface divergence $\bar{\nabla} \cdot \boldsymbol{u}$, the surface gradient $\bar{\nabla} \boldsymbol{u}$ as well as the curvature $\kappa = \bar{\nabla} \cdot \boldsymbol{n}$ have to be calculated. For simplicity, only constant surface tension is considered in Eq. 2.12 ($\bar{\nabla} \boldsymbol{\sigma} = 0$). The surface derivatives are discretized based on Gauss' integral theorem on the surface Γ_F and are calculated explicitly using the Finite-Area-Method (FAM). In principle, the FAM is nothing more than the FVM: instead of control *volumes* and *surfaces* between the volumes, there are control *surfaces* separated by *edges*.

The Gauss' integral theorem for a general quantity ψ defined on the region Γ_f of the surface f bounded by the closed line $\partial\Gamma$ is given by [6]

$$\int_{\Gamma_f} \bar{\nabla} \circ \psi \, \mathrm{d}S = \int_{\partial\Gamma_f} \boldsymbol{m} \circ \psi \, \mathrm{d}L - \int_{\Gamma_f} \kappa \boldsymbol{n} \circ \psi \, \mathrm{d}S, \tag{2.15}$$

where $\circ$ can be any product (inner product $\cdot$, cross product $\times$ or outer product $\otimes$), and $\boldsymbol{m}$ is the unit vector pointing from the edge center e to the face centre (see Figure 2.1). Assuming a locally linearly varying function ψ the value of ψ_e on an edge centre $\boldsymbol{x}_e$ can be approximated, similarly to the value in cell centres or face centres in the FVM with an integral across the edge Λ_e, as

$$\psi_e = \psi\left(t, x_e\right) \approx \frac{1}{L_e} \int_{\Lambda_e} \psi(t, x) \mathrm{d}L. \tag{2.16}$$

Here, L_e is the length of the edge e. Furthermore, analogous to integrals over a volume in the FVM, integrals over a face f can be approximated by

$$\int_{\Gamma_f} \psi \mathrm{d}S \approx \psi_f S_f. \tag{2.17}$$

Line integrals over the whole boundary $\partial\Gamma_f$ of the face region f can first be interpreted as the sum of the piecewise connected boundary edges Λ_e. The edge integrals can then be approximated with Eq. 2.16, which yields

$$\int_{\partial\Gamma_f} \mathrm{d}\boldsymbol{L} \circ \psi = \sum_{e}^{\Lambda_e \in \partial\Gamma_f} \int_{\Lambda_e}^{\Lambda_e \in \partial\Gamma_f} \mathrm{d}\boldsymbol{L} \circ \psi \approx \sum_{e} \boldsymbol{L}_e \circ \psi_e, \tag{2.18}$$

where ψ_e is the field value on the edge and $\boldsymbol{L}_e = \boldsymbol{m}_e L_e$ is the binormal vector weighted with the edge length L_e in the centroid $\boldsymbol{x}_e$ of the edge e. The normalised binormal vector $\boldsymbol{m}_e = \boldsymbol{t}_e \times \boldsymbol{n}_e$ with $|\boldsymbol{m}_e| = 1$ is calculated using the tangetial vector $\boldsymbol{t}_e$ pointing along the edge and the mean normal vector $\boldsymbol{n}_e$ in the edge centre. The latter is calculated as

$$\boldsymbol{n}_e = \frac{\boldsymbol{n}_{e,0} + \boldsymbol{n}_{e,1}}{|\boldsymbol{n}_{e,0} + \boldsymbol{n}_{e,1}|}, \tag{2.19}$$

where $\boldsymbol{n}_{e,i}$ are obtained by averaging the normal vectors $\boldsymbol{n}_f$ of the neighbouring faces on the start- and endpoints $\boldsymbol{x}_{e,i}$ of edge e. The vectors needed for this calculation are shown in Figure 2.1. For simplicity, only a curvature in one direction is considered.

With Eq.s 2.16 and 2.17, and together with Gauss' integral theorem in Eq. 2.15 any gradient, divergence, or curl of a quantity on the surface f needed in Eq.s 2.13 and 2.12 can be discretized with

$$\left\{\bar{\nabla} \circ \psi\right\}_f \approx \frac{1}{S_f} \int_{\Gamma_f} \bar{\nabla} \circ \psi \ \mathrm{d}S \approx \frac{1}{S_f} \sum_{e} \boldsymbol{m}_e \circ \psi_e L_e - \kappa_f \boldsymbol{n}_f \circ \psi_f \tag{2.20}$$

The main difference to the FVM is the additional term that considers the surface curvature, which can also be calculated using Eq. 2.20 with $\psi \hat{=} 1$ and $\circ$ being the outer product. Multiplying the inner product of $\boldsymbol{n}_f$ to both sides of the equation yields

$$\kappa_f = -\left\{\bar{\nabla} \cdot \boldsymbol{n}\right\}_f = \sum_{e} \boldsymbol{n}_f \cdot \boldsymbol{m}_e L_e. \tag{2.21}$$

The surface divergence $\bar{\nabla} \cdot \boldsymbol{u}$ and the surface gradient $\bar{\nabla}\boldsymbol{u}$ can therefore be discretised using the geometric mesh parameters $\boldsymbol{m}_e$, $\boldsymbol{n}_e$ and $\boldsymbol{L}_e$. The edge-center velocity $\boldsymbol{u}_e$ can be linearly interpolated from the face values $\boldsymbol{u}_f$, which in turn are extrapolated with common FVM methods from the cell-center values.

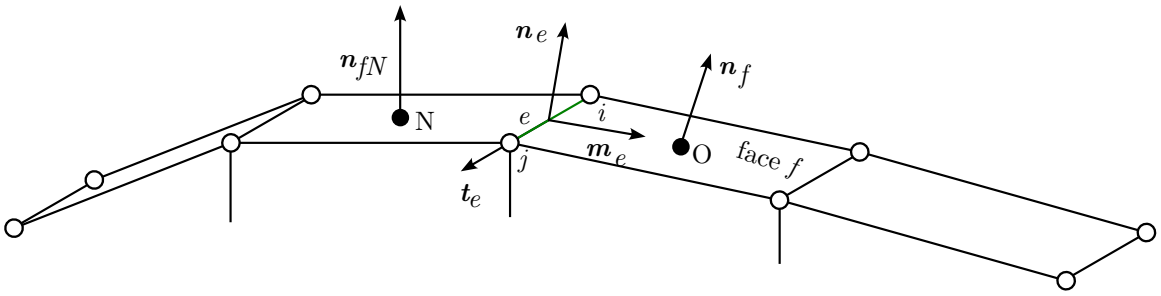


Figure 2.1: Surface area mesh with a one-dimensional curvature. Normal vector $\boldsymbol{n}_e$ is calculated using the normal vectors of neighboring faces.

2.2.2 Interface tracking procedure

The interface tracking procedure for the sharp interface in a two-phase fluid flow is implemented in OpenFOAM after the work of Tukovic and Jasak 2012 [5] and is an extension of the work of Muzaferija and Perić 1997 [6]. The basis of the algorithm is the kinematic condition in Eq. 2.6. A schematic of the procedure is schown in Figure 2.2.

In order to realize the interface tracking technique, when there is a free surface that moves with a fluid velocity of $\boldsymbol{u}$, the corresponding mesh has to move with an identical velocity $\boldsymbol{v}$, meaning

$$\boldsymbol{v}_f = \boldsymbol{u}_f. \tag{2.22}$$

By multiplying both sides of the equation with the face area S_f, a relation between the volume face flux $\dot{V}$, that is, the rate of change of the cell volume between two-time steps due to the face moving and the mass flux through the face $\dot{m}_f$ for each cell on the free surface is given by

$$\dot{V}_f = \boldsymbol{S}_f \cdot \boldsymbol{v}_f = \boldsymbol{S}_f \cdot \boldsymbol{u}_f = \frac{m_f}{\rho_f}. \tag{2.23}$$

The mass flux m_f through the face f on the boundary is calculated at the end of the PISO algorithm in the corrector step, where the velocity and face flux fields are corrected. In general, these mass fluxes will not be zero due to the prescribed boundary conditions, and the volume flux has to be corrected to satisfy Eq. 2.23. As shown by Tukovic and Jasak 2012 [5], this is accomplished by introducing a volume flux correction as

$$\dot{V}'_f = \frac{m_f}{\rho_f} - \dot{V}_f. \tag{2.24}$$

With this volume flux correction $\dot{V}'_f$, the required absolute change in the cell volume $\delta V'_f$ can be calculated with

$$\delta V'_f = C_{\mathrm{ddt}} \dot{V}'_f \Delta t, \tag{2.25}$$

where C_{ddt} is a constant that depends ont the time differencing scheme. For an Euler differencing scheme this constant is $C_{\mathrm{ddt}} = \frac{2}{3}$. The required displacement of the face $\delta h'_f$ can be calculated using Eq. 2.25, the face area S_f and face normal vector $\boldsymbol{n}_f$ as

$$\delta h'_f = \frac{\delta V'_f}{S_f \boldsymbol{n}_f \cdot \boldsymbol{d}_f}. \tag{2.26}$$

The motion direction vector $\boldsymbol{d}_f$ can be used to restrict the mesh motion direction. According to Muzaferija and Perić 1997 [6], the motion direction vector should be as parallel as possible to the effective force acting on the surface, for example, in the direction of the gravitational vector $\boldsymbol{g}$. If there is no clear preferred direction or in the case of a closed free surface, the mesh motion direction can be equal to the surface normal vector of the previous time step ($\boldsymbol{d}_f = \boldsymbol{n}_f$).

The direct displacement of the mesh points with Eq. 2.26 does not guarantee a smooth surface. The interface point displacement is therefore calculated based on the procedure in Muzaferija and Perić 1997 [6], where instead control points $\boldsymbol{r}_f$ are used that are attached on top of the centroid at $\boldsymbol{x}_f$ of each face (see Figure 2.2). The corrected position of the control points is calculated with Eq. 2.26 according to

$$\boldsymbol{r}_f = \boldsymbol{x}_f + \delta h'_f \boldsymbol{d}_f. \tag{2.27}$$

In other words, first, each surface is shifted from its initial position by a distance $\delta h'_f$ in the direction of $\boldsymbol{d}_f$, so that the volume of the prism formed with S_f equals the change $\delta V'_f$. The grid point adjustment is then calculated based on the position of the control points of neighboring faces using the least squares method [5]. In two dimensions, this is reduced to linear interpolation. Figure 2.2 shows the procedure in two dimensions.

