Westfälische Wilhelms-Universität Münster

Master Thesis

Spreading of simple and complex liquids - numerical approaches using Finite Element Methods

Submitted by:
Tobias Schemmelmann

First examiner:
Prof. Dr. Uwe Thiele
Second examiner:
Priv. Doz. Dr. Svetlana Gurevich

March 15, 2019

Contents

1 Introduction **1**

2 Mathematical concepts **3**
2.1 Weak formulation . . . 3
2.2 Weighted residual . . . 3
2.2.1 Integration by parts . . . 4
2.2.2 Function space . . . 4
2.3 Numerically calculating the weak solution . . . 4
2.3.1 Galerkin method . . . 5
2.3.2 Newton's method in higher dimensions . . . 6
2.3.3 Newton's method incorporating the Galerkin method . . . 7
2.4 Linear stability analysis . . . 7
2.4.1 Numerical linear stability analysis . . . 8
2.5 Finite element method . . . 9
2.5.1 Hat functions . . . 9
2.5.2 Higher order basis functions . . . 10
2.5.3 Facilitating elements . . . 10

3 Gradient dynamics **12**
3.1 General two-field model . . . 12
3.1.1 Calculating the weak solution . . . 13
3.1.2 Weighted residuals . . . 13
3.1.3 Galerkin method . . . 14
3.1.4 Calculating the Jacobian matrix analytically . . . 14
3.2 The `general2field` library . . . 15
3.2.1 General structure . . . 15
3.2.2 `General2FieldEquationsBase` . . . 16
3.2.3 `General2FieldEquations<unsigned DIM>` . . . 16
3.2.4 `General2FieldQElements<unsigned DIM, unsigned NNODE_1D>` . . . 20
3.2.5 `RefineableGeneral2FieldEquations<unsigned DIM>` . . . 22
3.2.6 `RefineableGeneral2FieldQElement<unsigned DIM, unsigned NNODE_1D>` 24

4 Delayed coalescence **25**
4.1 Mathematical modelling . . . 25
4.1.1 Rescaling the model . . . 27
4.2 Implementation using the `general2field` library . . . 27
4.3 The driver code . . . 28
4.3.1 `include` statements . . . 28
4.3.2 `G2FMDelayedCoalescenceFunctionsParameters` namespace . . . 28
4.3.3 `CustomRefineableGeneral2FieldQElements<unsigned DIM, unsigned NNODE_D>` . . . 31
4.3.4 `G2FMUnsteadyProblem` . . . 33
4.3.5 `G2FMUnsteadyProblem<ELEMENT>` methods . . . 33
4.3.6 Assembling the Jacobian matrix . . . 36
4.3.7 `int main(int argc, char* argv[])`, the actual driver code . . . 36
4.4 Spreading and coalescence of neighbouring droplets . . . 39

5 Thin-film equation **45**
5.1 Mathematical modelling . . . 45
5.1.1 Function space . . . 45
5.1.2 Discretisation . . . 46
5.1.3 Jacobian matrix . . . 47
5.2 Implementation in `oomph-lib` . . . 48
5.2.1 `CustomWettingElement<DIM, NNODE_1D>` . . . 48
5.2.2 `RefineableCustomWettingElement<DIM, NNODE_1D>` . . . 53

5.3 Introducing a volume constraint 54
5.3.1 `VolumeConstraintElement` 54
5.3.2 `CustomWettingElement<DIM, NNODE_1D>` 55
5.3.3 `UnsteadyHeatProblem<ELEMENT>` 57
5.4 The driver code 58
5.4.1 Defining system parameters and functions 58
5.4.2 `UnsteadyHeatProblem<ELEMENT>` 60
5.4.3 `int main(int argc, char* argv[])` 61
5.5 Numerical results 64
5.5.1 Approaching a stationary solution using time evolution 65
5.5.2 Continuation 65

6 Summary and outlook **68**

A Appendix **70**
A.1 `general2field` library UML class diagram 70
A.2 UML class diagram for the implementation of the thin-film equation 72
A.3 The (not-so-)quick users guide 74
A.3.1 How to build a `Problem` 74
A.3.2 How to build a `Mesh` 75
A.3.3 How to build a `FiniteElement` 77
A.3.4 How to build a new geometric element 80

