NTNU – Trondheim
Norwegian University of
Science and Technology

Development of an immersed boundary method for simulating contaminated fluid interfaces in two-phase flow

Morten Olsen Lysgaard

Master of Science in Physics and Mathematics
Submission date: June 2015
Supervisor: Helge Holden, MATH
Co-supervisor: Åsmund Ervik, EPT
Bernhard Müller, EPT
Svend Tollak Munkejord, SINTEF

Norwegian University of Science and Technology
Department of Mathematical Sciences

Abstract

This thesis covers the theory and implementation, verification and validation of an immersed boundary method for simulating the effect of an elastic membrane covering water drops in crude oil. First an introduction to incompressible two phase flow is given. Surface tension as well as elastic effects on the interface are also outlined. The discretization of the Navier-Stokes equations in space and time as well as the projection method are discussed. The penalization method for simulating obstacles in the computational domain is covered, as well as the level-set method for interface capturing, and the ghost-fluid method for handling the interface discontinuities. A thorough derivation of the immersed boundary method is done, and the details of the implementation is covered. A technique for coupling the immersed boundary method and the ghost fluid method is presented, as well as a final overview connecting together all the techniques used in the simulations into one coherent method.

Numerical evidence showing that the advection of the immersed boundary method is second order accurate in space is presented as well as numerical results showing that the immersed boundary method in some cases handles advection of the interface in a more accurate way than level-set. Numerical results comparing the immersed boundary method with the ghost-fluid method using the level-set method is presented, showing convergence of the two methods under grid refinement. Numerical comparison between the immersed boundary method and the ghost-fluid method using the level-set method for density and viscosity jumps is done showing consistent results. Simulations of drops with elastic membrane are performed, showing how the immersed boundary method enables the simulation of a larger set of physics than the previous method. The effect of the elastic membrane on a drop stretched in an electric field is simulated. Finally, crumpling of a drop interface is shown as mass is drained from

the drop, similar to results seen in lab experiments [45].

Sammendrag

Denne mastergraden omhandler teori, implementasjon, verifikasjon og validering av en immersed boundary metode for å simulere effekten av en elastisk membran som dekker vanndråper i råolje. Først gis en introduksjon til inkompressibel strømmning, overflatespenning og hvordan elastiske krefter påvirker overflaten. Så forklares diskertiseringen av Navier-Stokes likninger i rom og tid sammen med projeksjons metoder. Penalization metoden for å simulere solide obstruksjoner i domenet, level-set metoden for representasjon av grenseflaten og ghost-fluid metoden for å håndtere grenseflatediskontinuitetene blir dekket. En grundig gjennomgang av immersed boundary metoden blir gjort og implementasjonsdetaljer blir dekket. En teknikk for å koble sammen immersed boundary metoden og ghost-fluid metoden blir beskrevet. Til slutt gis en oversikt som kobler alle de forskjellige teknikkene sammen til en helhetlig metode.

Numerisk bevis som viser at adveksjon med immersed boundary metoden er andre orden i rom presenteres, i tillegg til resultater som viser at immersed boundary metoden noen ganger håndterer adveksjon på en mer korrekt måte enn level-set metoden. Sammenlikning av immersed boundary metode og ghost-fluid metoden med level-set metoden blir gjort. Disse viser at de to metodene konvergerer mot det samme resultatet etterhvert som griddet forfines. De to metodene blir også sammenliknet i simuleringer med tetthet og viskositetssprang hvor de viser seg at metodene gir konsistente resultater. Simuleringer av dråper med elastiske membraner blir gjort. Disse viser hvordan immersed boundary metoden tillater å simulere ny fysikk som ikke var tilgjengelig i de tidligere metodene. Effekten av den elastiske membranen på en dråpe i et eletrisk felt blir simulert. Til slutt gjøres et forsøk på å reprodusere effekten observert i [45]. Der viser labeksperimenter at dråper med elastiske membraner får ruglete overflater når man tømmer innholdet i de med en pipette.

Preface

This thesis is submitted to the Norwegian University of Science and Technology (NTNU) as part of my master's degree. It is the result of my work during the spring semester 2015, formally at the Department of Mathematical Sciences. The work has been carried out on the behalf of SINTEF Energy Research where it has been a part of the Compact Separation by Electrocoalescence project.

I would like to thank my supervisor, Mr. Åsmund Ervik, who gave me the opportunity to work on the project, always was open for my ideas, and who have been a true mentor for me. I want to thank Professor Helge Holden who have been my supervisor at the Department of Mathematical Sciences. I also want to thank my two co-supervisors, Dr. Svend Tollak Munkejord who helped me trough some of the intricacies of the numerical code and Professor Bernhard Müller for good advice in numerical methods for computational fluid dynamics. I would like to thank all my supervisors for meticulously proofreading my manuscript. Lastly I would like to thank my girlfriend for her love and support, and for helping me to forget work every once in a while.

Contents

Nomenclature

$\boldsymbol{\tau}$	Unit tangent vector	
$\boldsymbol{F}$	Lagrangian force density, section 4.2.3.	
$\boldsymbol{f}$	The Eulerian force field.	
$\boldsymbol{F}_i$	Force on Lagrangian point i, section 4.2.7	
$\boldsymbol{n}$	Unit normal vector	
$\boldsymbol{u}$	The Eulerian velocity field.	
$\boldsymbol{v}$	Perturbation of velocity field, section 4.2.3.	
$\boldsymbol{x}$	A point in the domain : $\boldsymbol{x} \in \Omega$.	
$\boldsymbol{X}(q,r,s,t)$	Position of Lagrangian point (q,r,s) at time t.	
Δ	The Eulerian grid spacing.	m
δ	Delta function.	
δ_Δ	Smooth approximation to the Dirac delta function.	
Γ	Parametrization of fluid interface.	
κ	Curvature of interface.	$1/\mathrm{m}$
μ	Dynamic viscosity of fluid.	$\mathrm{Pa} \cdot \mathrm{s}$
Ω	The spatial domain.	
$\partial\Omega$	The boundary of the spatial domain.	

ρ	Eulerian density field, section 4.2.3.	
ρ_i	Density of fluid i. section 3.1	
σ	Surface tension between fluids.	$\mathrm{N/m}$
$\wp$	Pertubation.	
A	Interface area, section 3.2.1.	
E	Energy functional, section 4.2.3.	
E	Potential energy in interface, section 3.2.1.	
K_a	Interface elasticity constant.	$\mathrm{N/m}$
M	Lagrangian mass, section 4.2.3.	
N	Dimension of linear system, section 4.3	
N	Number of grid points.	
p	The pressure field.	
S	Fiber tension functional, section 3.2.1.	
T	Interface tension.	
t	Time.	s

List of Figures

List of Tables

1. Introduction

Offshore oil production involves capturing the oil from deep subsea reservoirs. The oil recovered often contains a considerable fraction of seawater. The reservoirs are under high pressure, and to control the flow of oil to the surface, the pressure is reduced through a pressure reduction valve. This valve introduces a strong mixing of the oil and water. Before the oil can be sold as a product, the water content has to be reduced, normally down to about 1/2% in weight. The simplest way of separating the mixture is leaving it in a tank where the drops coalesce and gravity slowly deposits the water on the bottom until the two phases are sufficiently separated.

A model for a fluid drop falling trough another fluid is the Hadamard-Rybczynski relation. However, this relation has the requirement that the two fluids need to be perfectly clean. If either of the liquids have the slightest contamination a surface active layer will develop on the interface, the internal flow in the drop will stagnate. For this a better approximation of terminal velocity is that one of a solid sphere falling trough a viscous liquid [8, p. 35]. This is given by Stokes' law,

$$v_t = \frac{[\![\rho]\!] g 2r^2}{9\mu_2}. \tag{1.1}$$

Here fluid 1 is the drop, while fluid 2 is the oil. μ_2 is the viscosity of the surrounding fluid, $[\![\rho]\!] = \rho_2 - \rho_1$ is the difference in density between the two fluids. g is the gravitational acceleration and r is the drop radius. As we see the terminal velocity is dependent on r^2. For small drops this makes the terminal velocity very small, and almost no current is needed to keep the drops suspended in the oil. This makes the gravity coalescing process slow. Because of this, the tanks have to be large to handle the continuous stream of retrieved oil. To accelerate the process, electric fields can be applied to the oil-water mixture. This creates dipole moments

in the water drops which sets up forces pulling nearby drops together. Another technique used is adding demulsifiers to the oil which chemically enhances the separation rate. The oil industry is all about big scale, so naturally there are huge costs connected to this process. The issue of making the process more effective is not the only one. For example, if the operating conditions change, it may be challenging to control the separation equipment so as to maintain the specified maximum water concentration. This has lead to a considerable amount of research into the processes in an oil separator [24].

One such project is the "Fundamental understanding of electrocoalescence in heavy crude oils" project at SINTEF Energy Research. There the elementary physics of water drops in oil is studied both experimentally and numerically. The project goal is to gain fundamental knowledge about the physics of oil, water, electric fields, surfactants, crude oil components, and how they interact [41]. This should give a better understanding of the electrocoalescence process. One part of the project is about numerically simulating the small scale physics happening in the oil separator. In this part (together with other SINTEF Energy Research projects) a multiphase CFD code has been developed and is used to simulate the results from experimental research in the project. The code can simulate two-dimensional as well as axisymmetric two-phase flow. In [42] and [23] the code was used for simulations of clean water drops in oil under electric fields. Another use has been in the study of low emission LNG systems. Here the code was used for understanding of how LNG condensation happens in heat exchangers [22], [9] and [10]. This report focuses on the addition of a new interface tracking method to this code, the immersed boundary method, as well as the new physics it allows us to simulate.

2. Motivation

The dynamics of a fluid interface is governed by the Navier-Stokes equations including surface tension. As we will discuss in section 3.2.1, the forces from surface tension are only a function of the molecular composition of the two fluids as well as the shape of the current fluid interface. Experiments have shown that this is not enough to describe how water drops in crude oil behave. This stems from the very complex molecular composition of crude oil. A standard way of classifying a crude is a SARA [12] (Saturated, Aromatic, Resin and Asphaltene) analysis. It divides the different components of the crude into four groups based on their polarizability and polarity. The saturate part consists of nonpolar components such as branched, linear and cyclic saturated hydrocarbons, paraffins. Aromatic parts contain aromatic rings, which make them somewhat more polarizable. The two remaining classes have polar substituents. These are separated by that resins are miscible in heptane or pentane, while asphaltenes are not. SARA is only a very basic classification of crudes, and more advanced methods include mass spectrometry. In fact, the popularization of mass spectrometry is by some accredited to the demand for knowledge of the composition of crude oil[37]. Although highly complicated analysis methods have been developed one is still a far way from complete understanding of the composition of crude oil. Mass spectrometry analysis of a single crude sample has been able to identify $\approx$ 17000 different species[27], and this technique is only able to measure the polar species. Conservative estimates for the number of chemically distinct species in a crude oil are in the order of 50000[37]. The sheer number of different species and their diversity meant that up until 2008 there was still a dispute on the molecular weight of asphaltenes [17].

Because their composition is not well understood, modelling the interaction between crudes and water also is not well understood. What one does

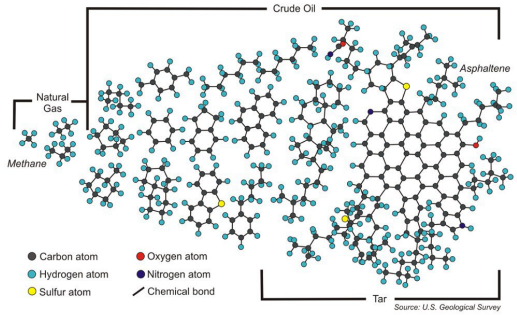


Figure 2.0.1.: Illustration of the molecular diversity present in crude oil. Source [44].

know, is that components of the crude are surface active and concentrate on the interface. In [31], experiments show how a membrane forms on the interface between a toluene drop with asphaltenes and water, and that this has an effect on the physics at the interface. We are interested in small water drops surrounded by crude oil, and how they behave under an electric field similarly to the electrocoalescence process used in offshore oil production. To understand this, one must be able to model, and simulate the membrane experienced in experiments. The goal of this master's thesis is the investigation into using the immersed boundary method for this purpose.

3. Governing equations

In this section we first talk about the single-phase Navier-Stokes equations, then these are expanded to two phase. Surface tension as well as elastic forces on the interface is discussed. Last, jump conditions are mentioned.

3.1. Navier-Stokes equations

Here a general introduction to the Navier-Stokes equations is given. For a more in depth view and an introduction to continuum mechanics in general, [4] is a good reference. First we introduce the single-phase, incompressible Navier-Stokes equations, then we extend this to two-phase flow.

Single-phase, viscous flow in the domain Ω is governed by the Cauchy equation,

$$\rho\left(\frac{\partial \boldsymbol{u}}{\partial t} + \boldsymbol{u}\cdot\nabla\boldsymbol{u}\right) = \nabla\cdot\boldsymbol{S} + \rho\boldsymbol{f}, \tag{3.1}$$

where ρ is the fluid density, $\boldsymbol{u}$ is the velocity vector, t is the time, $\boldsymbol{f}$ represents external acceleration and $\boldsymbol{S}$ denotes the stress tensor. The mass-conservation equation,

$$\frac{\partial \rho}{\partial t} + \nabla\cdot(\rho\boldsymbol{u}) = 0, \tag{3.2}$$

also needs to be satisfied. In this report, incompressibility is defined as

$$\rho = \text{const.} \tag{3.3}$$

Because of incompressibility, the mass-conservation equation reduces to

$$\nabla\cdot\boldsymbol{u} = 0. \tag{3.4}$$

In other words, the velocity field, $\boldsymbol{u}$, is divergence free and no mass is created or disappears at any time. If the fluid is Newtonian, the stress tensor is given by

$$\boldsymbol{S} = -p\boldsymbol{I} + 2\mu\boldsymbol{D} - \frac{2}{3}\mu(\text{trace}(\boldsymbol{D}))\boldsymbol{I}, \tag{3.5}$$

where μ is the dynamic viscosity, p the pressure and $\boldsymbol{D}$ is the rate of strain tensor,

$$\boldsymbol{D} = \frac{1}{2}\left(\nabla\boldsymbol{u} + (\nabla\boldsymbol{u})^T\right). \tag{3.6}$$

If we assume that viscosity is constant, the divergence of the stress tensor becomes

$$\nabla \cdot \boldsymbol{S} = -\nabla p + \mu\nabla^2\boldsymbol{u}. \tag{3.7}$$

Under the previous assumptions, the Navier-Stokes equations for incompressible, single-phase flow, with constant viscosity are,

$$\rho\left(\frac{\partial \boldsymbol{u}}{\partial t} + \boldsymbol{u}\cdot\nabla\boldsymbol{u}\right) = -\nabla p + \mu\nabla^2\boldsymbol{u} + \rho\boldsymbol{f}, \tag{3.8}$$

$$\nabla \cdot \boldsymbol{u} = 0, \tag{3.9}$$

$$\boldsymbol{u}(\boldsymbol{x}, 0) = \boldsymbol{u}_0(\boldsymbol{x}), \tag{3.10}$$

$$\boldsymbol{u}_{\partial\Omega}(t) = \boldsymbol{g}(t), \tag{3.11}$$

where $\partial\Omega$ is the domain boundary and $\boldsymbol{g}(t)$ is the velocity boundary condition.

We now want to extend the previous derivation to handle two fluid phases, with different viscosity and density. Let Ω_1 and Ω_2 denote the domains filled with fluid 1 and fluid 2, respectively and Γ denote the interface separating the two fluids. $\Omega = \Omega_1 \cup \Omega_2$.

On the interface, Γ, forces between the two fluids appear. These are discussed in section 3.2. Here, it is sufficient to conclude that the forces can be modeled as a contribution to eq. (3.8). This contribution is only present on the interface. If one moves in the normal direction to the interface, the contribution resembles a Dirac delta function.

$$\boldsymbol{f}_s(\boldsymbol{x}, t) = \int_\Gamma \left(\frac{\partial T(s)}{\partial s}\boldsymbol{\tau}(s) + T(s)\kappa(s)\boldsymbol{n}(s)\right)\delta(\boldsymbol{x} - \Gamma(s))\mathrm{d}\boldsymbol{s}, \tag{3.12}$$

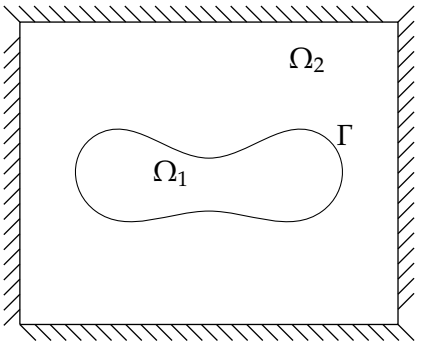


Figure 3.1.1.: Two phases of fluid and their interface Γ.

where T is the tension of the interface, $\frac{\partial T}{\partial s}$ is the derivative of tension along the interface, $\boldsymbol{\tau}$ is the unit interface tangent, κ is the local curvature of Γ, $\boldsymbol{n}$ is the principal unit normal vector, δ is the Dirac delta function, and $\boldsymbol{x}_I(s)$ is a parametrization of the interface. Adding this to eq. (3.8) gives

$$\rho\left(\frac{\partial \boldsymbol{u}}{\partial t} + \boldsymbol{u}\cdot\nabla\boldsymbol{u}\right) = -\nabla p + \mu\nabla^2\boldsymbol{u} + \rho\boldsymbol{f} + \boldsymbol{f}_s. \tag{3.13}$$

3.2. Interface forces

On the interface between the two clean fluids, forces appear. These stem from several phenomena on the molecular scale, which result in surface tension. As we have indicated in chapter 2, classical surface tension is not enough to describe the systems of interest for this report. To clear any ambiguity, in this thesis, surface tension refers to the phenomenon experienced in day to day life, e.g. the force that makes a raindrop shape like a drop. When the term interface tension or interface force is used, it is referring to a wider class of physics, including but not limited to surface tension. The elastic forces which are studied in this report are interface tension forces. Last, the term surface tension is not limited to a fluid-gas interface, e.g. the surface of a lake, but also includes fluid-fluid interfaces, like olive oil in water. The interface has a certain potential

energy. This energy stems from a tension, T, in the interface. By the principle of minimum potential energy, the interface will deform and displace until the potential energy reaches a minimum. Naturally, the form of T has a big impact on this process. This section first discusses interface tension for clean fluids and then extends this to a more general model for the systems studied in this report.

3.2.1. Surface tension

For a clean fluid interface, the interface tension is the change in potential energy, as a function of the change in interface area [21],

$$\sigma = \frac{\partial E}{\partial A}. \tag{3.14}$$

Interestingly, for a given temperature and pressure, this property is constant for all macroscopic interfaces between two clean fluids [1]. This stems from the fact that the interface tension is nothing but the sum of the inter-molecular forces acting on the two different types of fluid-molecules on the interface. If the area of the interface increases, the number of molecules on the interface will increase proportionally. Since the macroscopic interface tension is the sum of the inter-molecular forces, and the inter-molecular forces are constant per molecule, the sum of molecular strain, or potential energy, must also be proportional to the interface area. This leads to

$$E \propto A \Rightarrow \frac{\partial E}{\partial A} = \text{const} = \sigma. \tag{3.15}$$

Following [21, p. 250], we consider two fluids, separated by an arbitrary interface and an infinitesimally displaced version of this interface, as

[1] The underlying assumption for this is that on a molecular scale, the ratio of the range of inter-molecular forces and the local radius of the curvature is much smaller than 1. Or $r_m\kappa \ll 1$ where κ is the curvature. For a liquid water/water vapour interface a conservative cutoff for molecule interactions is 20 Ångström [26], or 2nm. The smallest drop relevant for the electrocoalescence process is around $r_m = 20\mu$m. To be on the safe side we assume drop radius $r_d = 1\mu$m which gives $\kappa = 1 \times 10^6$ and $r_m\kappa = 2 \times 10^{-9} \times 10^6 = 2 \times 10^{-3} \ll 1$. This assumption should thus hold for all situations related to electrocoalescence.

in fig. 3.2.1. On each point of the original interface, draw the interface normal towards the displaced interface. The length of this normal is denoted by $|\delta\zeta|$. The original interface element is denoted by $\mathrm{d}f$. Then the displaced volume for a interface element is $|\delta\zeta|\mathrm{d}f$. Now, let p_1 and p_2 be the pressures on either side of the interface. The work needed to change the volume is

$$\int(-p_1+p_2)|\delta\zeta|\mathrm{d}f. \tag{3.16}$$

In addition the interface has been stretched or compressed. This work is proportional to the infinitesimal stretching of each segment, if the infinitesimal stretching is written δf the total work must be

$$\delta W=\int(-p_1+p_2)|\delta\zeta|\mathrm{d}f+\sigma\delta f. \tag{3.17}$$

Where σ is the proportionality constant for stretching, similar to a spring constant in Hooke's law. From mechanical equilibrium, we have that $\delta W=0$.

Now, let R_1 and R_2 be the principal radii of curvature of two given points on the interface. If a radius is drawn into fluid 1, we consider it positive. Let $\mathrm{d}l_1$ and $\mathrm{d}l_2$ be the lengths of the two interface segments associated with the two radii. When the infinitesimal displacement is done, the change in length will be $\frac{|\delta\zeta|}{R_1}\mathrm{d}l_1$ and $\frac{|\delta\zeta|}{R_2}\mathrm{d}l_2$. By this, the area of the interface element $\mathrm{d}f=\mathrm{d}l_1\mathrm{d}l_2$ after displacement becomes $(1+\frac{|\delta\zeta|}{R_1})\mathrm{d}l_1(1+\frac{|\delta\zeta|}{R_2})\mathrm{d}l_2=\mathrm{d}l_1\mathrm{d}l_2(1+\frac{|\delta\zeta|}{R_1}+\frac{|\delta\zeta|}{R_2})$[2]. The change is $|\delta\zeta|\mathrm{d}f(\frac{1}{R_1}+\frac{1}{R_2})$ With this expression for the infinitesimal stretching of a interface element, the whole interface area will change under the displacement as

$$\delta f=\int|\delta\zeta|\left(\frac{1}{R_1}+\frac{1}{R_2}\right)\mathrm{d}f. \tag{3.18}$$

Inserting this into eq. (3.17) we get

$$\delta W=\int|\delta\zeta|\left((p_1-p_2)-\sigma\left(\frac{1}{R_1}+\frac{1}{R_2}\right)\right)\mathrm{d}f=0. \tag{3.19}$$

[2]We are working to first order in the displacements $|\delta\zeta|$ and δf.

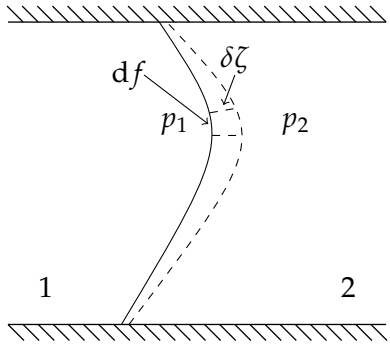


Figure 3.2.1.: Two fluids and their interface. The infinitesimally displaced interface is dashed. The dashed normals, $\delta\zeta$ are drawn and together with the two interfaces they form in infinitesimal displaced volume. The surface between the two interface normals, $\mathrm{d}f$, is the infinitesimal interface segment. The displaced segment, which is dashed, is to the right.

Since the choice of $\delta\zeta$ was arbitrary, it must hold for all $\delta\zeta$, this makes the whole integrand above equal to zero and we can rewrite

$$p_1 - p_2 = \sigma\left(\frac{1}{R_1} + \frac{1}{R_2}\right). \tag{3.20}$$

Equation (3.20) is known as Laplace's equation. It connects the jump in pressure to the curvature, and the interface tension, σ. It is more familiarly written

$$[\![p]\!] = \sigma\kappa, \tag{3.21}$$

where σ is the interface tension coefficient, and κ is the interface curvature and $[\![a]\!]$ is a general jump notation meaning the change in a as one moves from one side to the other of the interface. Here, the pressure is discontinuous over the interface, and the jump, is $p_1 - p_2 = [\![p]\!]$.

3.2.2. General tension

As previously mentioned, the most general form of interface tension is an arbitrary tension, T, may depend on various properties like interface geometry, temperature, electric fields, stretching, bending, molecular properties and so on.

To derive the effect of this tension on the liquids, we utilize the immersed boundary formulation [34]. We model the interface as a continuum of elastic fibers, immersed in the fluid. These fibers serve as a device for deriving the model. They do not have a mass nor a volume, but together with the fluid they are immersed in, they act as a viscoelastic material. The fibers are arranged in a structured mesh parametrized by three space coordinates, (q, r, s). With this framework, fixing two of the space coordinates, e.g. (q, r), uniquely determines a fiber. The last coordinate, s, is then a parametrization along the elastic fiber (q, r).

Following [34], we describe the motion of the material by

$$\boldsymbol{x} = \boldsymbol{X}(q, r, s, t). \tag{3.22}$$

If we consider an arbitrary fiber, (q, r),

$$\boldsymbol{\tau} = \frac{\frac{\partial \boldsymbol{X}}{\partial s}}{\left|\frac{\partial \boldsymbol{X}}{\partial s}\right|} \tag{3.23}$$

is the unit tangent along the fiber. The fiber tension is kept as general as possible,

$$T = S\left(\frac{\partial \boldsymbol{X}}{\partial s}; q, r, s, t\right). \tag{3.24}$$

Where $\frac{\partial \boldsymbol{X}}{\partial s}$ is the local fiber strain. Here the functional S is allowed to depend directly on q, r, s and t. This is important, as it lets the stress be completely general, and the dependence of other parameters, like temperature, can be added easily.

Now consider an infinitesimal bundle of elastic fibers of width $\mathrm{d}q$ and height $\mathrm{d}r$. The force transmitted by the bundle is $T\mathrm{d}q\mathrm{d}r$. Let B be an arbitrary part of the (q, r) plane, this is a bundle of fibers. We now only

look at the segment of the fibers that are between $s_1 < s < s_2$. As previously mentioned, the fibers are massless. From Newton's second law, this implies that the net force on the fibers must be zero, as they are in equilibrium. The only forces acting on the fibers are the forces from the surrounding fluid, as well as the force transmitted through the segment endpoints, s_1 and s_2. With $\boldsymbol{\tau}$ as the fiber tangent, this gives us:

$$\begin{aligned} 0 &= \text{Forces on fiber bundle segment} \\ &= \text{Force from fluid on fiber bundle segment} + \int_B (T\boldsymbol{\tau})\big|_{s_1}^{s_2} \mathrm{d}q\mathrm{d}r \end{aligned} \tag{3.25}$$

Now, an interesting trick can be used, applying Newton's third law to the force from the fluid we get

$$0 = -\text{Force from } \textit{fiber} \text{ bundle segment on } \textit{fluid} + \int_B (T\boldsymbol{\tau})\big|_{s_1}^{s_2} \mathrm{d}q\mathrm{d}r. \tag{3.26}$$

Rearrange and using the fundamental theorem of calculus on the integral gives

$$\text{Force from fiber bundle segment on fluid} = \int_{s_1}^{s_2} \int_B \frac{\partial (T\boldsymbol{\tau})}{\partial s} \mathrm{d}q\mathrm{d}r\mathrm{d}s. \tag{3.27}$$

The choice of B, s_1 and s_2 was arbitrary. This implies that the force density of the fibers acting on the fluids is

$$\boldsymbol{f} = \frac{\partial T\boldsymbol{\tau}}{\partial s}. \tag{3.28}$$

To get the force from the fibers on the fluid one simply integrates this, as in eq. (3.27). By using the product rule we can expand the derivative to

$$\boldsymbol{f} = \frac{\partial T}{\partial s}\boldsymbol{\tau} + T\frac{\partial \boldsymbol{\tau}}{\partial s}. \tag{3.29}$$

Since curvature is given by $\kappa = \left|\frac{\partial \boldsymbol{\tau}}{\partial s}\right| / \left|\frac{\partial \boldsymbol{X}}{\partial s}\right|$, and the interface normal by $\boldsymbol{n} = \left(\frac{\partial \boldsymbol{\tau}}{\partial s}\right) / \left|\frac{\partial \boldsymbol{\tau}}{\partial s}\right|$, eq. (3.29) can be written:

$$\boldsymbol{f} = \frac{\partial T}{\partial s}\boldsymbol{\tau} + T\left|\frac{\partial \boldsymbol{X}}{\partial s}\right| \kappa \boldsymbol{n}. \tag{3.30}$$

From eq. (3.30) we see that the force is consisting of a component along the fiber in the direction of $\boldsymbol{\tau}$, as well as a component in the principal normal direction, pointing towards the centre of the osculating circle[3] of the curve, $\boldsymbol{n}$. Similar to macroscopic experience with elastic materials, like a rubber band, there is no force in the binormal direction, $\boldsymbol{\tau} \times \boldsymbol{n}$. Note however, that if one considers another fiber bundle, say fixing (q, s) and varying r, this bundle can act with a force in this direction. When calculating the effect of the elastic fibres on the fluid, one has to take into account all fibres present at a material point, (q, r, s). This way forces in all directions can be generated by linear combination of forces from different fibres.

Interestingly, we see that if we assume constant interface tension,

$$T = \sigma = \text{const}, \tag{3.31}$$

eq. (3.30) becomes

$$\begin{aligned} \boldsymbol{f} &= \frac{\partial \sigma}{\partial s}\boldsymbol{\tau} + \sigma \left|\frac{\partial \boldsymbol{X}}{\partial s}\right| \kappa \boldsymbol{n} \\ &= \sigma \left|\frac{\partial \boldsymbol{X}}{\partial s}\right| \kappa \boldsymbol{n}, \end{aligned}$$

and the tangential force disappears.

At this point we can introduce the model used for the elastic membrane. In this thesis, a simple Hookean material was used for simulation

$$T = K_a \left(\left|\frac{\partial \boldsymbol{X}}{\partial s}\right| - 1 \right) + \sigma. \tag{3.32}$$

Here $K_a (\mathrm{N/m})$ is the force density equivalent of a spring constant. $\frac{\partial \boldsymbol{X}}{\partial s}$ is the relative stretching of an infinitesimal interface element. As detailed in chapter 2, there is not a full understanding of the interface between crude oil and water. Taking this into account, by application of Occam's razor, a Hookean law is the most logical choice. In section 6.2 this decision and its limitations is discussed further. In the code, the tension is discretized in a straightforward way and calculated for the points on the immersed boundary, then eq. (3.30) is used to calculate the force on each Lagrangian point. This is elaborated in section 4.2.8.

[3] http://mathworld.wolfram.com/OsculatingCircle.html

3.3. Jump conditions

As well as forces on the interface itself, the jump in viscosity and density across the interface changes the physics on each side of the domain. For example, because crude oil is highly viscous, a drop of water oscillating in crude oil will have a more damped oscillation than one in e.g. air. The constraints coming from these jumps are expressed as jump conditions. In [18] these are derived for two-phase flow. In [15] and [5] the effect of an electric field is added. In [23] Marangoni stresses coming from a varying interface tension is added. All together the conditions are

$$\llbracket \boldsymbol{u} \rrbracket = 0, \tag{3.33}$$

$$\llbracket p \rrbracket = 2\llbracket \mu \rrbracket \boldsymbol{n} \cdot \boldsymbol{\nabla u} \cdot \boldsymbol{n} + \boldsymbol{n} \cdot \llbracket \boldsymbol{M} \rrbracket \cdot \boldsymbol{n} + \sigma\kappa, \tag{3.34}$$

$$\llbracket \Psi \rrbracket = 0, \tag{3.35}$$

$$\begin{aligned} \llbracket \mu \boldsymbol{\nabla u} \rrbracket = \llbracket \mu \rrbracket & ((\boldsymbol{n} \cdot \boldsymbol{\nabla u} \cdot \boldsymbol{n})\boldsymbol{nn} + (\boldsymbol{n} \cdot \boldsymbol{\nabla u} \cdot \boldsymbol{t})\boldsymbol{nt} \\ & - (\boldsymbol{n} \cdot \boldsymbol{\nabla u} \cdot \boldsymbol{t})\boldsymbol{tn} + (\boldsymbol{t} \cdot \boldsymbol{\nabla u} \cdot \boldsymbol{t})\boldsymbol{tt}) \\ & - (\boldsymbol{t} \cdot \llbracket \boldsymbol{M} \rrbracket \cdot \boldsymbol{n})\boldsymbol{tn} - (\boldsymbol{t} \cdot \boldsymbol{\nabla}_\Gamma \sigma)\boldsymbol{tn}, \end{aligned} \tag{3.36}$$

where $\boldsymbol{t}$ is the unit tangent of the interface, $\boldsymbol{n}$ is the unit normal of the interface and $\boldsymbol{M}$ is the Maxwell stress tensor.

4. Numerical methods

In this section the discretization of the Navier-Stokes equations will be discussed. How to approximate the spatial derivatives, Chorin's projection method, and a brief overview of Runge-Kutta methods is given. The penalization method for simulating solid objects in the domain is discussed. The level-set method for interface capturing and the ghost-fluid method for handling the viscosity and density jumps on the interface is covered. A thorough introduction to the immersed boundary method is given and details regarding implementation are discussed. This includes cubic splines for parameterizing the interface as well as a routine for computing the level-set function based on the immersed boundary. Lastly an overview connecting all pieces together into one coherent method is given.

4.1. Discretization of the Navier-Stokes equations

This section discusses the discretization of the Navier-Stokes equations for single-phase, incompressible flow. Problems arising with a checkerboard pressure field is studied, how this relates to the null space of the discrete gradient operator, and how to correct this using a staggered grid. An overview of the spatial discretization of the Navier-Stokes equations and the differential operators needed on an orthogonal, rectilinear grid is given. General projection methods are considered and specifically Chorin's pressure projection method is derived. A short overview of Runge-Kutta methods for time integration is given and the penalization method for simulating solid objects in the domain is covered. Lastly the level-set method for interface capturing and the ghost-fluid method for handling the viscosity and density jumps are discussed.

4.1.1. Spatial Discretization

The discretization of the Navier-Stokes equations is not trivial, even for a structured grid using finite differences. Naively one would discretize eqs. (3.8) to (3.11) in space, store velocities, $\boldsymbol{u}$, pressure, p, and body forces $\boldsymbol{f}$ at each grid node and use a finite-difference stencil on this grid to approximate the needed differential operators. Historically, the first attempts at solving the Navier-Stokes equations proceeded using this approach, but problems quickly arose with the pressure oscillating out of control.

For the naive discretization, the finite-difference approximation to the derivative of the pressure would be determined by

$$\frac{\partial p_i}{\partial x} \approx \frac{p_{i+1} - p_{i-1}}{2\Delta_x}$$

where p_i denotes the pressure in grid node i and Δ_x is the grid cell distance along axis x. Given an oscillatory pressure field, e.g. $p_i = 1$, $p_{i+1} = -1$, $p_{i+2} = 1 \ldots$ the approximated derivative will be zero. In words, a checkerboard pressure is a part of the null-space of the finite-difference approximation of the derivative of pressure. One way of viewing this problem is that if you color your grid in a chess board pattern, the pressure gradients at black nodes will only depend on the pressures at black nodes, similarly the pressure gradient on white nodes, only depend on the pressures at the black nodes. To remedy this, a coupling between velocity components and adjacent pressures needs to be added. This can be done via grid staggering.

For the Navier-Stokes equations, staggering is normally done by storing scalar quantities, e.g. pressure, at the center of each grid cell while vector components are stored at the faces of each grid cell, cf. fig. 4.1.1. Using this staggering couples the pressure at p_i with p_{i+1}, p_{i-1}[1].

[1] Alternative solutions include Rhie-Chow interpolation [36], where all grid variables are collocated. The Rhie-Chow interpolations uses $\frac{\partial^4 p}{\partial x^4}$ to cancel the pressure oscillations, which has a stabilizing effect.