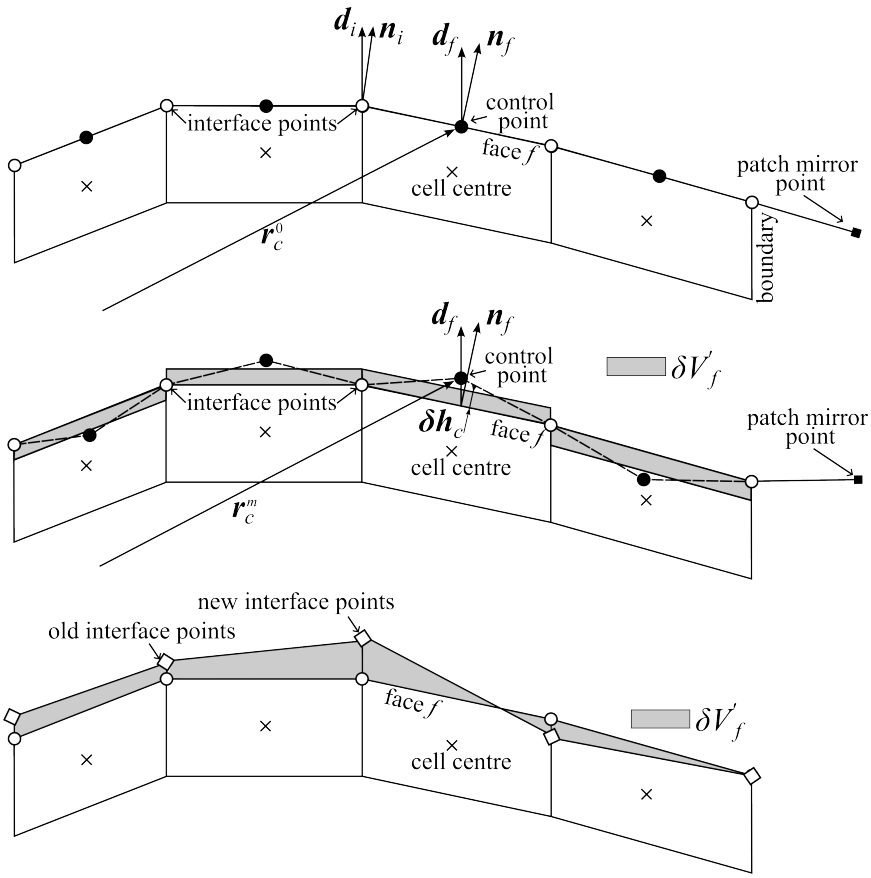


Figure 2.2: The motion direction of the interface points is either in direction of $\boldsymbol{d}_i$ (specified by the user) or $\boldsymbol{n}_i$. The control points start in the face center; mirror points are placed outside of the boundary. Control points are then moved according to Eq. 2.27. The displacement of the interface points is calculated using adjacent control points.

2.3 Contact angle boundary condition with pressure force

In a two-dimensional case, i.e., a one-dimensional curvature, the curvature calculated in Eq. 2.21 is reduced to

$$\kappa_f = \frac{1}{R'_f}, \tag{2.28}$$

where the curvature radius R'_f is the radius of a circle drawn through the face center of interest and its two neighboring face centers. At the boundaries, this circle would go through the boundary face center, its inner neighbor, and the mirror face center outside the region (see Figure 2.3b). It follows that the pressure boundary condition in Eq. 2.13 can be expressed for each face f as

$$p_f = p_\mathrm{a} - \rho \boldsymbol{g} \cdot \boldsymbol{x}_f - 2\eta \left(\bar{\nabla} \cdot \boldsymbol{u}_f\right) - \sigma \frac{1}{R'_f}. \tag{2.29}$$

The value p_a can be seen as the pressure of the surrounding fluid or as an unknown in the pressure jump condition. The pressure p_a might also include an arbitrary pressure level p^g_const from the hydrostatic pressure, which depends on the reference point of the coordinate $\boldsymbol{x}_f$ and has no influence on the flow itself. By introducing the gauge pressure $p_0 = p_a + p^g_\mathrm{const}$, the unknown value of the gauge pressure can be used to introduce an additional geometric boundary condition on the free surface [10].

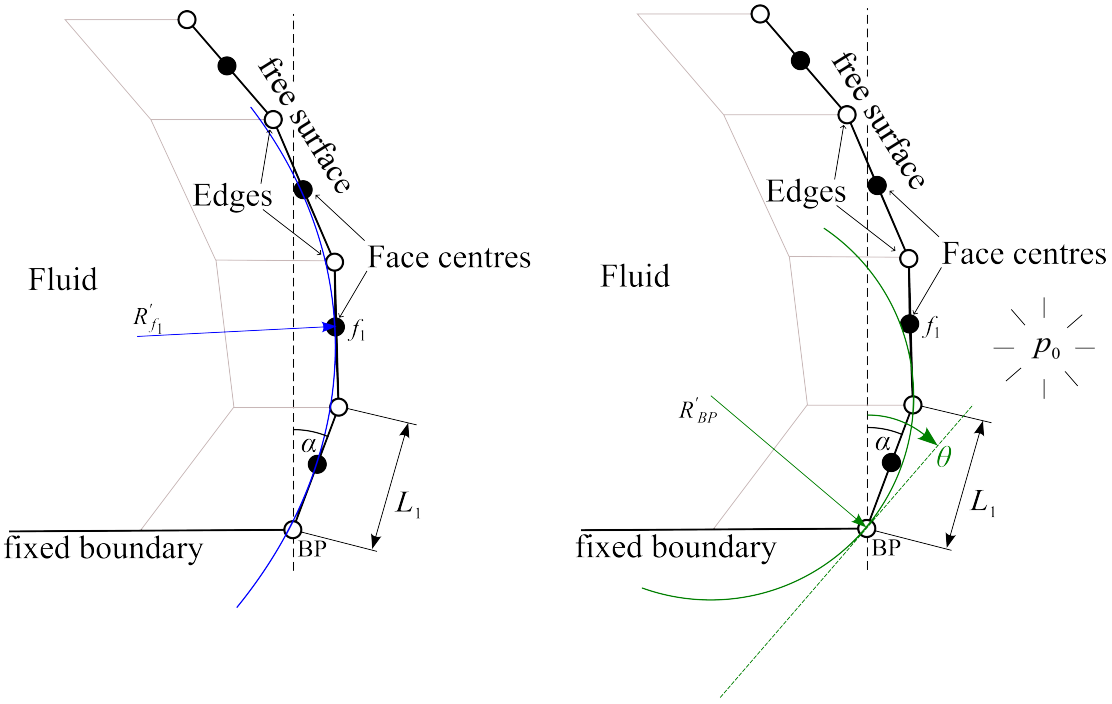
At a liquid-vapor-solid triple point, the contact angle θ is formed between these boundaries, as seen in Figure 2.3c. In the case described in Figure 1.1, this corresponds to the constant growth angle at the fixed external triple point and a constant crystal diameter. From Figure 2.3c it follows, that the curvature at the boundary point BP can be expressed through a circle with a radius, such that its tangent at BP makes the angle θ with the vertical and the circle goes through both vertices of the boundary face. The curvature can be expressed from geometric considerations following Ratnieks 2007 [10] as

$$\frac{1}{R'_\mathrm{BP}} = \frac{2\sin\left(\theta - \alpha_1\right)}{L_1}, \tag{2.30}$$

where α_1 is the current angle of the first face and L_1 is its length in the tangential direction. The total pressure at this face has to satisfy the Eq. 2.29 with the curvature calculated in Eq. 2.30. The pressure jump at this point can be calculated as

$$p_\mathrm{BP} = p_0 - \rho \boldsymbol{g} \cdot \boldsymbol{x}_\mathrm{BP} - 2\eta \left(\bar{\nabla} \cdot \boldsymbol{u}_\mathrm{BP}\right) - \sigma \frac{1}{R'_\mathrm{BP}}, \tag{2.31}$$

which can be interpreted as a pressure force, or gauge pressure acting on the free surface to reach the desired contact angle.



(a) Curvature at the first face center f_1

(b) Curvature at the first face center f_1

(c) curvature at the boundary point BP, defined by contact angle θ

Figure 2.3: Discretized free surface and curvature radii at the first face center and the first boundary point in two dimensions.

Chapter 3

Implementation of interface tracking in OpenFOAM

The theory described in Chapter 2 is common to both the `InterfaceTrackingFvMesh` library in OpenFOAMand also by the `interTrackFoam` solver in foam-extend. This chapter focuses on the implementation in OpenFOAM. First, the solution procedure is presented in a concise manner in Section 3.1 and some of the differences between the OpenFOAM and the foam-extend implementation are discussed in Section 3.2. Next, an overview of the library and its files are given in Section 3.3. The implementation of the boundary conditions is described in Section 3.4. Lastly, in Section 3.5, the `interFaceTrackingFvMesh` class and some relevant functions are described.

3.1 Solution procedure

The mathematical models and numerical methods shown in Chapter 2 are implemented in OpenFOAM using the dynamic motion library `interfaceTrackingFvMesh` that is called in the *outer PIMPLE iteration*. The procedure of one the outer iteration together with interface tracking is as follows:

1. Start dynamic mesh motion
   (a) Update displacement direction $\mathbf{d}_f$, mesh Courant number, transport properties and curvature κ
   (b) Interface Tracking procedure (see Section 2.2.2)
      i. Calculate the necessary cell volume change correction $\delta\dot{V}'_f$ to account for current relative flux through the interface cells (Eq. 2.25)
      ii. Calculate the required displacement $\delta\dot{h}'_f$ (Eq. 2.26, define the `controlPoints` and move them by $\delta\dot{h}'_f$ (Eq. 2.27)
      iii. If a contact angle θ_C is specified: rotate the face-normal vector of the faces adjacent to the edges, where the contact angle is specified
      iv. Calculate `patchMirrorPoints` mirroring `controlPoints` for boundary mesh points with neighboring face normal vectors (see Section 3.5.6.2)
      v. Do a least-squares plane fit (or linear interpolation for a two-dimensional case) for each mesh point using `controlPoints` and `patchMirrorPoints` and calculate the displacement of free surface mesh points (see Figure 2.2)
   (c) Displacement of the inner mesh with mesh motion solver. The displacement of the interface points is used as boundary conditions for the solution of the mesh motion problem
2. Update the pressure (Eq. 2.13) and velocity (Eq. 2.12) boundary conditions with the discretization methods described in 2.2.1

3. Assemble the discretised momentum Eq. 2.2, solve for $\boldsymbol{u}^t$ using the pressure p^{t-1} and the mass fluxes $\dot{m}^{t-1}$ on the new interface shape
4. Do at least 2 PISO iteration loops
5. Calculate new mass fluxes through the faces at the interfaces and restart the outer loop at point 1 if the number of outer correction loops is not reached.

An issue with this implementation is, that it is not checked if Eq. 2.23 is satisfied and the net mass flux through the free surface is converged to a small value. For a sufficient number of outer correction loops, this value converges to zero according to Muzaferija and Perić 1997 [6], but there is no convergence criterion implemented.