Bibliography **84**

1 Introduction

Surface tension accounts for effects that can be observed in day-to-day life, as well as in biological systems and modern technological applications. Water droplets of dew forming in the morning and droplets developing under the lid of a pot of hot water are examples of their occurrence. Inkjet printing is an important technological application and also a present topic of research interest [1, 2]. Hereby, small droplets of ink are placed on the paper by the printer and left to dry, creating the print. Marangoni drying refers to a cleaning process, which uses surface tension gradients to clean semiconductor surfaces [3, 4].
In biological systems, water striders are an example for the usage of surface tension effects. They use the surface tension of stagnant water to stand and walk on top of it [5]. On smaller length scales collective cell migration governed by surface tension effects is observed during the embryogenesis of zebrafish [6]. Considering the previously mentioned examples, we see that surface tension plays a crucial role for many phenomena. Additionally, these effects can be observed on a range of different length scales (see [7] and [8] for reviews). Hence, it is essential to understand these effects and to manipulate them for technological applications.
A surface tension gradient gives rise to the Marangoni effect. It corresponds to a flow driven by a gradient of surface tension along the interface of two fluids. We distinguish between the thermal Marangoni effect, which describes a surface tension gradient caused by a temperature gradient, and the solutal Marangoni effect, which corresponds to a surface tension gradient caused by a concentration gradient [9, 10, 11, 12].
Marangoni flows greatly influence the coalescence of liquid droplets, which are placed close to each other on a solid substrate. This coalescence effect is important in inkjet printing, since two droplets of different dye might coalesce, resulting in a mixture of the two colours and thusly reducing the contrast of the print [13]. Using Marangoni effects, the coalescence behaviour of droplets can be manipulated. Delayed coalescence of different, but completely miscible, liquids due to a surface tension difference has been shown in Ref. [14, 15, 16, 17].

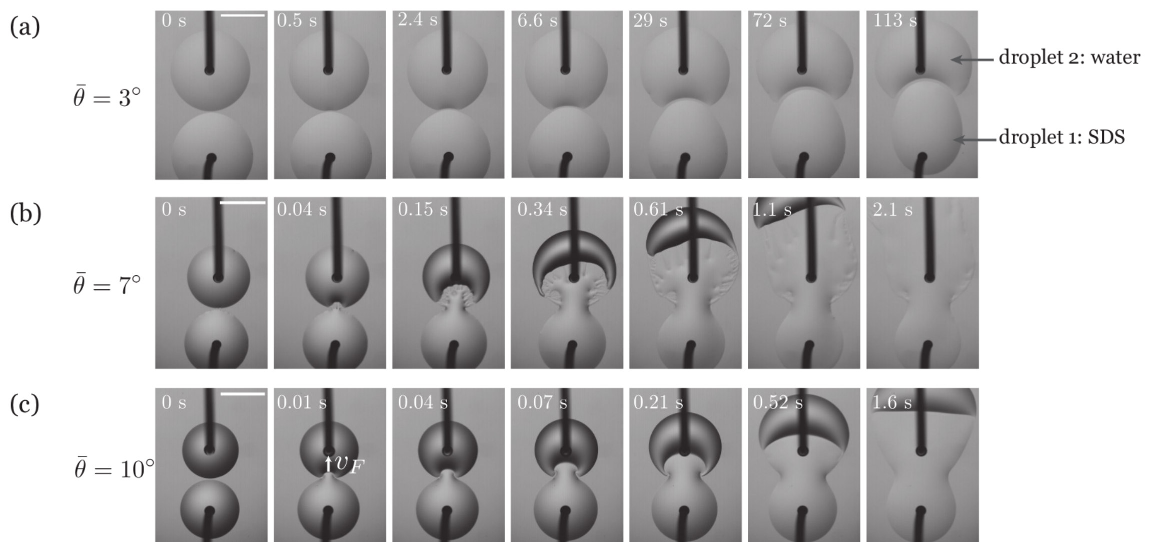

Figure 1: Coalescence behaviour for a surfactant drop (droplet 1) and a pure water drop (droplet 2) [18]. The figure shows that the lower surface tension drop (droplet 1) is attracted towards the higher surface tension drop (droplet 2). For a low mean contact angle (panel (a), $\bar{\theta} = 3°$) we observe delayed coalescence of the two droplets. Fingering instabilities occur in a regime of higher mean contact angles (panel (b), $\bar{\theta} = 7°$). Panel (c) ($\bar{\theta} = 10°$) shows the droplets connected by a stable neck and moving with the velocity v_F. The lower surface tension droplet (SDS) chases the other droplet and simultaneously flattens into a film. Reprinted with permission from Ref. [18] Copyright 2019 by the American Physical Society.