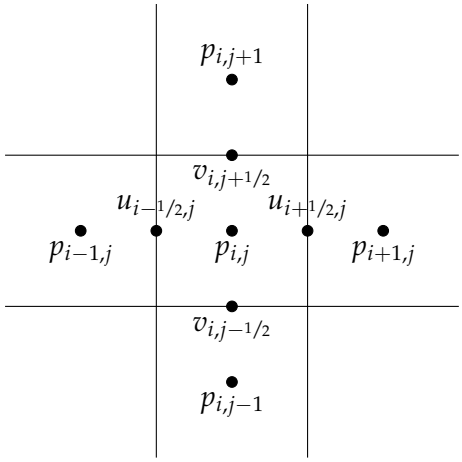


Figure 4.1.1.: A staggered grid cell. The pressure is located at the center of each cell, the x-velocities at the eastern and western, and y-velocities at the northern and southern faces. Here u and v denote the velocity in the x and y direction, respectively.

Finite difference approximations

The continuous differential operators in eqs. (3.8) to (3.11) have to be discretized in order to be computed in the simulation. During this discretization it is important to preserve a high enough order of accuracy such that the method as a whole has the expected order of accuracy. Spatially, the numerical code used in this work is second order, except locally at phase interfaces. Here the method is first order, this is due to the ghost-fluid method[29, sec. 15.8] explained in section 4.1.6. In time, the code is limited by the first order splitting error introduced by Chorin's projection method for the pressure discussed in the next section.

The gradient of a scalar field, g, is approximated with the second order finite difference scheme,

$$[(\nabla g)_{i,j}]_x \approx \frac{g_{i+\frac{1}{2},j} - g_{i-\frac{1}{2},j}}{\Delta_x}, \tag{4.1}$$

$$[(\nabla g)_{i,j}]_y \approx \frac{g_{i,j+\frac{1}{2}} - g_{i,j-\frac{1}{2}}}{\Delta_y}, \tag{4.2}$$

where $g_{i,j}$ is the value of the field g at grid node (i,j) and $[\boldsymbol{v}_{i,j}]_x$ is the value of field $\boldsymbol{v}$ at coordinate (i,j) in x direction. For y the same applies.

The Laplacian of a two-dimensional vector field is calculated with the second order finite difference approximation

$$(\nabla^2 \boldsymbol{f})_{i,j} \approx \frac{\boldsymbol{f}_{i+1,j} - 2\boldsymbol{f}_{i,j} + \boldsymbol{f}_{i-1,j}}{\Delta_x^2} + \frac{\boldsymbol{f}_{i,j+1} - 2\boldsymbol{f}_{i,j} + \boldsymbol{f}_{i,j-1}}{\Delta_y^2}. \tag{4.3}$$

To calculate the divergence of a 2D vector field, the second order finite difference approximation is used,

$$\nabla \cdot \boldsymbol{f}_{i,j} \approx \frac{\left[\boldsymbol{f}_{i+\frac{1}{2},j} - \boldsymbol{f}_{i-\frac{1}{2},j}\right]_x}{\Delta_x} + \frac{\left[\boldsymbol{f}_{i+\frac{1}{2},j} - \boldsymbol{f}_{i-\frac{1}{2},j}\right]_y}{\Delta_y}. \tag{4.4}$$

4.1.2. Chorin's projection method

A general projection method relies on the Helmholtz-Hodge theorem. A proof of this theorem can be found in [7, ch 1.3]. Here the theorem will

only be stated.

Theorem 4.1.1 (Helmholtz-Hodge theorem)**.** *An arbitrary vector field $\boldsymbol{w}$ can be uniquely decomposed in a bounded domain Ω such that*

$$\boldsymbol{w} = \boldsymbol{u} + \nabla g, \tag{4.5}$$
$$\boldsymbol{u} \cdot \boldsymbol{n} = 0 \text{ on } \partial\Omega, \tag{4.6}$$

where g is a scalar function, $\boldsymbol{n}$ is the outwards unit normal vector on $\partial\Omega$ and $\boldsymbol{u}$ is a divergence free vector field,

$$\nabla \cdot \boldsymbol{u} = 0 \text{ in } \Omega. \tag{4.7}$$

The Helmholtz-Hodge theorem asserts the existence and uniqueness of an orthogonal projection operator $\boldsymbol{P}$. This operator maps an arbitrary vector field into a divergence-free vector field,

$$\boldsymbol{u} = \boldsymbol{P}\boldsymbol{w} \tag{4.8}$$
$$= \boldsymbol{w} - \nabla g. \tag{4.9}$$

For further theory on projection operators, [38, sec. 1.12] is a good reference. A projection is defined by the property

$$\boldsymbol{P^2} = \boldsymbol{P}, \tag{4.10}$$

thus, applying $\boldsymbol{P}$ to eq. (4.8) and using eq. (4.10), we get

$$\boldsymbol{P}\boldsymbol{u} = \boldsymbol{P}^2\boldsymbol{w} \tag{4.11}$$
$$= \boldsymbol{P}\boldsymbol{w} \tag{4.12}$$
$$= \boldsymbol{u}.$$

Expanding eq. (4.11) using eq. (4.9) instead gives

$$\begin{aligned}\boldsymbol{P}\boldsymbol{u} &= \boldsymbol{P}^2\boldsymbol{w}, \\ &= \boldsymbol{P}(\boldsymbol{w} - \nabla g), \\ &= \boldsymbol{P}\boldsymbol{w} - \boldsymbol{P}(\nabla g).\end{aligned} \tag{4.13}$$

From eq. (4.12) we know that $\boldsymbol{P}\boldsymbol{u} = \boldsymbol{P}\boldsymbol{w}$, which implies that

$$\boldsymbol{P}(\nabla g) = 0. \tag{4.14}$$

To find the projection operator we use eq. (4.7)

$$\nabla \cdot \boldsymbol{u} = 0, \tag{4.15}$$
$$\nabla \cdot \boldsymbol{P}\boldsymbol{w} = 0, \tag{4.16}$$

by applying $(\nabla\cdot)$ to eq. (4.9) we get

$$\nabla \cdot \boldsymbol{P}\boldsymbol{w} = \nabla \cdot \boldsymbol{w} - \nabla^2 g, \tag{4.17}$$
$$0 = \nabla \cdot \boldsymbol{w} - \nabla^2 g, \tag{4.18}$$
$$\nabla^2 g = \nabla \cdot \boldsymbol{w}, \tag{4.19}$$

which is a Poisson equation for g. Using that $\boldsymbol{u} \cdot \boldsymbol{n} = 0$ on $\partial\Omega$ we can derive Neumann boundary conditions for the Poisson equation

$$\begin{aligned} \boldsymbol{u} \cdot \boldsymbol{n} &= 0, \\ &= (\boldsymbol{w} - \nabla g) \cdot \boldsymbol{n}, \\ &= \boldsymbol{w} \cdot \boldsymbol{n} - \nabla g \cdot \boldsymbol{n}, \qquad (4.20) \\ &\Downarrow \\ \boldsymbol{w} \cdot \boldsymbol{n} &= \nabla g \cdot \boldsymbol{n}. \qquad (4.21) \end{aligned}$$

To use the above results, we assume that $\boldsymbol{u}$ is smooth, and that it has a continuous first derivative. Equation (3.8) may be written in the form of eq. (4.5) and eq. (4.6) by using

$$g = p, \tag{4.22}$$
$$\boldsymbol{w} = \boldsymbol{a}, \tag{4.23}$$
$$\rho \frac{\partial \boldsymbol{u}}{\partial t} = \boldsymbol{a}(\boldsymbol{u}) - \nabla p, \tag{4.24}$$
$$\boldsymbol{a}(\boldsymbol{u}) = -\rho(\boldsymbol{u} \cdot \nabla \boldsymbol{u}) + \mu \nabla^2 \boldsymbol{u} + \boldsymbol{f}. \tag{4.25}$$

To continue the derivation of Chorin's projection method discretizing in time is also required. The code uses first order, forward Euler steps composed together in a higher order Runge-Kutta time step, see section 4.1.3.

The implementation is based on the routines outlined in [29]. Because of this, we only need to consider a first order Euler step. Let $\boldsymbol{u}^n$ be the velocity field at time $t = \Delta_t n$ and $\boldsymbol{u}^{n+1}$ at $t = \Delta_t(n+1)$, where Δ_t denotes the time step size. Equation (4.24) can be written in an semi-implicit way

$$\rho \frac{\boldsymbol{u}^{n+1} - \boldsymbol{u}^n}{\Delta_t} = \boldsymbol{a}(\boldsymbol{u}^n) - \nabla p^{n+1}. \tag{4.26}$$

recognizing $\rho \frac{\boldsymbol{u}^{n+1} - \boldsymbol{u}^n}{\Delta_t}$ as the divergence-free field and $\boldsymbol{a}(\boldsymbol{u})$ as our arbitrary vector field. Solving eq. (4.19) with Neumann boundary conditions, we can find p^{n+1} given $\boldsymbol{a}(\boldsymbol{u}^n)$. Next we can compute $\boldsymbol{u}^{n+1}$ using eq. (4.26).

The main advantage of this method is that the effect of pressure is separated from the effect of advection and viscosity. In the incompressible Navier-Stokes equations one assumes that the speed of sound is infinite, $c \to \infty$. This means that a local change in pressure affects the whole domain instantaneously. The advection and viscosity, on the other hand, are local phenomena and a local perturbation will only have a finite radius of effect a time, t, after the perturbation. The all-to-all nature of the pressure in the incompressible Navier-Stokes equations, makes it a computationally expensive problem to solve. A pure Poisson problem such as the one in Chorin's projection method is in many ways the simplest possible elliptic problem, and large amounts of research have been invested into efficient solvers. If one introduces the effects of advection and viscosity into the elliptic problem, which is perfectly legal, the computational effort needed to solve the system would increase significantly and one could no longer take advantage of the large amount of applied research on Poisson solvers.

In [40], a rigorous error analysis is performed and it is shown that if $\boldsymbol{u}$ and p solving eqs. (3.8) to (3.11) are sufficiently smooth, and the domain Ω satisfies a regularity condition, the error

$$\|\boldsymbol{u}' - \boldsymbol{u}\| = a(\boldsymbol{u}, p, T)\Delta_t = \mathcal{O}(\Delta_t), \tag{4.27}$$

is a constant a, independent of the time-step, Δ_t, such that the method is first order accurate. Here $\boldsymbol{u}'$ is the exact solution while $\boldsymbol{u}$ is the solution using Chorin's projection method, p is the pressure and T is the end time.

4.1.3. Time integration method

Time-integrating the Navier-Stokes equations, and more generally a system of equations, is a large field of research. In this work, a Runge-Kutta method is used. Generally we want to solve the problem: Given

$$\frac{\mathrm{d}\boldsymbol{u}(t)}{\mathrm{d}t} = \boldsymbol{f}(\boldsymbol{u}(t)), \tag{4.28}$$

$$\boldsymbol{u}(0) = \boldsymbol{u}_0, \tag{4.29}$$

find $\boldsymbol{u}(t)$. Runge-Kutta methods approximate a solution to this problem by evaluating $f(\cdot)$ at carefully chosen points in time and then composing these in such a way that their errors cancel out up to a certain order. In the code we use a special class of Runge-Kutta methods known as Strong Stability-Preserving. It has the property that if the spatial discretization is Total Variation Diminishing using forward Euler time integration, then the method will also be TVD when integrated using a SSP Runge-Kutta method. [20] is a good reference on strong stability-preserving Runge-Kutta methods.

4.1.4. Penalization method

As will be seen in section 5.5.3, we want to be able to simulate solid objects in our domain. To achieve this we utilize the penalization method. Following [2] the penalization method can be derived as follows. Let our domain contain n regular obstacles, $\Omega_y^s,\ i \in \{1 \dots n\}$, and let $\Omega^s = \cup_{i=1}^n \Omega_i^s$ be the obstructed, solid, part of the domain. Let $\Omega^f = \Omega \setminus \bar{\Omega}^s$ be the unobstructed, free, part of the domain. Lastly let $\Sigma_i^s = \partial\Omega_i^s$ be the boundary of each obstacle.

Our goal is now to solve eqs. (3.8) to (3.11) in Ω^f subject to the boundary conditions from the boundary of the domain, Γ, and from the boundaries of the internal solid parts Σ_i^s. This is not straightforward, as changing Ω to Ω^f changes the topology of the domain adding holes. Instead we try to modify the problem, solving it for the whole domain Ω but add a penalty for flow trough obstacles.

We modify the velocity and pressure in the following way

$$\boldsymbol{u}_\eta = \boldsymbol{u} + \eta\tilde{\boldsymbol{u}}, \tag{4.30}$$
$$p_\eta = p + \eta\tilde{p}. \tag{4.31}$$

That is, we allow a perturbation of the velocity field, $\tilde{\boldsymbol{u}}$, proportional to a parameter η. Then we solve a modified version of eqs. (3.8) to (3.11)

$$\rho\left(\frac{\partial \boldsymbol{u}_\eta}{\partial t} + \boldsymbol{u}_\eta \cdot \nabla \boldsymbol{u}_\eta\right) = -\nabla p_\eta + \mu\nabla^2\boldsymbol{u}_\eta - \frac{1}{\eta}\chi\boldsymbol{u}_\eta + \rho\boldsymbol{f}, \tag{4.32}$$
$$\nabla \cdot \boldsymbol{u}_\eta = 0, \tag{4.33}$$
$$\boldsymbol{u}_\eta(\boldsymbol{x}, 0) = \boldsymbol{u}_0(\boldsymbol{x}), \tag{4.34}$$
$$\boldsymbol{u}_{\eta\partial\Omega}(t) = \boldsymbol{g}(t), \tag{4.35}$$

where χ is a marker function defined as

$$\chi(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \boldsymbol{x} \in \Omega^s, \\ 0 & \text{otherwise.} \end{cases} \tag{4.36}$$

By substituting eq. (4.30) and eq. (4.31) into eq. (4.32) we get

$$\rho\left(\frac{\partial \boldsymbol{u} + \eta\tilde{\boldsymbol{u}}}{\partial t} + (\boldsymbol{u} + \eta\tilde{\boldsymbol{u}}) \cdot \nabla(\boldsymbol{u} + \eta\tilde{\boldsymbol{u}})\right) = \tag{4.37}$$
$$-\nabla(p + \eta\tilde{p}) + \mu\nabla^2(\boldsymbol{u} + \eta\tilde{\boldsymbol{u}}) - \frac{1}{\eta}\chi(\boldsymbol{u} + \eta\tilde{\boldsymbol{u}}) + \rho\boldsymbol{f}. \tag{4.38}$$

Even if we have perturbed the fields we still want eq. (3.8) to be satisfied. If we enforce that the velocities should be zero in the solid domain, $\chi\boldsymbol{u} = 0$, it is possible to split eq. (4.38) into two equations

$$\chi\boldsymbol{u} = 0, \tag{4.39}$$
$$\rho\left(\frac{\partial \boldsymbol{u}}{\partial t} + \boldsymbol{u} \cdot \nabla \boldsymbol{u}\right) = -\nabla p + \mu\nabla^2\boldsymbol{u} - \chi\tilde{\boldsymbol{u}} + \rho\boldsymbol{f}, \tag{4.40}$$
$$\rho\left(\frac{\partial \tilde{\boldsymbol{u}}}{\partial t} + \boldsymbol{u} \cdot \nabla \tilde{\boldsymbol{u}}\right) = -\nabla\tilde{p} + \mu\nabla^2\tilde{\boldsymbol{u}} - \tilde{\boldsymbol{u}} \cdot \nabla\boldsymbol{u} + 0, \tag{4.41}$$
$$\tag{4.42}$$

Since $\chi = 0$ in Ω^f, eq. (4.40) satisfies eq. (3.8). $\tilde{\boldsymbol{u}}$ satisfies

$$\tilde{\boldsymbol{u}} + \nabla p = 0, \qquad \text{in } \Omega^s \tag{4.43}$$

$$\rho\left(\frac{\partial \tilde{\boldsymbol{u}}}{\partial t} + \boldsymbol{u}\cdot\nabla\tilde{\boldsymbol{u}}\right) = -\nabla\tilde{p} + \mu\nabla^2\tilde{\boldsymbol{u}} - \tilde{\boldsymbol{u}}\cdot\nabla\boldsymbol{u} + 0, \qquad \text{in } \Omega^f \tag{4.44}$$

$$\nabla\cdot\tilde{\boldsymbol{u}} = 0, \qquad \text{in } \Omega \tag{4.45}$$

In [2] the existence, uniqueness as well as an error estimate are derived for this penalization method. The error of this penalization method is of the order of η. To get good results, one would naively set $\eta = 0$ and get infinite order, but as the time step needed for stability is inverse proportional to η this is a balancing act. In [6] it is shown that for a given grid there is on optimal η. For practical usage one sets up the simulation without penalization, then η is tuned so that it gives good performance on the given grid spacing, Δ, without a too small time step. To use the penalization method in the code, $-\frac{1}{\eta}\chi\boldsymbol{u}_\eta$ is added as a force term as seen in eq. (4.32). This force will then enter as a source term to the Poisson equation after using Chorin's projection method.

4.1.5. The level-set method

To solve the Navier-Stokes equations for two-phase flow, knowledge of the interface is required. The level-set method was proposed by Osher and Sethian[30]. Following [30] the interface is encoded as a signed scalar distance field

$$\varphi(\boldsymbol{x},t) = \begin{cases} d(\boldsymbol{x},t) & \text{if } \boldsymbol{x}\in\Omega_1 \\ -d(\boldsymbol{x},t) & \text{if } \boldsymbol{x}\in\Omega_2 \end{cases} \tag{4.46}$$

where

$$d(\boldsymbol{x},t) = \min_{\boldsymbol{x}'\in\Gamma(t)} \|\boldsymbol{x}-\boldsymbol{x}'\|. \tag{4.47}$$

This gives an implicit definition of the interface,

$$\Gamma(t) = \{\boldsymbol{x}\in\Omega \mid \varphi(\boldsymbol{x},t) = 0\}. \tag{4.48}$$

The interface moves according to the flow of the fluids. Time evolution is thus simply advecting the function with the velocity field

$$\frac{\partial \varphi}{\partial t} + \hat{\boldsymbol{u}} \cdot \boldsymbol{\nabla} \varphi = 0, \tag{4.49}$$

where $\hat{\boldsymbol{u}}$ is the fluid velocity field at the boundary, extrapolated to the whole domain. This quantity can be found by solving

$$\frac{\partial \hat{\boldsymbol{u}}}{\partial \tau} + S(\varphi)\boldsymbol{n} \cdot \boldsymbol{\nabla} \hat{\boldsymbol{u}} = 0, \quad \hat{\boldsymbol{u}}|_{\tau=0} = u, \tag{4.50}$$

Here τ is a pseudo time, and S is a smeared sign function which is zero at the interface,

$$S(\varphi) = \frac{\varphi}{\sqrt{\varphi^2 + 2\Delta_x^2}}. \tag{4.51}$$

We are interested in the steady-state solution of this equation [47, p. 193], in other words, the limit of the solution of eq. (4.50) as $\tau \to \infty$.

As the level-set field is advected by eq. (4.49) it will become distorted and lose its signed distance-property. Because of this, the level-set function is reinitialized at regular intervals by solving

$$\frac{\partial \varphi}{\partial \tau} + S(\varphi_0)(1\boldsymbol{\nabla}\varphi| - 1) = 0, \tag{4.52}$$

$$\varphi(\boldsymbol{x}, 0) = \varphi_0(\boldsymbol{x}), \tag{4.53}$$

to steady state [29, (7.4)]. Even though eq. (4.50) and eq. (4.52) are defined for the whole domain, we are only interested in the extrapolated velocity and the reinitialized field in a neighborhood around the interface. Interestingly the characteristics of both eq. (4.50) and eq. (4.52) originate at the interface going outwards. Because of this, solution to steady state in the whole domain is not needed to get the data we are interested in. In [1] this is detailed further.

The properties required to calculate forces coming from a fluid interface is the interface normal vectors and the interface curvature. Both of these

are available and can be computed from φ,

$$\boldsymbol{n} = \frac{\boldsymbol{\nabla}\varphi}{|\boldsymbol{\nabla}\varphi|}, \tag{4.54}$$

$$\kappa = \boldsymbol{\nabla} \cdot \left(\frac{\boldsymbol{\nabla}\varphi}{|\boldsymbol{\nabla}\varphi|} \right). \tag{4.55}$$

4.1.6. The ghost-fluid method

There are two main approaches in which the contact discontinuity at the interface can enter the numerical scheme. One is as a smeared out delta function. This is what is done in the immersed boundary method eq. (4.169). Here the effect of the singular forces at the interface is approximated as a smooth delta function, section 4.2.6, which spans several grid cells. With this method, the normal finite difference approximations to derivatives can be used as there are no discontinuities in the solution, but rather very steep, smooth, transitions. The other method is to incorporate the contact discontinuities in the numerics, handling them in a sharp manner. This means that there is an actual jump in the solution, and spatial derivatives are not defined. Instead, jump conditions are used to relate the values across the interface. This method require a whole deal more implementation work as the numerics is altered at the interface and logic has to be added to the code to handle this.

In the code, the ghost-fluid method ([14], [19] and [13]) , which is a sharp-interface method, is used. Here the method will be outlined for a one-dimensional model problem. The method is readily expanded to two or three dimensions because it is applied dimension by dimension. For more in-depth implementation details [19] is a good reference.

The key of this model problem is to demonstrate what has to be done with the finite-difference approximations of the spatial derivatives. For this, the one-dimensional Poisson problem is sufficient as an example

$$\frac{\partial}{\partial x}\left(\beta \frac{\partial u}{\partial x}\right) = f. \tag{4.56}$$

Here u and f are scalar functions while β is a constant on each side of the interface. The ghost-fluid method needs interface jump conditions, for our

model problem assume that they are given by

$$[\![u]\!] = a, \tag{4.57}$$

$$\left[\!\left[\beta \frac{\partial u}{\partial x}\right]\!\right] = b, \tag{4.58}$$

where $[\![\gamma]\!] = \gamma^+ - \gamma^-$ is the jump of a discontinuous variable γ over the interface, and γ^- and γ^+ is the value of γ at the left and right side of the interface respectively [2]. Now assume that the interface lies between grid cell k and $k+1$. Using a standard central difference approximation at grid cell k, similar to eq. (4.3), would yield

$$\frac{\beta_{k+1/2}\frac{u_{k+1}-u_k}{\Delta_x} - \beta_{k-1/2}\frac{u_k-u_{k-1}}{\Delta_x}}{\Delta_x} = f_k. \tag{4.59}$$

This approximation is second order accurate as long as the solution is smooth, but at the interface this is not the case. This leads to an error which smears the interface. The key trick in GFM is to replace the problematic value, u_{k+1}, with a ghost value, u^g. This value is extrapolated from the left side of the interface taking into account the jump conditions. We define

$$\theta = \frac{x_I - x_k}{\Delta_x} \tag{4.60}$$

as the normalized distance from the grid cell x_k to the interface x_I. Using this we can write u^g as

$$u^g = \frac{u^-(\theta - 1)u_k}{\theta}. \tag{4.61}$$

Recall that β is assumed constant on each side of the surface. Inserting eq. (4.61) into eq. (4.59) leaves us with

$$\frac{\beta\frac{u^- - u_k}{\theta\Delta_x} - \beta\frac{u_k-u_{k-1}}{\Delta_x}}{\Delta_x} = f_k. \tag{4.62}$$

[2] For two and three-dimensional problems the notion of left and right breaks down and the two fluids domains are decided by the sign of the level-set function, hence the convention is to use plus minus signs in superscript.

Now an expression for u^- is needed. To approximate it we discretize eq. (4.58) to get

$$\beta^+ \frac{u_{k+1} - u^+}{(1-\theta)\Delta_x} - \beta^- \frac{u^- - u_k}{\theta \Delta_x} = b. \tag{4.63}$$

Then we solve this for u^- giving

$$u^- = \frac{\theta\beta^+ u_{k+1} + (1+\theta)\beta^- u_k - \theta\beta^+ a - \theta(1-\theta)\Delta_x b}{\theta\beta^+ + (1-\theta)\beta^-}. \tag{4.64}$$

Inserting this approximation of the left interface value into eq. (4.62) gives

$$\frac{\hat{\beta}(u_{k+1} - u_k) - \beta^-(u_k - u_{k-1})}{\Delta_x^2} = f_k + \frac{\hat{\beta} a}{\Delta_x^2} + \frac{(1-\theta)\hat{\beta} b}{\beta^+ \Delta_x}, \tag{4.65}$$

where

$$\hat{\beta} = \frac{\beta^- \beta^+}{\theta\beta^+ + (1-\theta)\beta^-}. \tag{4.66}$$

Comparing with eq. (4.59) one sees that the modifications by GFM adds a source term to the right hand side of the equation as well as a scaling dependent on where the interface is located between grid cell k and $k+1$. If the interface is located between k and $k-1$, the same procedure as above would lead to

$$\frac{\beta^+(u_{k+1} - u_k) - \hat{\beta}(u_k - u_{k-1})}{\Delta_x^2} = f_k - \frac{\hat{\beta} a}{\Delta_x^2} + \frac{\theta\hat{\beta} b}{\beta^- \Delta_x}. \tag{4.67}$$

4.2. The immersed boundary method

4.2.1. Motivation

This section shows how the standard level-set method does not contain the information required to express the compression or stretching of an interface. To have compression of an interface in incompressible flow, assuming no sources or sinks are present, the velocity component tangential to the interface of the fluid flow has to be nonzero. If this is not the case, the interface will never be stretched or compressed.

Lemma 4.2.1. *Given any vector with unit norm, $\|\boldsymbol{n}\| = 1$, the operator $P_n = \boldsymbol{n}(\boldsymbol{n}\cdot)$ applied to an arbitrary vector $\boldsymbol{v}$ is a projection*

Proof. The defining property of a projection is $P^2 = P$.

$$\begin{aligned} P_n^2\boldsymbol{v} &= \boldsymbol{n}(\boldsymbol{n}\cdot\boldsymbol{n}(\boldsymbol{n}\cdot\boldsymbol{v})) \\ &= \boldsymbol{n}(1(\boldsymbol{n}\cdot\boldsymbol{v})) \\ &= \boldsymbol{n}(\boldsymbol{n}\cdot\boldsymbol{v}) \\ &= P_n\boldsymbol{v} \end{aligned} \tag{4.68}$$

□

Another property of projections is that if P_a is a projection, then $P_b = (1 - P_a)$ is also a projection.

$$\begin{aligned} P_b^2 &= (1-P_a)^2 \\ &= (1-P_a) - P_a(1-P_a) \\ &= 1 - P_a - P_a + P_a^2 \\ &= 1 - P_a - P_a + P_a \\ &= (1-P_a) \\ &= P_b \end{aligned} \tag{4.69}$$

Lemma 4.2.2.

$$P_{\|}(\varphi) = (1 - \nabla\varphi(\nabla\varphi\cdot))$$

is a projection operator.

Proof. φ is the signed distance from the interface, thus its gradient is always of unit norm, $\|\nabla\varphi\| = 1$. Using lemma 4.2.1, we see that $(\nabla\varphi(\nabla\varphi\cdot))$ is a projection. Using the second property of projections, eq. (4.69), we have that $P_{\|}(\varphi) = (1 - (\nabla\varphi(\nabla\varphi\cdot)))$ is also a projection. □

Given a level-set function, φ, $P_{\|}(\varphi)$ projects an arbitrary velocity field $\boldsymbol{u}$ into the space of velocity fields that are tangential to the interface.

With this operator we can decompose an arbitrary velocity field into

$$\boldsymbol{u} = \boldsymbol{u}_\perp + \boldsymbol{u}_\parallel \tag{4.70}$$
$$\boldsymbol{u}_\parallel = P_\parallel \boldsymbol{u} \tag{4.71}$$
$$\boldsymbol{u}_\perp = \boldsymbol{u} - \boldsymbol{u}_\parallel \tag{4.72}$$

where $\boldsymbol{u}_\perp$ is a velocity field normal to the interface and $\boldsymbol{u}_\parallel$ is a velocity tangential to the interface.

Recall that the goal of this thesis is to simulate the elastic effects observed on the interface of water-drops in crude-oil. To be able to simulate this, it is required to know how much an infinitesimal piece of the interface has been deformed from its equilibrium length. We will now show that standard level-set does not contain this information.

Theorem 4.2.3. *A signed-distance field does not contain the information needed to express compression or stretching of the interface, i.e.*

$$\frac{\partial}{\partial \boldsymbol{u}_\parallel} \dot{\varphi} = 0 \tag{4.73}$$

Proof. At each time step the signed distance function, φ, is advected using the advection equation eq. (4.49). Applying this advection to the decomposed velocity field gives

$$\dot{\varphi} = \boldsymbol{u} \cdot \nabla \varphi \tag{4.74}$$
$$\dot{\varphi} = (\boldsymbol{u}_\perp + \boldsymbol{u}_\parallel) \cdot \nabla \varphi \tag{4.75}$$
$$\dot{\varphi} = \boldsymbol{u}_\perp \cdot \nabla \varphi + \boldsymbol{u}_\parallel \cdot \nabla \varphi \tag{4.76}$$

By construction gradient of φ is always normal to the interface, $\boldsymbol{u}_\parallel \cdot \nabla \varphi = 0$, and

$$\dot{\varphi} = \boldsymbol{u}_\perp \cdot \nabla \varphi + \boldsymbol{u}_\parallel \cdot \nabla \varphi, \tag{4.77}$$
$$= \boldsymbol{u}_\perp \cdot \nabla \varphi. \tag{4.78}$$

□

In other words, the interface is invariant under velocity fields tangential to the interface. This, together with the statement that for incompressible flow, the part of the velocity field that stretches or compresses the interface is the tangential one tells about a fundamental limitation in the level-set formulation for interface tracking. There is no way of knowing whether a part of the interface is compressed or stretched, this information simply does not exist in φ.

The conclusion is that without an extension, the level-set method is not able to represent the compression of an interface. This lead to the search of an alternative method. One of the criteria for the choice of method was that it should have proven itself useful to simulate an elastic interface. This was found in the immersed boundary method. It was originally developed for simulating biological systems, e.g. blood flow trough a heart [33], which have many similarities to the simulation of drops covered in elastic membranes. Other than this, the immersed boundary method contains, as will be shown in the following sections:

- A rigorous mathematical framework of how to represent the interface, and how it interacts with the flow.
- The ability to represent an arbitrary shape.
- A uniquely defined delta function which guarantees several conservation properties.
- The possibility to solve Navier-Stokes on a standard Eulerian grid.

4.2.2. Introduction

The key innovation of the immersed boundary method, [32], is to allow solving the Navier-Stokes equations, or other continuum equations, on an Eulerian regular grid, while handling a large class of arbitrary, deformable and rigid bodies embedded in the continuum. These bodies are described in a natural Lagrangian way. The crux of the method is to be able to approximately transform Lagrangian information into Eulerian, and vice versa.

At first sight, it is not obvious why such a method is so desirable. If one only deals with simple geometries, a full Eulerian formulation together with appropriate transformations of the grid and operators will be sufficient and computationally efficient. If the geometries are not easily described by a coordinate transformation, then this method cannot be used. A way of handling such problems is an unstructured grid, together with an unstructured finite-difference, finite-volume, or finite-element, formulation. Here *all* the data is inherently unstructured and, because of this, all discretization operators and resulting linear systems will be in their most general forms. While theoretically attractive because of the generality an unstructured approach offers, the computational cost carried is normally massive. For problems where the geometry is spatially stationary, grid generation can be done once and used in each time step. However, if the problem has a geometry that changes over time, e.g. a deforming water drop in oil, the grid generation has to be done for each sub-time-step in the time integration method. Also, after generating the grid, the linear systems resulting from discretization of the problem are nowhere near as regular in their sparsity pattern and coefficient structure as problems discretized using finite differences.

For the problems where it can be used, the immersed boundary method takes the best from both worlds. Irregular geometries are represented in the natural Lagrangian way; points in space, and surfaces/volumes connecting the points. The PDEs of the problem are solved on an Eulerian grid, where all resulting linear systems have a nice structure. Any time-dependent geometric features of the problem are only present in the Lagrangian data, and a simple transformation between the Lagrangian and Eulerian data is used to compute tho forces from the Lagrangian structure to the Eulerian grid.

4.2.3. Derivation

This section gives a thorough derivation of the theory required to express the immersed boundary method. It follows Peskin's review paper of the method, [32] together with [28] for a less abstract perspective on the method. The result is a transformation between Lagrangian and Eulerian

data for a material immersed in a fluid. The transformation is shown to preserve physical quantities like mass, momentum, torque, effect, force, and energy. Here, capital letters are used for Lagrangian properties, while lower case letters are used for Eulerian properties. This way, $\boldsymbol{X}$ is the position of a Lagrangian point, while $\boldsymbol{x}$ is the position of an Eulerian grid node.

The elastic material

Let (q, r, s) be curvilinear coordinates that describe an incompressible elastic material filling a three-dimensional space. In this coordinate system, integer values of (q, r, s) represent a Lagrangian material point[3]. Let $\boldsymbol{X}(q, r, s, t)$ be the position of material point (q, r, s) at time t. Let $M(q, r, s)$ be the mass density of the material at (q, r, s) in such a way that

$$\int_Q M(q, r, s) \mathrm{d}q\mathrm{d}r\mathrm{d}s$$

is the mass of the material defined by $(q, r, s) \in Q$. Since we are modeling an incompressible material, mass is time independent and mass is conserved on a per material point basis.

Now, given a configuration of the material, $\boldsymbol{X}(\cdot, \cdot, \cdot, t)$, we assume there is a unique elastic potential energy. This energy is described by a functional, $E(\boldsymbol{X})$, meaning that $E(\boldsymbol{X}(\cdot, \cdot, \cdot, t))$ is the elastic potential energy at time t.

Incompressibility in Lagrangian form

We now want to consider a perturbation of the position, $\boldsymbol{X}$, and how it influences the energy functional. Denote a perturbation in $\boldsymbol{X}$ by $\wp\boldsymbol{X}(\cdot, \cdot, \cdot, t)$, then the accompanying perturbation in energy can always be expressed

[3] (q, r, s) can be viewed as coordinates into the data structure storing the variables of each Lagrangian point. For a three-dimensional material, a three-dimensional array where each element is a vector is used. In the actual code the domain is two and the interface is one-dimensional.

in the form

$$\wp E(\boldsymbol{X}(\cdot,\cdot,\cdot,t)) = \int (-\boldsymbol{F}(q,r,s,t)) \cdot \wp \boldsymbol{X}(q,r,s,t) \mathrm{d}q\mathrm{d}r\mathrm{d}s \tag{4.79}$$

where $\boldsymbol{F}(\cdot,\cdot,\cdot,t)$ is the Fréchet derivative of E at the material configuration $\boldsymbol{X}(\cdot,\cdot,\cdot,t)$. Physically, $\boldsymbol{F}$ can be interpreted as the force density resulting from the elasticity of the material in the configuration $\boldsymbol{X}$ with respect to (q,r,s). This agrees with the principle of virtual work[16, ch. 1], which roughly can be summarized as: "For all possible trajectories a particle can take in a potential field, the one taken will be a minimizer of the potential energy." A shorter way of expressing $\boldsymbol{F}$ is

$$\boldsymbol{F} = \frac{\wp E}{\wp \boldsymbol{X}}. \tag{4.80}$$

We now have an elegant framework for expressing an elastic material and its potential energy. Next we want to express the constraint of incompressibility. Let

$$J(q,r,s,t) = \det\left(\frac{\partial \boldsymbol{X}}{\partial q}, \frac{\partial \boldsymbol{X}}{\partial r}, \frac{\partial \boldsymbol{X}}{\partial s}\right) \tag{4.81}$$

be the volume density in such a way that the volume occupied by material point (q,r,s) at time t is equal to

$$\int_Q J(q,r,s,t)\mathrm{d}q\mathrm{d}r\mathrm{d}s. \tag{4.82}$$

Because of incompressibility, this should be constant for any choice of Q, which is equivalent to

$$\frac{\partial J}{\partial t} = 0. \tag{4.83}$$

Because of this, we can drop the t argument and instead write $J(q,r,s)$.