3.2 Differences between the interfaceTrackingFvMesh library and the interTrackFoam solver

The differences between the `interfaceTrackingFvMesh` library and the `interTrackFoam` solver in foam-extend-4.1 are mainly structural differences in the implementation. `interfaceTrackingFvMesh` is a motion solver library that inherits from `dynamicMotionSolverFvMesh` and includes the free surface boundary conditions for p and $\boldsymbol{u}$. The library can be used with any fluid solver, e.g., `pimpleFoam`. `interTrackFoam` is a solver with a modified PIMPLE algorithm described in Tukovic and Jasak 2012 [5]. It uses the `freeSurface` library with the `freeSurface` class, which is similar to the `interfaceTrackingFvMesh` described in this project, and the boundary conditions. Some major differences are that `interTrackFoam` can solve for the fluids on both sides of the interfaces, but it does not include a contact angle condition.

3.3 File structure

All files mentioned in this chapter are in reference to the directory

`$FOAM_SCR/dynamicFaMesh/interfaceTrackingFvMesh.`

The files contained in this directory are listed here:

`interTrackingFvMesh` directory

```
├── Make
│   ├── files
│   └── options
├── boundaryProcessorFaPatchPoints.H
├── freeSurfacePointDisplacement.C
├── functionObjects
│   ├── pointHistory
│   │   ├── pointHistory.C
│   │   └── pointHistory.H
│   └──writeFreeSurface
│       ├── writeFreeSurface.C
│       └── writeFreeSurface.H
├── fvPatchFields
│   ├── freeSurfacePressure
│   │   ├── freeSurfacePressureFvPatchScalarField.C
│   │   └── freeSurfacePressureFvPatchScalarField.H
│   └──freeSurfaceVelocity
│       ├── freeSurfaceVelocityFvPatchVectorField.C
│       └── freeSurfaceVelocityFvPatchVectorField.H
├── interfaceTrackingFvMesh.C
├── interfaceTrackingFvMesh.H
├── solveBulkSurfactant.H
└── surfactantProperties.H
```

In the interfaceTrackingFvMesh.* files the actual interfaceTrackingFvMesh class is defined. The file freeSurfacePointDisplacement.C includes additional member functions for the interfaceTrackingFvMesh class that handle the actual point displacement calculations described in Section 2.2.2. The freeSurfacePressure and freeSurfaceVelocity classes contain the definitions of the pressure boundary condition (see Eq. 2.13) and velocity boundary condition (see Eq. 2.12), respectively. In the file surfactantProperties.H, the surfactantProperties class is declared. This class handles different properties of surfactants on the surface and their effect on the surface tension. The solveBulkSurfactant.H file solves for the surfactant concentration on the surface and is included in the interfaceTrackingFvMesh.C file. The function object pointHistory can be used to track the absolute position of a point on the free surface, while the function object writeFreeSurface can be used to write the locations of the control points to a vtk file.

3.4 Boundary conditions in fvPatchFields

The classes for the pressure and velocity boundary conditions are inside the fvPatchField directory. Both boundary conditions inherit from the classes fixedValueFvPatchScalarField and fixedValueFvPatchVectorField respectively and use functions from the interfaceTrackingFvMesh class.

3.4.1 freeSurfacePressure

This boundary condition calculates the surface pressure jump in Eq. 2.13 and sets the pressure boundary condition. This is done in the updateCoeffs() function of the class:

freeSurfacePressureFvPatchScalarField.C

```
140 void Foam::freeSurfacePressureFvPatchScalarField::updateCoeffs()
141 {
142     if (updated())
143     {
144         return;
145     }
146
147     const fvMesh& mesh = patch().boundaryMesh().mesh();
148
149     interfaceTrackingFvMesh& itm =
150         refCast<interfaceTrackingFvMesh>
151         (
152             const_cast<dynamicFvMesh&>
153             (
154                 mesh.lookupObject<dynamicFvMesh>("fvSolution")
155             )
156         );
157
158     operator==
159     (
160         pa_ + itm.freeSurfacePressureJump()
161     );
162
163     fixedValueFvPatchScalarField::updateCoeffs();
164 }
```

The first if statement makes sure that the matrix coefficients are updated only once. Next, an object containing a constant reference to the boundary mesh is created. In the following lines, the current object itm of the interfaceTrackingFvMesh class is created. This enables access to functions and data of the object. The actual pressure at the boundary is specified in line 160. pa_ is an externally applied pressure, specified in the boundary condition dictionary. freeSurfacePressureJump is a member function of the interfaceTrackingFvMesh class and is described in Section 3.5.3

3.4.2 freeSurfaceVelocity

This boundary condition calculates and sets the velocity gradient boundary condition according to Eq. 2.12. This is done in the `updateCoeffs()` function of the class:

freeSurfaceVelocityFvPatchVectorField.C

```
90  void Foam::freeSurfaceVelocityFvPatchVectorField::updateCoeffs()
91  {
92      if (updated())
93      {
94          return;
95      }
96
97      const fvMesh& mesh = patch().boundaryMesh().mesh();
98
99      interfaceTrackingFvMesh& itm =
100         refCast<interfaceTrackingFvMesh>
101         (
102             const_cast<dynamicFvMesh&>
103             (
104                 mesh.lookupObject<dynamicFvMesh>("fvSolution")
105             )
106         );
107
108     gradient() = itm.freeSurfaceSnGradU();
109
110     fixedGradientFvPatchVectorField::updateCoeffs();
111 }
```

Here again, first an object of the `interfaceTrackingFvMesh` class is constructed (see Section 3.5.2). The applied gradient is calculated by the `freeSurfaceSnGradU` member function of the `interfaceTrackingFvMesh` class and is described in Section 3.5.5

3.5 interfaceTrackingFvMesh class

This class handles the interface tracking, displacement calculation, varying surface tension forces, and calculations needed for the free surface boundary conditions. This class inherits from the `dynamicMotionSolverFvMesh` class. In short, the necessary displacement of the free surface boundary is calculated and supplied to the `dynamicMotionSolverFvMesh` class as a boundary condition for the mesh motion problem of the entire mesh.

In the following sections, the class's member data and constructor are described as well as the most important functions:

- `freeSurfacePressureJump()`: Return free surface pressure jump
- `freeSurfaceSnGradUn()`: Return free surface normal derivative of the normal velocity component
- `freeSurfaceSnGradU()`: Return free surface normal derivative of velocity
- `pointDisplacement()`: Calculate free surface points displacement for given new control points position
- `maxCourantNumber()`: Calculates the maximal surface tension based Courant number
- `update()`: Update the mesh for both mesh motion and topology change. This is the function called by the top-level solver when using the library

How member data is accessed and handled is described later, in Section 5.3, where a new variable is implemented.

3.5.1 interfaceTrackingFvMesh.H

The class uses the following header files:

- dynamicMotionSolverFvMesh.H: includes the dynamicMotionSolverFvMesh class, which is a subclass of the dynamicFvMesh class and the superclass of interfaceTrackingFvMesh. In this class the motion of the mesh is solved by specifying a boundary condition and a diffusivity model. For more information on the dynamicFvMesh and dynamicMotionSolverFvMesh class the reader is referred to another project by Larsen 2016 [11]
- regIOobject.H: includes the abstract class regIOobject derived from the IOobject class. This object can handle automatic object registration with the objectRegistry.
- faCFD.H: This file includes, similarly to fvCFD.H, all necessary class declarations and methods for the Finite-Area-Method
- volFields.H: Volume fields are also needed for some fields, despite the FAM formulation
- surfaceFields.H: This includes the surfaceFields templated class that defines fields on surfaces similar to the volFields templated class
- surfactantProperties.H: This file includes the surfactantProperties which handles different properties of surfactants on the surface and their effect on the surface tension
- singlePhaseTransportModel.H: This file is needed to include a single-phase transport model based on the viscosityModel class
- demandDrivenData.H: This file includes the template functions to aid in the implementation of demand-driven data.

The member data included in this class is listed in the Table A.1. All member data in this class is private. The file furthermore includes the declaration of constructors for this class, some private member functions, some public member functions to return references to private member data and some additional public member functions. All relevant member functions will be discussed in the following sections together with their definition.

3.5.2 interfaceTrackingFvMesh constructor

The object constructor is declared as:

interfaceTrackingFvMesh.H: constructor declaration

```
253         //- Construct from IOobject
254         interfaceTrackingFvMesh(const IOobject& io, const bool doInit=true);
```

Here, the object is constructed from an IOobject and also takes a boolean variable doInit. This flag controls if the init() function should also be called in the constructor. In the definition of the constructor default values for the member data are set and the init() function is called:

interfaceTrackingFvMesh.C: constructor definition

```
1554 Foam::interfaceTrackingFvMesh::interfaceTrackingFvMesh
1555 (
1556     const IOobject& io,
1557     const bool doInit
1558 )
1559 :
1560     dynamicMotionSolverFvMesh(io, doInit),
1561     aMeshPtr_(nullptr),
1562     fsPatchIndex_(-1),
1563     fixedFreeSurfacePatches_(),
1564     nonReflectingFreeSurfacePatches_(),
1565     pointNormalsCorrectionPatches_(),
```

```
1566     normalMotionDir_(false),
1567     motionDir_(Zero),
1568     smoothing_(false),
1569     pureFreeSurface_(true),
1570     rigidFreeSurface_(false),
1571     correctContactLineNormals_(false),
1572     sigma0_("zero", dimForce/dimLength/dimDensity, Zero),
1573     rho_("one", dimDensity, 1.0),
1574     timeIndex_(-1),
1575     UsPtr_(nullptr),
1576     controlPointsPtr_(nullptr),
1577     motionPointsMaskPtr_(nullptr),
1578     pointsDisplacementDirPtr_(nullptr),
1579     facesDisplacementDirPtr_(nullptr),
1580     fsNetPhiPtr_(nullptr),
1581     phisPtr_(nullptr),
1582     surfactConcPtr_(nullptr),
1583     bulkSurfactConcPtr_(nullptr),
1584     surfaceTensionPtr_(nullptr),
1585     surfactantPtr_(nullptr),
1586     contactAnglePtr_(nullptr)
1587 {
1588     if (doInit)
1589     {
1590         init(false);    // do not initialise lower levels
1591     }
1592 }
```

The `init()` function is located in the file `interfaceTrackingFvMesh.C`. Here, first, the object is constructed from its superclass `dynamicMotionSolverFvMesh`, if specified by `init` flag (`true` by default), which also constructs the pointer to the `motionSolver` object. The `aMeshPtr_` is (re)set to the finite area mesh of the current object. Next, the variables are read from the `dynamicMeshDict`. Variables read with `get` are mandatory, while variables read with `getOrDefault` are set to a default value if not specified:

`interfaceTrackingFvMesh.C`: init function

```
1662 bool Foam::interfaceTrackingFvMesh::init(const bool doInit)
1663 {
1664     if (doInit)
1665     {
1666         dynamicMotionSolverFvMesh::init(doInit);
1667     }
1668
1669     aMeshPtr_.reset(new faMesh(*this));
1670
1671     // Set motion-based data
1672     fixedFreeSurfacePatches_ =
1673         motion().get<wordList>("fixedFreeSurfacePatches");
1674
1675     pointNormalsCorrectionPatches_ =
1676         motion().get<wordList>("pointNormalsCorrectionPatches");
1677
1678     normalMotionDir_ = motion().get<bool>("normalMotionDir");
1679     smoothing_ = motion().getOrDefault("smoothing", false);
1680     pureFreeSurface_ = motion().getOrDefault("pureFreeSurface", true);
1681
1682     initializeData();
1683
1684     return true;
1685 }
```

The `initializeData()` function is called, which is located in `interfaceTrackingFvMesh.C` as well (lines 75-127). In this function, some additional member data is initialized and checked, the vector for the required motion direction is read, if `normalMotionDir` was set to `false`. Additionnally, the function `makeContactAngle` is called (located in lines 531-630), which reads the `contactAngle`

boundary conditions, from the current time directory (i.e., `0/contactAngle`), if present. This is the function that creates the `contactAnglePtr_`:

3.5.3 `freeSurfacePressureJump()` function

This member function returns the free pressure jump to be used in the boundary condition `freeSurfacePressure` (see Section 3.4.1) and is defined as follows:

`interfaceTrackingFvMesh.C`: `freeSurfacePressureJump()`

```
1874 Foam::tmp<scalarField>
1875 Foam::interfaceTrackingFvMesh::freeSurfacePressureJump()
1876 {
1877     auto tPressureJump = tmp<scalarField>::New(aMesh().nFaces(), Zero);
1878     auto& pressureJump = tPressureJump.ref();
1879
1880     const scalarField& K = aMesh().faceCurvatures().internalField();
1881
1882     const uniformDimensionedVectorField& g =
1883         meshObjects::gravity::New(mesh().time());
1884
1885     const turbulenceModel& turbulence =
1886         mesh().lookupObject<turbulenceModel>("turbulenceProperties");
1887
1888     scalarField nu(turbulence.nuEff(fsPatchIndex()));
1889
1890     pressureJump =
1891       - (g.value() & mesh().Cf().boundaryField()[fsPatchIndex()])
1892       + 2.0*nu*freeSurfaceSnGradUn();
1893
1894     if (pureFreeSurface())
1895     {
1896         pressureJump -= sigma().value()*K;
1897     }
1898     else
1899     {
1900         pressureJump -= surfaceTension().internalField()*K;
1901     }
1902
1903     return tPressureJump;
1904 }
```

In this function, first, the necessary fields are created: the curvature κ is calculated in line 1880 from the area mesh `aMesh`, the gravitational acceleration $\boldsymbol{g}$ is read, and the viscosity ν is calculated based on the turbulence model. The pressure jump is then calculated either with a constant surface tension σ, if the surface should be pure, or with a variable surface tension reurned by `surfaceTension()`. The body force $\boldsymbol{g} \cdot \boldsymbol{x}_f$, where $\boldsymbol{x}_f$ are the positions of the face centres returned by

```
mesh().Cf().boundaryField()[fsPatchIndex()],
```

is subtracted, since the specified pressure `pa_` is only a static pressure. The equation for the pressure jump is already very reminiscent of Eq. 2.13.

3.5.4 `freeSurfaceSnGradUn()`

This function calculates the surface divergence of the surface velocity ($\bar{\nabla} \cdot \boldsymbol{u}$) The code is as follows:

`interfaceTrackingFvMesh.C`: `freeSurfaceSnGradUn()`

```
1857 Foam::interfaceTrackingFvMesh::freeSurfaceSnGradUn()
1858 {
1859     auto tSnGradUn = tmp<scalarField>::New(aMesh().nFaces(), Zero);
1860     auto& SnGradUn = tSnGradUn.ref();
1861
1862     areaScalarField divUs
```

```
1863     (
1864         fac::div(Us())
1865       - aMesh().faceCurvatures()*(aMesh().faceAreaNormals()&Us())
1866     );
1867
1868     SnGradUn = -divUs.internalField();
1869
1870     return tSnGradUn;
1871 }
```

First, a pointer to a `scalarField` of the size of the area mesh is created together with its reference. `divUs` is the surface divergence of the surface velocity, calculated with Eq. 2.15. It can be seen that `fac::div(Us())` returns the first term on the right side and the correction needed due to a surface curvature is done in line 1865. Since this term is subtracted in the pressure jump, but in `freeSurfacePressureJump()` the term is added, the function returns the negative value.

3.5.5 freeSurfaceSnGradU

This member function returns the free surface normal derivative of velocity to be used in the boundary condition `freeSurfaceVeloctityFvPatchVectorField.C` (see Section 3.4.2). The code is as follows:

interfaceTrackingFvMesh.C: freeSurfaceSnGradU

```
1813 Foam::tmp<Foam::vectorField>
1814 Foam::interfaceTrackingFvMesh::freeSurfaceSnGradU()
1815 {
1816     auto tSnGradU = tmp<vectorField>::New(aMesh().nFaces(), Zero);
1817     auto& SnGradU = tSnGradU.ref();
1818
1819     const vectorField& nA = aMesh().faceAreaNormals().internalField();
1820
1821     areaScalarField divUs
1822     (
1823         fac::div(Us())
1824       - aMesh().faceCurvatures()*(aMesh().faceAreaNormals()&Us())
1825     );
1826
1827     areaTensorField gradUs(fac::grad(Us()));
1828
1829     // Remove component of gradient normal to surface (area)
1830     const areaVectorField& n = aMesh().faceAreaNormals();
1831     gradUs -= n*(n & gradUs);
1832     gradUs.correctBoundaryConditions();
1833
1834     const turbulenceModel& turbulence =
1835         mesh().lookupObject<turbulenceModel>("turbulenceProperties");
1836
1837     scalarField nu(turbulence.nuEff(fsPatchIndex()));
1838
1839     vectorField tangentialSurfaceTensionForce(nA.size(), Zero);
1840
1841     if (!pureFreeSurface() && max(nu) > SMALL)
1842     {
1843         tangentialSurfaceTensionForce =
1844             surfaceTensionGrad()().internalField();
1845     }
1846
1847     SnGradU =
1848         tangentialSurfaceTensionForce/(nu + SMALL)
1849       - nA*divUs.internalField()
1850       - (gradUs.internalField()&nA);
1851
1852     return tSnGradU;
1853 }
```

The final gradient can be seen on lines 1847-1850 and resembles Eq. 2.12. The term `tangentialSurfaceTensionForce` is 0 for pure surfaces or calculated in `surfaceTensionGrad` and `divUs` is calculated as in 3.5.4. The surface gradient of `Us` is calculated as only the tangential part of the gradient on the surface. As a note, this is the same as the surface gradient defined in 2.8. In the code, `gradUs`$\hat{=}\nabla' \boldsymbol{u}_s$ is calculated as follows and can be rearranged to show the surface gradient operator defined in Eq. 2.8:

$$\nabla' \boldsymbol{u}_s = \nabla \boldsymbol{u}_s - \boldsymbol{n}_f \left(\boldsymbol{n}_f \cdot \nabla \boldsymbol{u}_s\right) = \left(\nabla - \boldsymbol{n}_f \left(\boldsymbol{n}_f \cdot \nabla\right)\right) \boldsymbol{u}_s = \left(\left(\boldsymbol{I} - \boldsymbol{n}_f \boldsymbol{n}_f\right) \cdot \nabla\right) \boldsymbol{u}_s = \bar{\nabla} \boldsymbol{u}_s \tag{3.1}$$

3.5.6 `pointDisplacement()` function in `freeSurfacePointDisplacement.C`

The `pointDisplacement()` function is located in the file `freeSurfacePointDisplacement.C`. This function calculates the displacement of the free surface points for given control points (the last step in figure 2.2) together with the `lsPlanePointAndNormal` function. The largest part of the function handles finding the corresponding `controlPoints` for each free surface point. `controlPoints` are specified for each face on the free surface. For boundary points, additional `patchMirrorPoints` have to be specified. The free surface points are split into four categories:

1. Inner points are the internal points of the `aMesh` and can be accessed with

```
53    labelList internalPoints = aMesh().internalPoints();
```

   The `controlPoints` for these points can be found by looping through the owning faces of a point. The labeling of points to faces is handled by the variable

```
45    const labelListList& pointFaces = aMesh().patch().pointFaces();
```

   which contains a list of all face labels of all points.

2. Boundary points are the points on the boundary of the `aMesh`. These points are shared with other boundary fields. They can be accessed with

```
232   labelList boundaryPoints = aMesh().boundaryPoints();
```

   The `controlPoints` of these points include the `controlPoints` of their inner faces and their `patchMirrorPoints`, which are the `controlPoints` mirrored at the boundary edge (see figure 2.2).

3. Points on a processor patch. The `controlPoints` for these points have to be grabbed from neighboring processor patches

4. Global processor patch points. The `controlPoints` for these points are either global points themselves, or have to be accessed from other processor patches.

3.5.6.1 `patchMirrorPoints`

The `patchMirrorPoints` are calculated as follows:

```
87    // Mirror control points
88    FieldField<Field, vector> patchMirrorPoints(aMesh().boundary().size());
89
90    // Old faMesh points
91    vectorField oldPoints(aMesh().nPoints(), Zero);
92    const labelList& meshPoints = aMesh().patch().meshPoints();
93    forAll(oldPoints, pI)
94    {
95        oldPoints[pI] =
96            mesh().oldPoints()[meshPoints[pI]];
97    }
98
99    forAll(patchMirrorPoints, patchI)
```

```
100     {
101         patchMirrorPoints.set
102         (
103             patchI,
104             new vectorField
105             (
106                 aMesh().boundary()[patchI].faPatch::size(),
107                 Zero
108             )
109         );
110 
111         vectorField N
112         (
113             aMesh().boundary()[patchI].ngbPolyPatchFaceNormals()
114         );
115 
116         const labelList& eFaces =
117             aMesh().boundary()[patchI].edgeFaces();
118 
119         // Correct N according to specified contact angle
120         if (contactAnglePtr_)
121         {
```

```
197         }
198 
199         const labelList peFaces =
200             labelList::subList
201             (
202                 aMesh().edgeOwner(),
203                 aMesh().boundary()[patchI].faPatch::size(),
204                 aMesh().boundary()[patchI].start()
205             );
206 
207         const labelList& pEdges = aMesh().boundary()[patchI];
208 
209         vectorField peCentres(pEdges.size(), Zero);
210         forAll(peCentres, edgeI)
211         {
212             peCentres[edgeI] =
213                 edges[pEdges[edgeI]].centre(points);
214         }
215 
216         vectorField delta
217         (
218             vectorField(controlPoints(), peFaces)
219           - peCentres
220         );
221 
222         // Info<< aMesh().boundary()[patchI].name() << endl;
223         // Info<< vectorField(controlPoints(), peFaces) << endl;
224 
225         patchMirrorPoints[patchI] =
226             peCentres + ((I - 2*N*N)&delta);
227 
228         // Info<< patchMirrorPoints[patchI] << endl;
229     }
```

The outer `forAll` loop loops through all free surface boundary patches (Remember: the fa-boundaries of the free surface mesh are edges). The locations of the `patchMirrorPoints` are stored in a `FieldField<Field, vector>`, meaning it is a field of the size of the total fa-boundary patches (4). Each of these fields contains a number of vectors (coordinates) equal to the number of edges in that fa-patch. Each edge on that fa-boundary has a neighboring face from another fv-boundary. The face normals of these faces are stored in `N`. The next lines calculate the `vectorField` delta which points from the edge center to the `controlPoint` of the inner face. This point is that mirrored at the edge and written to `patchMirrorPoints`.