By the principle of least action [16, ch. 2], the system will evolve to minimize the action of S,

$$S = \int_0^T L(t)\mathrm{d}t, \tag{4.84}$$

where L is the Lagrangian of the system. The minimization is constrained by the incompressibility eq. (4.83) and initial and final conditions

$$\boldsymbol{X}(q,r,s,0) = \boldsymbol{X}_0(q,r,s), \tag{4.85}$$
$$\boldsymbol{X}(q,r,s,T) = \boldsymbol{X}_T(q,r,s). \tag{4.86}$$

A Lagrangian is the difference between kinetic and potential energy. For us it is

$$L(t) = \frac{1}{2}\int M(q,r,s)\left|\frac{\partial \boldsymbol{X}}{\partial t}(q,r,s,t)\right|^2 \mathrm{d}q\mathrm{d}r\mathrm{d}s - E(\boldsymbol{X}(\cdot,\cdot,\cdot,t)). \tag{4.87}$$

For a perturbation we get

$$\wp L(t) = \frac{1}{2}\int M(q,r,s)\left|\frac{\partial(\wp \boldsymbol{X})}{\partial t}(q,r,s,t)\right|^2 \mathrm{d}q\mathrm{d}r\mathrm{d}s - \wp E(\boldsymbol{X}(\cdot,\cdot,\cdot,t)).\mathrm{d}t \tag{4.88}$$

We then integrate over time to get S, the quantity we want to minimize

$$\begin{aligned}\wp S &= \int_0^T \wp L(t)\mathrm{d}t \\ &= \frac{1}{2}\int_0^T \int M(q,r,s)\left|\frac{\partial(\wp \boldsymbol{X})}{\partial t}(q,r,s,t)\right|^2 \mathrm{d}q\mathrm{d}r\mathrm{d}s\mathrm{d}t \\ &- \int_0^T \wp E(\boldsymbol{X}(\cdot,\cdot,\cdot,t))\mathrm{d}t.\end{aligned}$$

We substitute eq. (4.79) for the last term in the above equation and get

$$\begin{aligned}\wp S &= \int_0^T \wp L(t)\mathrm{d}t \\ &= \frac{1}{2}\int_0^T \int M(q,r,s)\left|\frac{\partial(\wp \boldsymbol{X})}{\partial t}(q,r,s,t)\right|^2 \mathrm{d}q\mathrm{d}r\mathrm{d}s\mathrm{d}t \\ &+ \int_0^T \int \boldsymbol{F}(q,r,s,t)\cdot \wp \boldsymbol{X}(q,r,s,t)\mathrm{d}q\mathrm{d}r\mathrm{d}s\mathrm{d}t.\end{aligned}$$

The two integrands can now be collected:

$$\wp S = \frac{1}{2}\int_0^T \int M(q,r,s)\left|\frac{\partial(\wp \boldsymbol{X})}{\partial t}(q,r,s,t)\right|^2 + \boldsymbol{F}(q,r,s,t)\cdot\wp\boldsymbol{X}(q,r,s,t)\mathrm{d}q\mathrm{d}r\mathrm{d}s\mathrm{d}t. \tag{4.89}$$

Applying integration by parts to the first term we get

$$\wp S = \frac{1}{2}\int_0^T \int \left(-M(q,r,s)\frac{\partial^2 \boldsymbol{X}}{\partial t^2}(q,r,s,t) + \boldsymbol{F}(q,r,s,t)\right)\cdot\wp\boldsymbol{X}(q,r,s,t)\mathrm{d}q\mathrm{d}r\mathrm{d}s\mathrm{d}t. \tag{4.90}$$

This is valid for arbitrary $\wp\boldsymbol{X}$, but the perturbations are not arbitrary, they must be consistent with the incompressibility constraint eq. (4.83). There is no obvious way to enforce this constraint on eq. (4.90).

Incompressibility in Eulerian form

Incompressibility has a particularly nice form in Eulerian variables,

$$\nabla\cdot\boldsymbol{u} = 0. \tag{4.91}$$

Thus it is possible that a change to Eulerian variables would resolve this difficulty.

To check this, we introduce two new quantities, the velocity field $\boldsymbol{u}$ and the virtual velocity field $\boldsymbol{v}$.

$$\boldsymbol{u}(\boldsymbol{X}(q,r,s,t),t) = \frac{\partial \boldsymbol{X}}{\partial t}(q,r,s,t), \tag{4.92}$$

$$\boldsymbol{v}(\boldsymbol{X}(q,r,s,t),t) = \wp\boldsymbol{X}(q,r,s,t). \tag{4.93}$$

$\boldsymbol{u}(\boldsymbol{X},t)$ is the familiar velocity field, while $\boldsymbol{v}(\boldsymbol{X},t)$ is the perturbation (difference from unperturbed velocity) experienced by a particle at position $\boldsymbol{X}$ at time t.

The material derivative is also needed

$$\frac{\mathrm{D}\boldsymbol{u}}{\mathrm{D}t} = \frac{\partial \boldsymbol{u}}{\partial t} + \boldsymbol{u}\cdot\nabla u, \tag{4.94}$$

and the identity

$$\frac{\mathrm{D}\boldsymbol{u}}{\mathrm{D}t} = \frac{\partial^2 \boldsymbol{X}}{\partial t^2}(q, r, s, t). \tag{4.95}$$

An insight is that the perturbations have to be incompressible not just globally but also locally. That is, the density has to be both spatially and temporally constant. Given this, substituting $\boldsymbol{X} + \wp\boldsymbol{X}$ for $\boldsymbol{X}$ in J should not change it up to first order terms of $\wp\boldsymbol{X}$.

We introduce

$$\boldsymbol{a} = (q, r, s) \tag{4.96}$$

as a coordinate vector. Let

$$\frac{\partial \boldsymbol{X}}{\partial \boldsymbol{a}} = \begin{bmatrix} \frac{\partial X_x}{\partial q} & \frac{\partial X_y}{\partial q} & \frac{\partial X_z}{\partial q} \\ \frac{\partial X_x}{\partial r} & \frac{\partial X_y}{\partial r} & \frac{\partial X_z}{\partial r} \\ \frac{\partial X_x}{\partial s} & \frac{\partial X_y}{\partial s} & \frac{\partial X_z}{\partial s} \end{bmatrix}. \tag{4.97}$$

Using this we get a new way to write J,

$$J = \det\left(\frac{\partial \boldsymbol{X}}{\partial \boldsymbol{a}}\right). \tag{4.98}$$

We want to see what happens when we perturb J, to do this we need the following identity for perturbations of the determinant. Given that A is a nonsingular square matrix, then

$$\wp \log(\det(A)) = \mathrm{Tr}((\wp A)A^{-1}). \tag{4.99}$$

Recall eq. (4.93). By differentiating on both sides with respect to $\boldsymbol{a}$ we get

$$\frac{\partial \wp\boldsymbol{X}}{\partial \boldsymbol{a}} = \frac{\partial \boldsymbol{v}}{\partial \boldsymbol{x}}\frac{\partial \boldsymbol{X}}{\partial \boldsymbol{a}}. \tag{4.100}$$

By changing the order of perturbation and differentiation and multiplying with $\left(\frac{\partial \boldsymbol{X}}{\partial \boldsymbol{a}}\right)^{-1}$ one arrives at

$$\left(\wp\frac{\partial \boldsymbol{X}}{\partial \boldsymbol{a}}\right) = \frac{\partial \boldsymbol{v}}{\partial \boldsymbol{x}}\frac{\partial \boldsymbol{X}}{\partial \boldsymbol{a}}, \tag{4.101}$$

$$\left(\wp\frac{\partial \boldsymbol{X}}{\partial \boldsymbol{a}}\right)\left(\frac{\partial \boldsymbol{X}}{\partial \boldsymbol{a}}\right)^{-1} = \frac{\partial \boldsymbol{v}}{\partial \boldsymbol{x}}. \tag{4.102}$$

Now we take the trace on both sides

$$\mathrm{Tr}\left(\left(\wp\frac{\partial \boldsymbol{X}}{\partial \boldsymbol{a}}\right)\left(\frac{\partial \boldsymbol{X}}{\partial \boldsymbol{a}}\right)^{-1}\right) = \mathrm{Tr}\left(\frac{\partial \boldsymbol{v}}{\partial \boldsymbol{x}}\right). \tag{4.103}$$

Substituting eq. (4.99) for the left-hand side gives

$$\wp \log(\det\left(\frac{\partial \boldsymbol{X}}{\partial \boldsymbol{a}}\right)) = \mathrm{Tr}\left(\frac{\partial \boldsymbol{v}}{\partial \boldsymbol{x}}\right). \tag{4.104}$$

Inspecting eq. (4.97) we see that the trace on the right-hand side is nothing but the divergence in Eulerian coordinates, we also insert J for the derivative on the left-hand side and get

$$\wp \log(J(q,r,s)) = \nabla \cdot \boldsymbol{v}(\boldsymbol{X}(q,r,s,t),t). \tag{4.105}$$

From this we can conclude that the incompressibility constraint in Lagrangian variables, $\wp J = 0$, is nothing but the familiar zero-divergence constraint in Eulerian variables.

$$\wp \log(J(q,r,s)) = 0, \tag{4.106}$$
$$\nabla \cdot \boldsymbol{v}(\boldsymbol{X}(q,r,s,t),t) = 0, \tag{4.107}$$
$$\Rightarrow \nabla \cdot \boldsymbol{v} = 0. \tag{4.108}$$

The same derivation can be done to the velocity field $\boldsymbol{u}$ and it yields the same constraint,

$$\nabla \cdot \boldsymbol{u} = 0. \tag{4.109}$$

Transforming between Lagrangian and Eulerian variables

We can now start expressing the relationship between Lagrangian and Eulerian variables. To translate between the two, the defining property of the delta function is needed,

$$\int \delta(x - x_0) f(x) = f(x_0). \tag{4.110}$$

If we want to express eq. (4.90) in Eulerian variables, we will need to convert perturbed position and acceleration times perturbed position. The

relationship between the Lagrangian perturbed position, virtual velocity, and the Eulerian virtual velocity can, using a 3-dimensional delta function, be written

$$\wp \boldsymbol{X}(q,r,s,t) = \int \boldsymbol{v}(\boldsymbol{x},t)\delta(\boldsymbol{x} - \boldsymbol{X}(q,r,s,t))\mathrm{d}\boldsymbol{x}. \tag{4.111}$$

Multiplying with the acceleration on both sides and using the identity eq. (4.95) we get

$$\frac{\partial^2 \boldsymbol{X}}{\partial t^2}(q,r,s,t) \cdot \wp \boldsymbol{X}(q,r,s,t) = \int \frac{\mathrm{D}\boldsymbol{u}}{\mathrm{D}t}(\boldsymbol{x},t) \cdot \boldsymbol{v}(\boldsymbol{x},t)\delta(\boldsymbol{x} - \boldsymbol{X}(q,r,s,t))\mathrm{d}\boldsymbol{x}. \tag{4.112}$$

Substitute eq. (4.111) and eq. (4.112) into eq. (4.90) gives

$$0 = \frac{1}{2}\int_0^T \int\int \left(-M(q,r,s)\frac{\mathrm{D}\boldsymbol{u}}{\mathrm{D}t}(\boldsymbol{x},t) + \boldsymbol{F}(q,r,s,t)\right) \cdot \boldsymbol{v}(\boldsymbol{x},t)\delta(\boldsymbol{x} - \boldsymbol{X}(q,r,s,t))\mathrm{d}\boldsymbol{x}\mathrm{d}q\mathrm{d}r\mathrm{d}s\mathrm{d}t. \tag{4.113}$$

There are still Lagrangian variables left in the expression, namely M and $\boldsymbol{F}$. Using the same delta function technique one can define

$$\rho(\boldsymbol{x},t) = \int M(q,r,s)\delta(\boldsymbol{x} - \boldsymbol{X}(q,r,s,t))\mathrm{d}q\mathrm{d}r\mathrm{d}s, \tag{4.114}$$

$$\boldsymbol{f}(\boldsymbol{x},t) = \int \boldsymbol{F}(q,r,s,t)\delta(\boldsymbol{x} - \boldsymbol{X}(q,r,s,t))\mathrm{d}q\mathrm{d}r\mathrm{d}s, \tag{4.115}$$

which is the Eulerian mass density and elastic force density, respectively. Looking at eq. (4.113) we can substitute in eq. (4.114) and eq. (4.115), getting rid of the integral over q, r, s,

$$0 = \frac{1}{2}\int_0^T \int \left(-\rho(\boldsymbol{x},t)\frac{\mathrm{D}\boldsymbol{u}}{\mathrm{D}t}(\boldsymbol{x},t) + \boldsymbol{f}(\boldsymbol{x},t)\right) \cdot \boldsymbol{v}(\boldsymbol{x},t)\mathrm{d}\boldsymbol{x}\mathrm{d}t. \tag{4.116}$$

Equation (4.116) now only contains Eulerian variables. It holds for arbitrary $\boldsymbol{v}$ as long as they are within the constraints given,

$$\boldsymbol{v}(\boldsymbol{x},0) = 0, \tag{4.117}$$

$$\boldsymbol{v}(\boldsymbol{x},T) = 0, \tag{4.118}$$

$$\nabla \cdot \boldsymbol{v} = 0. \tag{4.119}$$

Using Helmholtz-Hodge decomposition, theorem 4.1.1, we can always write an arbitrary vector field as the sum of a gradient and a divergence-free vector field. We use this on the first part of the integrand in eq. (4.116):

$$\rho \frac{\mathrm{D}\boldsymbol{u}}{\mathrm{D}t} - \boldsymbol{f} = -\nabla p + \boldsymbol{w} \tag{4.120}$$

$$\nabla \cdot \boldsymbol{w} = 0 \tag{4.121}$$

Now, if $\boldsymbol{w} = 0$, this would be the incompressible Navier-Stokes equations without viscosity effects. To check if this is true we use the freedom in $\boldsymbol{v}$ to choose

$$\boldsymbol{v}(\boldsymbol{x}, t) = \xi(t)\boldsymbol{w}(\boldsymbol{x}, t). \tag{4.122}$$

Since $\nabla \cdot \boldsymbol{w} = 0$ this satisfies the constraints, eq. (4.119), on $\boldsymbol{v}$ as long as $\xi(0) = \xi(T) = 0$. Going forward we choose

$$\xi(t) > 0 \quad \forall t \in (0, T), \tag{4.123}$$

which inserted into eq. (4.116) gives

$$0 = \int_0^T \xi(t) \int (-\nabla p(\boldsymbol{x}, t) + \boldsymbol{w}(\boldsymbol{x}, t)) \cdot \boldsymbol{w}(\boldsymbol{x}, t) \mathrm{d}\boldsymbol{x}\mathrm{d}t. \tag{4.124}$$

Since $\nabla \cdot \boldsymbol{w} = 0$, the term with ∇p disappears and the integral ends up as

$$0 = \int_0^T \xi(t) \int |\boldsymbol{w}(\boldsymbol{x}, t)|^2 \, \mathrm{d}\boldsymbol{x}\mathrm{d}t. \tag{4.125}$$

The only way for this to be true if $\xi(t) > 0$ is if $\boldsymbol{w} = 0$.

Summary

To sum up, we have the following equations for the immersed boundary formulation of an incompressible elastic material. The viscosity, which

was left out, is added assuming a Newtonian fluid.

$$\rho\left(\frac{\partial \boldsymbol{u}}{\partial t}+\boldsymbol{u}\cdot\nabla\boldsymbol{u}\right)+\nabla p=\mu\nabla^2\boldsymbol{u}+\boldsymbol{f}, \tag{4.126}$$

$$\nabla\cdot\boldsymbol{u}=0, \tag{4.127}$$

$$\rho(\boldsymbol{x},t)=\int M(q,r,s)\delta(\boldsymbol{x}-\boldsymbol{X}(q,r,s,t))\mathrm{d}q\mathrm{d}r\mathrm{d}s, \tag{4.128}$$

$$\boldsymbol{f}(\boldsymbol{x},t)=\int \boldsymbol{F}(q,r,s,t)\delta(\boldsymbol{x}-\boldsymbol{X}(q,r,s,t))\mathrm{d}q\mathrm{d}r\mathrm{d}s, \tag{4.129}$$

$$\begin{aligned}\frac{\partial \boldsymbol{X}}{\partial t}(q,r,s,t)&=\boldsymbol{u}(\boldsymbol{X}(q,r,s,t),t)\\&=\int \boldsymbol{u}(\boldsymbol{x},t)\delta(\boldsymbol{x}-\boldsymbol{X}(q,r,s,t)\mathrm{d}\boldsymbol{x},\end{aligned} \tag{4.130}$$

$$\boldsymbol{F}=-\frac{\wp E}{\wp \boldsymbol{X}}. \tag{4.131}$$

In the previous section, the immersed boundary equations for a viscous elastic material filling all of the domain was derived. If the material does not fill the whole domain, but is immersed in a fluid, the equations have to be modified. The key change is that the mass term M in eq. (4.128) must be modified to account for the buoyancy force experienced when submerged in the fluid.

Another important case is when the material is an interface, e.g. a balloon filled with water, immersed in water. This interface only needs two variables to be parametrized, (r,s), which changes eq. (4.128)-eq. (4.130) to

$$\rho(\boldsymbol{x},t)=\rho_0+\int M(r,s)\delta(\boldsymbol{x}-\boldsymbol{X}(r,s,t))\mathrm{d}r\mathrm{d}s, \tag{4.132}$$

$$\boldsymbol{f}(\boldsymbol{x},t)=\int \boldsymbol{F}(r,s,t)\delta(\boldsymbol{x}-\boldsymbol{X}(r,s,t))\mathrm{d}r\mathrm{d}s, \tag{4.133}$$

$$\frac{\partial \boldsymbol{X}}{\partial t}(r,s,t)=\boldsymbol{u}(\boldsymbol{X}(r,s,t),t)=\int \boldsymbol{u}(\boldsymbol{x},t)\delta(\boldsymbol{x}-\boldsymbol{X}(r,s,t)\mathrm{d}\boldsymbol{x}, \tag{4.134}$$

where ρ_0 is the density of the fluid the material is immersed in. Note that since a surface has no volume, it does not displace any fluid, and one

thus does not have to take into account the buoyancy force and M is not modified. δ is still three-dimensional, but in the integral for ρ and $\boldsymbol{f}$, only two dimensions are integrated. Because of this, ρ and $\boldsymbol{f}$ are distributions, similar to a one-dimensional delta function that when integrated normally to the surface takes the appropriate value. It is common to call the operations taking a Lagrangian value and distributing it over the Eulerian grid "spreading". This makes eq. (4.132) and eq. (4.133) spreading equations. Operations that construct a value for a Lagrangian point from an Eulerian field are called interpolating, such as eq. (4.134).

4.2.4. Discretization of space

Up to now we have only discussed how to derive Eulerian variables from Lagrangian ones and vice-versa. We will now start the discretization of these equations, leading to a numerical method that can be implemented on a computer. The basic idea is this: Take eq. (4.126)-eq. (4.131), approximate integrals as discrete sums and the delta functions as smooth approximations of a delta function. Derivatives are approximated by finite difference stencils on the Lagrangian grid. Note that in this section Δ is used to symbolize a discretized version of continuous variables, e.g. δ_Δ and as grid distance in Eulerian and Lagrangian coordinates.

The Eulerian grid g_Δ is the standard, orthogonal, uniform grid of the form

$$\begin{aligned} \boldsymbol{x} &= \boldsymbol{j}\Delta \\ \boldsymbol{j} &= (j_1, j_2, j_3). \end{aligned} \tag{4.135}$$

where the $\boldsymbol{j}$ vector is an integer index vector for the coordinates [4]. The Lagrangian grid is a set, G_Δ, of integer vectors (q, r, s) of the form $(k_q\Delta q, k_r\Delta r, k_s\Delta s)$, where $(k_q, k_r, k_s) = \boldsymbol{k}$. If the distance between two Lagrangian points is too big, the discrete spreading operations will not approximate the continuous versions correctly. Because of this we have to demand that two

[4] the j vector uniquely specifies one grid node in the data structure storing the variables on the Eulerian grid.

Lagrangian points never are further apart than 1/2 Eulerian grid cell,

$$|\boldsymbol{X}(q+\Delta q, r, s, t) - \boldsymbol{X}(q, r, s, t)| < \frac{\Delta}{2} \quad \forall (q, r, s, t) \tag{4.136}$$

and similarly for Δr and Δs.

In the continuous form, the functional for the elastic potential energy is normally an integral over a local energy density $\mathcal{E}$. In the discrete case the integral is approximated by a sum as follows

$$E_\Delta = \sum_{k'} \mathcal{E}_{k'}(\dots \boldsymbol{X}_k \dots)\Delta q \Delta r \Delta s. \tag{4.137}$$

Perturbing this functional, we get

$$\wp E_\Delta = \sum_k \sum_{k'} \frac{\partial \mathcal{E}_{k'}}{\partial \boldsymbol{X}_k} \cdot \wp \boldsymbol{X}_k \Delta q \Delta r \Delta s. \tag{4.138}$$

If we let

$$\boldsymbol{F}_k = -\sum_{k'} \frac{\partial \mathcal{E}_{k'}}{\partial \boldsymbol{X}_k}, \tag{4.139}$$

this can be written

$$\wp E_\Delta = -\sum_k \boldsymbol{F}_k \cdot \wp \boldsymbol{X}_k \Delta q \Delta r \Delta s. \tag{4.140}$$

Looking at eq. (4.139) we see that it is equivalent to

$$\boldsymbol{F}_k \Delta q \Delta r \Delta s = -\frac{\partial E_\Delta}{\partial \boldsymbol{X}_k}, \tag{4.141}$$

which is the force acting on the material point (q, r, s). By a similar argument if M_Δ is the mass density of point (q, r, s) the actual mass is $M_\Delta \Delta q \Delta r \Delta s$.

Given a discretization of the Navier-Stokes equations, sections 4.1.1 to 4.1.2, and a smooth approximation to the delta function

$$\lim_{\Delta \to 0} \delta_\Delta = \delta \tag{4.142}$$

that obeys certain criteria, which will be discussed in section 4.2.6, we can now define the discrete versions of the spreading and interpolating equations

$$\rho(\boldsymbol{x},t) = \sum_{(q,r,s)\in G_\Delta} M(q,r,s)\delta_\Delta(\boldsymbol{x}-\boldsymbol{X}(q,r,s,t))\Delta q\Delta r\Delta s, \tag{4.143}$$

$$\boldsymbol{f}(\boldsymbol{x},t) = \sum_{(q,r,s)\in G_\Delta} \boldsymbol{F}(q,r,s,t)\delta_\Delta(\boldsymbol{x}-\boldsymbol{X}(q,r,s,t))\Delta q\Delta r\Delta s, \tag{4.144}$$

$$\frac{\mathrm{d}\boldsymbol{X}}{\mathrm{d}t}(q,r,s,t) = \sum_{\boldsymbol{x}\in g_\Delta} \boldsymbol{u}(\boldsymbol{x},t)\delta_\Delta(\boldsymbol{x}-\boldsymbol{X}(q,r,s,t)\Delta^3, \tag{4.145}$$

$$\boldsymbol{F} = -\frac{\partial}{\partial \boldsymbol{X}(q,r,s)} E_\Delta(\dots \boldsymbol{X}(q',r',s')\dots) \tag{4.146}$$

This is a system of ordinary differential equations, with $\boldsymbol{x}\in g_\Delta$ and (q,r,s) and $(q',r',s')\in G_\Delta$ which can be integrated with a Runge-Kutta method, see section 4.1.3. The code calculating eq. (4.146) is listed in listing 5. Spreading of forces, eq. (4.144) is in listing 12. Finally the interpolation of velocities to Lagrangian points, eq. (4.145), is in listing 9 and listing 4.

4.2.5. Physical identities

If eqs. (4.143) to (4.146) are to be correct, physical quantities should be the same before and after being converted between Lagrangian and Eulerian, and vice versa.

We start with the Eulerian form of momentum

$$\sum_{\boldsymbol{x}\in g_\Delta} \rho(\boldsymbol{x},t)\boldsymbol{u}(\boldsymbol{x},t)\Delta^3 \tag{4.147}$$

$$= \sum_{\boldsymbol{x}\in g_\Delta}\sum_{(q,r,s)\in G_\Delta} M(q,r,s)\delta_\Delta(\boldsymbol{x}-\boldsymbol{X}(q,r,s,t))\Delta q\Delta r\Delta s\boldsymbol{u}(\boldsymbol{x},t)\Delta^3 \tag{4.148}$$

$$= \sum_{(q,r,s)\in G_\Delta} M(q,r,s)\frac{\mathrm{d}\boldsymbol{X}}{\mathrm{d}t}(q,r,s,t)\Delta q\Delta r\Delta s, \tag{4.149}$$

which is the Lagrangian form of momentum. This means that momentum is conserved trough the transformation. Here, first we used eq. (4.143) and

then eq. (4.145). One important note is that if the spreading and interpolating operations use different δ, this identity does not hold. This is important because no momentum should be created or destroyed when transforming the representation of the physical system. The exact same argument can be used to see that the effect from the elastic force is preserved.

Next we consider mass. To do this we need to enforce some properties on the approximate delta function,

$$\sum_{\boldsymbol{x}\in g_\Delta} \delta_\Delta(\boldsymbol{x}-\boldsymbol{X})\Delta^3 = 1 \quad \forall \boldsymbol{X}, \tag{4.150}$$

$$\sum_{\boldsymbol{x}\in g_\Delta} (\boldsymbol{x}-\boldsymbol{X})\delta_\Delta(\boldsymbol{x}-\boldsymbol{X})\Delta^3 = 0 \quad \forall \boldsymbol{X}, \tag{4.151}$$

$$\sum_{\boldsymbol{x}\in g_\Delta} \boldsymbol{x}\delta_\Delta(\boldsymbol{x}-\boldsymbol{X})\Delta^3 = \boldsymbol{X} \quad \forall \boldsymbol{X}. \tag{4.152}$$

All of these properties are true for δ, and should thus also be true for a good approximation δ_Δ. Setting up the Eulerian equation for mass we have

$$\sum_{\boldsymbol{x}\in g_\Delta} \rho(\boldsymbol{x},t)\Delta^3 \tag{4.153}$$

$$= \sum_{\boldsymbol{x}\in g_\Delta} \sum_{(q,r,s)\in G_\Delta} M(q,r,s)\delta_\Delta(\boldsymbol{x}-\boldsymbol{X}(q,r,s,t)))\Delta q\Delta r\Delta s\ \Delta^3, \tag{4.154}$$

$$= \sum_{(q,r,s)\in G_\Delta} M(q,r,s)\Delta q\Delta r\Delta s. \tag{4.155}$$

Here we used eq. (4.143) and then eq. (4.150), interestingly this binds the time varying $\rho(\boldsymbol{x},t)$ to the constant $\sum M$ meaning that mass is conserved. The same can be done for the force:

$$\sum_{\boldsymbol{x}\in g_\Delta} \boldsymbol{f}(\boldsymbol{x},t)\Delta^3 \tag{4.156}$$

$$= \sum_{\boldsymbol{x}\in g_\Delta} \sum_{(q,r,s)\in G_\Delta} \boldsymbol{F}(q,r,s,t)\delta_\Delta(\boldsymbol{x}-\boldsymbol{X}(q,r,s,t)))\Delta q\Delta r\Delta s\ \Delta^3, \tag{4.157}$$

$$= \sum_{(q,r,s)\in G_\Delta} \boldsymbol{F}(q,r,s,t)\Delta q\Delta r\Delta s. \tag{4.158}$$

Lastly, using eq. (4.152), we can show that torque is converted consistently

$$\sum_{\boldsymbol{x}\in g_\Delta} \boldsymbol{x}\times \boldsymbol{f}(\boldsymbol{x},t)\Delta^3 \tag{4.159}$$

$$= \sum_{\boldsymbol{x}\in g_\Delta}\sum_{(q,r,s)\in G_\Delta} \boldsymbol{x}\times \boldsymbol{F}(q,r,s,t)\delta_\Delta(\boldsymbol{x}-\boldsymbol{X}(q,r,s,t)))\Delta q\Delta r\Delta s\ \Delta^3, \tag{4.160}$$

$$= \sum_{(q,r,s)\in G_\Delta} \boldsymbol{X}(q,r,s,t)\times \boldsymbol{F}(q,r,s,t)\Delta q\Delta r\Delta s. \tag{4.161}$$

4.2.6. The delta function, δ_Δ

In the previous sections we have referred to a smooth approximate delta function, δ_Δ, several times, without specifying further. This section studies the properties and the derivation of the delta function.

First, we assume that the 3D delta function is the product of three one-dimensional delta functions:

$$\delta_\Delta(\boldsymbol{x}) = \frac{1}{\Delta^3}\gamma\left(\frac{[\boldsymbol{x}]_x}{h}\right)\gamma\left(\frac{[\boldsymbol{x}]_y}{h}\right)\gamma\left(\frac{[\boldsymbol{x}]_z}{h}\right). \tag{4.162}$$

With this we can now focus on the function $\gamma(r)$ and its properties. Following [32, sec. 6] they are

$$\gamma(r) \text{ is continuous } \forall\, r\in\mathbb{R}, \tag{4.163}$$

$$\gamma(r) = 0 \quad \text{for } |r|\geq 2, \tag{4.164}$$

$$\sum_{j\,\text{even}} \gamma(r-j) = \sum_{j\,\text{odd}} \gamma(r-j) = \frac{1}{2} \quad \forall\, r\in\mathbb{R}, \tag{4.165}$$

$$\sum_j (r-j)\gamma(r-j) = 0 \quad \forall\, r\in\mathbb{R}, \tag{4.166}$$

$$\sum_j (\gamma(r-j))^2 = C \quad \forall\, r\in\mathbb{R}, \tag{4.167}$$

where C is independent of r.

These postulates have different justification. First, composing the three-dimensional delta function from one-dimensional delta functions, eq. (4.162),

reduces the complexity a lot, while it preserves the essential property that $\lim_{\Delta\to 0}\delta_\Delta = \delta$. The first property of the one-dimensional delta function, continuity, makes sense in that one wants a smooth approximation of the delta function, and that it does not introduce any jump in the Eulerian grid as a point moves around. Equation (4.164) stems from a very practical need, being computationally effective. Because of this bounded support, each Lagrangian point has a finite radius of effect in the Eulerian grid. Equation (4.165) implies

$$\sum_j \gamma(r-j) = 1 \quad \forall\, r \in \mathbb{R}, \tag{4.168}$$

which implies eq. (4.150) which is needed to derive the identities for mass, torque and force in section 4.2.5. Equation (4.166) is needed for the identity eq. (4.150) which is part of the derivation of torque in section 4.2.5. The last property, eq. (4.167) is coming from the fact that ideally, the immersed boundary should be translation invariant with respect to its effect on the Eulerian grid. It can be proven that this is impossible with compact support. In [32, sec. 6] it is shown that eq. (4.167) results in a good approximation to translation invariance.

One interesting fact about these postulates, is that they together uniquely identify a single delta function, derived in [32, sec. 6],

$$\gamma(r) = \begin{cases} 0, & r \le -2 \\ \frac{1}{8}\left(5 + 2r - \sqrt{-7 - 12r - 4r^2}\right), & -2 \le r \le -1 \\ \frac{1}{8}\left(3 + 2r + \sqrt{1 - 4r - 4r^2}\right), & -1 \le r \le -0 \\ \frac{1}{8}\left(3 - 2r + \sqrt{1 + 4r - 4r^2}\right), & 0 \le r \le 1 \\ \frac{1}{8}\left(5 - 2r - \sqrt{-7 + 12r - 4r^2}\right), & 1 \le r \le 2 \\ 0, & 2 \le r. \end{cases} \tag{4.169}$$

This is quite different to most other diffuse-interface methods where it is not clear which delta function is the most appropriate. The code for the delta function is in listing 15.

Discretizing the immersed boundary to get interface tension

As we have seen, the immersed boundary method can express the coupling between an immersed elastic membrane and the fluid it is immersed in. We will now take a closer look at what physical phenomena this enables us to simulate. We derive ordinary surface tension using the results from section 3.2.1. Then the results from section 3.2.2 is discretized to get a general viscoelastic tension.

4.2.7. Deriving surface tension for the immersed boundary method

The immersed boundary elements are Lagrangian, so the natural way of thinking about them is from a standard rigid body, Newtonian physics perspective. Calculate all forces acting on each segment, and move it according to the sum of these forces. In the most general way, every force acting on the immersed boundary must follow this framework:

$$\dot{\boldsymbol{u}}_i = \frac{1}{m_i} \sum \boldsymbol{F}_i, \tag{4.170}$$

where $\dot{\boldsymbol{u}}_i$ is the acceleration, m_i is the mass and $\boldsymbol{F}_i$ is the force force interface segment i. From section 3.2.1, we know that given two fluids surface tension is only a function of the shape of the surface. Thus, for a linear segment of the interface, the only variables are the pressure on each side, and the size and orientation of the segment. The pressure force is proportional to area and acts normal to the surface. Thus the force on each segment must be the area of the surface element times the difference in pressure

$$\boldsymbol{F}_i = (p_{1,i} - p_{2,i}) A_i \boldsymbol{n}_i, \tag{4.171}$$

where $\boldsymbol{n}_i$ is a unit-normal vector to the segment i, pointing towards phase 2. The surface elements themselves have a defined length, position, curvature, equilibrium length and equilibrium curvature. There is no notion of pressure for the interface itself. This makes physical sense, as the pressure is discontinuous over the interface and thus the interface pressure is not defined. To calculate the forces on the elements, an expression for the

pressure jump is needed. This is where eq. (3.21) comes in. Substituting it for the pressure in eq. (4.171) gives

$$\boldsymbol{F}_i = \sigma_i \kappa_i A_i \boldsymbol{n}_i. \tag{4.172}$$

Interestingly this is a discrete version of the expression arrived at in eq. (3.21), although a completely different path was taken to derive it. This shows how the immersed boundary method can provide a solid mathematical framework for describing *any* surface interacting with a fluid.

The only thing missing now is the density, which is simply

$$\frac{\rho_{1,i} + \rho_{2,i}}{2}. \tag{4.173}$$

The final equation which is the one used in the code as a force contribution in the Poisson equation is as follows

$$\dot{\boldsymbol{u}}_i = \frac{\sigma_i \kappa_i A_i}{1/2(\rho_{1,i} + \rho_{2,i})} \boldsymbol{n}_i. \tag{4.174}$$

4.2.8. Generalized viscoelastic interface for immersed boundary method

Stretching and compression is implemented using the fully general tension eq. (3.32) and eq. (3.30) relating this tension to a force density.

Let

$$\|\boldsymbol{X}\|_i^k = \|\boldsymbol{X}_k - \boldsymbol{X}_i\| \tag{4.175}$$

be the Euclidean distance between Lagrangian points i and k. Discretizing eq. (3.32) gives the following expression for the tension at point i:

$$T_i = K_{a,i} \left(\frac{\|\boldsymbol{X}\|_i^{i+1} + \|\boldsymbol{X}\|_{i-1}^i}{e_i + e_{i-1}} - 1 \right) + \sigma_i, \tag{4.176}$$

where e_i is the equilibrium length between point $\boldsymbol{X}_i$ and $\boldsymbol{X}_{i+1}$. $K_{a,i}$ is the spring constant, and σ_i the surface tension of segment i. Using this, the

tension for each Lagrangian point along the boundary is computed. This is then used in a discretized version of eq. (3.30),

$$\boldsymbol{f}_i = \frac{T_{i+1} - T_{i-1}}{2}\boldsymbol{\tau}_i + T_i \frac{\|\boldsymbol{X}\|_i^{i+1} + \|\boldsymbol{X}\|_{i-1}^{i}}{2}\kappa \boldsymbol{n}. \tag{4.177}$$

The equivalent code can be seen in listing 5.