3.5.6.2 patchMirrorPoints with specified contact angle

The code in lines 119-197 handles the contact angle boundary condition if specified in the current time directory. The vector N, which is used for calculating the patchMirrorPoints, is rotated with Rodriguez formula as

$$\boldsymbol{N}^{\mathrm{rot}} = \boldsymbol{N}\cos\theta + \boldsymbol{e}_r\left(\boldsymbol{e}_r \cdot \boldsymbol{N}\right)(1-\cos\theta) + \left(\boldsymbol{e}_r \times \boldsymbol{N}\right), \tag{3.2}$$

with θ beeing the angle as seen in figure 1.1 and equal to 90 - contactAngle and $\boldsymbol{e}_r$ is the axis of rotation. This is only done if the contactAnglePtr_ exists, the neighboring fv-Patch is of type wall, and the contactAngle boundary is of type calculated. The location $\boldsymbol{r}^{\mathrm{PMP}}$ of the patchMirrorPoints is then calculated with

$$\boldsymbol{r}^{\mathrm{PMP}} = \mathbf{r}^{\mathrm{eC}} + \left(\boldsymbol{I} - 2\boldsymbol{N}^{\mathrm{rot}}\boldsymbol{N}^{\mathrm{rot}}\right)\cdot\boldsymbol{\delta}, \tag{3.3}$$

where $\mathbf{r}^{\mathrm{eC}}$ is the center of the edge, and $\boldsymbol{\delta}$ is the vector pointing from the edge center to the inner controlPoint. As can be seen from this formulation, the contact angle condition is applied in an indirect way by setting the patchMirrorPoints, which in turn are used to calculate the optimal displacement of the face centers. In the tutorial presented in Section 4 this contact angle is not reached. It seems this condition is not strong enough; some alternative implementations are discussed in Sections 5.5 and 5.5.

Chapter 4

Tutorial Case

4.1 Description

This tutorial describes preparing and running a case using the `interTrackFvMesh` library and the `freeSurfaceVelocity` and `freeSurfacePressure` boundary conditions. The tutorial case deals with the formation of a free surface in a cavity with a specified contact angle. The top boundary is the free surface, while for the other boundaries the `slip` condition is set. The results after a simulation time of $t = 0.2$ s are shown in figure 4.1. The boundary with the `freeSurface` condition is at the top, while the other boundaries are set to `slip`. The `frontAndBack` boundaries are type `empty`. It should be noted that the `freeSurface` starts horizontally and, while it changes a bit over time, it converges to the shown result. Since this report focuses on the free surface simulation, the actual flow in the domain is not be shown.

4.2 Setup

In order to run this tutorial you can execute the following commands in your `run` directory:

```
cp -r $FOAM_TUTORIALS/incompressible/pimpleFoam/laminar/contactAngleCavity .
./Allrun
```

The tutorial can also be run with the accompanying files, including some post-processing:

```
cd contactAngleCavity
./Allrun
```

4.3 Files

The file structure of the case is as follows:

```
├── 0.orig
│   ├── U
│   ├── contactAngle
│   ├── p
│   └── pointMotionU
├── Allclean
├── Allrun
├── constant
│   ├── dynamicMeshDict
│   ├── g
│   ├── transportProperties
│   └── turbulenceProperties
└── system
    ├── blockMeshDict
```

```
    ├── controlDict
    ├── decomposeParDict
    ├── faMeshDefinition
    ├── faSchemes
    ├── faSolution
    ├── fvSchemes
    └── fvSolution
```

This is in the standard OpenFOAM structure. Some additional files are `faSolution`, `faSchemes`, `faMeshDefinition`, `dynamicMeshDict` and `contactAngle`. In order to use the `interFaceTrackingFvMesh` library, it is included in the `controlDict` as

```
libs    (interFaceTrackingFvMesh);
```

4.3.1 `faMeshDefinition` file

This file includes the definition of the *finite-area-mesh*. It has two subdictionaries `polyMeshPatches` and `boundary`. In `polyMeshPatches` the name of the `polyPatch` that is converted to a *FA-mesh* is provided. The `boundary` dictionary is similar to the one in a `blockMeshDict`, but here the defined patches are boundaries of the *FA-mesh*, meaning they are the *edges* of the free surface. Additionally the neighboring `polyPatch` has to be specified. In the below example the boundary `left` is created and contains all the edges that are between the `polyPatch top` and the `polyPatch left`. By running `makeFaMesh` in the terminal, the *FA-mesh* is created. This includes the files `faBoundary` and `faFaceLabels`. `faFaceLabels` contains the poly patch face labels of the free surface boundary. `faBoundary` contains the boundary mesh.

system/faMeshDefinition

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2112                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      faMeshDefinition;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

polyMeshPatches  ( top );

boundary
{
    left
    {
        type                patch;
        neighbourPolyPatch  left;
    }

    right
    {
        type                patch;
        neighbourPolyPatch  right;
    }

    frontAndBack
    {
        type                empty;
        neighbourPolyPatch  frontAndBack;
    }
```

```
}


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

4.3.2 faSolution file

This file is empty in this tutorial. It can include solution settings for transport equations that are solved on the free surface, like a surfactant transport equation. This is done in another tutorial:

```
$FOAM_TUTORIALS/incompressible/pimpleFoam/laminar/contaminatedDroplet2D
```

4.3.3 faSchemes file

This file contains the numerical schemes used to discretize the operators on the *FA-mesh*. If the surfactant transport equation is solved, additional `divSchemes` have to be specified.

4.3.4 dynamicMeshDict file

In this file, the settings for the dynamic mesh motion solver are set. First, the `motionSolverLibs` are specified. The `dynamicFvMesh` is the class type of the dynamic mesh. In this case, it should be of type `interfaceTrackingFvMesh`. The `motionSolver` and `diffusivity` settings are specifying the type of motion solver.

dynamicMeshDict

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2112                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      dynamicMeshDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

motionSolverLibs (fvMotionSolvers interfaceTrackingFvMesh);

dynamicFvMesh   interfaceTrackingFvMesh;

motionSolver    velocityLaplacian;

diffusivity     uniform; //onTimeChange inverseDistance 1(top);


// Free surface data

fsPatchName top;

fixedFreeSurfacePatches ( );

pointNormalsCorrectionPatches ();
// pointNormalsCorrectionPatches ( left right );

normalMotionDir false;

motionDir (0 1 0);

```

```
39
40 // ************************************************************************* //
```

The settings for the `interfaceTrackingFvMesh` class are:

- `fsPatchName`: Name of free surface `polyPatch`
- `fixedFreeSurfacePatches`: Names of fixed free surfaces. Here a free surface patch can be specified to not move
- `pointNormalCorrectionPatches`: Free surface patches for which point normals must be corrected
- `pointNormalCorrectionPatches`: Free surface patches where a wave should not be reflected
- `normalMotionDir`:
  `true`: motion is in point normal direction (see $\boldsymbol{n}_i$ in figure 2.2)
  `false`: motion is in direction of `motionDir` (see $\boldsymbol{d}_i$ in figure 2.2)

Additional entries are

- `pureFreeSurface` of type boolean. This is `false` by default. If set to `true` the surface tension is dependent on the surfactant concentrations. The properties can be set in the `surfactantProperties` dictionary as can be seen in

  ```
  $FOAM_TUTORIALS/incompressible/pimpleFoam/laminar/contaminatedDroplet2D/\
  constant/dynamicMeshDict
  ```

4.3.5 `contactAngle` file

In this file, the boundary condition for the contact angle is set. This angle is specified as the angle between the free surface tangent and the wall. It is specified for the boundaries of the *FA-mesh*, i.e., the edges of the free surface, denoted as `left` and `right`. The condition is activated if this file exists.

`contactAngle`

```
1  /*--------------------------------*- C++ -*----------------------------------*\
2  | =========                 |                                                 |
3  | \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
4  |  \\    /   O peration     | Version:  v2112                                 |
5  |   \\  /    A nd           | Website:  www.openfoam.com                      |
6  |    \\/     M anipulation  |                                                 |
7  \*---------------------------------------------------------------------------*/
8  FoamFile
9  {
10     version     2.0;
11     format      ascii;
12     class       areaScalarField;
13     object      contactAngle;
14 }
15 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
16
17 dimensions      [0 0 0 0 0 0 0];
18
19 internalField   uniform 0;
20
21 boundaryField
22 {
23     left
24     {
25         type            calculated;
26         value           uniform 70;
27     }
```

```
28
29      right
30      {
31          type            calculated;
32          value           uniform 70;
33      }
34
35      frontAndBack
36      {
37          type            empty;
38      }
39  }
40
41
42  // ************************************************************************* //
```

4.4 Results

Run the tutorial provided in the accompanying files by

```
cd contactAngleCavity
./Allrun
```

This includes a sample utility that writes out the free surface location and pressure values.

The resulting files in a one time directory are:

```
1  U_0
2  Uf
3  Uf_0
4  Us
5  V0
6  cellMotionU
7  contactAngle
8  controlPoints
9  freeSurfaceControlPoints.vtk
10 fsNetPhi
11 meshPhi
12 p
13 phi
14 pointMotionU
15 polyMesh/
16 uniform/
```

Some of those files are fields of the *FA-mesh* and can not be loaded into `paraFoam`. This is addressed in a modification described in section 5.4. The file `freeSurfaceControlPoints.vtk` contains the locations of the `controlPoints`. The actual free surface shape has to be extracted from the `polyMesh` with `foamToVTK`, as seen in the `Allrun` script. Figure 4.2 shows the resulting free surface shape in this tutorial. It should be noted that this shape results only from the contact angle condition. There is no gravity. Figure 4.3 shows the point history of the two leftmost mesh points and the contact angle calculated between the tangent formed by those points and the horizontal wall. The set contact angle of 70 is not reached in this simulation, despite the boundary conditions, requiring further investigations and possible modifications.

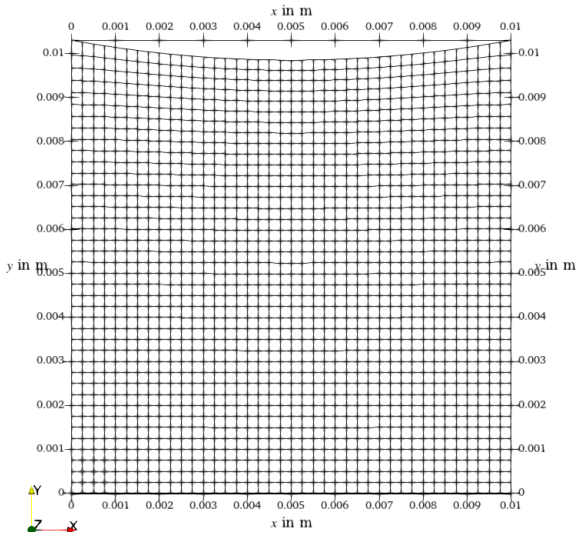


Figure 4.1: Result of `contactAngleCavity` tutorial case with gravity after $t = 0.2$ s

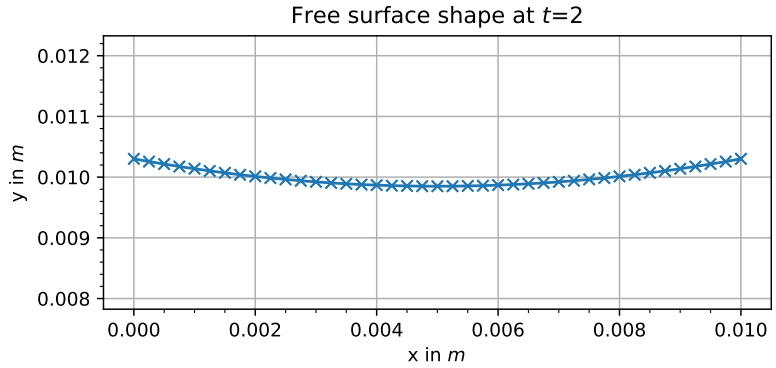


Figure 4.2: Resulting free surface shape of the `contactAngleCavity` tutorial case with after $t = 0.2$ s)

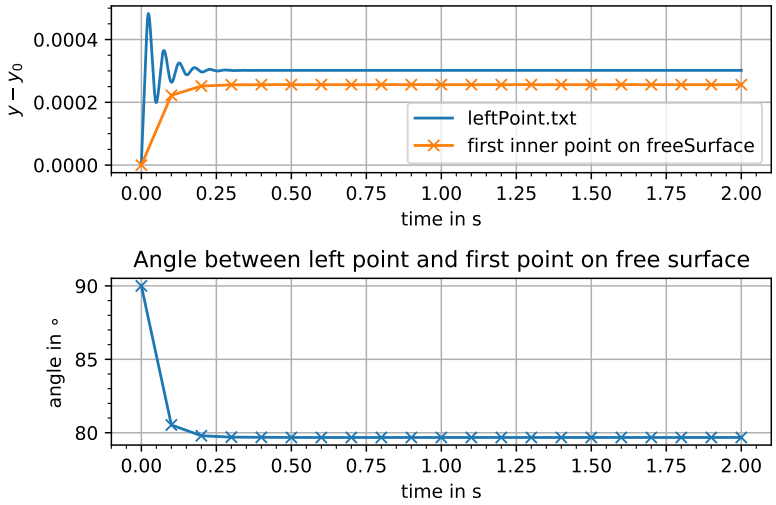


Figure 4.3: Top: time evolution of left-most point on the free surface and first inner point. Bottom: contact angle between those two points

Chapter 5

Modifications

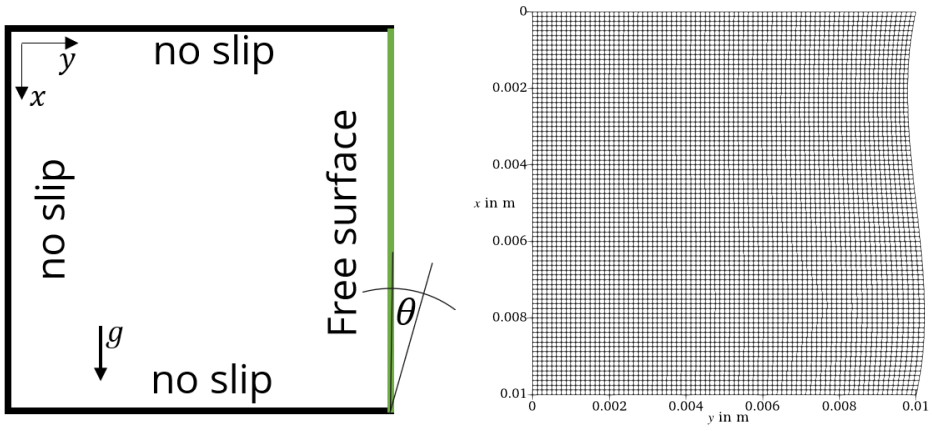
5.1 Handling the code

In order to use the developed code please download the accompanying files, and run

```
tar xf Report_IasonTsiapkinis.tar
cd Report_IasonTsiapkinis
./Allwmake
./Allrun
```

5.2 New case setup `contactAngleColumn`

A new test case was set up for testing and showing the implemented modifications, resembling the process described in Figure 1.1 with a modified `contactAngleCavity` tutorial. In the new case, the working fluid is silicon with a density of $\rho = 2580\frac{\text{kg}}{\text{m}^3}$, kinematic viscosity of $\nu = 3.333 \cdot 10^{-7}\frac{\text{m}^2}{\text{s}}$, surface tension of $\sigma = 0.88\frac{\text{N}}{\text{m}}$, and contact angle $\theta = 11°$. In this setup, gravity is included by adding it in the `constant/g` file.



(a) Setup (b) Free surface shape

Figure 5.1: `contactAngleColumn` case for demonstrating the modifications

Run the case with the `./Allrun` command and look at the results with `paraFoam`.

5.3 Calculate face angles

This section describes how to calculate, store and access a new variable inside the library. The free surface fields are handled with `mutable` pointers and accessed through *public* member functions of that class. The value that should be calculated is the angle between each face and a direction, which the user specifies as `verticalDir` in the `dynamicMeshDict`. Calculating these values helps with checking the contact angle without additional and complicated post-processing. It is also needed for the implementation of the pressure boundary conditions described in Section 5.6. The following additional *private* member data needs to be declared in `myInterfaceTrackingFvMesh.H`, anywhere before the `public` area:

- `vector verticalDir_;`: This is the vertical vector provided by the user for the calculation of the face angle. It is provided in the `dynamicMeshDictionary`.
- `mutable areaScalarField* faceAnglesPtr_;`: This is the pointer to the face angle values

To access the user-defined variable `verticalDir`, it has to be added to the `init()` function as seen below. By using `getOrDefault`, this variable is not necessarily required. The default value is a zero vector, in which case the calculated face angles will also be all zero.

```
1729    verticalDir_ = normalised(motion().getOrDefault<vector>("verticalDir",Zero));
```

Additionally, the member data `verticalDir_` has to be added to the constructor (see line 1610).

Additionally the following *private* member functions need to be declared in the file `interfaceTrackingFvMesh.H`, again, anywhere before the `public` area:

- `void makeFaceAngles() const;` This function creates the field to store the face angles
- `void updateFaceAngles()`: This function calculates the face angles and updates them

In order to access the field, two *public* member functions are declared:

- `const areaScalarField& faceAngles() const;`: This function returns a constant reference to the `faceAnglesPtr_` field
- `areaScalarField& faceAngles();`: This function returns a reference to the `faceAnglesPtr_` field

The member functions are defined in `myInterfaceTrackingFvMesh.C`. The two functions named `faceAngles()` return either a constant reference or a reference to an `areaScalarField`. If it does not exist, they also create the pointer by calling the `makeFaceAngles()` function. These are the functions that should be called to return the face angle field.

`interfaceTrackingFvMesh.C` - `faceAngles()` member function

```
1780 Foam::areaScalarField& Foam::myInterfaceTrackingFvMesh::faceAngles()
1781 {
1782     if (!faceAnglesPtr_)
1783     {
1784         makeFaceAngles();
1785     }
1786
1787     return *faceAnglesPtr_;
1788 }
1789
1790
1791 const Foam::areaScalarField& Foam::myInterfaceTrackingFvMesh::faceAngles() const
1792 {
1793     if (!faceAnglesPtr_)
1794     {
1795         makeFaceAngles();
1796     }
1797
1798     return *faceAnglesPtr_;
1799 }
```

In the `makeFaceAngles()` function, a new `areaScalarField` object is created as an `IOobject`, which is in turn set to `AUTO_WRITE` to be written together with the other variables. The object is assigned to the pointer `faceAnglesPtr_`. This function also makes sure, that the allocation happens only once and prints additional information.

`interfaceTrackingFvMesh.C` - `makeFaceAngles()` member function

```
284 void Foam::myInterfaceTrackingFvMesh::makeFaceAngles() const
285 {
286     DebugInFunction
287         << "making surface face angles" << nl;
288
289     if (faceAnglesPtr_)
290     {
291         FatalErrorInFunction
292             << "surface face angles already exists"
293             << abort(FatalError);
294     }
295
296     Info<< "Making surface face angles" << endl;
297
298     faceAnglesPtr_ = new areaScalarField
299     (
300         IOobject
301         (
302             "faceAngles",
303             mesh().time().timeName(),
304             mesh(),
305             IOobject::NO_READ,
306             IOobject::AUTO_WRITE
307         ),
308         aMesh(),
309         dimensionedScalar(Zero)
310     );
311 }
```

The `updateFaceAngles()` calculates the current face angles and ensures that the field is only updated when specified and not every time the face angle is called. It is added in the `update()` function of the class after the call of the `updateProperties()` function. The angles are calculated between the face-normal vector `Nf` of each face and the direction `verticalDir_` specified by the user by taking the inner product of these two vectors and calculating the angle between them. The result is in degrees. The sign of the angle is set to the sign of the curvature.

`interfaceTrackingFvMesh.C` - `updateFaceAngles()` member function

```
660 void Foam::myInterfaceTrackingFvMesh::updateFaceAngles()
661 {
662     // Calculate local angle of face
663     const vectorField& Nf = aMesh().faceAreaNormals().internalField();
664
665     forAll(faceAngles(), faceI)
666     {
667         faceAngles()[faceI] =  90 - radToDeg(acos((Nf[faceI]&verticalDir_)));
668         faceAngles()[faceI] *= sign(aMesh().faceCurvatures()[faceI]);
669     }
670 }
```

5.4 Write out additional surface data

The `areaFields` written to the time directories are not picked up by `paraFoam`. The `writeVTK()` function in the file `interfaceTrackingFvMesh.C` is expanded into writing out the free surface geometry and some of the fields. The type of the `writer` object needs to be changed to be able to write out `areaFields` to `vtk` format.

interfaceTrackingFvMesh.C - writeVTK() member function