One nice thing about this approach is that it encapsulates all the surface effects we need to simulate in one coherent framework. It is able to simulate constant and varying surface tension, as well as an elastic membrane effect. If, say, the elasticity of the material is a function of temperature, this is trivially added, only one line of code has to be changed. This way, new forces from physics can be added in an easy manner by modifying eq. (4.176).

4.2.9. CFL condition

Following [18, sec 3.8] we have the following convective CFL number

$$\mathrm{CFL}_c = \Delta t \left(\frac{|\boldsymbol{u}_x|}{\Delta_x} + \frac{|\boldsymbol{u}_y|}{\Delta_y} \right) \leq 1, \tag{4.178}$$

and a viscous CFL number given by

$$\mathrm{CFL}_v = \Delta t \left(\max\left(\frac{\mu_1}{\rho_1}, \frac{\mu_2}{\rho_2} \right) \left(\frac{2}{(\Delta_x)^2} + \frac{2}{(\Delta_y)^2} \right) \right) \leq 1. \tag{4.179}$$

An added force $\boldsymbol{f}$, e.g. surface tension, can be taken into account using the relation

$$\frac{\Delta t}{2} \left((\mathrm{CFL}_c + \mathrm{CFL}_v) + \sqrt{(\mathrm{CFL}_c + \mathrm{CFL}_v)^2 + \frac{4|\boldsymbol{f}_x|}{\Delta_x} + \frac{4|\boldsymbol{f}_y|}{\Delta_y}} \right) \leq 1 \tag{4.180}$$

From eq. (4.144) we have that

$$\boldsymbol{f} \propto \boldsymbol{F}\delta_\Delta \tag{4.181}$$

where $\boldsymbol{F}$ is a force density on the interface. As $\delta_\Delta \propto \frac{1}{\Delta_x}$ eq. (4.180) can be written

$$\frac{\Delta t}{2}\left((\mathrm{CFL}_c + \mathrm{CFL}_v) + \sqrt{(\mathrm{CFL}_c + \mathrm{CFL}_v)^2 + \frac{4|\boldsymbol{F}_x|}{(\Delta_x)^2} + \frac{4|\boldsymbol{F}_y|}{(\Delta_y)^2}}\right) \leq 1. \tag{4.182}$$

Equation (4.178), eq. (4.179) and eq. (4.182) are used in the code to determine the appropriate time step restriction when using dynamic time stepping. This helps a lot for simulating time dependent problems since several of the simulations have a big temporal variation in both convective and surface force CFL numbers, as potential energy from the immersed boundary is converted into kinetic energy in the flow field and vice versa. This calculation is done in the end of the function in listing 5

4.2.10. Computing the level-set function from the immersed boundary

When using both the immersed boundary method and the ghost-fluid method to calculate interface forces, special care has to be taken to make the methods consistent. The following technique is proposed. The geometry is completely determined by the Lagrangian points along the interface. In each sub time-step, the shortest distance from the Eulerian points to the Lagrangian boundary is computed. In other words, we compute the level-set function purely from the immersed boundary.

This has several advantages. First, advection is moved from the level-set function to the immersed boundary. When no advection of the level-set function is required, it is no longer needed to reinitialize it, eq. (4.52), or extrapolate the velocity, eq. (4.50). These routines are costly, and their saving leads to a 25% reduction in wall clock run time for two-phase simulations. Second, using this, the level-set function is always the best possible approximation to the exact signed distance function for the given Eulerian grid. Third, given equal initial conditions for the immersed boundary and the level-set field, the two descriptions of the interface will not be consistent with respect to each other, meaning that after some time, t, the advection of the level-set function and the Lagrangian points will

not be exactly the same [5]. This is highly problematic because the singular interface forces will appear at two different interfaces rather than one. This inconsistency disappears when reinitializing the level-set function from the immersed boundary at every sub-step.

The algorithm for reinitialization is as follows. For each line segment on the Lagrangian boundary, compute its bounding box. Grow the bounding box such that it contains the biggest stencil, 4δ. For each grid node inside the bounding box, compute the distance to the line segment and whether the point is inside or outside the closed interface. If the current distance is the smallest distance found, save it with sign according to if the grid cell is inside or outside the boundary. For point to line distance, well known formulas are used, and to calculate if a grid cell is inside or outside the boundary, a point-inside-polyhedron algorithm, commonly used in computer graphics is used. The idea behind the algorithm is beautifully simple: Given an arbitrary point that is either inside or outside a closed polyhedron. If one travels on a ray from infinity to the point, one will cross the surface of the polyhedron an odd number of times if it is inside the polyhedron, and an even number of times if it is not. Using this algorithm together with the distance to the line segment gives an algorithm for computing the level-set function in a narrow band around the immersed boundary. The overall algorithm can be seen in listing 6, the distance from line segment to point in listing 7 and the inside outside algorithm in listing 8.

4.2.11. Implementation details

When implementing the immersed boundary method, certain choices have to be made regarding where different quantities are stored in the discretization, much in resemblance to how vector and scalar quantities are stored on staggered and collocated grids in many CFD simulations.

[5] The reason for this is that the immersed boundary points can have sub grid details. This means that inside a grid-cell there will be differences between the level-set and the immersed boundary. Over time these will grow bigger than one grid cell because of advection. At this point the two interface descriptions are not consistent with each other.

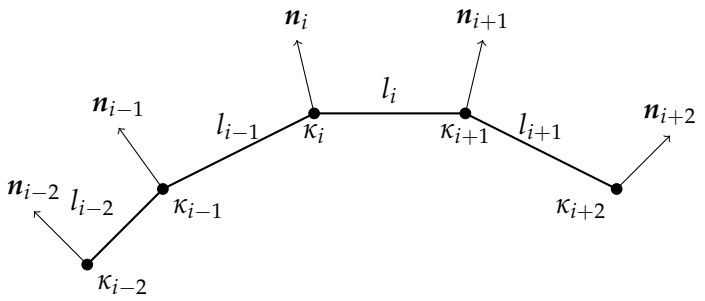


Figure 4.2.1.: Part of immersed boundary grid showing where different values are located.

Figure 4.2.1 shows the immersed boundary elements around index i together with their different properties, and where they are defined. The cubic spline fitted to the points is only evaluated at the knot points. Because of this curvature is only available at the nodes. The same applies to the normal vectors, which are directly calculated from the first derivative of the spline at the knot points. On the other hand, line segments are computed as the difference in position between two adjacent nodes. This means that lengths are defined on the segments, and not on the nodes. Because the grid is staggered, interpolation will have to take place at some point. Note that the cubic spline going trough the points is not shown here, although it is used to derive the curvature, κ, and the normal vectors $\boldsymbol{n}$.

One of the fundamental decisions of the implementation was the usage of cubic splines to generate a smooth analytic parametrization of the interface. The main advantage of this is that properties like derivatives, curvatures, tangents and normal vectors are all naturally defined for a cubic spline, while for a line segment, one has to resort to approximate difference formulas that span several nodes to recover the same values. The previously mentioned staggering can be seen in fig. 4.2.1. Unit normal vectors and curvature are defined at the nodes, while segment-lengths are derived as the difference in node position and thus defined on the

segments between the nodes. This leads to problems when computing forces on the boundary because all variables have to be collocated to compute the force. One apparent solution to this staggering would be to evaluate the curvature and normal vectors of the cubic spline not at the control points, but rather at the midpoints, with respect to arc-length, between each control point. In this way, all properties would be centered on the line segment midpoint, and the staggering would disappear. There is, however, a catch. To do this, knowledge of the length of the cubic spline is needed. Between the control points, the spline is parametrized as

$$f_i(t) = (x_i(t), y_i(t)), \tag{4.183}$$
$$x_i(t) = a_{ix} + b_{ix}t + c_{ix}t^2 + d_{ix}t^3, \tag{4.184}$$
$$y_i(t) = a_{iy} + b_{iy}t + c_{iy}t^2 + d_{iy}t^3, \tag{4.185}$$
$$\tag{4.186}$$

where $f(t = 0) = p_i$ and $f(t = 1) = p_{i+1}$. To find the midpoint, a mapping,

$$t = m\left(\frac{s}{s_i}\right), \tag{4.187}$$

is needed. Here s is an arc-length along the cubic spline from point i to $i + 1$, s_i is the total arc-length of the cubic spline between point i and $i + 1$ and $m(\frac{s}{s_i})$ is the value of t such that $p = f(m(\frac{s}{s_i}))$ is the midpoint with respect to arc-length along the spline. There are two issues with creating this mapping. First, calculating the arc length, s_i, requires evaluation of a non-trivial elliptic integral. Second, the mapping m is the inverse of this elliptic integral so that the arc length ratio can be mapped to the parameter t. Because of this, the idea was not pursued further as it would incur a considerable cost, both for implementation and computational time with only questionable gain in accuracy. After all, the existing framework is only of first order, calculating a higher order solution for the immersed boundary itself would not enhance the simulation as other discretization errors are dominating.

The end result was that splines are only evaluated at points $t = 0$, where they coincide with the control points, and thus have a priori known

positions. Lengths of line segments are derived from the length between control points, and not arc length on the spline, while curvature and unit normal vectors are derived analytically from the cubic spline. The calculation of this can be seen in listing 32.

4.3. Cubic splines

An arbitrary, closed, parametric curve in two dimensions can be written in the form

$$\gamma(s) = (x(s), y(s)),\ s \in [0, S] \tag{4.188}$$

where $x(s)$ and $y(s)$ are the x and y coordinates of the curve. s runs from the start of the curve, $s = 0$ to the end, $s = S$. Since the curve is closed

$$(x(0), y(0)) = (x(S), y(S)). \tag{4.189}$$

We see that if we can determine the one-dimensional functions $x(s)$ and $y(s)$, the two-dimensional curve is completely determined. Thus, following [3], it is enough to consider the one-dimensional problem.

Given $n + 1$ values, $\{y_0, y_1, \ldots, y_n\}$, at regular intervals, Δ_x, a cubic spline is a piece-wise third-order polynomial that goes through all the points, $(i\Delta_x, y_i)$, in order, has a smooth first derivative, and a continuous second derivative. For our purposes we can assume that $\Delta_x = 1$ which makes s run from i to $i + 1$ on spline segment i. For convenience we define a local parameterization,

$$t = s - i \ \ \in [0, 1]. \tag{4.190}$$

The ith segment of the spline is represented by

$$Y_i(t) = a_i + b_i t + c_i t^2 + d_i t^3 \tag{4.191}$$

where t is in the domain $[0, 1]$ and $i = 0, \ldots, n - 1$. Then

$$Y_i(0) = y_i = a_i, \tag{4.192}$$
$$Y_i(1) = y_{i+1} = a_i + b_i + c_i + d_i. \tag{4.193}$$

The derivative of $y_i(t)$ at each interval gives

$$Y_i'(0) = b_i = E_i, \tag{4.194}$$
$$Y_i'(1) = b_i + 2c_i + 3d_i = E_{i+1}, \tag{4.195}$$

where E_i and E_{i+1} are unknowns. Solving eqs. (4.192) to (4.195) for a_i, b_i, c_i, d_i we get

$$a_i = y_i \tag{4.196}$$
$$b_i = E_i \tag{4.197}$$
$$c_i = 3(y_{i+1} - y_i) - 2E_i - E_{i+1} \tag{4.198}$$
$$d_i = 2(y_i - y_{i+1}) + E_i + E_{i+1} \tag{4.199}$$

A requirement for the curve is that it is continuous and that the first and second derivatives should match at the knot points

$$Y_{i-1}(1) = y_i. \tag{4.200}$$
$$Y_i(0) = y_i, \tag{4.201}$$
$$Y_{i-1}'(1) = Y_i'(0), \tag{4.202}$$
$$Y_{i-1}''(1) = Y_i''(0). \tag{4.203}$$

Because of periodicity we have

$$Y_n(1) = y_0. \tag{4.204}$$
$$Y_0(0) = y_0, \tag{4.205}$$
$$Y_n'(1) = Y_0'(0), \tag{4.206}$$
$$Y_n''(1) = Y_0''(0). \tag{4.207}$$

This leaves us with $4n$ equations for $4n$ unknowns which can be ex-

pressed as a linear system

$$A\boldsymbol{E} = \boldsymbol{f}$$

$$\begin{bmatrix} 4 & 1 & & & & & 1 \\ 1 & 4 & 1 & & & & \\ & 1 & 4 & 1 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & 1 & 4 & 1 & \\ & & & & 1 & 4 & 1 \\ 1 & & & & & 1 & 4 \end{bmatrix} \begin{bmatrix} E_1 \\ E_2 \\ E_3 \\ \vdots \\ E_{n-2} \\ E_{n-1} \\ E_n \end{bmatrix} = 3 \begin{bmatrix} (y_2 - y_n) \\ (y_3 - y_1) \\ (y_4 - y_2) \\ \vdots \\ (y_{n-1} - y_{n-3}) \\ (y_n - y_{n-2}) \\ (y_1 - y_{n-1}) \end{bmatrix}. \tag{4.208}$$

By solving this linear system for $\boldsymbol{E}$ one can compute $Y_i(t)$ using eqs. (4.196) to (4.199) and eq. (4.191).

By creating two one-dimensional splines, $Y_i(t)$ and $X_i(t)$ and composing them

$$\gamma_i(t) = (X_i(t), Y_i(t)), \ t \in [0, 1] \tag{4.209}$$

we get a full two-dimensional cubic spline. The code performing the above operations and calculating the spline coefficients is listed in listing 26. For axisymmetric flow instead of a periodic spline a normal cubic spline is needed. The calculation of this is done in listing 27.

Curvature of a spline

As previously mentioned, the two-dimensional cubic spline is a parametric smooth curve where each segment of the curve, from (x_i, y_i) to (x_{i+1}, y_{i+1}), is parametrized by

$$\gamma_i(t) = (x_i(t), y_i(t)), \tag{4.210}$$
$$x_i(t) = a_{xi} + b_{xi}t + c_{xi}t^2 + d_{xi}t^3, \tag{4.211}$$
$$y_i(t) = a_{yi} + b_{yi}t + c_{yi}t^2 + d_{yi}t^3. \tag{4.212}$$

Being cubic polynomials, both the x and y directions have analytical

expressions for both the first and second derivatives given by

$$x_i(t)' = b_{xi} + 2c_{xi}t + 3d_{xi}t^2, \tag{4.213}$$
$$x_i(t)'' = 2c_{xi} + 6d_{xi}t. \tag{4.214}$$
$$\tag{4.215}$$

The signed curvature of a parametric curve, $\gamma(t) = (x(t), y(t))$, in two dimensions, is given by

$$\kappa = \frac{x'y'' - y'x''}{(x'^2 + y'^2)^{3/2}}. \tag{4.216}$$

Using this one can efficiently compute the curvature of the spline. In addition, if one is only interested in the curvature at the control points, then $t = 0$ and the derivatives are even simpler,

$$x_i' = b_{xi}, \tag{4.217}$$
$$x_i'' = 2c_{xi}. \tag{4.218}$$
$$\tag{4.219}$$

When simulating using axisymmetry, even though the computational domain is two dimensional, the physics is three dimensional. This changes how curvature is computed. For a two dimensional surface embedded in 3D space, a different measure of curvature is needed. This can easily be seen by plotting a saddle function. Depending on which direction one is looking, the curvature at the saddle point can be both positive, negative or zero. As we saw in the derivation of surface tension in section 3.2.1 in 3D we need to compute the mean curvature, which is the mean between the minimum and the maximum curvature at a point. Using [35] the following relation was found. Given a parametrization of a surface of revolution

$$x(\theta, t) = \phi(t)\cos(\theta) \tag{4.220}$$
$$y(\theta, t) = \phi(t)\sin(\theta) \tag{4.221}$$
$$z(\theta, t) = \Psi(t). \tag{4.222}$$

where t is the parameter along the spline and θ is the angle of rotation. The mean curvature is given by

$$\kappa = \frac{\phi\left(\frac{\partial^2\phi}{\partial t^2}\frac{\partial\Psi}{\partial t} - \frac{\partial\phi}{\partial t}\frac{\partial^2\Psi}{\partial t^2}\right) - \frac{\partial\Psi}{\partial t}\left(\frac{\partial\phi}{\partial t}^2 + \frac{\partial\phi}{\partial t}^2\right)}{2|\phi|\left(\frac{\partial\phi}{\partial t}^2 + \frac{\partial\Psi}{\partial t}^2\right)^{3/2}} \tag{4.223}$$

The code for calculating the curvature of a spline is listed in listing 29. Also, the alternative method of using an osculating circle is available in listing 30, but only used for comparison with the spline method.

Numerical performance of curvature estimate using cubic splines

A test was set up to evaluate the performance of the curvature calculation based on splines versus a naive three-point circle estimate. The most naive way of calculating the local curvature of a function is to take three successive points on the curve, draw the circle that is defined by those three points, calculate the radius of the circle, and then the curvature.

The radius of a circle that passes trough three points,

$$\mathcal{P} = \{(x_i, y_i), \quad i \in \{1,2,3\}\}, \tag{4.224}$$

is given by

$$r(\mathcal{P}) = \frac{\sqrt{\left((x_2-x_1)^2+(y_2-y_1)^2\right)\left((x_2-x_3)^2+(y_2-y_3)^2\right)\left((x_3-x_1)^2+(y_3-y_1)^2\right)}}{2\,|x_1\,y_2 + x_2\,y_3 + x_3\,y_1 - x_1\,y_3 - x_2\,y_1 - x_3\,y_2|}. \tag{4.225}$$

The curvature is defined as

$$\kappa = \frac{1}{r(\mathcal{P})}. \tag{4.226}$$

This method was compared to the more elaborate method using splines explained in section 4.3. The test function was chosen to be

$$y(x) = \sin(x), \quad x \in [0, \pi]. \tag{4.227}$$

Which has an exact solution for the curvature

$$\kappa = \frac{|y''|}{(1+y'^2)^{3/2}} = \frac{|-\sin(x)|}{(1+\cos^2(x))^{3/2}} = \frac{|\sin(x)|}{(1+\cos^2(x))^{3/2}}. \tag{4.228}$$

The function was discretized with n points, $n = 8 \times 2^i, \quad i \in 0 \ldots 9$, and the L_1, L_2 and L_∞ errors of the two methods were compared in fig. 4.3.1. We observe a difference in the error for the two methods, but it is not very pronounced. Interestingly the L_2 error is of order 2.5 while L_1 and L_∞ is of order 2. This test does not seem to justify the usage of splines, as it is much more complicated implementation wise.

The above tested function, $\sin(x)$, is a good example of a smooth, easy to approximate, function. The interfaces we are interested in, are not this smooth, they are crumpled, have dimples and kinks. Because of this a more demanding test function was investigated,

$$\sin\left(\frac{1}{x}\right), \quad x \in [0.2, 0.3], \tag{4.229}$$

which can be seen in fig. 4.3.2. The tricky part of this function is at $x \approx 0.21$. Here, the function is has a large curvature. This can be seen as a spike for the dashed line in fig. 4.3.2. The curvature was estimated using both a cubic spline and a three-point circle. The errors of the two estimates is shown in fig. 4.3.3. Here, there is a much more distinct difference between the circle and the cubic spline estimates. They are both of the same order, and the L_2 norm still shows a 2.5 order, but the relative difference in error almost 2 orders of magnitude. This much more clearly advocates the usage of splines. This conclusion was also strengthened later, in experiments, where a test case using three-point circle for estimating curvature would compute extreme curvature values making the simulation blow up, while cubic splines gave no such problems.

Modified Thomas Algorithm

For each sub-time step in the code, the curvature information is required to compute the forces from the immersed boundary on the fluid. For

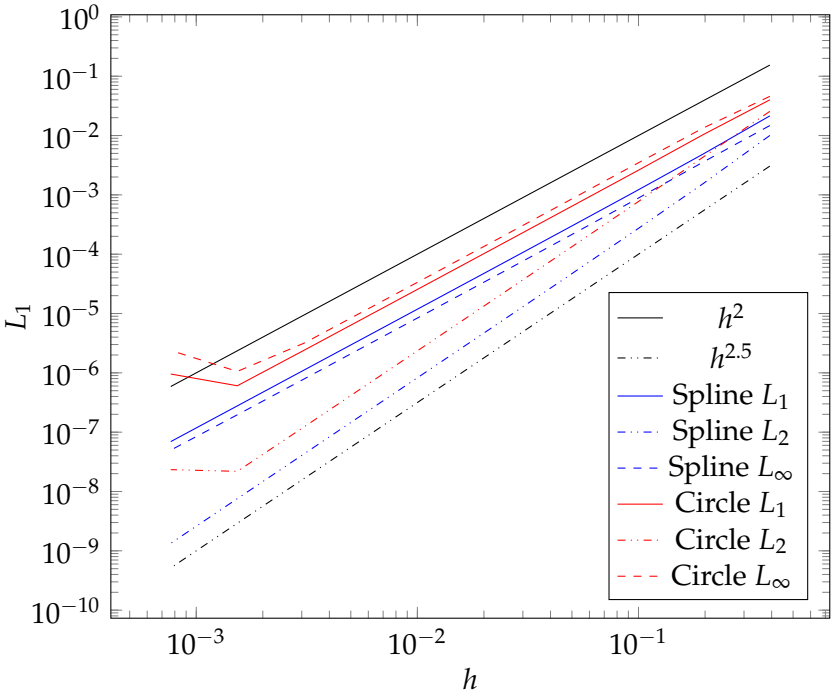


Figure 4.3.1.: Convergence of curvature estimates for the first test case, $\sin(x)$.

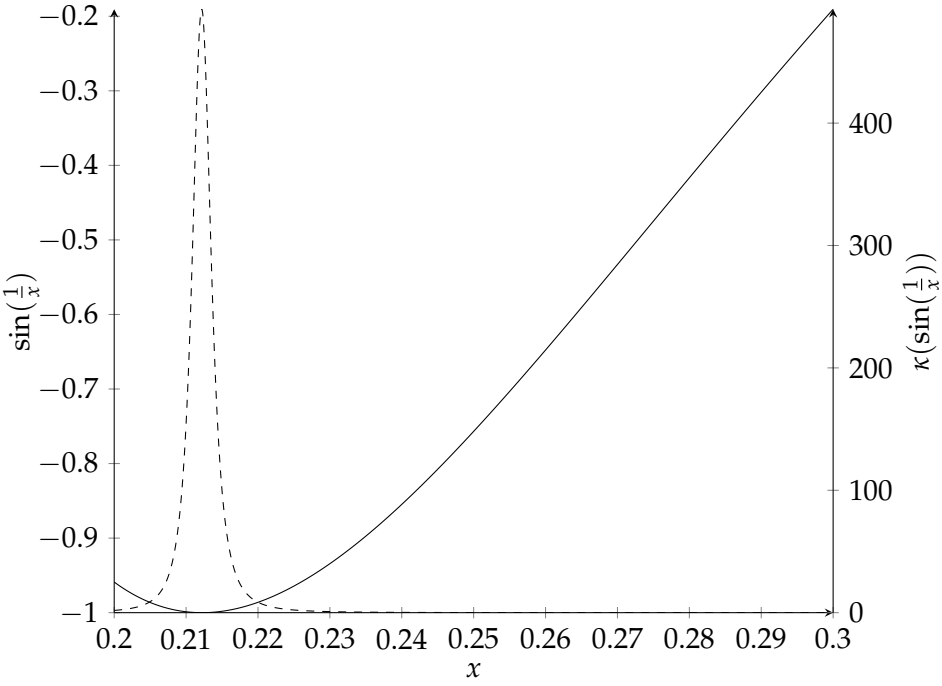


Figure 4.3.2.: The function $\sin(\frac{1}{x})$, solid line, and the curvature of $\kappa(y(x)) = \kappa(\sin(\frac{1}{x}))$, dashed line. Note the difference in scales for the left and right y-axis. The function has a large curvature around $x \approx 0.21$.

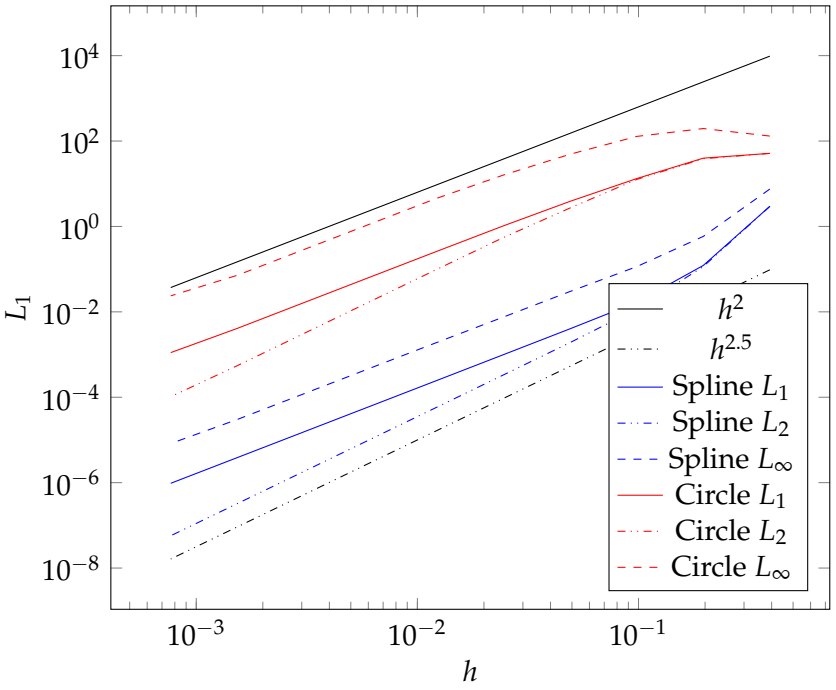


Figure 4.3.3.: Convergence of curvature estimates for the second test case, $\sin(\frac{1}{x})$. Figure 4.3.1 shows the function and its curvature.

accurate curvature estimates, as seen in the previous section, a cubic spline is needed. This means that a cubic spline has to be computed often, and an efficient algorithm is favourable.

Recall the linear system eq. (4.208). It is almost tridiagonal, but because of the periodicity it has a bandwidth equal to its dimension, N. Had it been tridiagonal it would have been straightforward to solve it utilizing the Thomas algorithm, forward-backward substitution, which is an $\mathcal{O}(N)$ algorithm. Luckily, for this special kind of periodic tridiagonal matrix, a reduction can be made [43, Algorithm 4, p. 320].

Given a symmetric, diagonally dominant, constant coefficient, periodic, system of the form

$$A\boldsymbol{x} = \boldsymbol{b}$$

$$\begin{bmatrix} \lambda & 1 & & & & & 1 \\ 1 & \lambda & 1 & & & & \\ & 1 & \lambda & 1 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & 1 & \lambda & 1 & \\ & & & & 1 & \lambda & 1 \\ 1 & & & & & 1 & \lambda \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-2} \\ x_{n-1} \\ x_n \end{bmatrix} = 3 \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_{n-2} \\ b_{n-1} \\ b_n \end{bmatrix}, \tag{4.230}$$

let E be the resulting matrix removing the first row and column of A. Define

$$\hat{\boldsymbol{f}} = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}, \quad \hat{\boldsymbol{x}} = \begin{bmatrix} x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix}, \quad \hat{\boldsymbol{b}} = \begin{bmatrix} b_2 \\ b_3 \\ \vdots \\ b_{n-1} \\ b_n \end{bmatrix}, \tag{4.231}$$

all being vectors of dimension $n-1$.

Now suppose that we know the value of x_1, then the reduced system

can be written

$$E\hat{\boldsymbol{x}} = \hat{\boldsymbol{b}} - \hat{\boldsymbol{f}}x_1 \tag{4.232}$$

$$\begin{bmatrix} \lambda & 1 & & & & \\ 1 & \lambda & 1 & & & \\ & 1 & \lambda & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & \lambda & 1 \\ & & & & 1 & \lambda \end{bmatrix} \begin{bmatrix} x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} b_2 \\ b_3 \\ \vdots \\ b_{n-1} \\ b_n \end{bmatrix} - \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} x_1. \tag{4.233}$$

This is a tridiagonal system that can be solved efficiently with the Thomas Algorithm. The only thing stopping us is the unknown value of x_1.

Let $\boldsymbol{r} = (r_1, r_2, \ldots, r_n)$ denote the first row of A^{-1}. Given this x_1 is computable by the dot product

$$x_1 = \boldsymbol{r} \cdot \boldsymbol{b} \tag{4.234}$$

$$x_1 = \sum_{i=1}^{n} r_i b_i \tag{4.235}$$

Because A is a symmetric periodic matrix, so is A^{-1}, thus

$$r_i = r_{n+2-i}, \quad 2 \leq i \leq n. \tag{4.236}$$

Let $m = \lfloor \frac{n+1}{2} \rfloor$, the floor of $\frac{n+1}{2}$, then eq. (4.235) can be rewritten

$$x_1 = r_1 b_1 + \sum_{i=2}^{m} r_i (b_i + b_{n+2-i}), \qquad \text{if } n \text{ is odd} \tag{4.237}$$

$$x_1 = r_1 b_1 + r_{m+1} b_{m+1} + \sum_{i=2}^{m} r_i (b_i + b_{n+2-i}), \qquad \text{if } n \text{ is even} \tag{4.238}$$

Define

$$\alpha = \frac{-\lambda + \operatorname{sgn}(\lambda)\sqrt{\lambda^2 - 4}}{2}, \tag{4.239}$$

$$\sigma = \frac{1}{(\alpha + \alpha^{n-1}) + \lambda(1 + \alpha^n)}, \tag{4.240}$$

where $\text{sgn}(x)$ is the sign of x.

It can be shown that

$$r_{i+1} = \sigma(\alpha^i + \alpha^{n-i}), \tag{4.241}$$

Now we can first use eq. (4.241) to calculate the first row of A^{-1}, then use eq. (4.235) to compute x_1, then lastly solve the linear system eq. (4.232). This gives us the full solution to a periodic, symmetric, constant coefficient, tridiagonal system in $\mathcal{O}(N)$ flops, where N is the number of knot-points.

Different versions of the above algorithm were implemented as stand-alone functions in Fortran, listing 50, listing 51 and listing 52. These were then tested against the default linear system solver in Matlab and a reference implementation of the above algorithm in Matlab. All the algorithms where in agreement down to machine epsilon.

4.4. The proposed method

We have now reached the point where we can assemble all the theory and numerics into one complete coherent simulation of a drop with density and viscosity jumps with respect to the bulk fluid, as well as variable surface tension and an elastic membrane.

We start off by discretizing our domain as explained in section 4.1.1. The simulation is incompressible. Thus we can use Chorin's projection method, section 4.1.2, to get a Poisson equation for pressure, split the effect of pressure from advection and viscosity, and thereby lower the computational cost. In time we integrate using the SSP 2-2 Runge-Kutta method, section 4.1.3. When static solid objects are wanted in the domain, the penalization method is used, section 4.1.4. The geometry of the interface is described by Lagrangian points, following the immersed boundary method, section 4.2. The forces from the boundary are modeled as a source term, eq. (3.12), of the Navier-Stokes equation. These forces are computed by two different methods. The effect of the density and viscosity jump is calculated by the ghost-fluid method, section 4.1.6. In section 4.2.10 it is explained how the level-set field given to the GFM is calculated purely from the immersed boundary enabling simultaneous use of immersed

boundary and GFM. Given the nature of GFM, the force from the jumps is computed in a sharp manner. The forces from the surface tension and elastic membrane are computed on the immersed boundary, as explained in sections 4.2.6 to 4.2.8 . From eq. (4.144) we see that these forces are spread to the Eulerian grid in a smooth way using a delta function. Because of this, we have a hybrid GFM immersed-boundary method having both sharp and smooth interface discretizations. As will be shown in chapter 5, this gives results consistent with existing simulations and is able to simulate additional physics from the elastic membrane that was previously not possible.

5. Numerical results

In this section the results from the numerical simulations performed for this master's thesis is presented. First, the advection of the immersed boundary method is compared with analytic advection. Then the immersed boundary method is compared with the level-set method. Simulating surface tension with the two methods is compared. Then viscosity and density jumps are added and the effect of these are discussed. Lastly, simulations with an elastic membrane is done. An elliptical drop with elastic membrane relaxing to equilibrium driven by surface tension is studied. The effect of an elastic membrane when stretching a drop in an electric field is shown. The last simulation is an effort to reproduce the crumpling effect seen in lab experiments.

5.1. Analytic advection

The routines responsible for advecting the immersed boundary are important. The general entry point for calculating advection is listing 4. From there listing 9 takes over, and the details are contained in listing 11. These routines correspond to eq. (4.145). To be certain that they were implemented correctly, and to investigate their order of error, a test case was created.

The setup is seen in fig. 5.1.1. The initial configuration is a circle of radius 0.1 centered at (½, ½) in a 1×1 domain. The velocity field is given as

$$\boldsymbol{u}(x, y) = [-(x - 1/2), y - 1/2]. \tag{5.1}$$

Two separate solutions for the advection are computed. The first, reference method, performs forward Euler integration of the velocity. The velocity is computed by evaluating eq. (5.1) at each Lagrangian point. In the

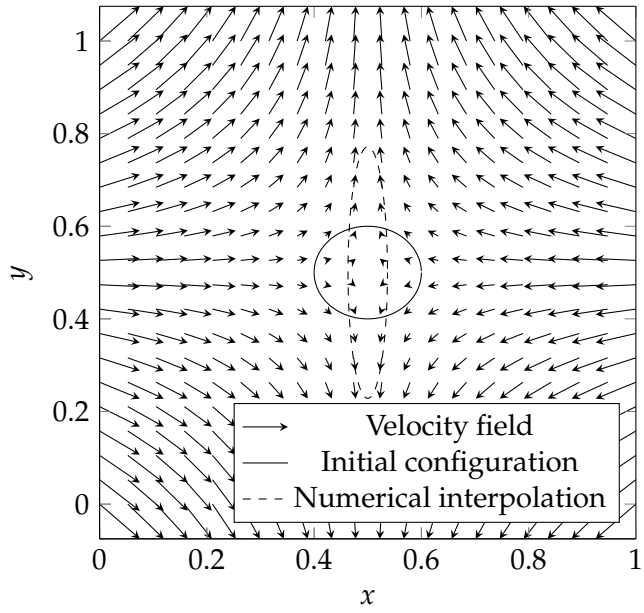


Figure 5.1.1.: Initial and end configuration in analytical advection test. The solid black line is the initial boundary, the arrows show the velocity field, and the dashed ellipse shows the end configuration after advecting the boundary.

second method, the domain is discretized using a $N \times N$ grid, and a discrete velocity field is computed by evaluating eq. (5.1) at each grid node. To compute the velocity at a Lagrangian point, the immersed-boundary interpolation equation, eq. (4.145), is used. This interpolated velocity is then forward Euler integrated, for each point, in time with the same time step as for the prescribed velocity field. The goal of this test is quantifying how big the error is when interpolating a discretized field versus using the analytical one in the context of time integration. For the test N was chosen small, $N = 4$, to amplify any errors. The integration was 1000 time steps at $\Delta t = 0.01\,\mathrm{s}$. The result from this test was that the two methods of calculating the velocity and thus also advecting the boundary were equal down to machine precision. The reason for this lies in the chosen velocity field, eq. (5.1), being linear. This means that a 1st or higher order interpolation of the velocity field will be exact. Because of this, eq. (4.145) is exact for this field.

Although being exact for linear fields is nice, this result does not reveal the error. To do this, the velocity field

$$\boldsymbol{u}(x, y) = [\cos(x), \sin(y)] \tag{5.2}$$

was used. Figure 5.1.2 shows this configuration. Again, 1000 time steps were taken with $\Delta t = 5 \times 10^{-4}$ to keep the advected particles inside the domain.

The error between the interpolated and the analytically evolved boundary was recorded for different grid resolutions $N = \{4, 8, 16, 32, 64, 128\}$. Figure 5.1.3 shows how the interpolation introduces a second-order error. Chorin's projection method, section 4.1.2, has a first-order splitting error in time. With this in mind it is safe to say that a second-order advection error is sufficient, and using the immersed boundary method should not degrade the order of the solution.

5.2. Drop in vortex

A standard test of advection for interface tracking methods is the drop in a potential vortex [25]. Here a drop is placed in the unit box, and a static

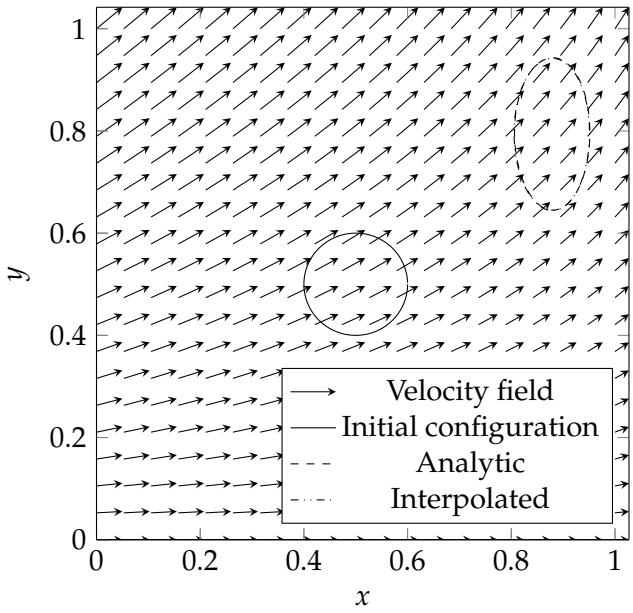


Figure 5.1.2.: Initial and end configuration in analytic advection test. The solid black line is the initial boundary, the arrows show the advection field, and the dashed ellipse shows the end configuration after advecting the boundary. The error is almost invisible.

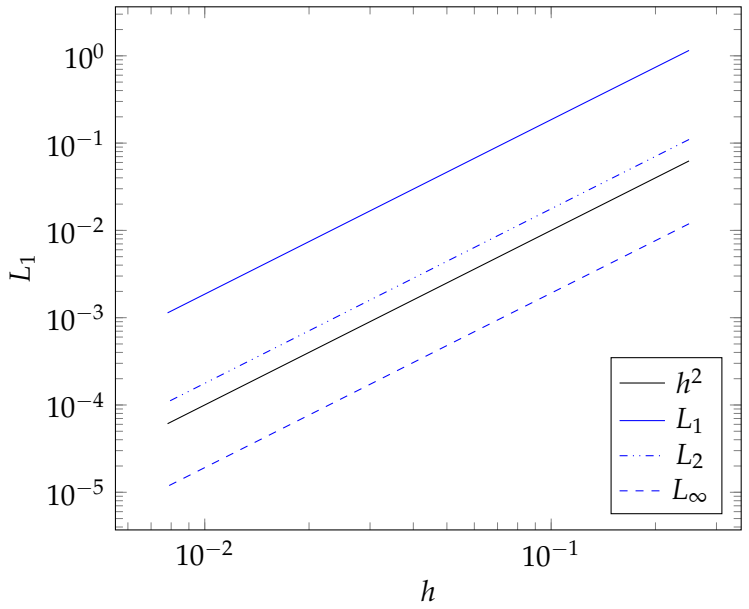
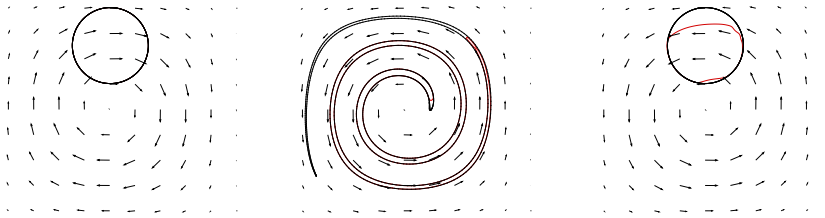


Figure 5.1.3.: Error for advection. L_1, L_2 and L_∞ error of the error stemming from interpolating a velocity field rather than using the analytical expression when preforming time integration. The interpolation is second order after time integration. Here $h = 1/N$.

Table 5.1.: Parameters for the drop in potential vortex.

Parameter	Symbol	Value
Drop radius	r	0.15
Domain size	Ω	1×1
CFL		0.5
Euler grid nodes	N	200×200
Lagrangian point density		$5/\Delta$



(a) $t = 0$ (b) $t = 3.5$ (c) $t = 7$

Figure 5.2.1.: Drop in potential vortex. Red is level-set solution while black is the immersed boundary.

potential vortex advects it. The velocity field is given by

$$\boldsymbol{u}_x = -2(\sin(\pi(x - 1/2)))^2 \cos(\pi(y - 1/2)) \sin(\pi(y - 1/2)) \tag{5.3}$$
$$\boldsymbol{u}_y = -2(\cos(\pi(y - 1/2)))^2 \sin(\pi(x - 1/2)) \cos(\pi(x - 1/2)) \tag{5.4}$$

The rest of the parameters for this test are in table 5.1. At some time $t = \frac{t_e}{2}$, the flow field is reversed, and the simulation is run until $t = t_e$. Then the initial interface is compared with the final one. Figure 5.2.1 shows the initial condition (a), the interface at half time (b) where $t = \frac{t_e}{2} = 3.5$, and the final interface for both the level-set and the immersed boundary method. We see that the immersed boundary method has no visible mass loss, while the level-set representation loses mass when the drop gets stretched thinner than a grid cell. The reason for this is that when

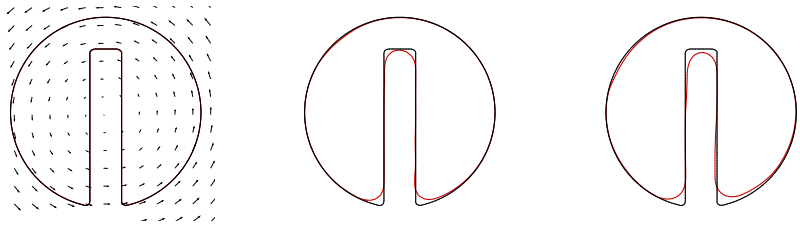
two interfaces are this close, the discrete level-set function does not have the required resolution to switch sign. The immersed boundary method does not have this restriction. If one wanted to represent the smaller features with level-set representation, one choice would be to double the grid resolution. For two dimensions this would make the computational cost increase quadratically. To get the same increase in resolution with the immersed boundary method, one would need to double the number of points, this would only double the amount of work needed. Thus immersed boundary scales considerably better than level-set as a function of interface resolution. The previous argument makes immersed boundary seem superior to level-set when it comes to resolution. However this is not the whole story. For the immersed boundary to represent a non-smooth sub-grid feature, the Lagrangian points have to be advected in a sub-grid way. From eq. (4.134) we see that the Lagrangian points are advected using a delta function interpolation of the velocity field from the Eulerian grid. This means that the highest wave number that can be created by the immersed boundary is proportional to $1\Delta x$. With this in mind, for non smooth velocity fields, we can conclude that for each Eulerian grid, there exists an optimal Lagrangian point density. For smooth velocity fields, like this potential vortex, the immersed boundary method has some sub grid resolution. This is because it can accurately represent stretching, squishing and other smooth transformations that lead to sub grid details. Another way of thinking about this is that the immersed boundary method has some sub grid detail as long there is no turbulence.

5.3. Zalesak's disk

Another interesting difference between an Eulerian and Lagrangian description of geometry is the effect of grid alignment. For a Lagrangian description, the grid is by definition aligned with the geometry. For the level-set method, this is not the case. As long as the interface is smooth, the level set can accurately represent it, but in the presence of corners, the interface is not resolved sharply unless the corner is aligned with the grid. This effect can be seen in the next test, Zalesak's disk [46]. Here, a

Table 5.2.: Parameters for the Zalesak's disk test.

Parameter	Symbol	Value
Disk radius	r	$1/3$
Domain size	Ω	1×1
CFL		0.5
Euler grid nodes	N	64×64
Velocity field	$\boldsymbol{u}$	$\frac{\pi}{10}[1/2 - y, x - 1/2]$
Lagrangian point density		$5/\Delta$



(a) $t = 0$ (b) $t = 40$ (c) $t = 80$

Figure 5.3.1.: Zalesak's disk for 0, 1 and 2 revolutions. Red shows level-set interface while black shows immersed boundary. The velocity field is constant in time and represents pure rotation.

slotted disk is put in a velocity field that has constant angular velocity. The boundary is advected one or more revolutions and the result is inspected.

From fig. 5.3.1 it is clear that the immersed boundary resolves the rotated disk better than the level-set function. During the rotation, information is lost in the level-set, while the immersed boundary is virtually not affected. The reason for this is that the level set, based on an Eulerian grid, cannot represent nonsmooth features that are not aligned with the grid perfectly. This means that over the duration of the rotation, small errors in the interface position creeps in as a consequence of the interface not being straight and aligned with the grid. In the immersed boundary method, the

grid has no preference about the orientation of the interface. In the drop-in-vortex test, section 5.2, it was argued that the difference between the two methods were exaggerated by the specifics of the test, very thin stretched parts of the drop disappear. There does not seem to be any such argument for Zalesak's disk. The immersed boundary method is fundamentally better at preserving non-smooth features like corners without smearing. In real life, non smooth interfaces appear several places, e.g. when two drops meet and coalesce.

5.4. Comparison with reference method

We have now verified that the immersed boundary method captures the interface correctly under advection. Next we need to verify that the forces from the boundary on the fluid are implemented correctly. The forces from the viscosity and density jumps must also be verified to be correctly coupled with the immersed boundary. The technique chosen for this was to compare the proposed method, section 4.4, with a reference method, the level-set method with the ghost fluid method, which had previously been verified to be correct [42], [23], [22], [9] and [10]. The main reason for using a reference method, instead of an analytical solution, is that as far as the author knows, analytical solutions for relaxing drops with large eccentricity, viscosity and density jumps are not known. To measure the drops, the horizontal and vertical axis lengths are used. This stems from the usage of Fiji [39] in the experimental setup. Conveniently the top/bottom and left/right sides of the drops also are the part of the drop that is most rapidly advected, with highest pressure differences and sharpest curvature, thus, if there is any difference, it would be most pronounced here. All simulations were done with zero gravity and a Lagrangian point density of 5 per Δ. Unless otherwise stated simulations are in two-dimensions. The stretching of a drop in an electric field was only done for axisymmetric flow. Crumpling was only done in two-dimensions because crumpling as a phenomenon is not axisymmetric, it would thus violate the assumptions of axisymmetric flow to simulate crumpling.

Table 5.3.: Parameters for the elliptical drop driven by surface tension.

Parameter	Symbol	Value
Drop density	ρ_1	$10^3\,\mathrm{kg/m^3}$
Matrix density	ρ_2	$10^3\,\mathrm{kg/m^3}$
Drop viscosity	μ_1	$10^{-3}\,\mathrm{Pa\cdot s}$
Matrix viscosity	μ_2	$10^{-3}\,\mathrm{Pa\cdot s}$
Surface tension	σ	$15\times10^{-3}\,\mathrm{N/m}$
Drop radius	r	$10^{-3}\,\mathrm{m}$
Drop axis length ratio	$\frac{a}{b}$	1.16
Domain size	Ω	$0.007\times0.007\,\mathrm{m}$
CFL		0.2
Grid nodes	N	$\{100, 200, 400, 800\}$

5.4.1. Immersed boundary-driven surface tension

In this test, an elliptical drop is relaxing to its equilibrium, a sphere, driven by surface tension. The purpose of this test is to verify that surface tension simulated with the immersed boundary gives the same results as when simulated with the level-set method. To isolate the effect of surface tension, no density or viscosity jumps are present. There is also zero gravity. This way, all forces are generated by surface tension as it drives the ellipse to equilibrium. The parameters of the test are listed in table 5.3. The densities and viscosities are equal in each phase. This way the only force generated comes from surface tension. The test was run for increasing grid resolution to see how the two methods compare under refinement. The result for 2D and axisymmetric simulation can be seen in fig. 5.4.1 and fig. 5.4.2 respectively. The length of the ellipse axis as a function of time can be seen in fig. 5.4.1 and fig. 5.4.2. We see that under grid refinement, the proposed method converges to the same answer as the previously verified method. This indicates that both the theoretical work deriving surface tension for the immersed boundary, and the implementation of the proposed method have been done correctly. For low grid resolution,

there is a visible difference in the two results. This stems from the main difference between the two methods. The reference method resolves surface tension using the ghost fluid method, which is a sharp-interface method. The proposed method uses a smeared delta function, resulting in a diffuse interface spanning a distance of 4δ. As the grid resolution is increased, the diffuse interface approximates a sharp interface closer, which can be seen in fig. 5.4.1 and fig. 5.4.2. The level-set method with the ghost-fluid method is the most correct for this simulation, as the sharp interface works better.

5.4.2. Relaxing ellipse with density and viscosity jump

The previous test shows that the reference method converges to the same solution as our proposed method for a relaxing ellipse driven by surface tension. There is, however, no jump in density or viscosity in this test. As elaborated in section 4.2.10 the proposed method will treat density and viscosity jumps in a sharp fashion, while surface tension will be diffuse. If the proposed level-set reinitialization from immersed boundary does not work, or there is a flaw in the assumptions that it is possible to use both sharp and diffuse interface forces at a same time, adding a viscosity and density jump should uncover these.

A simulation with the same geometry as in section 5.4.1, of a relaxing ellipse driven by surface tension was set up. Instead of equal density and viscosity, the relative density jump was 2, and the relative viscosity jump was 10. The simulation was run on a moderately fine grid, $N = 400$, which showed good agreement between the two methods in the previous test. The full set of parameters for the simulation are listed in table 5.4. These parameters correspond roughly to a water drop in oil. The simulations were run both for 2D and axisymmetric flow for both methods.

As seen in fig. 5.4.3 the two methods are in agreement both for two-dimensional and axisymmetric flow. This shows that the usage of both sharp and diffuse interface forces in the proposed method as well as the implementation works consistently with the reference method which only has sharp interface forces.

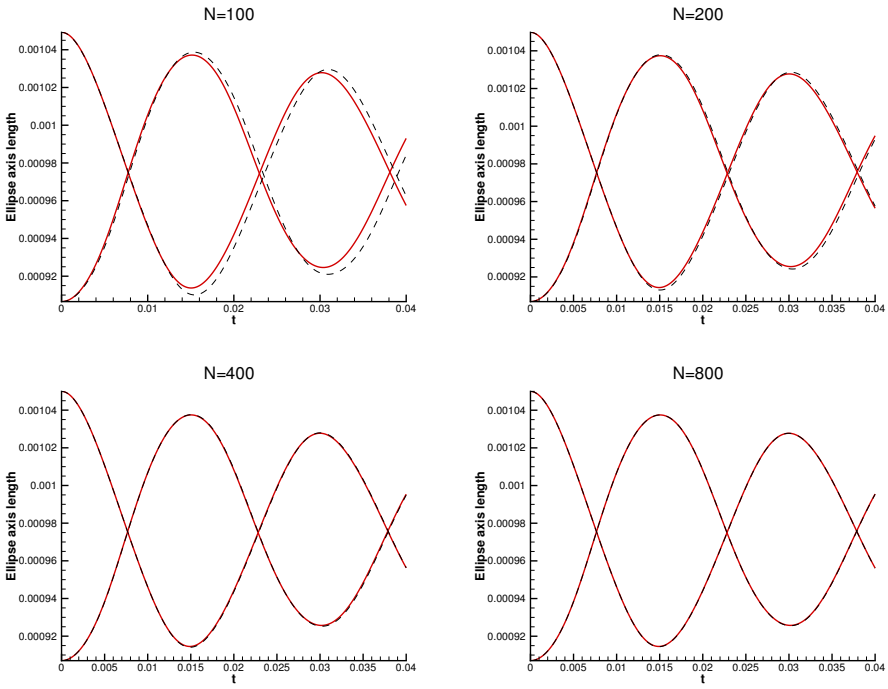


Figure 5.4.1.: Drop axis lengths for the two-dimensional relaxing drop, section 5.4.1. Red is the reference solution, dashed black is immersed boundary solution. The two methods converge as the grid is refined.

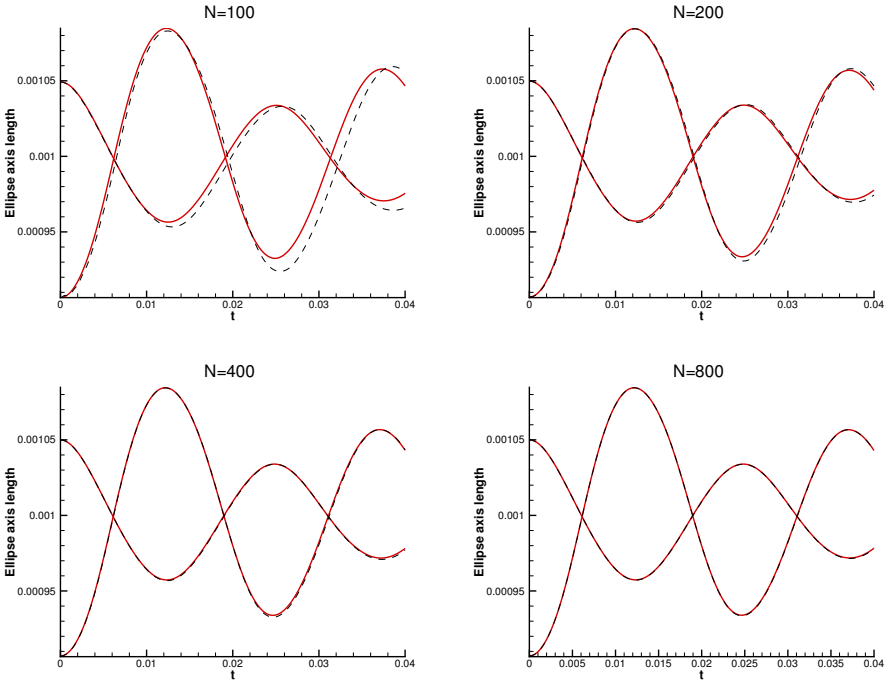
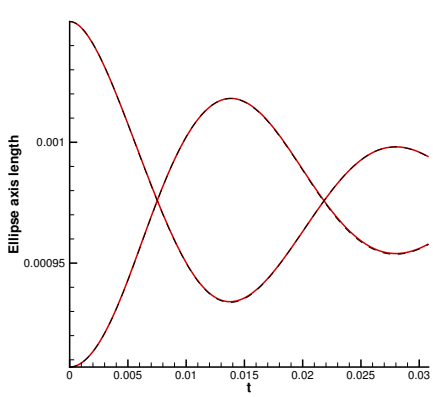


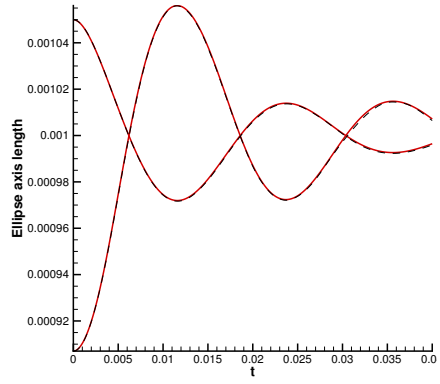
Figure 5.4.2.: Drop axis lengths for the axisymmetric, three-dimensional relaxing drop, section 5.4.1. Red is the reference solution, dashed black is immersed boundary solution. The two methods converge as the grid is refined.

Table 5.4.: Parameters for relaxing drop with viscosity and density jump.

Parameter	Symbol	Value
Drop density	ρ_1	$10^3\,\mathrm{kg/m^3}$
Matrix density	ρ_2	$5 \times 10^2\,\mathrm{kg/m^3}$
Drop viscosity	μ_1	$10^{-3}\,\mathrm{Pa \cdot s}$
Matrix viscosity	μ_2	$10^{-2}\,\mathrm{Pa \cdot s}$
Surface tension	σ	$15 \times 10^{-3}\,\mathrm{N/m}$
Drop radius	r	$10^{-3}\,\mathrm{m}$
Drop axis length ratio	$\frac{a}{b}$	1.16
Domain size	Ω	$0.007 \times 0.007\,\mathrm{m}$
CFL		0.2
Grid nodes	N	400



(a) Two-dimensional drop.



(b) Three-dimensional, axisymmetric drop.

Figure 5.4.3.: Comparison of reference method and proposed method with a viscosity and density jump, section 5.4.2. Red is reference method while dashed black is proposed method.

5.4.3. Effect of adding sharp forces on the diffuse interface

A hypothesis was that if the proposed method correctly simulated density and viscosity jumps together with surface tension, the difference between the proposed method and the reference method should be smaller than if simulating something only driven by surface tension. The rationale behind this hypothesis is that if only simulating surface tension, the proposed method is a fully diffuse interface method, while the reference method is fully sharp. When adding density and viscosity jumps, sharp forces are added to the proposed method, these will not suffer the same problems as the diffuse interface. This means that the relative amount of diffuse interface effects decreases for the proposed method when introducing density and viscosity jumps. It thus makes sense that the effects from the diffuse interface would be less pronounced for a simulation with density and viscosity jumps.

To test this hypothesis, a coarse grid of the above test was compared to a coarse simulation with only surface tension. The result can be seen in fig. 5.4.4. As can be seen, the higher viscosity and density of the surrounding liquid damps the oscillation, in other words the physics have changed. From the figure there is a smaller error when viscosity and density jumps are present. Still one cannot conclude that the method is better with jumps, as this is not an apples to apples comparison. Even with this limitation it is reassuring to see that adding more forces to the method, does not degrade the solution. The oscillation frequency has a pronounced improvement when adding the sharp effects of density and viscosity.

This test shows that the proposed method consistently combines the surface tension from the diffuse interface, with the sharp forces from viscosity and density jumps. This is interesting, because it enables the implementation of interface forces in the Lagrangian formulation as well as in an Eulerian. Having this flexibility can be beneficial for a multiphysics code as different phenomena are more naturally expressed in either Eulerian or Lagrangian coordinate systems.

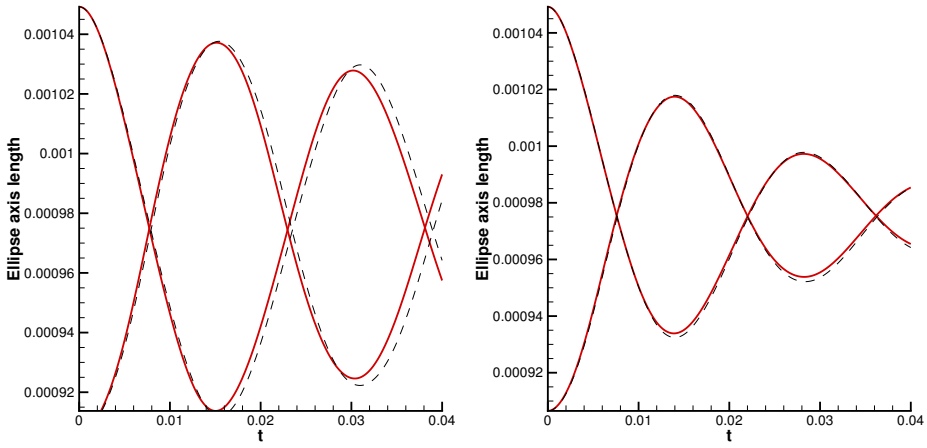


Figure 5.4.4.: Comparison with and without viscosity and density jump, section 5.4.3. Left figure shows relaxation driven purely by surface tension, right shows relaxation with a jump in viscosity and density. Red is reference method, dashed black is the proposed method.

Table 5.5.: Parameters for relaxing drop with an elastic membrane.

Parameter	Symbol	Value
Drop density	ρ_1	$10^3\,\mathrm{kg/m^3}$
Matrix density	ρ_2	$10^3\,\mathrm{kg/m^3}$
Drop viscosity	μ_1	$10^{-3}\,\mathrm{Pa\cdot s}$
Matrix viscosity	μ_2	$10^{-3}\,\mathrm{Pa\cdot s}$
Surface tension	σ	$15\times10^{-3}\,\mathrm{N/m}$
Elastic tension	K_a	0 and $15\times10^{-2}\,\mathrm{N/m}$
Drop radius	r	$10^{-3}\,\mathrm{m}$
Drop axis length ratio	$\frac{a}{b}$	3.0
Domain size	Ω	$0.007\times0.007\,\mathrm{m}$
CFL		0.5
Grid nodes	N	150

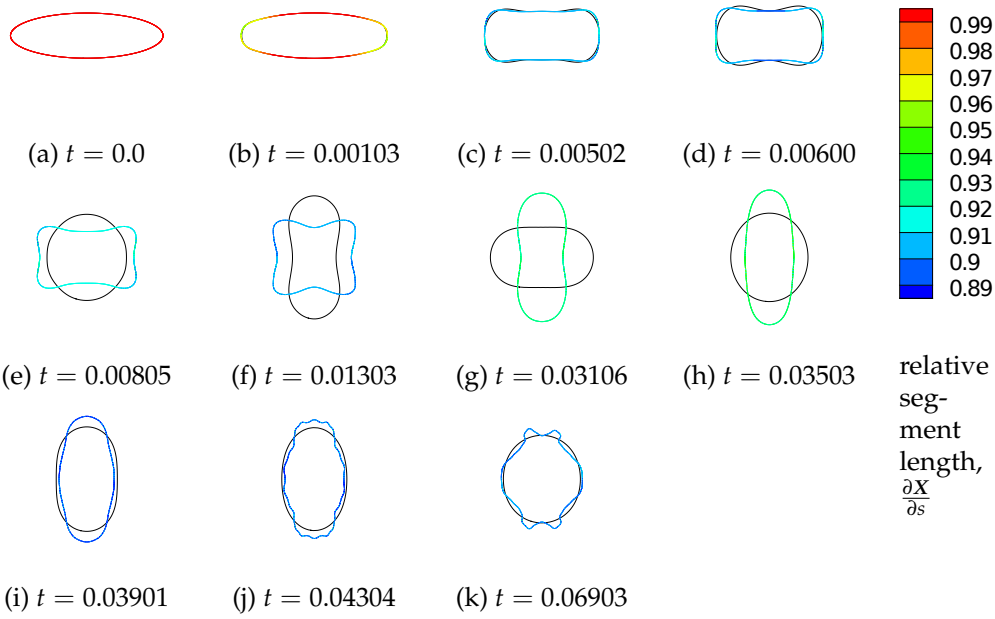
5.5. Simulations with general interface tension

After verifying that the proposed method can simulate surface tension, viscosity and density jumps we are now in a position where we can simulate the elastic membrane, which was not previously possible. With this we are leaving the classical area of research about drops with constant surface tension and little material exist on the analytical or approximate solutions. Because of this we rely on qualitatively replicating experiments, rather than exact or reference solutions.

5.5.1. Relaxing drop with elastic membrane

The first test of the effect of an elastic membrane is simply comparing the previous simulation with one having an elastic membrane. In this test, an ellipse with a surface membrane is set to relax under surface tension in a box. The parameters for this test are in table 5.5. Initially , at $t = 0$, the elastic membrane is in equilibrium. This means that $\frac{\partial \boldsymbol{X}}{\partial s} = 1$ in eq. (3.32) and the membrane is neither stretched nor compressed.

The interface starts in equilibrium, red, in fig. 5.5.1a. Because of its



(a) $t = 0.0$ (b) $t = 0.00103$ (c) $t = 0.00502$ (d) $t = 0.00600$

(e) $t = 0.00805$ (f) $t = 0.01303$ (g) $t = 0.03106$ (h) $t = 0.03503$

(i) $t = 0.03901$ (j) $t = 0.04304$ (k) $t = 0.06903$

Figure 5.5.1.: Several frames of the simulation with elastic membrane, (colored) together with the clean interface, (black). The colors indicate the relative length of the interface compared to its equilibrium length.

eccentricity, surface tension is relatively strong on the left and right side of the drop and it is quickly compressed, fig. 5.5.1b. After $5 \times 10^{-}3\,\mathrm{s}$ the drop is almost fully compressed fig. 5.5.1c. In this simulation K_a is 10 times larger than than σ, this means that eq. (3.32) will be close to zero when $\frac{\partial X}{\partial s} \approx 0.9$. In other words, when the membrane is compressed to 90% of its original length, elastic forces and surface tension forces will be in balance. This means that the interface no longer introduces any force and modulo any viscosity or density differences the simulation can be considered single phase! As there is a quite strong flow field present in fig. 5.5.1c, advection of the interface continues. The part of the potential kinetic energy that is not dissipated by viscosity goes into deforming and again stretching the interface. At $t = 3.5 \times 10^{-}2$, fig. 5.5.1g, this potential energy has stretched the interface maximally again, and the flow field is close to zero. Now there is not enough potential energy in the membrane to do another oscillation and it is critically damped. The interface contracts creating a crumpled drop as seen in fig. 5.5.1h to fig. 5.5.1i. As this has happened the normal drop in black has oscillated towards its equilibrium shape, a spherical. To see the difference the elastic membrane makes for the oscillation, the axis lengths of the two simulations were plotted in fig. 5.5.2. It is clear that for these parameters, the elastic membrane has a significant effect on the time evolution of the drop, dampening its response. The elastic membrane introduces a new potential energy to the system. For clean fluids, the equilibrium interface is always the one that has the minimal interface area. The elastic potential changes this equilibrium to a more chaotic and unpredictable one. The equilibrium state is no longer obvious given the initial conditions. One insight from this simulation is that for a clean drop without an elastic membrane, there exists a unique spherical equilibrium state, only given by the initial volume of the drop. On the other hand, for the drop with an elastic membrane, the equilibrium is not just a function of the initial volume, but also its shape. This is because the initial shape affects what parts of the drop is stretched and compressed which has a big impact on the final steady state. This shows how the evolution of a drop with an elastic membrane is more complex than one without.

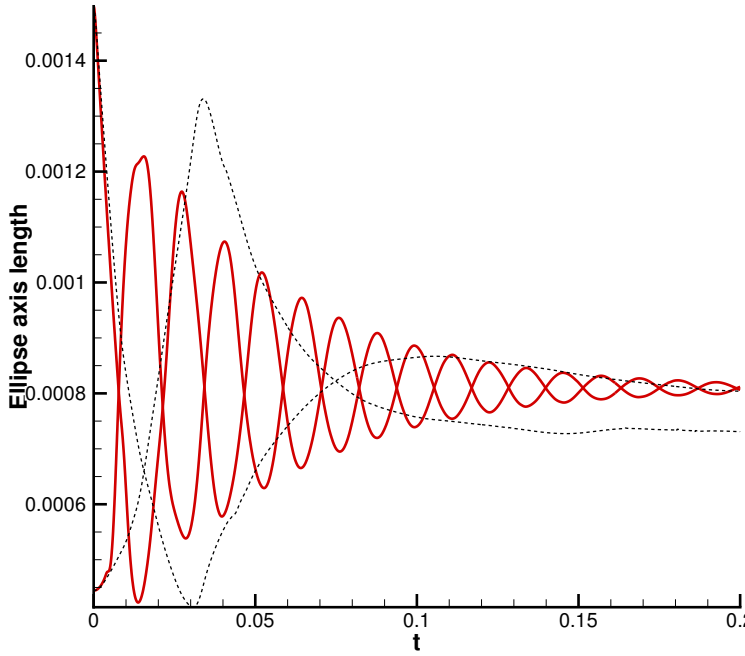


Figure 5.5.2.: Red is clean interface, dashed black is drop with elastic membrane. The elastic membrane dampens the oscillations.

Table 5.6.: Parameters for drop stretched in electric field.

Parameter	Symbol	Value
Drop density	ρ_1	$1000\,\mathrm{kg/m^3}$
Matrix density	ρ_2	$830\,\mathrm{kg/m^3}$
Drop viscosity	μ_1	$1.03 \times 10^{-3}\,\mathrm{Pa \cdot s}$
Matrix viscosity	μ_2	$12.4 \times 10^{-3}\,\mathrm{Pa \cdot s}$
Surface tension	σ	$40 \times 10^{-3}\,\mathrm{N/m}$
Elastic tension	K_a	0 and $50 \times 10^{-3}\,\mathrm{N/m}$
Drop radius	r	$4.5 \times 10^{-4}\,\mathrm{m}$
Drop axis length ratio	$\frac{a}{b}$	1.0
Domain size	Ω	$0.001 \times 0.002\,\mathrm{m}$ (axisymmetric)
CFL		0.5
Grid nodes	N	100×200 (axisymmetric)

5.5.2. Drop stretched in electric field

For the electrocoalescence process, it is interesting to see what the elastic membrane does to a drop stretched in an electric field. To do this, a simulation with a spherical drop, in zero gravity, was set up. Then an electric field was applied to the drop. This sets up dipole moments in the water and creates a force stretching the drop. For the elasticity, $K_a = 50 \times 10^{-3}$ which is $5/4 \times \sigma$, is used. The rest of the parameters of the test are summarized in table 5.6. The measured property was the ratio of the horizontal and vertical ellipse axis. A comparison with, and without elastic membrane was done. From fig. 5.5.3 we see that the elastic membrane does not visibly effect the eigenfrequency of the drop. However, it does make the drop harder to stretch. Also note that the steady state solution with elastic membrane is more spherical than without. Conclusions about what consequences this has for electrocoalescence is left as further work, here it is sufficient to state that there is an effect.

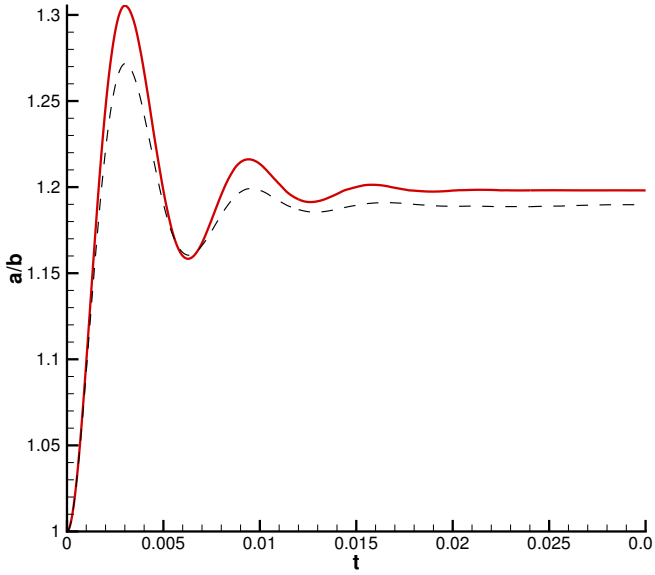


Figure 5.5.3.: Drop stretched in electric field test. Showing drop axis ratio as a function of time. Red is solution with $K_a = 0$, dashed black is with $K_a = 50 \times 10^{-3}$.

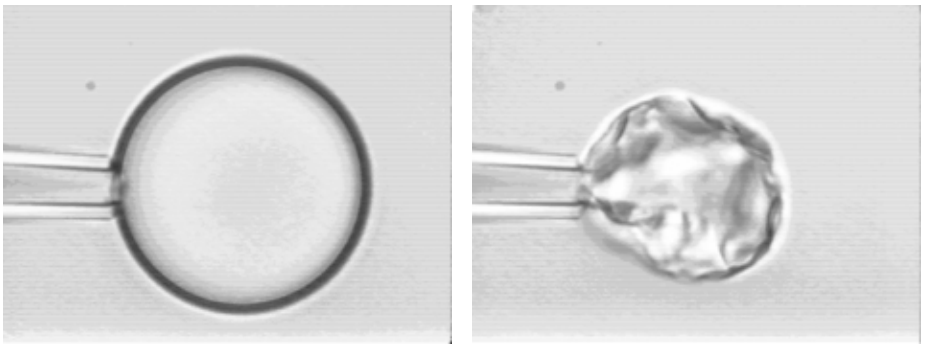


Figure 5.5.4.: Left, the initial water drop, $r \approx 25 \times 10^{-6}$ m. Right, drop after draining some of its volume using the pipette. Images from [45].