```
2450 void Foam::myInterfaceTrackingFvMesh::writeVTK() const
2451 {
2452     // GenericPatchGeoFieldsWriter<uindirectPrimitivePatch>
2453     vtk::GenericPatchGeoFieldsWriter<uindirectPrimitivePatch> writer
2454     (
2455         aMesh().patch(),
2456         vtk::formatType::LEGACY_ASCII,
2457         mesh().time().timePath()/"freeSurface",
2458         false // serial only
2459     );
2460     writer.writeGeometry();
2461     writer.beginCellData(3);
2462     writer.write(Us());
2463     writer.write(fsNetPhi());
2464     writer.write(aMesh().faceCurvatures());
2465     writer.write(faceAngles());
2466 }
```

This function is called in the `writeFreeSurface functionObject` after the call to `itm.writeVTKControlPoints();`:

```
itm.writeVTKControlPoints();
itm.writeVTK();
```

It can be added to the `system/controlDict.functionObjects` as

```
writeFreeSurface
{
    type                writeFreeSurface;
}
```

The file `freeSurface.vtk`, which includes the above fields, is written in each time directory and can be imported into ParaView. Figure 5.2 shows the face curvatures and angles loaded into ParaView through the `freeSurface.vtk` file of the last time step. It is the same free surface as in Figure 5.1b. Creating these files and writing out the fields enables easier post-processing.

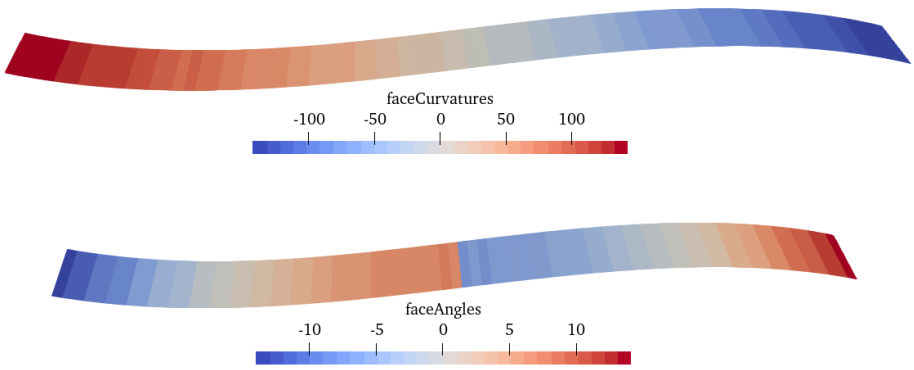


Figure 5.2: Face curvature and face angles loaded into ParaView through `freeSurface.vtk`

5.5 New contact angle condition inside `pointDisplacement`

In this section, a new condition to calculate the `controlPoints` at a patch, with a specified contact angle, is proposed and implemented inside the `pointDisplacement()` function. The following code is added after the calculation of `delta`:

pointDisplacement()— in freeSurfacePointDisplacement.C

```
234         if (contactAnglePtr_)
235         {
236             label ngbPolyPatchID =
237                 aMesh().boundary()[patchI].ngbPolyPatchIndex();
238             if
239             (
240                 mesh().boundary()[ngbPolyPatchID].type()
241                 == wallFvPatch::typeName
242                 &&
243                 contactAnglePtr_->boundaryField()[patchI].type()
244                 == "fixedValue"
245             )
246             {
247                 forAll(peCentres, edgeI)
248                 {
249                     controlPoints()[eFaces[edgeI]] = peCentres[edgeI]
250                                     - ((-1*N[edgeI])&delta[edgeI])*N[edgeI];
251                 }
252             }
253         }
```

The new `type fixedValue` for the `contactAngle` can be specified in the `0/contactAngle` file. The `forAll` loop goes through all edges on the current *FA-Patch*. The new `controlPoints` are placed such that the angle between $\boldsymbol{\delta} = \boldsymbol{r}^{\mathrm{CP}} - \boldsymbol{r}^{\mathrm{eC}}$ and the adjacent wall is equal to the contact angle. The vector `delta` points from the edge center to the control point and is calculated previously (see Section 3.5.6.1) by using the normal vector `N`, which was already rotated by the `contactAngle` (see Section 3.5.6.2). This is analogous to the calculation of the `patchMirrorPoints`, albeit here, the inner control points are set in order to satisfy the angle. This ensures that for the face at a boundary, the `patchMirrorPoint`, the edge center, and the `controlPoint` are on one line (this yields the configuration on the right boundary of Figure 2.2 (top). The angle between this line and the neighboring wall is the `contactAngle` as shown in Figure 2.3c.

It is not always desired to specify a contact angle at every boundary. Adding the `0/contactAngle` file to a simulation requires the user to supply a `contactAngle` for every *FA-patch*. Adding an `if`-statement (as seen in the code below), such that `N` is only rotated for these two types, allows the user to specify another type, e.g., `type zeroGradient`. For this *FA-Patch* type, there will be no calculations based on a contact angle. Therefore, the following types are available in the `0/contactAngle` file:

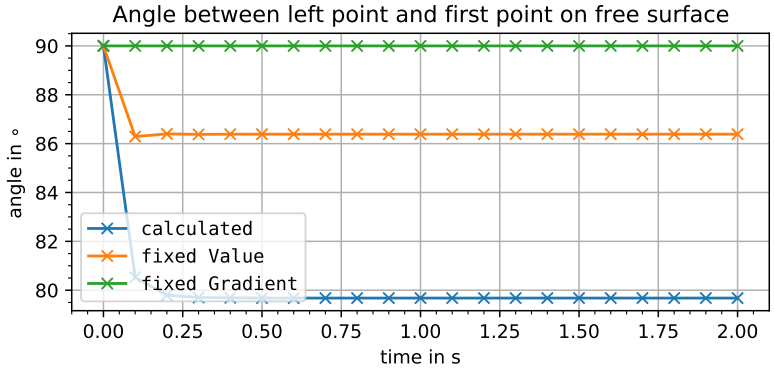
- `type calculated`: uses the default calculation explained in Section 2.3
- `type fixedValue`: uses the calculation explained in this Section
- `type zeroGradient`: no contact angle calculation is done at this *FA-Patch*

pointDisplacement() in freeSurfacePointDisplacement.C

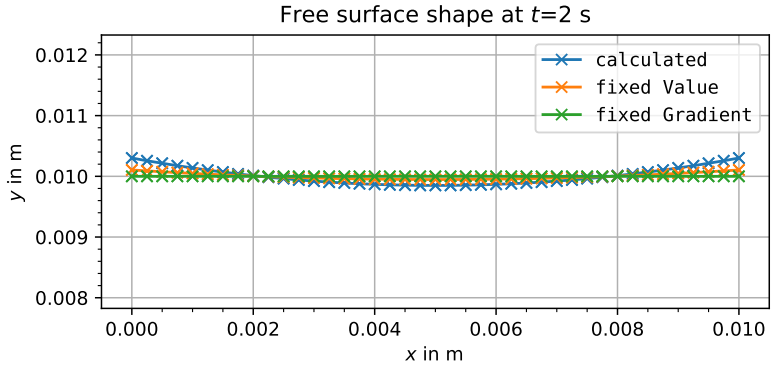
```
120         if (contactAnglePtr_)
121         {
122             label ngbPolyPatchID =
123                 aMesh().boundary()[patchI].ngbPolyPatchIndex();
124
125             if (ngbPolyPatchID != -1)
126             {
127                 if
128                 (
129                     mesh().boundary()[ngbPolyPatchID].type()
130                  == wallFvPatch::typeName
131                  &&
132                  (
133                  contactAnglePtr_->boundaryField()[patchI].type()
134                  == "fixedValue"
```

```
                    ||
                    contactAnglePtr_->boundaryField()[patchI].type()
                    == "calculated"
                    )
                  )
```

Figure 5.3 shows the contact angle and the free surface shape of the cavity tutorial case from Chapter 4 with a fixed contact angle of 70° and for the three different boundary conditions. The `fixedGradient` condition works as intended, i.e., the code setting the `contactAngle` is skipped. The new `type fixedValue` condition performs worse than the default `type calculated` and requires further development.



(a) Face angle of the leftmost face over time.



(b) Free surface shape

Figure 5.3: `contactAngleCavity` case with a contact angle of 70° and for the three contact angel boundary condition types

5.6 Pressure boundary condition for constant contact angle

Implementing this boundary condition follows the theory described in Section 2.3. Start with the existing `freeSurfacePressure` boundary condition and rename the files and the type name to `freeSurfaceContactAnglePressure`:

```
cp -r fvPatchFields/freeSurfacePressure \
fvPatchFields/freeSurfaceContactAnglePressure
mv fvPatchFields/freeSurfaceContactAnglePressure/freeSurfacePressure.H \
fvPatchFields/freeSurfaceContactAnglePressure/freeSurfaceContactAnglePressure.H
```

```
mv fvPatchFields/freeSurfaceContactAnglePressure/freeSurfacePressure.C \
fvPatchFields/freeSurfaceContactAnglePressure/freeSurfaceContactAnglePressure.C
sed -i "s/freeSurfacePressure/freeSurfaceContactAnglePressure/g"
 \ freeSurfaceContactAnglePressure/*
```

Two additional variables need to be declared. `contactAnglePatch_` is of type `word` and defines the *FA-patch*, i.e., the edge, on which the contact angle should be applied. The variable `contactAngle_` stores the desired contact angle in degrees. Both variables are declared as protected member data in the file `freeSurfaceContactAnglePressureFvPatchScalarField.H`:

`freeSurfaceContactAnglePressureFvPatchScalarField.H`

```
89 class freeSurfaceContactAnglePressureFvPatchScalarField
90 :
91     public fixedValueFvPatchScalarField
92 {
93 protected:
94
95     // Protected data
96
97         //- Ambient pressure
98         scalarField pa_;
99
100        //- Name of fa boundary for contact angle condition
101        word contactAnglePatch_;
102
103        //- Desired contact angle
104        scalar contactAngle_;
```

The default values for the two variables are set in the first constructor in lines 48-49 of `freeSurfaceContactAnglePressureFvPatchScalarField.C`. The second constructor reads the boundary dictionary and checks if the `contactAnglePatch_` is valid. In the `updateCoeff` seen below, first, an object to the current `interfaceTrackingFvMesh` is created. This enables the use of its function inside this boundary condition. In line 174, the `ID` for the `contactAnglePatch_` is searched for in the *FA-mesh* and written to `patchI`. In line 176 it is checked if the `patchI` is valid. In lines 186-192, the face next to the `contactAnglePatch_` is returned. The `currentAngleDiff` of this face is then calculated. Next, the edge length `Le` is calculated using `delta`. The `pressureForce` is calculated according to Eq. 2.30 in lines 218-219, and multiplied by the surface tension and added to the `pressureJump` in lines 221-228 yielding Eq. 2.31.

`updateCoeffs()` in `freeSurfaceContactAnglePressureFvPatchScalarField.C`