5.5.3. Pipette draining a water drop in crude-oil

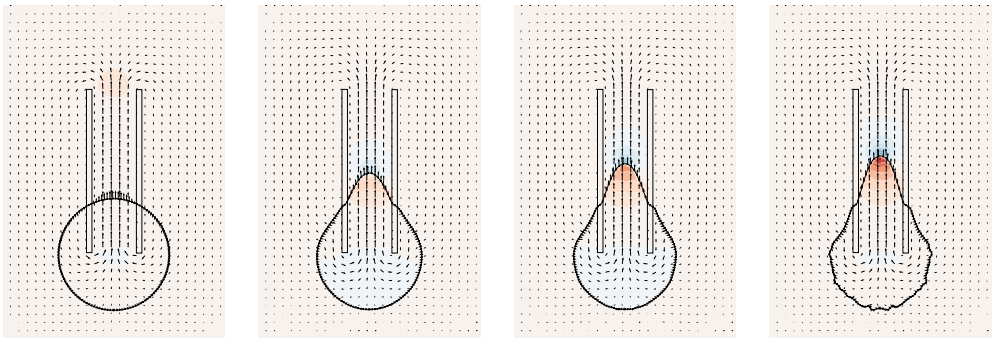
The last test with an elastic membrane is an effort to reproduce the effect seen in [45]. There, a bitumen (asphalt), extracted from crude oil, is added to a solvent consisting of one part n-heptane and one part toluene. Then a water drop with $r \approx 25 \times 10^{-6}$ m is inserted into the bulk using a micropipette. The drop is aged and then drained using the pipette while observed under a microscope. Instead of shrinking spherically, as expected, the drop crumples as seen in fig. 5.5.4.

To simulate this experiment the proposed method, section 4.4, was used. To simulate the pipette walls, and to enforce the flow trough the pipette tube, the penalization method, section 4.1.4, was used. The result can be observed in fig. 5.5.5. The parameters for the simulation was are listed in table 5.7. The parameters are representative for a water drop in a model crude oil. As long as the membrane is compressed less than $1 - \frac{\sigma}{K_a}$ the interface is still driven by surface tension. During this phase the drop will shrink spherically, as tension builds up. When the interface is compressed enough, the tension in the membrane will be zero, and crumples will appear. To accelerate the simulation the membrane was pre-tensioned close to $1 - \frac{\sigma}{K_a}$. This is equivalent to an initial condition where some of

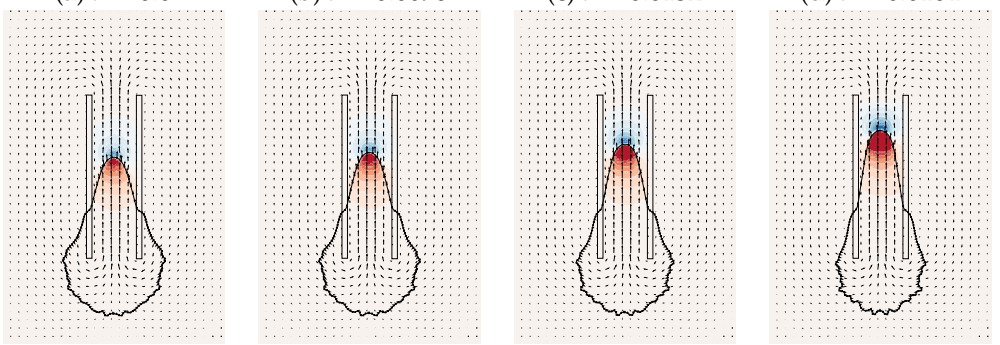
Table 5.7.: Parameters for the pipette draining drop case.

Parameter	Symbol	Value
Drop density	ρ_1	$1000\,\mathrm{kg/m^3}$
Matrix density	ρ_2	$830\,\mathrm{kg/m^3}$
Drop viscosity	μ_1	$1.03 \times 10^{-3}\,\mathrm{Pa \cdot s}$
Matrix viscosity	μ_2	$12.4 \times 10^{-3}\,\mathrm{Pa \cdot s}$
Surface tension	σ	$40 \times 10^{-3}\,\mathrm{N/m}$
Elastic tension	K_a	$50 \times 10^{-3}\,\mathrm{N/m}$
Drop radius	r	$5 \times 10^{-4}\,\mathrm{m}$
Domain size	Ω	$(2 \times 10^{-3}) \times (3 \times 10^{-3})\,\mathrm{m}$
CFL		0.5
Grid nodes	N	132×200
Penalization	η	5×10^{-6}

the drop mass already has been removed. Because of this, a relatively small amount of mass had to be removed from the drop to induce the crumpling. There is a good qualitative similarity between fig. 5.5.5 and fig. 5.5.4. Clearly the physics inside the pipette is not correct, as contact angle handling has been done. Also the trick of enforcing the penalization domain across a fluid interface is questionable. The numerical methods are being pushed to their limit, nevertheless the crumpling seems qualitatively correctly captured, which is very interesting.



(a) $t = 0.0$ (b) $t = 0.0096$ (c) $t = 0.0132$ (d) $t = 0.0164$



(e) $t = 0.0202$ (f) $t = 0.0202$ (g) $t = 0.0238$ (h) $t = 0.0300$

Figure 5.5.5.: Frames from the simulation of a micropipette draining a water drop. Color denotes pressure; red s high, blue is low. Velocities are plotted for every 5th grid point and every 10th Lagrangian point is plotted. The simulation bears a good qualitative resemblance to the photographs in fig. 5.5.4.

6. Concluding remarks

6.1. Conclusion

The goal of this thesis was to develop an immersed boundary method for simulating contaminated fluid interfaces in two-phase flow. The immersed boundary method was chosen for its history of simulating biological systems, e.g. blood flow trough a heart [33], which share many properties with the drops in question. A thorough derivation of interface forces was given, and then extended using the mathematical tools of the immersed boundary formulation to describe a general viscoelastic membrane. The projection method for simulating the Navier-Stokes equations were reviewed, both in space, time and how to split the pressure from viscous and advective effects. Interface capturing with the level-set method and handling of interface discontinuities with the ghost fluid method where also discussed.

The mathematical framework of the immersed boundary method was reviewed. The discretization as well as practical implementation details of the method were discussed. Specific contributions are the usage of cubic splines to parametrize the boundary. To the author's knowledge, the approach of using the immersed boundary to generate a level-set function that is consistent in such a way that it enables simulating interface forces from both a sharp Eulerian method, as well as the diffuse immersed boundary method, is novel.

The developed code has been verified by unit testing, appendix A, and full case testing. The numerical results show the expected performance for known test cases and the ability to reproduce experiments not previously possible. For the new method, combining the strengths of the immersed boundary method and the level-set method has allowed to simulate the dynamics of elastic membranes covering water drops immersed

in crude oil. The elastic membranes represent interfaces contaminated by surfactants. This was used to reproduce crumpling of a drop, showing good qualitative resemblance to the results seen in lab experiments [45]. Other work reproducing the crumpling phenomenon seen in [45] using simulation has not been found and to the author's knowledge, this is new. Based on the experiences from this master's thesis, the immersed boundary method is a good choice for simulation of contaminated fluid interfaces in two-phase flow.

6.2. Future work

During the work on this thesis, new questions have emerged that needs to be addressed in future work. First, the expression for tension used in this project is a Hookean linear law, eq. (3.32). As always, a linearization is only a good approximation close to the linearization point. As stated in chapter 2, the exact details of the forces acting on the interface are not known and because of this, it is uncertain if the linear model is accurate enough to represent the interface behaviour of water drops in oil correctly. Further investigation could involve devising microscopic experiments measuring the interfacial properties. Another approach, which could bring more insight into the mechanics of these interfaces, would be new and accurate molecular dynamics simulations of the interfaces. One of the author's supervisors, Åsmund Ervik, has started this work [11] using molecular dynamics software to simulate the interface on a nanometer level. The preliminary results are promising, but a lot of work still has to be done with regards to modeling of the surface active molecules in the crude.

Another limitation of the model used in this thesis is the assumption that there is no exchange of interfacially-active molecules between the bulk fluid and the interface. To get a full understanding of the electrocoalescence process, this will have to be addressed. The author hypothesizes that as a crude representation, this diffusion can be modeled as a one-parameter model where the equilibrium length of the immersed boundary segments is computed by low-pass filtering the instantaneous length. The

diffusion then determines the time constant of the low-pass filter. More advanced models would include an added Eulerian field holding the surfactant concentration and a diffusion equation between the immersed boundary and this field.

Last, even if section 4.2.1 proves that the level-set method alone cannot simulate compression or stretching of the interface, this does not mean that it is impossible in an Eulerian formulation. To do this, a new Eulerian field would have to be added. This field would represent the interface density. At each time step, this field would be advected, in a compressible way, by the tangential part of the velocity field, $\boldsymbol{u}_{\parallel}$. This tangential part can be found using the projection operator derived in section 4.2.1. To handle deformation of the interface, a reinitialization routine similar to the velocity extrapolation, eq. (4.50), could be used. This would have some advantages in that it would allow usage of the ghost-fluid method for sharp interface forces, and would again unify all the methods into an Eulerian framework. The downside would be simulations with crumpled interfaces and kinks. For these simulations, the immersed boundary method would probably still be better, because of its ability to represent the sub-grid features and more stable curvature estimates on these features. Further work could look into the implementation details of such an Eulerian routine.

Bibliography

[1] D. Adalsteinsson and J. Sethian. "A Fast Level Set Method for Propagating Interfaces". In: *Journal of Computational Physics* 118.2 (1995). cited By 0, pp. 269–277. DOI: 10.1006/jcph.1995.1098 (cit. on p. 25).

[2] P. Angot, C.-H. Bruneau, and P. Fabrie. "A penalization method to take into account obstacles in incompressible viscous flows". In: *Numerische Mathematik* 81.4 (1999), pp. 497–520. DOI: 10.1007/s002110050401 (cit. on pp. 22, 24).

[3] R. H. Bartels, J. Beatty, and B. Barsky. *An Introduction to Splines for Use in Computer Graphics and Geometric Modeling*. Morgan Kaufmann Series in Computer Graphics and Geometric Modeling. Morgan Kaufmann, 1995. ISBN: 9781558604001 (cit. on p. 55).

[4] G. Batchelor. *An Introduction to Fluid Dynamics*. Cambridge Mathematical Library. Cambridge University Press, 2000. ISBN: 9780521663960 (cit. on p. 5).

[5] E. Bjørklund. "The level-set method applied to droplet dynamics in the presence of an electric field". In: *Computers & Fluids* 38.2 (2009), pp. 358 –369. ISSN: 0045-7930. DOI: 10.1016/j.compfluid.2008.04.008 (cit. on p. 14).

[6] G. Carbou and P. Fabrie. "Boundary layer for a penalization method for viscous incompressible flow". In: *Advances in Differential Equations* 8.12 (2003). cited By 0, pp. 1453–1480. ISSN: 10799389. URL: http://projecteuclid.org/euclid.ade/1355867981 (visited on 06/28/2015) (cit. on p. 24).

[7] A. Chorin and J. E. Marsden. *A Mathematical Introduction to Fluid Mechanics*. Springer, 2000. ISBN: 9780387904061. DOI: 10.1007/978-1-4684-0082-3 (cit. on p. 18).

[8] R. Clift, J. Grace, and E. Weber. *Bubbles, Drops, and Particles*. Dover Civil and Mechanical Engineering. Dover Publications, 2013. ISBN: 9780486317748 (cit. on p. 1).

[9] Åsmund. Ervik. "The local level-set extraction method for robust calculation of geometric quantities in the level-set method". MA thesis. NTNU, 2012. URL: http://daim.idi.ntnu.no/masteroppgaver/008/8230/masteroppgave.pdf/ (visited on 06/28/2015) (cit. on pp. 2, 77).

[10] Åsmund. Ervik, K. Lervåg, and S. Munkejord. "A robust method for calculating interface curvature and normal vectors using an extracted local level set". In: *Journal of Computational Physics* 257, Part A.0 (2014), pp. 259 –277. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2013.09.053 (cit. on pp. 2, 77).

[11] Ervik et al. *A Multiscale Coarse-Grained Molecular and Continuum Model for the Flow of Droplets in Oil With Asphaltenes*. English. 2015. URL: http://folk.ntnu.no/asmunder/petrophase2015.html#/ (visited on 06/28/2015) (cit. on p. 96).

[12] T. Fan, J. Wang, and J. Buckley. "Evaluating Crude Oils by SARA Analysis". In: cited By 0. 2002, pp. 883–889. DOI: 10.2118/75228-MS (cit. on p. 3).

[13] R. P. Fedkiw and X. D. Liu. *The Ghost Fluid Method for Viscous Flows*. Presented at the "Solutions of PDE" Conference in honour of Prof. Phil Roe. 1998. URL: http://physbam.stanford.edu/~fedkiw/papers/cam1998-44.pdf (visited on 06/28/2015) (cit. on p. 26).

[14] R. P. Fedkiw et al. "A Non-oscillatory Eulerian Approach to Interfaces in Multimaterial Flows (the Ghost Fluid Method)". In: *Journal of Computational Physics* 152.2 (1999), pp. 457 –492. ISSN: 0021-9991. DOI: 10.1006/jcph.1999.6236 (cit. on p. 26).

[15] E. Hansen. “Numerical simulation of droplet dynamics in the presence of an electric field”. PhD thesis. NTNU, 2005. URL: http://www.diva-portal.org/smash/get/diva2:370048/FULLTEXT01.pdf (visited on 06/28/2015) (cit. on p. 14).

[16] J. S. Herbert Goldstein Charles Poole. *Classical Mechanics*. 3rd. Springer, 2000. ISBN: 9780201657029 (cit. on p. 34).

[17] A. A. Herod, K. D. Bartle, and R. Kandiyoti. “Comment on a Paper by Mullins, Martinez-Haya, and Marshall "Contrasting Perspective on Asphaltene Molecular Weight. This Comment vs the Overview of A. A. Herod, K. D. Bartle, and R. Kandiyoti"”. In: *Energy & Fuels* 22.6 (2008), pp. 4312–4317. DOI: 10.1021/ef8006036 (cit. on p. 3).

[18] M. Kang, R. P. Fedkiw, and X.-D. Liu. “A Boundary Condition Capturing Method for Multiphase Incompressible Flow”. In: *Journal of Scientific Computing* 15 (2000), pp. 323–360. ISSN: 0885-7474. DOI: 10.1023/A:1011178417620 (cit. on pp. 14, 50).

[19] M. Kang, R. Fedkiw, and X.-D. Liu. “A Boundary Condition Capturing Method for Multiphase Incompressible Flow”. English. In: *Journal of Scientific Computing* 15.3 (2000), pp. 323–360. ISSN: 0885-7474. DOI: 10.1023/A:1011178417620 (cit. on p. 26).

[20] D. I. Ketcheson and A. C. Robinson. “On the practical importance of the SSP property for Runge-Kutta time integrators for some common Godunov-type schemes”. In: *International Journal for Numerical Methods in Fluids* 48.3 (2005), pp. 271–303. ISSN: 1097-0363. DOI: 10.1002/fld.837 (cit. on p. 22).

[21] L. Landau and E. Lifshitz. *Fluid Mechanics*. v. 6. Elsevier Science, 2013. ISBN: 9781483140506 (cit. on p. 8).

[22] K. Lervåg. “Calculation of the interface curvatures with the level-set method for two-phase flow simulations and a second-order diffuse-domain method for elliptic problems in complex geometries”. PhD thesis. NTNU, 2013. URL: http://www.diva-portal.org/smash/get/diva2:649166/FULLTEXT01.pdf (visited on 06/28/2015) (cit. on pp. 2, 77).

[23] K. Lervåg. "Simulation of two-phase flows with varying surface tension". MA thesis. NTNU, 2008. URL: http://ntnu.diva-portal.org/smash/get/diva2:348658/COVER01 (visited on 06/28/2015) (cit. on pp. 2, 14, 77).

[24] S. Less et al. "Electrostatic destabilization of water-in-crude oil emulsions: Application to a real case and evaluation of the Aibel VIEC technology". In: *Fuel* 87.12 (2008), pp. 2572 –2581. ISSN: 0016-2361. DOI: 10.1016/j.fuel.2008.03.004 (cit. on p. 2).

[25] R. J. Leveque. "High-Resolution Conservative Algorithms for Advection in Incompressible Flow". English. In: *SIAM Journal on Numerical Analysis* 33.2 (1996), pp. 627–665. ISSN: 00361429. DOI: 10.1137/0733033 (cit. on p. 71).

[26] P. Mark and L. Nilsson. "Structure and dynamics of liquid water with different long-range interaction truncation and temperature control methods in molecular dynamics simulations". In: *Journal of Computational Chemistry* 23.13 (2002), pp. 1211–1219. ISSN: 1096-987X. DOI: 10.1002/jcc.10117 (cit. on p. 8).

[27] A. b. Marshall and R. b. Rodgers. "Petroleomics: The Next Grand Challenge for Chemical Analysis". In: *Accounts of Chemical Research* 37.1 (2004). cited By 307, pp. 53–59. DOI: 10.1021/ar020177t (cit. on p. 3).

[28] R. Mittal and G. Iaccarino. "Immersed boundary methods". In: *Annu. Rev. Fluid Mech.* 37 (2005), pp. 239–261. DOI: 10.1146/annurev.fluid.37.061903.175743 (cit. on p. 32).

[29] S. Osher and R. Fedkiw. *Level Set Methods and Dynamic Implicit Surfaces*. Applied Mathematical Sciences. Springer New York, 2012. ISBN: 9781468492514 (cit. on pp. 18, 21, 25).

[30] S. Osher and J. A. Sethian. "Fronts propagating with curvature-dependent speed: Algorithms based on Hamilton-Jacobi formulations". In: *Journal of Computational Physics* 79.1 (1988), pp. 12 –49. ISSN: 0021-9991. DOI: 10.1016/0021-9991(88)90002-2 (cit. on p. 24).

[31] V. Pauchard, J. P. Rane, and S. Banerjee. "Asphaltene-Laden Interfaces Form Soft Glassy Layers in Contraction Experiments: A Mechanism for Coalescence Blocking". In: *Langmuir* 30.43 (2014). PMID: 25330092, pp. 12795–12803. DOI: 10.1021/la5028042 (cit. on p. 4).

[32] C. S. Peskin. "The immersed boundary method". In: *Acta Numerica* 11 (Jan. 2002), pp. 479–517. ISSN: 1474-0508. DOI: 10.1017/S0962492902000077. URL: http://journals.cambridge.org/article_S0962492902000077 (cit. on pp. 31, 32, 46, 47).

[33] C. Peskin. "Numerical analysis of blood flow in the heart". In: *Journal of Computational Physics* 25.3 (1977). cited By 1276, pp. 220–252. DOI: 10.1016/0021-9991(77)90100-0 (cit. on pp. 31, 95).

[34] C. Peskin and D. McQueen. "A general method for the computer simulation of biological systems interacting with fluids". In: *Symposia of the society for Experimental Biology*. Vol. 49. cited By 58. 1995, pp. 265–276. URL: http://www.math.nyu.edu/~mcqueen/Public/papers/seb/SEB_19971216/SEB_19971216.html (visited on 06/28/2015) (cit. on p. 11).

[35] C. Pozrikidis. *Introduction to Theoretical and Computational Fluid Dynamics*. Oxford University Press, USA, 2011. ISBN: 9780199909124 (cit. on p. 58).

[36] C. M. Rhie and W. L. Chow. "Numerical study of the turbulent flow past an airfoil with trailing edge separation". In: *AIAA Journal* 21.11 (1983), pp. 1525–1532. DOI: 10.2514/3.8284 (cit. on p. 16).

[37] R. b. c. f. Rodgers, T. d. Schaub, and A. e. Marshall. "PETROLEOMICS: MS returns to its roots". In: *Analytical Chemistry* 77.1 (2005). cited By 0, 20 A–27 A. DOI: 10.1021/ac053302y (cit. on p. 3).

[38] Y. Saad. *Iterative Methods for Sparse Linear Systems*. 2nd. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003. ISBN: 0898715342 (cit. on p. 19).

[39] J. Schindelin et al. "Fiji: an open-source platform for biological-image analysis". In: *Nat Meth* 9.7 (July 2012), pp. 676–682. ISSN: 1548-7091. DOI: 10.1038/nmeth.2019 (cit. on p. 77).

[40] J. Shen. "On Error Estimates of the Projection Methods for the Navier-Stokes Equations: Second-Order Schemes". English. In: *Mathematics of Computation* 65.215 (1996), pp. 1039–1065. ISSN: 00255718. URL: http://www.jstor.org/stable/2153791 (visited on 06/28/2015) (cit. on p. 21).

[41] SINTEF. *Compact Separation by Electrocoalescence*. 2014. URL: http://www.sintef.no/home/projects/sintef-energy-research/Compact-Separation-by-Electrocoalescence-/ (visited on 06/28/2015) (cit. on p. 2).

[42] K. Teigen. "Development and use of interface-capturing methods for investigation of surfactant-covered drops in electric fields". PhD thesis. NTNU, 2010. URL: http://www.diva-portal.org/smash/get/diva2:370048/FULLTEXT01.pdf (visited on 06/28/2015) (cit. on pp. 2, 77).

[43] C. Temperton. "Algorithms for the solution of cyclic tridiagonal systems". In: *Journal of Computational Physics* 19.3 (1975), pp. 317–323. ISSN: 0021-9991. DOI: 10.1016/0021-9991(75)90081-9 (cit. on p. 64).

[44] USGS. *Organic Origins of Petroleum*. 2015. URL: http://energy.usgs.gov/GeochemistryGeophysics/GeochemistryResearch/OrganicOriginsofPetroleum.aspx (visited on 06/28/2015) (cit. on p. 4).

[45] A. Yeung et al. "On the interfacial properties of micrometre-sized water droplets in crude oil". In: *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 455.1990 (1999), pp. 3709–3723. ISSN: 1364-5021. DOI: 10.1098/rspa.1999.0473 (cit. on pp. ii, iii, 91, 96).

[46] S. T. Zalesak. "Fully multidimensional flux-corrected transport algorithms for fluids". In: *Journal of Computational Physics* 31.3 (1979), pp. 335 –362. ISSN: 0021-9991. DOI: 10.1016/0021-9991(79)90051-2 (cit. on p. 75).

[47] H.-K. Zhao et al. "A Variational Level Set Approach to Multiphase Motion". In: *Journal of Computational Physics* 127.1 (1996), pp. 179 –195. ISSN: 0021-9991. DOI: 10.1006/jcph.1996.0167. URL: http://www.sciencedirect.com/science/article/pii/S0021999196901679 (cit. on p. 25).

A. Coding conventions

As this master's thesis was an implementation project code was written. This code will go into the upstream SINTEF repository and available for future simulations. Because of this, the author early decided to take the issue of code quality seriously. The following goals were set, from general to more specific.

- Clearly communicate any assumptions the code makes in comments.
- In the documentation, not only say what something is, but how it works.
- Considerate use of abstractions. This is a balancing act between a tower of abstraction only those who create them understand, and an ocean of tedious explicitness. Both are unwanted.
- Independently test all non trivial routines and verify them correct for a realistic input space.
- Minimize the use of `subroutine`s because they can mutate their environment, instead use `function`s. When done correctly, this has negligible runtime cost.
- In a similar spirit, strive to make as many of the functions `pure`, and in the ideal case `elemental`. This gives static guarantees of their behaviour and enables automatic inlining removing the cost of function calls.
- Minimize the usage of global variables. Treating programs like a big state machine where everything has a potential interaction trough a global variable is harder to understand than the isolation given by `function`s.

The overall idea behind these points is to enforce a certain structure on the program, such that if one knows a part of the code, reading the rest of it will feel familiar. The specific points about favouring functions over subroutines and discourage global variables are to lower the mental burden of state management between different parts of the code. If a function is pure, its output is uniquely defined by its input, and it does not have any effect on any other function in the code. The points above served as guidelines and had to be violated several times because of limitations in Fortran, like not being able to return multiple variables from a function, architecture of existing code or the points simply being impractical to follow.

Testing and verifying correctness of all non trivial functions was a goal. This implies at least one test per non-trivial function, which quickly adds up to quite a lot. To efficiently work with that many tests automation was needed. A minimal testing framework was created. Using it adding a new unit test was as simple as creating a single file. When compiled and linked with the rest of the source code this file would create a binary which when run would output any errors. Then a wrapper script was created to compile, run and report back the status of all tests. This way, verifying that nothing was broken in a new revision of the code was as simple as running a single command. This proved essential several times during development and saved many hours of debugging.

B. Core immersed boundary and linear algebra routines developed

The most essential parts of developed code are listed below, they are also embedded in the PDF file and can be obtained by pressing the following links: immersed boundary, linear algebra.

Listing 1: immersed_boundary_module_header

```
module immersed_boundary
  !
  !> @file
  !> Immersed boundary
  !>
  !> MOL, 2015-02-03.
  !> This is a 2D implementation of the immersed boundary method based
  !> on Peskins review paper on the method (2002).
  !>
  !> The main use of the code is to enable simulation of elastic
   ↪  membranes
  !> coming from asphaltene surfactants. See masterthesis of
  !> Morten Olsen Lysgaard (NTNU 2015) for further reference.
  !> This works for both 2D and axisymmetry, but note that for
   ↪  axisymmetry
  !> it is unphysical for the drop to crumple axisymmetricly.
  !>
  !> The code has been verified to simulate surface tension correctly
  !> together with density and viscosity jumps. It also has been tested
  !> Together with electric fields. In general, it is compatible with
  !> any sharp interface forces coming from the GFM method.
  !> It should also work together with the CSF method, but
  !> this has not been verified.
  !> The code is also verified for axisymmetry.
  !>
```

```
24  !> Variable surface tension has been implemented, but not verified to
25  !> work correctly.
26  !>
27  !> The code can simulate a single closed interface, e.g. a drop,
28  !> although adding support for several interfaces should be quite easy.
29  !> This would preferably be done using a dynamic datastructure that can
30  !> hold the data for the different boundaries.
31  !>
32  !> Code for topological change, e.g. collission is not implemented.
33  !> This depends on support for multiple interfaces first.
34  !> Once multiple interfaces is implemented it should be easy to
35  !> implement for the 2D case.
36  !
37  implicit none
38  private
39  save
40  !
41  ! Logical telling if (general) surface tension is simulated with the IB
      ↪  method.
42  logical :: libsigma=.false.
43  !
44  ! The Lagrangian point density per Eulerian grid cell.
45  integer :: ibdensity=0
46  !
47  ! The maximum number of Lagrangian points we can handle.
48  ! The limit arises because of the current storage method used
49  ! for easy integration with the Runge-Kutta methods.
50  integer :: ib_max_points=0
51  !
52  ! The current number of Lagrangian points <= ib_max_points.
53  integer :: npoints=0
54  !
55  ! The surface tension of the immersed boundary.
56  real :: ibsigma=0.0
57  !
58  ! The elastic membrane Hookean spring constant.
59  real :: ibKa=0.0
60  !
61  ! The CFL number coming from surface forces, this is used by the outer
      ↪  routines
62  ! calculating the time step.
```

```
63   real :: ib_cfl_st=0.0
64   !
65   ! Public procedures
66   public :: init_ib,init_ib_from_userinp
67   public :: rhs_ib
68   public :: find_boundary
69   public :: ib_force
70   public :: ellipsoid_ib
71   public :: reinitialize_level_set_from_ib
72   public :: write_ib_to_tecplot
73   !
74   ! Public variables
75   public :: npoints
76   public :: libsigma, ibsigma, ib_cfl_st
77   !
78   ! In an ideal world these would not be public, as they are not used by
       ↪  the outer routines.
79   ! However, they are used by the unit tests, because of this they need
       ↪  to be public.
80   public :: ib_max_points, ibdensity
81   public :: calculate_curvature, cubic_spline, cubic_spline_normal
82   public :: spread_vector_lagrangian_to_eulerian,
       ↪  interpolate_vector_eulerian_to_lagrangian
83   public :: heaviside
84   !
85   ! Static variables used for saving the immersed boundary to a TecPlot
       ↪  compatible file
86   ! This is used for visualization.
87   integer, parameter :: itec_points  =51
88   character(len=24)  :: tec_pointfile="level2-points.tec"
89 contains
```

Listing 2: init_ib_from_userinp

```
1    subroutine init_ib_from_userinp
2      ! This routine is used to initialize the IB-method from user
3      ! input (user.inp file).
4      !
5      !---------------------------------------------------------------------
6      ! Created: MOL, 2015-02-03.
```

```
7     ! Changed: MOL, 2015-02-18 seperated out find_boundary
8     !---------------------------------------------------------------------
9     use userinp
10    integer :: pibdensity
11    !
12    ! Get immersed boundary point density
13    pibdensity=ivalue("Number of points per grid cell, (0 == No
        ↪  IB-method) recommended ~ 5", &
14      'I',"spi.ib.ibdensity",'0')
15    libsigma=lvalue("Use Immersed Boundary method for surface tension
        ↪  calculation", &
16      'L',"spi.ib.libsigma",'0')
17    ibKa=rvalue("Immersed Boundary coefficient of elasticity,
        ↪  Ka",'R',"spi.ib.ibKa",'0.0')
18    call init_ib(pibdensity)
19  end subroutine init_ib_from_userinp
```

Listing 3: init_ib

```
1   subroutine init_ib(pibdensity)
2     ! This routine calculates the maximum number of points and sets up
3     ! logical to signal that we are either using or not using the IB
4     ! method.
5     !
6     !---------------------------------------------------------------------
7     ! Created: MOL, 2015-03-20.
8     !---------------------------------------------------------------------
9     use grid, only: imax, jmax
10    use rhs_var, only: lib
11    ! the Lagrangian point density per Eulerian grid cell
12    integer, intent(in) :: pibdensity
13    ibdensity = pibdensity
14    if(ibdensity>0) then
15      ! We are using the IB-method
16      lib=.true.
17      !
18      ! calculate the limit of number of points we can handle
19      ib_max_points = (imax*jmax)/2
20    endif
21  end subroutine init_ib
```

Listing 4: rhs_ib

```
subroutine rhs_ib(ibp, ibdk, f, dpdt, dibdkdt)
  ! This is the main advection routine. It uses the immersed
  ! boundary delta function interpolation to interpolate an
  ! Eulerian velocity field to the Lagrangian grid points.
  !
  ! Calculates the right hand side of
  !
  ! dp_i/dt = rhs_i
  !
  ! where p_i is position of point i at the current time.
  ! This right hand side is simply an Euler step of an ordinary ODE.
  ! These Euler steps are composed in a higher order Runge-Kutta
  ! Method from the calling code.
  !-----------------------------------------------------------------------
  ! Created: MOL, 2015-02-03.
  ! Changed: MOL, 2015-02-18 Several bug fixes fixing a sign error
    ↪  caused
  !                          by a bug in the interpolation function
  !-----------------------------------------------------------------------
  use grid, only: ib1,ibn, jb1, jbn
  use rhs_var, only: lvar_st
  use surfactants, only: surf_diff
  real, intent(in) :: f(ib1:ibn,jb1:jbn,2) ! eulerian staggered
    ↪  velocity field
  real, intent(in) :: ibp(2,ib_max_points) ! current point positions
  real, intent(in) :: ibdk(2,ib_max_points) ! current point equilibrium
    ↪  distance and curvature
  real, intent(out) :: dpdt(2,ib_max_points) ! output velocities of
    ↪  each point
  real, intent(out) :: dibdkdt(2,ib_max_points) ! output rate of change
    ↪  for surfactant
  real :: consentration(npoints) ! consentration of surfactant at a
    ↪  point
  real :: d_mid(npoints) ! array containing the distance between 2
    ↪  points
  integer :: i

  dpdt(:,1:npoints) = interpolate_velocity_field(ibp, f)
```

```
32
33    dibdkdt = 0.0
34    if(lvar_st) then
35      write(*,*) 'MOL: Varying surface tension has not been validated to
    ↪  be correct for the immersed boundary.'
36      write(*,*) '      This warning only applies if you want to '
37      write(*,*) '      tension from _both_ elastic membrane and insoluble
          ↪  surfactants (soap)'
38      write(*,*) '      at the same time.'
39      write(*,*) ''
40      write(*,*) '      Summary:'
41      write(*,*) '      Constant surface tension = Use level-set + GFM or
          ↪  immersed boundary, both are verified.'
42      write(*,*) '      Varying surface tension = Use level-set + GFM,
          ↪  verified.'
43      write(*,*) '      Elastic forces + constant surface tension = Use
          ↪  immersed boundary, verified.'
44      write(*,*) '      Elastic forces + varying surface tension = Use
          ↪  immersed boundary, but remove this warning and verify that it
          ↪  works correctly.'
45      write(*,*) "      Program will now terminate so that you don't get
          ↪  results you can't trust ;)"
46      stop
47
48      d_mid = calculate_middle_dist(ibp)
49      consentration = ibdk(2,1:npoints)/d_mid
50
51      do i=1,npoints
52        dibdkdt(2,i) =
            ↪  surf_diff*(consentration(iprev(i))-2.0*consentration(i)+ &
53          consentration(inext(i)))/d_mid(i)**2.0 ! laplace term
54      end do
55    end if
56  end subroutine rhs_ib
```

Listing 5: ib_force

```
1   subroutine ib_force(ibp, ibdk, fi, f)
2     ! Calculates the forces from the interface on the
3     ! fluid.
```

```
4     !
5     ! This enters as a right hand side term in the pressure equation.
6     !----------------------------------------------------------------------
7     ! Created: MOL, 2015-03-02.
8     !----------------------------------------------------------------------
9     use grid, only: ib1, ibn, jb1, jbn, dxymin
10    use rhs_var, only: rho, laxisym, lvar_st
11    use surfactants, only: max_surfactant_packing, elasticity
12    real, intent(out) :: f(ib1:ibn,jb1:jbn,2) ! eulerian staggered force
        ↪  field
13    real, intent(in) :: fi(ib1:ibn, jb1:jbn) ! level-set function
14    real, intent(in) :: ibp(2,ib_max_points) ! current point positions
15    real, intent(in) :: ibdk(2,ib_max_points) ! equilibrium distances and
        ↪  curvatures, [(d1,k1)...(dn,kn)]
16    real :: k(npoints) ! current curvature
17    real :: d_mid(npoints) ! segment length interpolated to point
18    real :: fp(2,npoints) ! sum of forces on particle
19    real, dimension(2,npoints,4) :: abcd ! coeffecients for cubic spline
20    real, dimension(2,npoints) :: tangents ! normalized xy tangent vector
        ↪  to interface
21    real, dimension(2,npoints) :: inward_normal ! unit normal vector to
        ↪  interface
22    real :: T(npoints) ! tension in the boundary
23    real :: dTds(npoints) ! derivative of tension in boundary
24    real :: mass_density(ib1:ibn, jb1:jbn) ! Eulerian mass density
25    integer :: i
26    fp=0.0
27
28    d_mid = calculate_middle_dist(ibp)
29
30    if(laxisym) then
31      abcd = cubic_spline_normal(ibp)
32    else
33      abcd = cubic_spline(ibp)
34    endif
35    tangents = calculate_tangent(abcd)
36    k = calculate_curvature(ibp, abcd)
37
38    if(lvar_st) then
```

```
39      T(:) = ibKa*(d_mid(:)/ibdk(1,1:npoints) - 1.0) +
    ↪   ibsigma*(1.0+elasticity*log(1.0-
    ↪   (ibdk(2,1:npoints)/d_mid(:))/max_surfactant_packing))
40    else
41      T(:) = ibKa*(d_mid(:)/ibdk(1,1:npoints) - 1.0) + ibsigma
42    end if
43
44    do i=1,npoints
45      dTds(i) = (T(inext(i))-T(iprev(i)))/2.0
46    end do
47
48    mass_density = rho(1) + heaviside_fi(fi)*(rho(2)-rho(1)) ! heaviside
    ↪   smooth eulerian mass density
49
50    if(libsigma) then
51      inward_normal = -calc_outward_normal(ibp, tangents, fi)
52      do i=1,npoints
53        fp(:,i) = ( &
54                    dTds(i)*tangents(:,i) + &
55                    T(i)*k(i)*d_mid(i)*inward_normal(:,i) &
56                  )/interpolate_scalar_eulerian_to_lagrangian(ibp(1,i),
                      ↪   ibp(2,i), mass_density)
57      end do
58    end if
59
60    ! When the simulation is axisymmetrix the x-forces on points at y==0
        ↪   are in
61    ! equilibrium, in code: if(y==0) then fx=0
62    ! Since the points having y==0 are known, the first and last point on
        ↪   the bondary,
63    ! we simply set the forces there to zero.
64    if(laxisym) then
65      fp(1,1) = 0.0
66      fp(1,npoints) = 0.0
67    end if
68
69    ib_cfl_st=sqrt(maxval(abs(norm2(fp,1)/d_mid))/dxymin**2)
70
71    f = 0.0
72    do i=1,npoints
```

```
73      call spread_vector_lagrangian_to_eulerian(ibp(1,i), ibp(2,i),
          ↪  fp(1,i), fp(2,i), f)
74    end do
75  end subroutine ib_force
```

Listing 6: reinitialize_level_set_from_ib

```
1   subroutine reinitialize_level_set_from_ib(ibp, fi)
2     ! Reinitialize the level-set function from the immersed boundary.
3     !
4     ! This function takes the immersed boundary points, ibp,
5     ! and outputs a level-set field, a signed distance to
6     ! the immersed boundary.
7     !
8     ! The general algorithm is outlined in Morten Olsen Lysgaards
        ↪  masterthesis and works
9     ! as follows:
10    !
11    ! * For each segment of the immersed boundary create a bounding box,
        ↪  and
12    ! then grow the box by 4 grid cells in all directions.
13    !
14    ! * For each grid cell in this bounding box,
15    !   calculate the distance between the grid cell node and the line
        ↪  segment.
16    !   If this distance is the smallest yet discovered, store it in a
        ↪  temporary field.
17    !
18    ! * When all segments have been evaluated, go trough the temporary
        ↪  field node by node.
19    !   If there is a saved distance for the node, calculate wheter the
        ↪  node is inside
20    !   or outside the polyhedron defined by ibp. This decides the sign
        ↪  of the distance.
21    !
22    ! * Save the signed distance for the updated nodes in the original fi
        ↪  field.
23    !
24    !-----------------------------------------------------------------------
25    ! Created: MOL, 2015-04-24.
```

```
26    !---------------------------------------------------------------------
27    use grid, only: x, y, ib1,ibn,jb1,jbn, p2ij, nbord
28    use constants, only: almost_infinite
29    real, intent(in) :: ibp(2,ib_max_points) ! current point positions
30    real, intent(inout) :: fi(ib1:ibn, jb1:jbn) ! the level-set field
31    real :: fiTmp(ib1:ibn, jb1:jbn) ! temporary level-set field
32
33    integer :: i,j,k
34    integer :: ij1(2), ij2(2), ijtmp(2)
35
36    fiTmp = almost_infinite ! set temp array to inf because fi never
        ↪  should be inf
37
38    do k=1,npoints
39      ij1 = p2ij(ibp(:,k))
40      ij2 = p2ij(pnext(ibp,k))
41      ijtmp = ij1
42      ij1 = [min(ij1(1), ij2(1)), min(ij1(2), ij2(2))] - 4
43      ij1 = [max(ij1(1), ib1), max(ij1(2), jb1)]
44      ij2 = [max(ijtmp(1), ij2(1)), max(ijtmp(2), ij2(2))] + 4
45      ij2 = [min(ij2(1), ibn), min(ij2(2), jbn)]
46
47      do i=ij1(1),ij2(1)
48        do j=ij1(2),ij2(2)
49          fiTmp(i,j) = min(fiTmp(i,j), dist_point_to_line(ibp(:,k),
              ↪  pnext(ibp,k), [x(i), y(j)]))
50        end do
51      end do
52
53    end do
54
55    !$OMP PARALLEL DO schedule(guided, 10) private(i, j) shared(ib1, ibn,
        ↪  jb1, jbn, fi, fiTmp, ibp)
56    do i=ib1,ibn
57      do j=jb1,jbn
58        if(fiTmp(i,j) < almost_infinite) then ! if this value was
            ↪  updated
59          fi(i,j) = fiTmp(i,j)*point_inside(ibp,[x(i),y(j)])
60        end if
61      end do
62    end do
```

```
63    !$OMP END PARALLEL DO
64  end subroutine reinitialize_level_set_from_ib
```

Listing 7: dist_point_to_line

```
1   pure real function dist_point_to_line(v, w, p) result(d)
2     ! Distance between a point, p, and a line segment defined by the
3     ! points v and w.
4     !---------------------------------------------------------------------
5     ! Created: MOL, 2015-04-24.
6     !---------------------------------------------------------------------
7     real, dimension(2), intent(in) :: p(2) ! the point in question
8     real, dimension(2), intent(in) :: v, w ! the endpoints of the
        ↪  linesegment
9     real :: l2, t, projection(2)
10
11    l2 = sum((w-v)**2)  ! length squared i.e. |w-v|^2
12    if (l2 == 0.0) then ! v == w case
13      d = distp(p- v)
14      return
15    end if
16    ! Consider the line extending the segment, parameterized as v + t (w
        ↪  - v).
17    ! We find projection of point p onto the line.
18    ! It falls where t = [(p-v) . (w-v)] / |w-v|^2
19    t = dot_product(p - v, w - v) / l2;
20    if (t < 0.0) then ! Beyond the 'v' end of the segment
21      d = distp(p- v)
22      return
23    else if (t > 1.0) then
24      d = distp(p- w) ! Beyond the 'w' end of the segment
25      return
26    end if
27    projection = v + t * (w - v) ! Projection falls on the segment
28    d = distp(p- projection)
29  end function dist_point_to_line
```

Listing 8: point_inside

```
pure real function point_inside(ibp, p) result(sgn)
  ! Returns negative if a point is inside the closed polyhedron
  ! defined by ibp, positive if not.
  !
  ! Algorithm follows this principle:
  !
  ! * Given a closed polyhedron G, and an arbitrary point p.
  !
  ! * Count the number of times crossing the interface of G when
  !   traveling on a ray from infinity to the point p.
  !
  ! * If the number is odd, p is inside the polyhedron,
  !   if it is even, p is outside.
  !
  !------------------------------------------------------------------------
  ! Created: MOL, 2015-04-24.
  !------------------------------------------------------------------------
  real, intent(in) :: ibp(2,ib_max_points) ! current point positions
  real, dimension(2), intent(in) :: p(2) ! the point in question
  integer :: i
  real ymin, ymax

  sgn = 1.0 ! starting outside

  do i=1,npoints
    if(p(1) < ibp(1,i)) then ! only count crossing to the right
      ymin = min(ibp(2,i), ibp(2,inext(i)))
      ymax = max(ibp(2,i), ibp(2,inext(i)))
      if(ymin <= p(2) .and. p(2) < ymax) then ! this is a crossing
        sgn = -1.0*sgn
      end if
    end if
  end do
end function point_inside
```

Listing 9: interpolate_velocity_field

```
pure function interpolate_velocity_field(ibp, eulerVel)
  ↪  result(pointVel)
  ! Interpolates an Eulerian velocity field to the Lagrange points
```

```
3     !
4     !------------------------------------------------------------------
5     ! Created: MOL, 2015-04-17.
6     !------------------------------------------------------------------
7     use grid, only: ib1,ibn, jb1, jbn
8     use rhs_var, only: laxisym
9
10    real, intent(in) :: ibp(2,ib_max_points) ! current point positions
11    real, intent(in) :: eulerVel(ib1:ibn,jb1:jbn,2) ! Eulerian staggered
        ↪  velocity field
12    real :: pointVel(2,npoints) ! output velocities of each point
13    integer :: i
14
15    forall(i=1:npoints)
16      pointVel(:,i) = interpolate_vector_eulerian_to_lagrangian(ibp(1,i),
          ↪  ibp(2,i), eulerVel)
17    endforall
18
19    ! If the simulation is axysymmetric the mirror points (x==0) are
        ↪  constarined
20    ! to the y axis (enforce vx == 0)
21    if(laxisym) then
22      pointVel(1,1) = 0.0
23      pointVel(1,npoints) = 0.0
24    end if
25  end function
```

Listing 10: interpolate_scalar_eulerian_to_lagrangian

```
1   pure function interpolate_scalar_eulerian_to_lagrangian(px, py, F)
      ↪  result(res)
2     ! Calculates the effect of a scalar field on a lagrangian particle
3     ! using the deltafunction interpolation from the Peskin review paper.
4     !
5     ! The deltafunction has a radius of 2 times the discretization size.
6     ! As the figure below illustrates this mean that one for an
        ↪  arbitrary
7     ! point inside a grid cell, have to consider 16 different points
8     ! to be able to interpolate the value correctly.
9     ! In the figure p is the arbitrary point, and x is the grid points
```

```
10    ! that need to be considered in order to interpolate the from the
11    ! eulerian grid to the point p
12    !
13    !   -------------------------
14    !   | dx | dx | dx | dx | dx |
15    !   |dy  |    |    |    |    |
16    !s+2|----x----x----x----x----|
17    !   |    |    |    |    |    |
18    !   |dy  |    |    |    |    |
19    !s+1|----x----x----x----x----|
20    !   |    |    |  p |    |    |
21    !   |dy  |    |    |    |    |
22    !  s|----x----x----x----x----|
23    !   |    |    |    |    |    |
24    !   |dy  |    |    |    |    |
25    !s-1|----x----x----x----x----|
26    !   |    |    |    |    |    |
27    !   |dy  |    |    |    |    |
28    !   |----|----|----|----|----|
29    !       w-1   w   w+1  w+2
30    !
31    ! Also note the point (w,s), it is this point we use
32    ! as basis for calculating the 16 points.
33    !
34    !-----------------------------------------------------------------------
35    ! Created: MOL, 2015-06-10.
36    !-----------------------------------------------------------------------
37    use grid, only: ib1,ibn,jb1,jbn,x2i,y2j, x, y, dx, dy
38    ! Input/Output
39    real, intent(in) :: px,py ! x and y position of current point
40    real, intent(in), dimension(ib1:ibn,jb1:jbn) :: F ! Eulerian scalar
        ↪  field
41    real res ! output interpolated scalar value
42    !
43    ! Local variables
44    integer :: nstencil ! number of Eulerian points to consider
45    parameter (nstencil = 16) ! for a stencil that has radius 2h, we have
        ↪  to consider 16 points
46    real :: Fs(nstencil) ! values of the scalar field at our 16 points
47    integer :: s ! lower y-bound in the eulerian grid for cell containing
        ↪  (px,py)
```

```
48    integer :: w ! lower x-bound in the eulerian grid for cell containing
        ↪  (px,py)
49    integer :: xvec(nstencil)  ! x-coordinates of the 16 points
50    integer :: yvec(nstencil)  ! y-coordinates of the 16 points
51    integer :: i,j,k
52
53    ! find south-west corner of the cell for point (x,y)
54    w=x2i(px)
55    s=y2j(py)
56
57    ! calculate coordinates for 16 points
58    k=1
59    do i=-1,2
60      do j=-1,2
61        xvec(k) = w+i
62        yvec(k) = s+j
63        k = k+1
64      end do
65    end do
66
67    ! calculate scalar at the 16 points
68    forall(i=1:nstencil)
69      Fs(i) = F(xvec(i),yvec(i))
70    end forall
71
72    ! Compute interpolated scalar
73    !
74    ! Here we use the fact that fortran supports vector arguments to
        ↪  elemental functions
75    ! to compute all 16 points in one go.
76    res = sum(delta2d(px-x(xvec),dx(xvec),py-y(yvec),dy(yvec))*Fs(:))
77  end function interpolate_scalar_eulerian_to_lagrangian
```

Listing 11: interpolate_vector_eulerian_to_lagrangian

```
1   pure function interpolate_vector_eulerian_to_lagrangian(px, py, F)
      ↪  result(res)
2     ! Calculates the effect of a vector field on a lagrangian particle
3     ! using the deltafunction interpolation from the Peskin review paper.
4     !
```

```
5     ! The deltafunction has a radius of 2 times the discretization size.
6     ! As the figure below illustrates this mean that one for an
        ↪  arbitrary
7     ! point inside a grid cell, have to consider 16 different points
8     ! to be able to interpolate the value correctly.
9     ! In the figure p is the arbitrary point, and x is the grid points
10    ! that need to be considered in order to interpolate the from the
11    ! eulerian grid to the point p
12    !
13    !   -------------------------
14    !   | dx | dx | dx | dx | dx |
15    !   |dy  |    |    |    |    |
16    !s+2|----x----x----x----x----|
17    !   |    |    |    |    |    |
18    !   |dy  |    |    |    |    |
19    !s+1|----x----x----x----x----|
20    !   |    |    |  p |    |    |
21    !   |dy  |    |    |    |    |
22    !  s|----x----x----x----x----|
23    !   |    |    |    |    |    |
24    !   |dy  |    |    |    |    |
25    !s-1|----x----x----x----x----|
26    !   |    |    |    |    |    |
27    !   |dy  |    |    |    |    |
28    !   |----|----|----|----|----|
29    !       w-1   w   w+1  w+2
30    !
31    ! Also note how the point (w,s) is denoted, it is this point we use
32    ! as basis for calculating the 16 points.
33    !
34    ! This function takes into account the staggered grid used for
        ↪  velocities.
35    !-----------------------------------------------------------------------
36    ! Created: MOL, 2015-02-05.
37    ! Updated: MOL, 2015-02-18 - Fixed sign and swap of u,v coordinates
38    !-----------------------------------------------------------------------
39    use grid, only: ib1,ibn,jb1,jbn,x2i,xu2i,y2j,yv2j, x, xu, y, yv, dx,
        ↪  dy, dxu, dyv
40    ! Input/Output
41    real, intent(in) :: px,py ! x and y position of point
```

```
42    real, intent(in), dimension(ib1:ibn,jb1:jbn,2) :: F ! eulerian
        ↪ velocity field
43    real res(2) ! output vector containing the interpolated x,y
        ↪ velocities
44    !
45    ! Local variables
46    integer :: nstencil ! the number of eulerian points to consider
47    parameter (nstencil = 16) ! for a stencil that has radius 2h, we have
        ↪ to consider 16 points
48    real :: Fs(nstencil,2) ! values of the velocity field at our 16
        ↪ points
49    integer :: s,sv ! lower y-bound in the eulerian grid for cell
        ↪ containing (px,py)
50    integer :: w,wu ! lower x-bound in the eulerian grid for cell
        ↪ containing (px,py)
51    integer :: xvec(nstencil)  ! x-coordinates of the 16 points
52    integer :: xuvec(nstencil) ! xu-coordinates of the 16 points
53    integer :: yvec(nstencil)  ! y-coordinates of the 16 points
54    integer :: yvvec(nstencil) ! yv-coordinates of the 16 points
55    integer :: i,j,k
56
57    ! find south-west corner of the cell for point (x,y)
58    w=x2i(px)
59    wu=xu2i(px)
60    s=y2j(py)
61    sv=yv2j(py)
62
63    ! calculate coordinates for 16 points
64    k=1
65    do i=-1,2
66      do j=-1,2
67        xvec(k) = w+i
68        yvec(k) = s+j
69        xuvec(k) = wu+i
70        yvvec(k) = sv+j
71        k = k+1
72      end do
73    end do
74
75    ! calculate velocities at the 16 points
76    forall(i=1:nstencil)
```

```
77      Fs(i,:) = [F(xuvec(i),yvec(i),1), F(xvec(i),yvvec(i),2)]
78    end forall
79
80    ! Compute interpolated velocies
81    !
82    ! Here we use the fact that fortran supports vector arguments to
        ↪  elemental functions
83    ! to compute all 16 points in one go.
84    res(1) = sum(delta2d(px-xu(xuvec),dxu(xuvec),py-
        ↪  y(yvec),dy(yvec))*Fs(:,1))
85    res(2) = sum(delta2d(px-x(xvec),dx(xvec),py-
        ↪  yv(yvvec),dyv(yvvec))*Fs(:,2))
86  end function interpolate_vector_eulerian_to_lagrangian
```

Listing 12: spread_vector_lagrangian_to_eulerian

```
1   subroutine spread_vector_lagrangian_to_eulerian(px, py, fx, fy, F)
2     ! Calculates the effect of a lagrangian particle on a vector field
3     ! using the deltafunction interpolation from the Peskin review paper.
4     !
5     ! The deltafunction has a radius of 2 times the discretization size.
6     ! As the figure below illustrates this mean that one for an
        ↪  arbitrary
7     ! particle inside a grid cell, have to interpolate the particle value
        ↪  to 16 different grid nodes.
8     ! In the figure, p is the arbitrary point, x is the grid nodes
9     ! that need to be considered
10    !
11    !   ---------------------------
12    !   | dx | dx | dx | dx | dx |
13    !   |dy  |    |    |    |    |
14    !s+2|----x----x----x----x----|
15    !   |    |    |    |    |    |
16    !   |dy  |    |    |    |    |
17    !s+1|----x----x----x----x----|
18    !   |    |    |  p |    |    |
19    !   |dy  |    |    |    |    |
20    !  s|----x----x----x----x----|
21    !   |    |    |    |    |    |
22    !   |dy  |    |    |    |    |
```

```
23    !s-1|----x----x----x----x----|
24    !   |    |    |    |    |    |
25    !   |dy  |    |    |    |    |
26    !   |----|----|----|----|----|
27    !       w-1   w   w+1  w+2
28    !
29    ! Also note how the point (w,s) is denoted, it is this point we use
30    ! as basis for calculating the 16 points.
31    !
32    ! This function takes into account the staggered grid used for
        ↪ velocities.
33    !-----------------------------------------------------------------------
34    ! Created: MOL, 2015-03-04
35    ! Updated: MOL, 2015-04-17 - fixed scaling of deltafunction
36    !-----------------------------------------------------------------------
37    use grid, only: ib1,ibn,jb1,jbn,x2i,xu2i,y2j,yv2j, x, xu, y, yv, dx,
        ↪ dy, dxu, dyv
38    ! Input/Output
39    real, intent(in) :: px,py ! x and y position of our arbitrary point
40    real, intent(in) :: fx,fy ! x and y forces on our arbitrary point
41    real, intent(out), dimension(ib1:ibn,jb1:jbn,2) :: F ! eulerian
        ↪ vector field
42    !
43    ! Local variables
44    integer :: s,sv ! lower y-bound in the eulerian grid for cell
        ↪ containing (px,py)
45    integer :: w,wu ! lower x-bound in the eulerian grid for cell
        ↪ containing (px,py)
46    integer :: i,j,k
47
48    ! find south-west corner of the cell for point (x,y)
49    w=x2i(px)
50    wu=xu2i(px)
51    s=y2j(py)
52    sv=yv2j(py)
53
54    forall (i=-1:2,j=-1:2)
55      F(wu+i,s+j,1) = F(wu+i,s+j,1) + delta2d(px-xu(wu+i),dxu(wu+i),py-
          ↪ y(s+j),dy(s+j))/(dxu(wu+i)*dy(s+j))*fx
56      F(w+i,sv+j,2) = F(w+i,sv+j,2) + delta2d(px-x(w+i),dx(w+i),py-
          ↪ yv(sv+j),dyv(sv+j))/(dx(wu+i)*dyv(s+j))*fy
```

```
57    end forall
58  end subroutine spread_vector_lagrangian_to_eulerian
```

Listing 13: spread_scalar_lagrangian_to_eulerian

```
1   function spread_scalar_lagrangian_to_eulerian(ibp, fin) result(F)
2     ! Spreads/distributes a scalar value from all the Lagrangian point
3     ! to an Eulerian field.
4     !
5     ! Uses the deltafunction interpolation from the Peskin review paper.
6     !-----------------------------------------------------------------------
7     ! Created: MOL, 2015-05-07
8     !-----------------------------------------------------------------------
9     use grid, only: ib1,ibn,jb1,jbn,x2i,y2j, x, y, dx, dy
10    ! Input/Output
11    real, intent(in) :: ibp(2,ib_max_points) ! x and y position of our
        ↪  arbitrary point
12    real, intent(in) :: fin(npoints) ! scalar value on points
13    real, dimension(ib1:ibn,jb1:jbn) :: F ! eulerian scalar field
14    integer, dimension(ib1:ibn,jb1:jbn) :: N ! normalization field
15    integer :: i
16
17    N = 0
18    F = 0.0
19    do i=1,npoints
20      call spread_scalar_lagrangian_to_eulerian_inner(ibp(1,i), ibp(2,i),
          ↪  fin(i), F, N)
21    end do
22    F = F/real(max(N,1)) ! normalize
23  end function spread_scalar_lagrangian_to_eulerian
```

Listing 14: spread_scalar_lagrangian_to_eulerian_inner

```
1   subroutine spread_scalar_lagrangian_to_eulerian_inner(px, py, fin, F,
      ↪  N)
2     ! Spreads/distributes a scalar value from a single point to an
        ↪  Eulerian
3     ! field. Also updates a normilazition field which is used in the
        ↪  outer routine.
```

```
4     !
5     !---------------------------------------------------------------------
6     ! Created: MOL, 2015-05-07
7     !---------------------------------------------------------------------
8     use grid, only: ib1,ibn,jb1,jbn,x2i,y2j, x, y, dx, dy
9     ! Input/Output
10    real, intent(in) :: px,py ! x and y position of our arbitrary point
11    real, intent(in) :: fin ! x and y forces on our arbitrary point
12    real, intent(inout), dimension(ib1:ibn,jb1:jbn) :: F ! eulerian
        ↪  scalar field
13    integer, intent(inout), dimension(ib1:ibn,jb1:jbn) :: N !
        ↪  normalization field
14    !
15    ! Local variables
16    integer :: s ! lower y-bound in the eulerian grid for cell containing
        ↪  (px,py)
17    integer :: w ! lower x-bound in the eulerian grid for cell containing
        ↪  (px,py)
18    integer :: i,j
19
20    ! find south-west corner of the cell for point (x,y)
21    w=x2i(px)
22    s=y2j(py)
23
24    do i=-1,2
25      do j=-1,2
26        F(w+i,s+j) = F(w+i,s+j) +
            ↪  delta2d(px-x(w+i),dx(w+i),py-y(s+j),dy(s+j))*fin
27        N(w+i,s+j) = N(w+i,s+j) + 1
28      end do
29    end do
30  end subroutine spread_scalar_lagrangian_to_eulerian_inner
```

Listing 15: delta2d

```
1   elemental real function delta2d(x,hx,y,hy)
2     ! 2D delta function from Peskins review (2002) paper on the
        ↪  IB-method
3     ! Centered around (x,y)=(0,0)
4     !
```

```
    ! NOTE: one slight difference from the paper is that this
      ↪  deltafunction
    ! is not divided by the step size.
    ! This is because for interpolation(spreading) we need an unscaled
    ! deltafunction. The caller of this function is thus responsible of
    ! scaling correctly.
    !
    !-----------------------------------------------------------------------
    ! Created: MOL, 2015-02-03.
    ! Updated: MOL, 2015-02-18 - updated documentation
    !-----------------------------------------------------------------------
    real, intent(in) :: x,y,hx,hy

    delta2d = delta1d(x,hx)*delta1d(y,hy)
  end function delta2d
```

Listing 16: delta1d

```
  elemental real function delta1d(r, h)
    ! 1D delta function from Peskins review (2002) paper on the
      ↪  IB-method
    ! Centered around r=0
    !
    ! NOTE: one slight difference from the paper is that this
      ↪  deltafunction
    ! is not divided by the step size.
    ! This is because for interpolation(spreading) we need an unscaled
    ! deltafunction. The caller of this function is thus responsible of
    ! scaling correctly.
    !
    !-----------------------------------------------------------------------
    ! Created: MOL, 2015-02-03.
    ! Updated: MOL, 2015-02-18 - updated documentation
    !-----------------------------------------------------------------------
    real, intent(in) :: r, h
    delta1d = phi(r/h)
  end function delta1d
```

Listing 17: phi

```
elemental real function phi(rin)
  ! 1D phi function from Peskins review (2002) paper on the IB-method
  ! Centered around rin=0
  !
  !-----------------------------------------------------------------------
  ! Created: MOL, 2015-02-03.
  ! Updated: MOL, 2015-02-18 - simplified and updated documentation
  !-----------------------------------------------------------------------
  real, intent(in) :: rin
  real r
  ! since the delta function is symmetric we only consider positive
    ↪ distances
  ! this saves unnessesary complexity
  r = abs(rin)
  if (2.0 < r) then
    phi=0.0
  else if (1.0 < r) then
    phi=1.0/8.0 * (5.0 -2.0*r -sqrt(-7.0 +12.0*r -4.0*r*r))
  else ! if r <= 1.0
    phi=1.0/8.0 * (3.0 -2.0*r +sqrt(1.0 +4.0*r -4.0*r*r))
  endif
end function phi
```

Listing 18: heaviside_fi

```
function heaviside_fi(fi)
  ! Take heaviside of level-set field.
  ! Used for calculating density in the routine ib_force.
  !
  !-----------------------------------------------------------------------
  ! Created: MOL, 2015-06-16.
  !-----------------------------------------------------------------------
  use grid, only: dx, dy, ib1, ibn, jb1, jbn, str_method
  real, intent(in) :: fi(ib1:ibn, jb1:jbn) ! level-set function
  real :: heaviside_fi(ib1:ibn, jb1:jbn) ! output heaviside of
    ↪ level-set function
  integer :: i, j

  if (str_method==0) then
    heaviside_fi(:,:) = heaviside(fi(:,:)/dx(1))
```

```
15    else
16      forall (i=ib1:ibn, j=jb1:jbn)
17        heaviside_fi(i,j) = heaviside(fi(i,j)/dx(i))
18      end forall
19    end if
20  end function heaviside_fi
```

Listing 19: heaviside

```
1   elemental real function heaviside(r)
2     ! 1D heaviside function from Peskins review (2002) paper on the
        ↪  IB-method
3     ! Centered around r=0
4     !
5     !-------------------------------------------------------------------------
6     ! Created: MOL, 2015-05-03.
7     !-------------------------------------------------------------------------
8     use constants, only: pi
9     real, intent(in) :: r
10    if (2.0 < r) then
11      heaviside=1.0
12    else if(r < -2.0) then
13      heaviside=0.0
14    else if (r < -1.0) then
15      ! integrate from -inf to -2<r<-1
16      heaviside = -(pi-46.0)/64.0-(2.0*asin((2.0*r+3.0)/sqrt(2.0)) +
          ↪  (2.0*r+3.0)*sqrt(-4.0*r**2-12.0*r-7.0) -
          ↪  4.0*r**2-20.0*r)/32.0
17    else if (r < 0.0) then
18      ! integrate from -inf to -1<r<0
19      heaviside = (2.0*asin((2.0*r+1.0)/sqrt(2.0)) +
          ↪  (2.0*r+1.0)*sqrt(-4.0*r**2-4.0*r+1.0)+4.0*r**2+12.0*r)/32.0 +
          ↪  (pi+18.0)/64.0-(pi-6.0)/32.0
20    else if (r < 1.0) then
21      ! integrate from -inf to 0<r<1
22      heaviside = (2.0*asin((2.0*r-1.0)/sqrt(2.0)) +
          ↪  (2.0*r-1.0)*sqrt(-4.0*r**2+4.0*r+1.0)-4*r**2+12.0*r)/32.0 +
          ↪  (pi+2.0)/64.0+0.5
23    else if (r < 2.0) then
24      ! integrate from -inf to 1<r<2
```

```
25      heaviside = -(2.0*asin((2.0*r-3.0)/sqrt(2.0)) +
          ↪  (2.0*r-3.0)*sqrt(-4.0*r**2+12.0*r-7.0)+4.0*r**2-20.0*r)/32.0 -
          ↪  (pi+34.0)/64.0+(pi+26.0)/32.0
26    endif
27  end function heaviside
```

Listing 20: find_boundary

```
1   subroutine find_boundary(ibp, ibdk, fi)
2     ! Given a discretized eulerian scalar field, the level-set function
        ↪  fi,
3     ! this function computes a piecewice linear path such that all points
        ↪  along
4     ! the path lie on fi = 0.
5     ! The distance between the points is calculated from the immersed
        ↪  boundary
6     ! point density.
7     ! The actual algorithm roughly works like this:
8     !
9     ! * Find the point in fi that has smallest absolute value.
10    !
11    ! * Use bilinear interpolation ta make the discrete fi continous.
12    !
13    ! * From the point with smallest absolute value, consider all
14    !   points that are on the circle with radius
        ↪  p=1.01*sqrt(dx**2+dy**2) away from it.
15    !
16    ! * Use a bisection search algorithm to find the angle that
        ↪  corresponds
17    !   to the value closest to zero.
18    !
19    ! * To make sure that the algorithm finishes it only looks for the
        ↪  next
20    !   point within a given sector based on the previous point. This is
        ↪  to
21    !   keep it from turning 180 deg and never finish the curve.
22    !
23    ! * Iteratively find new points untill the current point is closer
        ↪  than
24    !   pd to the starting point. This means we have closed the curve.
```

```
25    !
26    ! * Fit a cubic spline to the rough points
27    !
28    ! * Intersperse the rough points with new points such that the
        ↪  distance
29    !   from one to the next is pd
30    !
31    !-----------------------------------------------------------------------
32    ! Created: MOL, 2015-02-17.
33    !-----------------------------------------------------------------------
34    use grid, only: ij2p, dx, dy, ib1, ibn, jb1, jbn
35    use rhs_var, only: laxisym, lvar_st
36    use surfactants, only: surf0
37    real :: heading ! current heading (rad), this is the tangent of fi=0
        ↪  at p
38    real :: boundary_length ! estimated length of the closed curve fi=0
39    real :: pd ! step length between two points on the curve fi=0
40    real :: last_p_dist ! The distance between the first and the last
        ↪  points in the curve.
41    real :: p(2) ! The previous point added to the curve
42    real :: pn(2) ! The new point to be added to the curve
43    real, intent(in) :: fi(ib1:ibn,jb1:jbn) ! level-set function, we want
        ↪  to find a closed curve where fi=0
44    real, intent(inout) :: ibp(2,ib_max_points) ! array containing the
        ↪  points (xi, yi) along the closed curve
45    real :: ibp_tmp(2,ib_max_points) ! tmp array containing the points
        ↪  (xi, yi) along the closed curve
46    real, intent(inout) :: ibdk(2,ib_max_points) ! array containing the
        ↪  equilibrium distance and curvature
47    real, allocatable, dimension(:,:,:) :: abcd ! coeffecients cubic
        ↪  spline for points
48    integer :: i,j
49    real :: t
50
51    ! calculate the mean point distance
52    pd = 0.5*sqrt(dx(1)**2+dy(1)**2) ! pd is longest diagonal plus 1%
53
54    last_p_dist=10.0*pd ! set to value to stop uninitialized memory
        ↪  valgrind error
55
```

```
56    ! bootstrap the algorithm by finding the points on the eulerian grid
        ↪  with
57    ! smalles absolute value
58    if(laxisym) then
59      p = find_axisym_edge(fi, pd)
60    else
61      p = find_start_point(fi, pd)
62    end if
63    heading=0.0
64    npoints=0
65    ibp_tmp=0.0
66    ! while we have not closed the circle or have to few points.
67    do while (last_p_dist>pd .or. npoints<3)
68      ! find next point and add it to the curve
69      call find_next_point(p, heading, fi, pd, pn)
70      npoints = npoints+1
71      ibp_tmp(:,npoints) = pn
72      p = pn
73      if(laxisym) then
74        ! calculate the distance between the currently added point and
            ↪  the y-axis (x==0)
75        last_p_dist = abs(ibp_tmp(1,npoints))
76      else
77        ! calculate the distance between the beginning of the curve and
            ↪  the currently added point
78        last_p_dist = distp(ibp_tmp(:,1)-ibp_tmp(:,npoints))
79      end if
80    end do
81    ! calculate the approximated boundary length
82    boundary_length = (npoints-2)*pd + last_p_dist
83    ! calculate new pd to get wanted amount of points
84    pd = boundary_length/real(npoints-1)
85
86    ibp = ibp_tmp
87    allocate(abcd(2,npoints,4))
88
89    ! we are done placing the rough points
90    ! now calculate a spline following the points, and insert more points
        ↪  on it
91    if(laxisym) then
92      abcd = cubic_spline_normal(ibp)
```

```
93    else
94      abcd = cubic_spline(ibp)
95    endif
96
97    ibp = 0.0
98    do i=1,npoints
99      do j=1,ibdensity
100       t=real(j-1)/real(ibdensity)
101       ibp(:,(i-1)*ibdensity+j) = eval_cubic_spline(abcd, i, t)
102     end do
103   end do
104   npoints = npoints + npoints*(ibdensity-1)
105
106   ! calculate equilibrium distances
107   ibdk(1,1:npoints) = calculate_middle_dist(ibp)
108
109   if(lvar_st) then
110     ibdk(2,1:npoints) = surf0*ibdk(1,1:npoints)
111   else
112     ! calculate equilibrium curvatures
113     ibdk(2,1:npoints) = calculate_curvature(ibp, cubic_spline(ibp))
114   end if
115
116   ! cleanup
117   deallocate(abcd)
118 end subroutine find_boundary
```

Listing 21: find_start_point

```
1  function find_start_point(fi, pd) result(p)
2    ! Finds the point in the level-set field fi
3    ! which has the smalles absolute value.
4    ! This is the starting point for the search in the
5    ! find_boundary routine. For axisymmetric simulations
6    ! the find_axisym_edge is used instead.
7    !
8    ! This function finds a minimum of the bilinear interpolation
9    ! of abs(fi), not the minimum of the discrete field.
10   !
11   !---------------------------------------------------------------------
```

```
12    ! Created: MOL, 2015-04-12.
13    !-----------------------------------------------------------------------
14    use grid, only: ij2p, dx, dy, imax, jmax, ib1, ibn, jb1, jbn
15    real, intent(in) :: pd ! step length for search
16    real, intent(in) :: fi(ib1:ibn,jb1:jbn) ! level-set function, we want
        ↪  to find the minimum of abs(fi)
17    real :: p(2) ! the previous point
18    real :: pn(2) ! the new point
19    real :: heading ! current heading (rad) for the algorithm
20    logical :: on_boundary
21
22    ! inital seed for search
23    p = ij2p(minloc(abs(fi)))
24
25    heading=0.0
26    ! while we have not closed the circle or have to few points.
27    do while (.not. on_boundary)
28      ! gradient decent ontoo the level-set function
29      call find_next_point(p, heading, fi, pd, pn, on_boundary)
30      p = pn
31    end do
32  end function find_start_point
```