```
156 void Foam::freeSurfaceContactAnglePressureFvPatchScalarField::updateCoeffs()
157 {
158     if (updated())
159     {
160         return;
161     }
162
163     const fvMesh& mesh = patch().boundaryMesh().mesh();
164
165     myInterfaceTrackingFvMesh& itm =
166         refCast<myInterfaceTrackingFvMesh>
167         (
168             const_cast<dynamicFvMesh&>
169             (
170                 mesh.lookupObject<dynamicFvMesh>("fvSolution")
171             )
172         );
173
174     const label& patchI = itm.aMesh().boundary().findPatchID(contactAnglePatch_);
175     // fvPatch patch = this->patch();
176     if (patchI==-1)
177     {
178         FatalErrorInFunction
```

```
            << "contactAnglePatch '" << contactAnglePatch_
            << "'in contactAnglePatch"
            << " in p." << this->patch().name()
            << "not found in faMesh"
            << abort(FatalError);
    }

    const labelList peFaces =
                labelList::subList
                (
                    itm.aMesh().edgeOwner(),
                    itm.aMesh().boundary()[patchI].faPatch::size(),
                    itm.aMesh().boundary()[patchI].start()
                );

    const scalar currentAngleDiff(90-contactAngle_-itm.faceAngles()[peFaces[0]]);
    Info << "Current faceAngle of faPatch '" << contactAnglePatch_
        << "' of fvPatch '" << this->patch().name()
        << "' = " << itm.faceAngles()[peFaces[0]]
        << endl;

    const pointField& points = itm.aMesh().patch().localPoints();
    const edgeList& edges = itm.aMesh().patch().edges();
    const labelList& pEdges = itm.aMesh().boundary()[patchI]; // the one edge
    vectorField peCentres(pEdges.size(), Zero);
    forAll(peCentres, edgeI)
    {
        peCentres[edgeI] =
            edges[pEdges[edgeI]].centre(points);
    }

    scalarField pressureJump = itm.freeSurfacePressureJump();
    vectorField delta
                (
                    itm.aMesh().areaCentres().internalField()[peFaces[0]]
                    - peCentres
                );
    scalarField Le(2*mag(delta));

    scalarField pressureForce(pressureJump.size(),
                2/Le[0]*sin(currentAngleDiff*constant::mathematical::pi/180));

    if (itm.pureFreeSurface())
    {
        pressureJump -= itm.sigma().value()*pressureForce;
    }
    else
    {
        pressureJump -= itm.surfaceTension().internalField()*pressureForce;
    }

    Info << "Current angle diff = " << currentAngleDiff
        << "; Current curvature correction = "
        << pressureForce[peFaces[0]]
        << "; Current gauge pressure = "
        << pressureJump[peFaces[0]]
        << endl;

    operator==
    (
        pa_ + pressureJump
    );

    fixedValueFvPatchScalarField::updateCoeffs();
}


void Foam::freeSurfaceContactAnglePressureFvPatchScalarField::write(Ostream& os) const
```

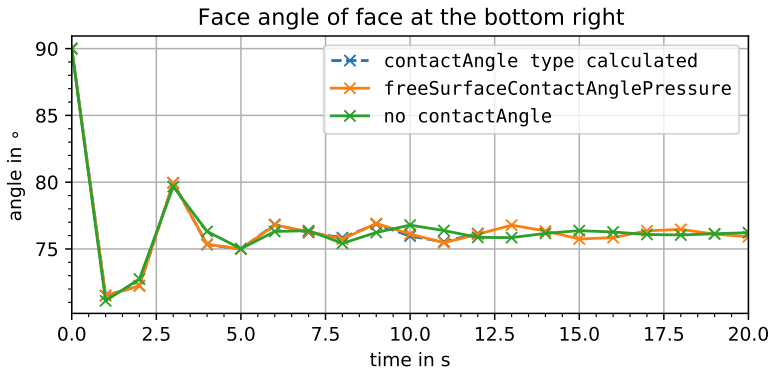
```
247 {
248     fvPatchScalarField::write(os);
249     pa_.writeEntry("pa", os);
250     os.writeEntry("contactAnglePatch_", contactAnglePatch_);
```

The `freeSurfaceContactAnglePressure` boundary condition can be used in the case by selecting it in the file `0.orig/p` and providing the `contactAnglePatch` and the desired `contactAngle`:

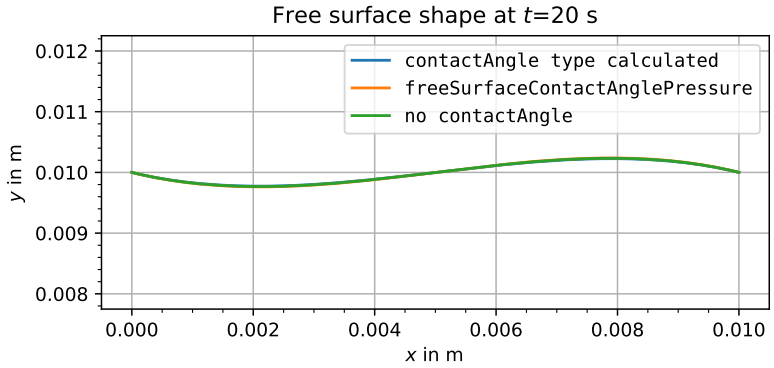
`0.orig/p`

```
23     top
24     {
25         type                    freeSurfaceContactAnglePressure;
26         pa                      uniform 0;
27         gaugePressurePatch      right;
28         contactAngle            79;
29     }
```

Figure 5.4 shows the face angle at the specified `gaugePressurePatch` in comparison with the default implementation of the `contactAngle` and without a specified `contactAngle`. The free surface is almost exactly the same and all contact angles converge to a value of around 76°. This hints at an issue with contact angle conditions in general, when gravity is defining the shape.



(a) Face angle of the leftmost face over time.



(b) Free surface shape

Figure 5.4: `contactAngleColumn` case with a desired contact angle of 79° and for the three boundary condition types: (1) `freeSurfaceContactAnglePressure` boundary condition for the pressure (see Section 5.6, (2) `type calculated` boundary condition for the `contactAngle`, i.e., with the default code, and (3) no `contactAngle` file specified.

Bibliography

[1] A. Muiznieks, J. Virbulis, A. Lüdge, H. Riemann, and N. Werner, "Floating Zone Growth of Silicon," in *Handbook of Crystal Growth*, pp. 241–279, Elsevier, 2015.

[2] V. N. Kurlov, S. N. Rossolenko, N. V. Abrosimov, and K. Lebbou, "Shaped Crystal Growth," in *Crystal Growth Processes Based on Capillarity* (T. Duffar, ed.), pp. 277–354, Chichester, UK: John Wiley & Sons, Ltd, Apr. 2010.

[3] X.-F. Han, X. Liu, S. Nakano, H. Harada, Y. Miyamura, and K. Kakimoto, "3D numerical study of the asymmetric phenomenon in 200 mm floating zone silicon crystal growth," *Journal of Crystal Growth*, vol. 532, p. 125403, Feb. 2020.

[4] P. Beckstein, "Methodenentwicklung zur Simulation von Strömungen mit freier Oberfläche unter dem Einfluss elektromagnetischer Wechselfelder," *Technische Universität Dresden*, 2018.

[5] Z. Tuković and H. Jasak, "A moving mesh finite volume interface tracking method for surface tension dominated interfacial fluid flow," *Computers & Fluids*, vol. 55, pp. 70–84, Feb. 2012.

[6] S. Muzaferija and M. Perić, "Computation of free-surface flows using the finite-volume-method and moving grids," *Numerical Heat Transfer, Part B: Fundamentals*, vol. 32, pp. 369–384, Dec. 1997.

[7] G. K. Batchelor, *An Introduction to Fluid Dynamics*. Cambridge Mathematical Library, Cambridge University Press, 2000.

[8] J. H. Ferziger and M. Perić, *Computational methods for fluid dynamics*. Berlin ; New York: Springer, 3rd, rev. ed ed., 2002.

[9] R. I. Issa, A. D. Gosman, and A. P. Watkins, "The computation of compressible and incompressible recirculating flows by a non-iterative implicit scheme," *Journal of Computational Physics*, vol. 62, pp. 66–82, Jan. 1986.

[10] G. Ratnieks, "Modelling of the Floating Zone Growth of Silicon Single Crystals with Diameter up to 8 Inch," *University of Latvia*, 2007.

[11] B. Eltard-Larsen, "How to make a dynamicMotionRefineFvMesh class," in *Proceedings of CFD with OpenSource Software* (H. Nilsson, ed.), 2016.

Study questions

1. What is an interface tracking technique?
2. How are the boundary conditions for a free surface derived?
3. What is the difference between the Finite-Area-Method and the Finite-Volume-Method?
4. What is the purpose of the Finite-Area-Method inside the `interfaceTrackingFvMesh` library?
5. What is the difference between `interfaceTrackingFvMesh` in OpenFOAM and `interTrackFoam` in foam-extend?

Appendix A

Member data

member data	type	description
aMeshPtr	autoPtr<faMesh>	Finite area mesh
fsPatchIndex	label	Free surface patch index
fixedFreeSurfacePatches	wordList	Free surface faPatches which do not move
nonReflectingFreeSurfacePatches	wordList	Free surface faPatches where wave should not reflect
pointNormalsCorrectionPatches	wordList	Free surface patches for which point normals must be corrected
normalMotionDir	Switch	`True`: free-surface points displacement direction is parallel with free-surface point normals; `False`: `motionDir` has to be specified
motionDir	vector	Free-surface points displacement direction if not normal motion direction
smoothing	Switch	Interface smoothing at the beginning of time step
pureFreeSurface	Switch	Pure free-surface
rigidFreeSurface	Switch	Rigid free-surface
correctContactLineNormals	Switch	Correct contact line point normals
sigma0	dimensionedScalar	Surface tension coefficient of pure free-surface
rho	dimensionedScalar	Fluid density
timeIndex	label	Current interface tracking time index
areaVectorField* UsPtr	mutable	Free-surface velocity field
controlPointsPtr	mutable vectorIOField*	Points which are attached to the free-surface A side faces and which defines the free-surface shape
motionPointsMaskPtr	mutable labelList*	Field which additionally determines the motion of free-surface points
pointsDisplacementDirPtr	mutable vectorField*	Displacement direction of free-surface points

facesDisplacementDirPtr	mutable vectorField*	Displacement direction of free-surface control points
fsNetPhiPtr	mutable areaScalarField*	Free-surface net flux
phisPtr	mutable edgeScalarField*	Free-surface flux
surfactConcPtr	mutable areaScalarField*	Free-surface surfactant concetration
bulkSurfactConcPtr	mutable volScalarField*	Volume surfactant concetration
surfaceTensionPtr	mutable areaScalarField*	Surface tension field
surfactantPtr	mutable surfactantProperties*	Surfactant properties
contactAnglePtr	mutable areaScalarField*	Contact angle

Table A.1: Private member data in `interfaceTrackingFvMesh class`