Listing 22: find_axisym_edge

```
1   function find_axisym_edge(fi, pd) result(p)
2     ! Finds a point in the level-set field fi
3     ! which has the properties fi==0 and x==0
4     !
5     ! This is the starting point for the search in the
6     ! find_boundary routine when simulating an
7     ! axisymmetric case. For 2D simulations
8     ! the find_start_point is used instead.
9     !
10    ! This function finds a minimum of the bilinear interpolation
11    ! of abs(fi), not the minimum of the discrete field.
12    !
13    !-----------------------------------------------------------------------
14    ! Created: MOL, 2015-04-12.
15    !-----------------------------------------------------------------------
```

```
16    use grid, only: ij2p, dx, dy, imax, jmax, ib1, ibn, jb1, jbn
17    real :: heading ! current heading (rad) for the algorithm
18    real, intent(in) :: pd ! search step length between two points on the
        ↪  curve fi=0
19    real :: p(2) ! the previous point
20    real :: pn(2) ! the new point
21    real, intent(in) :: fi(ib1:ibn,jb1:jbn) ! level-set function
22
23    ! bootstrap the algorithm by finding the points on the eulerian grid
        ↪  with
24    ! smalles absolute value
25    p = ij2p(minloc(abs(fi)))
26    heading=0.0
27    ! while we have not reach the axysymmetric axis
28    do while (p(1)>0.0)
29      ! find next point
30      call find_next_point(p, heading, fi, pd, pn)
31      if(pn(1)<0.0) then
32        p = p+(pn-p)*abs(p(1))/abs(pn(1)-p(1))
33      else
34        p = pn
35      end if
36    end do
37  end function find_axisym_edge
```

Listing 23: find_next_point

```
1   subroutine find_next_point(pa, heading, fi, pd, pb, on_boundary)
2     ! Given a point, pa, level-set function, fi, radius, pb, heading
3     ! output the next point, pb, which is on the curve fi=0 and distance
4     ! pd from pa in the direction of heading +-  pi/1.7.
5     ! A bisection search is used to find the correct angle.
6     !---------------------------------------------------------------------
7     ! Created: MOL, 2015-02-17.
8     !---------------------------------------------------------------------
9     use constants, only: pi
10    use grid, only: ib1,ibn, jb1,jbn
11    real, intent(in) :: pd ! radius of search circle
12    real, intent(in) :: pa(2) ! current point
13    real, intent(in) :: fi(ib1:ibn,jb1:jbn) ! level-set function
```

```
14    real, intent(out) :: pb(2) ! output next point
15    real, intent(inout) :: heading ! the current heading, search
        ↪  direction for next point
16    logical, optional, intent(out) :: on_boundary ! whenether the
        ↪  returned point is on the boundary
17    !
18    ! how much we look left and right for the next point.
19    ! We deliberately do not want to look backwards, because
20    ! this can make us repeatedly turn 180 degrees not making progress.
21    real :: minmaxtheta
22    !
23    ! when the bisection sector has size thres (rad), the bisection is
        ↪  done
24    real :: thres
25    parameter (thres = 1E-12)
26    ! variables for bisection algorithm. t=angle, f=level-set, p=point
27    real :: tlow, thigh, tmid, flow, fhigh, fmid, phigh(2), plow(2),
        ↪  pmid(2)
28
29    ! assume we are on the boundary untill prooven otherwise
30    if(present(on_boundary)) on_boundary=.true.
31
32    minmaxtheta = pi/1.7
33    fmid = huge(fmid)
34    tlow = -minmaxtheta
35    thigh = minmaxtheta
36
37    ! bisection search
38    do while (abs(thigh-tlow) > thres)
39      plow  = pa + pd*dir2p(heading+tlow)
40      phigh = pa + pd*dir2p(heading+thigh)
41
42      flow = bilinear(plow, fi)
43      fhigh = bilinear(phigh, fi)
44      ! swap if low is high
45      if (flow > fhigh) then
46        call swap_real(tlow,thigh)
47        call swap_real(flow,fhigh)
48        call swap_array(plow,phigh)
49      end if
50
```

```
51      if(flow>0.0 .or. fhigh<0.0) then
52        if(present(on_boundary)) on_boundary=.false.
53      end if
54
55      tmid = (tlow+thigh)/2.0
56      pmid = pa + pd*dir2p(heading+tmid)
57      fmid = bilinear(pmid, fi)
58
59      if (0.0 < fmid) then
60        thigh = tmid
61      else
62        tlow = tmid
63      end if
64    end do
65
66    pb = pa + pd*dir2p(heading+tmid)
67    heading = mod(heading + tmid, 2.0*pi)
68  end subroutine find_next_point
```

Listing 24: ellipsoid_ib

```
1   subroutine ellipsoid_ib(x_c, y_c, a, b, ibp, ibdk)
2     !
3     ! Initialize the immersed boundayr as a ellipsoid
4     ! with centre (x_c, y_c) and half axis lengths (a,b)
5     !
6     ! Does not evenly place points out. This is not a problem
7     ! as the method handles this using a variable equilibrium
8     ! distance for each line segment.
9     !
10    !-----------------------------------------------------------------------
11    ! Created: MOL, 2015-04-13.
12    !-----------------------------------------------------------------------
13    use grid, only: dx, dy
14    use constants, only: pi
15    use rhs_var, only: laxisym, lvar_st
16    use surfactants, only: surf0
17    real, intent(in) :: x_c, y_c ! x and y ellipsoid centre
18    real, intent(in) :: a, b ! x and y half axis length
19    real, dimension(2,ib_max_points) :: ibp
```

```
20    real, dimension(2,ib_max_points) :: ibdk
21    real :: pd,arcLength
22    integer :: i,di
23
24    pd = min(dx(1),dy(1))
25    if(laxisym) then
26      arcLength=pi
27      di=1
28    else
29      arcLength=2.0*pi
30      di=0
31    end if
32
33    npoints=ibdensity*int(arcLength*max(a,b)/pd)+di
34    do i=1,npoints
35      ibp(:,i) = [a,b] * &
36        [cos(arcLength*real(i-di)/real(npoints-di)-pi/2.0), &
37         sin(arcLength*real(i-di)/real(npoints-di)-pi/2.0)] + &
38        [x_c, y_c]
39    end do
40
41    ibdk(1,1:npoints) = calculate_middle_dist(ibp)
42
43    if(lvar_st) then
44      ibdk(2,1:npoints) = surf0*ibdk(1,1:npoints)
45    else
46      ! calculate curvature of relaxed ellipse (circle)
47      ibdk(2,1:npoints) = 1.0/((a**2*b)**(1.0/3.0))
48    end if
49
50  end subroutine ellipsoid_ib
```

Listing 25: bilinear

```
1   pure real function bilinear(p, f) result(v)
2     ! Bilinear interpolation of a discrete scalar field for an arbitrary
        ↪  point
3     ! in this field.
4     ! Takes an Eulerian field, f, and a point, p, returns
5     ! the bilinear interpolation of f to p.
```

```
    !
    ! Ref:
      ↪  https://en.wikipedia.org/wiki/Bilinear_interpolation#Algorithm
    !-----------------------------------------------------------------------
    ! Created: MOL, 2015-02-17.
    !-----------------------------------------------------------------------
    use grid, only: ij2p, p2ij, ib1, ibn, jb1, jbn
    real, intent(in) :: p(2) ! point we want to interpolate to
    real, intent(in) :: f(ib1:ibn, jb1:jbn) ! discrete field to
      ↪  interpolate from
    integer :: ij(2)
    real :: xy1(2), xy2(2), x1, x2, y1, y2, Q11, Q12, Q21, Q22
    ij = p2ij(p)
    xy1 = ij2p(ij)
    xy2 = ij2p(ij+1)
    x1 = xy1(1)
    y1 = xy1(2)
    x2 = xy2(1)
    y2 = xy2(2)
    Q11 = f(ij(1),ij(2))
    Q12 = f(ij(1),ij(2)+1)
    Q22 = f(ij(1)+1,ij(2)+1)
    Q21 = f(ij(1)+1,ij(2))
    v = (Q11*(x2-p(1))*(y2-p(2)) + Q21*(p(1)-x1)*(y2-p(2)) +
      ↪  Q12*(x2-p(1))*(p(2)-y1) +
      ↪  Q22*(p(1)-x1)*(p(2)-y1))/((x2-x1)*(y2-y1))
end function bilinear
```

Listing 26: cubic_spline

```
function cubic_spline(ibp) result(abcd)
    ! Calculate a periodic cubic spline contaning the points ibp
    ! Uses an efficient modified Thomas algorithm for solving the
    ! periodic tridiagonal system.
    !
    ! Ref: http://mathworld.wolfram.com/CubicSpline.html
    !-----------------------------------------------------------------------
    ! Created: MOL, 2015-04-04.
    !-----------------------------------------------------------------------
    use linalg, only: solve_constant_symmetric_tridiag_periodic
```

```
11    real, intent(in) :: ibp(2,ib_max_points) ! current point positions
12    real, dimension(2,npoints) :: coeffs
13    real, dimension(2,npoints) :: ibpdiff,sn,sm
14    real, dimension(2,npoints,4) :: abcd ! coeffecients for cubic spline
15    integer :: n,i
16
17    ! alias to make code more readable
18    n = npoints
19
20    ! calculate point distances
21    do i=1,n
22      ibpdiff(:,i) = ibp(:,inext(i)) - ibp(:,iprev(i))
23    end do
24
25    coeffs(1,:) = solve_constant_symmetric_tridiag_periodic(4.0, 1.0,
    ↪  3.0*ibpdiff(1,1:n), n)
26    coeffs(2,:) = solve_constant_symmetric_tridiag_periodic(4.0, 1.0,
        ↪  3.0*ibpdiff(2,1:n), n)
27
28    ! calculate coefficients in polynomials
29    ! a + b*t + c*t^2 + d*t^3
30    ! t in [0,1] for each a, b, c and d
31
32    ! a is the points
33    abcd(:,:,1) = ibp(:,1:n)
34    ! b is the coefficients
35    abcd(:,:,2) = coeffs(:,:)
36
37    do i=1,n
38      ! c
39      abcd(:,i,3) = 3.0*(ibp(:,inext(i))-ibp(:,i))-2.0*coeffs(:,i)-
          ↪  coeffs(:,inext(i))
40      ! d
41      abcd(:,i,4) =
          ↪  2.0*(ibp(:,i)-ibp(:,inext(i)))+coeffs(:,i)+coeffs(:,inext(i))
42    end do
43  end function cubic_spline
```

Listing 27: cubic_spline_normal

```
function cubic_spline_normal(ibp) result(abcd)
  ! Calculate a normal cubic spline contaning the points ibp
  ! mirroring the first and last point around the x-axis.
  ! This special case cubic spline is used for solving the
  ! axisymmetric case, where the spline is not periodic.
  ! Uses an efficient Thomas algorithm for solving the
  ! periodic tridiagonal system.
  !
  ! Ref: http://mathworld.wolfram.com/CubicSpline.html
  !-----------------------------------------------------------------------
  ! Created: MOL, 2015-04-15.
  !-----------------------------------------------------------------------
  use linalg, only: solve_tridiag
  real, intent(in) :: ibp(2,ib_max_points) ! current point positions
  real, dimension(2,npoints+2) :: coeffs
  real, dimension(2,npoints+2) :: ibpdiff,sn,sm
  real, dimension(2,npoints,4) :: abcd ! coeffecients for cubic spline
  real :: ac(npoints+2), b(npoints+2)
  integer :: n,i,ii

  ! alias to make code more readable
  n = npoints

  ! calculate point distances
  do i=1,n
    ibpdiff(:,i+1) = pnext(ibp,i) - pprev(ibp,i)
  end do
  ibpdiff(:,1) = ibp(:,1) - pprev(ibp,1)
  ibpdiff(:,n+2) = pnext(ibp,n) - ibp(:,n)

  ac = 1.0
  b = 4.0
  b(1) = 2.0
  b(n+2) = 2.0

  coeffs(1,:) = solve_tridiag(ac, b, ac, 3.0*ibpdiff(1,:), n+2)
  coeffs(2,:) = solve_tridiag(ac, b, ac, 3.0*ibpdiff(2,:), n+2)

  ! calculate coefficients in polynomials
  ! a + b*t + c*t^2 + d*t^3
  ! t in [0,1] for each a, b, c and d
```

```
42
43    ! a is the points
44    abcd(:,:,1) = ibp(:,1:n)
45    ! b is the coefficients
46    abcd(:,:,2) = coeffs(:,2:n+1)
47
48    do i=1,n
49      ! c
50      abcd(:,i,3) =
        ↪  3.0*(pnext(ibp,i)-ibp(:,i))-2.0*coeffs(:,i+1)-coeffs(:,i+2)
51      ! d
52      abcd(:,i,4) =
        ↪  2.0*(ibp(:,i)-pnext(ibp,i))+coeffs(:,i+1)+coeffs(:,i+2)
53    end do
54  end function cubic_spline_normal
```

Listing 28: eval_cubic_spline

```
1   pure function eval_cubic_spline(abcd, i, t) result(p)
2     ! Evaluate a spline given by the coeficcients abcd
3     ! at spline segment i at t, t in [0,1].
4     !
5     ! Ref: http://mathworld.wolfram.com/CubicSpline.html
6     !-----------------------------------------------------------------------
7     ! Created: MOL, 2015-04-13.
8     !-----------------------------------------------------------------------
9     real, intent(in), dimension(2,npoints,4) :: abcd ! coeffecients cubic
        ↪  spline
10    integer, intent(in) :: i ! index of segment
11    real, intent(in) :: t ! parameter [0,1] on segment i
12    real :: p(2)
13
14    p = abcd(:,i,1) + abcd(:,i,2)*t + abcd(:,i,3)*t**2 + abcd(:,i,4)*t**3
15
16  end function eval_cubic_spline
```

Listing 29: calculate_curvature

```
1   function calculate_curvature(ibp, abcd) result(k)
2     ! Calculate the curvature at knot points of a spline given by the
```

```
3     ! coeficcients abcd.
4     !
5     ! Ref: http://mathworld.wolfram.com/CubicSpline.html
6     !------------------------------------------------------------------------
7     ! Created: MOL, 2015-04-02.
8     !------------------------------------------------------------------------
9     use rhs_var, only: rho, laxisym
10    real, intent(in), dimension(2,npoints,4) :: abcd ! coeffecients for
        ↪  cubic spline
11    real, intent(in), dimension(2,ib_max_points) :: ibp ! current point
        ↪  positions
12    real, dimension(2,npoints) :: d ! first derivative
13    real, dimension(2,npoints) :: dd ! second derivative
14    real, dimension(npoints) :: k ! curvature
15    integer :: n
16
17    n = npoints ! convenience to make code more readable
18
19    ! first derivative at t=0 is b
20    d = abcd(:,:,2)
21
22    ! second derivative at t=0 is 2c
23    dd = 2.0*abcd(:,:,3)
24
25    ! 2d curvature formula, ref:
        ↪  http://mathworld.wolfram.com/Curvature.html
26
27    ! Also, the above formula could be expanded, check
        ↪  level_set_geometry.f90:1231
28    if(laxisym) then
29      k(2:n-1) = -2.0*(ibp(1,2:n-1)*(dd(1,2:n-1)*d(2,2:n-1) -
    ↪   d(1,2:n-1)*dd(2,2:n-1)) - d(2,2:n-1) * sum(d(:,2:n-1)**2,1)) /
    ↪   (2.0*ibp(1,2:n-1)*(sum(d(:,2:n-1)**2,1)**(3.0/2.0)))
30      k(1) = 2.0*(dd(2,1)*d(1,1)-
          ↪  dd(1,1)*d(2,1))/(sum(d(:,1)**2,1)**(3.0/2.0))
31      k(n) = 2.0*(dd(2,n) * d(1,n)-dd(1,n)*d(2,n)) /
          ↪  (sum(d(:,n)**2,1)**(3.0/2.0))
32    else
33      k = (dd(2,:)*d(1,:)-dd(1,:)*d(2,:))/(sum(d**2,1)**(3.0/2.0))
34    end if
35  end function calculate_curvature
```

Listing 30: calculate_curvature_circle

```
function calculate_curvature_circle(ibp) result(k)
  ! Calculate the curvature at points using three point circle
    ↪  approximation.
  ! This gives approximately 2 orders of magnitude bigger errors than
  ! using calculate_curvature which uses a spline.
  !
  ! When used for relaxing ellipse this method was unstable and lead
  ! to huge errors in curvature, enogh to make the simulation blow up.
  !
  ! Ref: http://en.wikipedia.org/wiki/Curvature#Local_expressions
  !---------------------------------------------------------------------
  ! Created: MOL, 2015-04-02.
  !---------------------------------------------------------------------
  real, intent(in), dimension(2,ib_max_points) :: ibp ! coeffecients
    ↪  for polynomial at each point
  real, dimension(npoints) :: k ! curvature
  real :: p1(2), p2(2), p3(2), r
  integer :: i

  do i=1,npoints
    p1 = pprev(ibp,i)
    p2 = ibp(:,i)
    p3 = pnext(ibp,i)

    ! spline approximated curvature
    r = sqrt(((p2(1)-p1(1))**2+(p2(2)-p1(2))**2) *
      ↪  ((p2(1)-p3(1))**2+(p2(2)-p3(2))**2) *
      ↪  ((p3(1)-p1(1))**2+(p3(2)-p1(2))**2)) / (2.0*abs(p1(1)*p2(2) +
      ↪  p2(1)*p3(2) + p3(1)*p1(2) - p1(1)*p3(2) - p2(1)*p1(2) -
      ↪  p3(1)*p2(2)))
    k(i) = 1.0/r
  end do

end function calculate_curvature_circle
```

Listing 31: calculate_tangent

```
pure function calculate_tangent(abcd) result(tangent)
  ! Calculate the tangent at knot points of a spline given by abcd
  !---------------------------------------------------------------------
  ! Created: MOL, 2015-04-10.
  !---------------------------------------------------------------------
  real, dimension(2,npoints,4), intent(in) :: abcd ! coeffecients for
    ↪  cubic spline
  real, dimension(2,npoints) :: tangent ! unit tangent to immersed
    ↪  boundary
  integer :: i
  forall (i=1:npoints)
    tangent(:,i) = abcd(:,i,2)/norm2(abcd(:,i,2))
  end forall
end function calculate_tangent
```

Listing 32: calculate_middle_dist

```
pure function calculate_middle_dist(ibp) result(dist)
  ! Calculate the mean distance between point i-1, i and i, i+1.
  ! This distance is an approximation to the segment length, centered
    ↪  on
  ! points.
  ! This resolves the problem that all quantities except lengths
  ! are stored on points.
  !---------------------------------------------------------------------
  ! Created: MOL, 2015-04-28.
  !---------------------------------------------------------------------
  real, intent(in) :: ibp(2,ib_max_points)
  real :: dist(npoints)
  integer :: i

  ! calculate distance between points
  forall(i=1:npoints)
    dist(i) = (distprev(ibp,i)+distnext(ibp,i))/2.0 ! mean distance
      ↪  between points
  endforall
end function calculate_middle_dist
```

Listing 33: calculate_dist

```
pure function calculate_dist(ibp) result(dist)
  ! Calculate the distance of segment between point i and i+1
  !---------------------------------------------------------------------
  ! Created: MOL, 2015-04-28.
  !---------------------------------------------------------------------
  real, intent(in) :: ibp(2,ib_max_points)
  real :: dist(npoints)
  integer :: i

  ! calculate distance between points
  forall(i=1:npoints)
    dist(i) = distnext(ibp,i)
  endforall
end function calculate_dist
```

Listing 34: calc_outward_normal

```
function calc_outward_normal(ibp, tangent, fi) result(outward_normal)
  ! Calculate the outward normal, this normal points towards
  ! bigger level-set function. It will point from phase2 to phase1.
  !---------------------------------------------------------------------
  ! Created: MOL, 2015-04-09.
  !---------------------------------------------------------------------
  use grid, only: dx, ib1,ibn,jb1,jbn
  real, intent(in) :: ibp(2,ib_max_points) ! current point positions
  real, dimension(2,npoints), intent(in) :: tangent ! unit tangent to
    ↪ immersed boundary
  real, intent(in) :: fi(ib1:ibn,jb1:jbn) ! level-set function
  real, dimension(2,npoints) :: outward_normal ! unit normal to
    ↪ immersed boundary
  integer :: i
  real :: p(2)

  !$OMP PARALLEL DO schedule(guided, 30) private(i, p) shared(ibp, dx,
    ↪ tangent, fi, outward_normal)
  do i=1,npoints
    ! rotate tangent 90 degrees, and walk a small amount in that
      ↪ direction, check if fi is bigger there
    ! if it is this is the outwards normal
    p(1) = ibp(1,i) - dx(1)*tangent(2,i)
```

```
20      p(2) = ibp(2,i) + dx(1)*tangent(1,i)
21
22      if(bilinear(p,fi) > bilinear(ibp(:,i),fi)) then
23        outward_normal(1,i) = -tangent(2,i)
24        outward_normal(2,i) = tangent(1,i)
25      else
26        outward_normal(1,i) = tangent(2,i)
27        outward_normal(2,i) = -tangent(1,i)
28      end if
29    end do
30    !$OMP END PARALLEL DO
31  end function calc_outward_normal
```

Listing 35: inext

```
1   pure integer function inext(i)
2     ! Returns the index of point after point i
3     !---------------------------------------------------------------------
4     ! Created: MOL, 2015-04-06.
5     !---------------------------------------------------------------------
6     use rhs_var, only: laxisym
7     integer, intent(in) :: i
8
9     if(laxisym) then
10      if (i==npoints) then
11        inext=npoints-1
12        return
13      end if
14    end if
15
16    inext = modulo(i,npoints)+1
17  end function inext
```

Listing 36: iprev

```
1   pure integer function iprev(i)
2     ! Returns the index of point before point i
3     !---------------------------------------------------------------------
4     ! Created: MOL, 2015-04-06.
```

```
5     !---------------------------------------------------------------------
6     use rhs_var, only: laxisym
7     integer, intent(in) :: i
8
9     if(laxisym) then
10      if (i==1) then
11        iprev=2
12        return
13      end if
14    end if
15
16    iprev = modulo(i-2,npoints)+1
17  end function iprev
```

Listing 37: pprev

```
1   pure function pprev(points,i)
2     ! Returns the point before point i
3     !---------------------------------------------------------------------
4     ! Created: MOL, 2015-04-06.
5     !---------------------------------------------------------------------
6     use rhs_var, only: laxisym
7     integer, intent(in) :: i
8     real, intent(in) :: points(2, ib_max_points)
9     real pprev(2)
10
11    pprev = points(:,iprev(i))
12
13    ! flip x axis if axisym and on edge
14    if(laxisym .and. i==1) then
15      pprev(1) = -pprev(1)
16    end if
17  end function pprev
```

Listing 38: pnext

```
1   pure function pnext(points,i)
2     ! Returns the point after point i
3     !---------------------------------------------------------------------
```

```
    ! Created: MOL, 2015-04-06.
    !-------------------------------------------------------------------
    use rhs_var, only: laxisym
    integer, intent(in) :: i
    real, intent(in) :: points(2, ib_max_points)
    real pnext(2)

    pnext = points(:,inext(i))

    ! flip x axis if axisym and on edge
    if(laxisym .and. i==npoints) then
      pnext(1) = -pnext(1)
    end if
  end function pnext
```

Listing 39: dir2p

```
  pure function dir2p(d) result(p)
    ! Angle (radians) to unit length vector pointing in
    ! the direction of the angle
    !-------------------------------------------------------------------
    ! Created: MOL, 2015-02-18.
    !-------------------------------------------------------------------
    real, intent(in) :: d
    real :: p(2)
    p = [cos(d), sin(d)]
  end function dir2p
```

Listing 40: dist

```
  elemental real function dist(x,y)
    ! Euclidean norm, component version
    !-------------------------------------------------------------------
    ! Created: MOL, 2015-02-18.
    !-------------------------------------------------------------------
    real, intent(in) :: x, y
    dist=sqrt(x*x+y*y)
  end function dist
```

Listing 41: distp

```
pure real function distp(p)
  ! Euclidean norm, 2-vector version
  !---------------------------------------------------------------------
  ! Created: MOL, 2015-02-18.
  !---------------------------------------------------------------------
  real, intent(in), dimension(2) :: p
  distp=dist(p(1),p(2))
end function distp
```

Listing 42: dist2p

```
pure real function dist2p(ibp, i, j)
  ! Euclidean norm of difference between point i and j
  ! (Distance between point i and j)
  !---------------------------------------------------------------------
  ! Created: MOL, 2015-02-18.
  !---------------------------------------------------------------------
  real, intent(in), dimension(2,ib_max_points) :: ibp
  integer, intent(in) :: i, j
  dist2p=distp(ibp(:,j)-ibp(:,i))
end function dist2p
```

Listing 43: distprev

```
pure real function distprev(ibp, i)
  ! Euclidean distance between point i and the point before it on the
  ! immersed boundary
  !---------------------------------------------------------------------
  ! Created: MOL, 2015-02-18.
  !---------------------------------------------------------------------
  real, intent(in), dimension(2,ib_max_points) :: ibp
  integer, intent(in) :: i
  distprev=distp(pprev(ibp,i)-ibp(:,i))
end function distprev
```

Listing 44: distnext

```
pure real function distnext(ibp, i)
  ! Euclidean distance between point i and the point after it on the
  ! immersed boundary
  !---------------------------------------------------------------------
  ! Created: MOL, 2015-02-18.
  !---------------------------------------------------------------------
  real, intent(in), dimension(2,ib_max_points) :: ibp
  integer, intent(in) :: i
  distnext=distp(pnext(ibp,i)-ibp(:,i))
end function distnext
```

Listing 45: write_ib_to_tecplot

```
subroutine write_ib_to_tecplot(ibp, ibdk, f, fi, t)
  ! Writes the current immersed boundary to a tecplot file with name
  ! levelZ-points.tec
  !
  ! The file contains the  variables:
  !  x,y - position
  !  u,v  - velocity
  !  k - curvature
  !
  !---------------------------------------------------------------------
  ! Created: MOL, 2015-04-17.
  !---------------------------------------------------------------------
  use grid, only: ib1,ibn, jb1, jbn
  use rhs_var, only: laxisym

  real, intent(in) :: f(ib1:ibn,jb1:jbn,2) ! Eulerian staggered
    ↪  velocity field
  real, intent(in) :: fi(ib1:ibn,jb1:jbn,2) ! Eulerian staggered
    ↪  velocity field
  real, intent(in) :: ibp(2,ib_max_points) ! current point positions
  real, intent(in) :: ibdk(2,ib_max_points) ! current point equilibrium
    ↪  distance and curvature
  real, intent(in) :: t ! time
```

```
21    real :: tangent(2,npoints) ! tangents to immersed boundary
22    real :: k(npoints) ! curvature
23    integer :: i
24
25    if(laxisym) then
26      tangent = calculate_tangent(cubic_spline_normal(ibp))
27      k = calculate_curvature(ibp, cubic_spline_normal(ibp))
28    else
29      tangent = calculate_tangent(cubic_spline(ibp))
30      k = calculate_curvature(ibp, cubic_spline(ibp))
31    endif
32
33    call tec_points_2vec_2scalar(ibp(:,1:npoints),
        ↪  interpolate_velocity_field(ibp, f), -calc_outward_normal(ibp,
        ↪  tangent, fi),
34    calculate_middle_dist(ibp), k, npoints, t)
35  end subroutine write_ib_to_tecplot
```

Listing 46: tec_points_2vec_2scalar

```
1   subroutine tec_points_2vec_2scalar(points, vec1, vec2, scalar1,
      ↪  scalar2, npoints, t)
2     ! Output the a list of points, a vector quantity, curvatures in
        ↪  tecplot format to level2-points.tec
3     ! We are using tecplots FELINESEG which can represent a 2D
        ↪  linesegment,
4     ! or a list of points.
5     ! Only one of tec_points and tec_points_vel should be used at a time.
6     ! MOL, 2015-02-04.
7     !-------------------------------------------------------------------
        ↪  -------
8     ! Local variables
9     logical, save :: lopen=.false.
10    integer :: ierr=0,ivar,i,npoints
11    real, intent(in) :: t
12    real, intent(in), dimension(2, npoints) :: points
13    real, intent(in), dimension(2, npoints) :: vec1, vec2
14    real, intent(in), dimension(npoints) :: scalar1, scalar2
15    !-------------------------------------------------------------------
        ↪  -------
16    !
```

```
17    ! Open  file
18    if (.not. lopen) then
19      lopen=.true.
20      open(itec_points,file=tec_pointfile(1:len(tec_pointfile)),&
21           status='unknown',form='formatted',iostat=ierr)
22      if (ierr /= 0) then
23        write(*,*) 'Error while opening file
    ↪  ',trim(tec_pointfile),':',ierr
24        call stoperror('')
25      else
26        ! write header if open successfull
27        write(itec_points,*) 'TITLE = "levelZ - Immersed Boundary"'
28        write(itec_points,*) 'VARIABLES = x, y, u, v, nx, ny, d, k'
29      end if
30    end if
31    !
32    ! Write current ZONE and IB-points to file
33    write(itec_points, 74) t, npoints, npoints, t
34    do i=1,npoints
35      write(itec_points, 78) points(1,i), points(2,i), vec1(1,i),
        ↪  vec1(2,i), vec2(1,i), vec2(2,i), scalar1(i), scalar2(i)
36    enddo
37    write(itec_points,*) ''
38    do i=1,npoints
39      write(itec_points, 79) i, mod(i,npoints)+1
40    enddo
41    ! Under Linux (at least with pgf compilers), output is buffered.
42    ! If the program (or the computer) crashes before the file is
        ↪  properly
43    ! closed, the buffered output is lost. This is fixed by the following
        ↪  call,
44    ! which, however, does not seem to be Fortran standard. Neither of
        ↪  ifort,
45    ! pgf90, or f90 on OSF1 complain, though.
46    flush(itec_points)
47    !
48 74   format(' ZONE T="t=',es12.3,'", DATAPACKING=POINT, NODES=',I9,',
    ↪  ELEMENTS=',I9,', &
49         &ZONETYPE=FELINESEG, DT=(DOUBLE DOUBLE), SOLUTIONTIME=',es12.3)
50 78   format(es18.10, ', ', es18.10, ', ', es18.10, ', ', es18.10, ', ',
    ↪  es18.10, ', ', es18.10, ', ', es18.10, ', ', es18.10)
```

```
79  format(I9,' ',I9)
 end subroutine tec_points_2vec_2scalar
```

Listing 47: swap_real

```
 subroutine swap_real(a,b)
   ! Swap two real values
   !---------------------------------------------------------------------
   ! Created: MOL, 2015-02-18.
   !---------------------------------------------------------------------
   real, intent(inout) :: a,b
   real :: c
   c = a
   a = b
   b = c
 end subroutine swap_real
```

Listing 48: swap_array

```
 subroutine swap_array(a,b)
   ! Swap the values in two arrays
   !---------------------------------------------------------------------
   ! Created: MOL, 2015-02-18.
   !---------------------------------------------------------------------
   real, intent(inout), dimension(:) :: a,b
   real, dimension(size(a)) :: c
   c = a
   a = b
   b = c
 end subroutine swap_array
```

Listing 49: linalg_module_header

```
module linalg
 !> @file
 !> Immersed boundary
 !>
 !> MOL, 2015-02-03.
```

```
6   !> Linear algebra routines, currently only for solving tridiagonal
      ↪  systems of differents sorts.
7   !> Used in cubic spline aprroximation for immersed boundary.
8   !> The routines have been tested up agains the standard Matlab linear
      ↪  system
9   !> solver and they give the same result.
10  implicit none
11  public
12  save
13 contains
```

Listing 50: solve_tridiag

```
1   function solve_tridiag(a,b,c,d,n) result(x)
2     !  a - sub-diagonal variable coeffecient (means it is the diagonal
        ↪  below the main diagonal)
3     !  b - the main diagonal variable coefficient
4     !  c - sup-diagonal variable coeffecient (means it is the diagonal
        ↪  above the main diagonal)
5     !  d - right part
6     !  x - the answer
7     !  n - number of equations
8
9     integer,intent(in) :: n
10    real,dimension(n),intent(in) :: a,b,c,d
11    real,dimension(n) :: x
12    real,dimension(n) :: cp,dp
13    real :: m
14    integer :: i
15
16    ! initialize c-prime and d-prime
17    cp(1) = c(1)/b(1)
18    dp(1) = d(1)/b(1)
19    ! solve for vectors c-prime and d-prime
20    do i = 2,n
21      m = b(i)-cp(i-1)*a(i)
22      cp(i) = c(i)/m
23      dp(i) = (d(i)-dp(i-1)*a(i))/m
24    enddo
25    ! initialize x
```

```
26    x(n) = dp(n)
27    ! solve for x from the vectors c-prime and d-prime
28    do i = n-1, 1, -1
29      x(i) = dp(i)-cp(i)*x(i+1)
30    end do
31  end function solve_tridiag
```

Listing 51: solve_constant_tridiag

```
1   function solve_constant_tridiag(a,b,c,d,n) result(x)
2     !  a - sub-diagonal (means it is the diagonal below the main
        ↪  diagonal)
3     !  b - the main diagonal
4     !  c - sup-diagonal (means it is the diagonal above the main
        ↪  diagonal)
5     !  d - right part
6     !  x - the answer
7     !  n - number of equations
8     integer, intent(in) :: n
9     real, intent(in) :: a,b,c
10    real, dimension(n),intent(in) :: d
11    real, dimension(n) :: x
12    real, dimension(n) :: cp,dp
13    real :: m
14    integer i
15
16    ! initialize c-prime and d-prime
17    cp(1) = c/b
18    dp(1) = d(1)/b
19    ! solve for vectors c-prime and d-prime
20    do i = 2,n
21      m = b-cp(i-1)*a
22      cp(i) = c/m
23      dp(i) = (d(i)-dp(i-1)*a)/m
24    enddo
25    ! initialize x
26    x(n) = dp(n)
27    ! solve for x from the vectors c-prime and d-prime
28    do i = n-1, 1, -1
29      x(i) = dp(i)-cp(i)*x(i+1)
```

```
30    end do
31  end function solve_constant_tridiag
```

Listing 52: solve_constant_symmetric_tridiag_periodic

```
1   function solve_constant_symmetric_tridiag_periodic(a,b,d,n) result(x)
2     ! a - the main diagonal
3     ! b - sub and super-diagonal
4     ! d - right part
5     ! x - the answer
6     ! n - number of equations
7     ! Using tactic for periodic systems from:
        ↪  http://www.cfm.brown.edu/people/gk/chap6/node14.html
8     ! Essentialy reduce the tridiagonal periodic system to two n-1
        ↪  tridiagonal NON-periodic systems.
9     ! http://www.sciencedirect.com/science/article/pii/0021999175900819
10    integer, intent(in) :: n
11    real, intent(in) :: a,b
12    real, dimension(n),intent(in) :: d
13    real, dimension(n) :: x,r
14    real, dimension(n-1) :: d2
15    real :: lambda, alpha, sigma
16    integer :: m,i
17
18    ! calculate factors for first unknown
19    lambda = a/b
20
21    if (lambda > 2.0) then
22      alpha = (-lambda+sqrt(lambda*lambda-4.0))/2.0
23    else if(lambda < -2.0) then
24      alpha = (-lambda-sqrt(lambda*lambda-4.0))/2.0
25    else
26      write(*,*) 'linalg.f90: Error: system not diagonally dominant'
27      stop
28    endif
29
30    sigma = (1.0+alpha*alpha)/(lambda*(1.0-alpha*alpha)*(1.0-alpha**n)*b)
31
32    forall (i=0:n-1)
33      r(i+1) = sigma*(alpha**i+alpha**(n-i))
```

```
34    end forall
35
36    m = (n+1)/2
37
38    x(1) = 0.0
39    do i=2,m
40      x(1) = x(1) + r(i)*(d(i)+d(n+2-i))
41    end do
42
43    if (mod(n,2)==0) then
44      x(1) = x(1) + r(m+1)*d(m+1)
45    end if
46    x(1) = x(1) + r(1)*d(1)
47
48    ! create modified rhs
49    d2 = d(2:n)
50    d2(1) = d2(1)-b*x(1)
51    d2(n-1) = d2(n-1)-b*x(1)
52
53    ! solve rest of the system
54    x(2:n) = solve_constant_tridiag(b,a,b,d2,n-1)
55
56  end function solve_constant_symmetric_tridiag_periodic
```