Fig. 1 shows the coalescence behaviour of two neighbouring droplets, of which one contains surfactant and the other droplet is pure water. The surfactant-containing droplet's surface tension

is lower than the surface tension of the pure water droplet. Due to the resulting Marangoni flows different phenomena are observed. A temporary state of non-coalescence, fingering instabilities and motion of droplets connected by stable neck occur. The coalescence behaviour is mainly influenced by the surface tension difference between the two liquids and the mean contact angle of the droplets at their contact line [18].
Despite the Marangoni effect, surface and interface tension give rise to capillarity. It describes the interaction of a liquid with a solid and a gas phase. The interface tension between the liquid and the solid induces a Derjaguin pressure, which drives dewetting of partially wetting liquids. Dewetting describes the process of liquid retraction of a solid substrate, which can result in the formation of droplets separated by dry regions [19].

In this thesis we investigate the influence of surface tension effects on the formation of droplets on a horizontal substrate. Therefore, we consider a partially wetting fluid without surfactant and a fluid that may be covered by insoluble surfactant molecules. Since the partially wetting fluid can be described using one order parameter field equation it is considered a “simple” liquid. Fluids covered by surfactant molecules or containing solutes are considered “complex”, because two or more order parameter field equations are needed for their description. To describe these systems we use partial differential equations (PDEs).
In section 2, we introduce mathematical methods to find solutions of partial differential equations. At first, we introduce the weak formulation and the concept of the weak solution, which does not require the solution to be continuously differentiable in the classical sense. We use a Galerkin method to develop a numerical scheme for the approximation of such solutions. Subsequently, we introduce the finite element method, which is a numerical method that ensures efficient computations.
Section 3 presents a general gradient dynamics model for the description of thin films of complex fluids with two order parameter fields (the general two-field model). It consists of two PDEs that depend on the variations of an underlying free energy functional. We apply the mathematical concepts of the previous section to derive a finite element method for its numerical calculation. After establishing the theoretical concept, we develop a C++ library, the `general2field` library, based on finite element methods.
Many software packages are available for the numerical approximation of solutions of partial differential equations, which are based on finite element or finite difference methods. The `pde2path` package for Matlab handles continuation routines and bifurcation point tracking [20], while `DUNE` calculates the temporal evolution of a system [21]. For the implementation of the `general2field` library, we use the `oomph-lib` framework, since it allows us to calculate the temporal evolution of the system and to perform continuation methods [22].
In section 4, we consider a thin liquid film, which is covered by insoluble surfactant molecules. This system can be described in context of the general two-field model, using the height of the liquid layer and the surfactant concentration as the two order parameter fields. The previously developed `general2field` library allows us to calculate the temporal evolution of the system. We study the coalescence behaviour of two droplets placed next to each other on a flat substrate. A difference in surfactant concentration introduces a surface tension gradient into the system, which can delay the coalescence of the two droplets.
In section 5, we investigate a simple, partially wetting fluid using the thin-film equation. We develop a finite element method for the numerical calculation and implement a C++ program using the `oomph-lib` framework. Hereby, we facilitate continuation routines as well as time evolution methods. In order to use continuation routines, we need to introduce an auxiliary condition into the finite element method. We add an integral auxiliary condition to the system and discuss its implementation. It enables us to determine a bifurcation diagram, which we compare to literature results obtained using `pde2path` [23].
Section 6 concludes the thesis with a summary and an outlook.

2 Mathematical concepts

For the mathematical description of the systems discussed in this thesis we use partial differential equations (PDEs). They are employed to describe the temporal evolution of an observable field-like property of the system (e.g. the height profile of a liquid film). This chapter covers the mathematical concepts, which are used to solve and calculate the PDEs.
Since finding a classical solution for PDEs is in most cases very complicated, sometimes impossible, we introduce the weak formulation as a solution concept. It allows for solutions of PDEs, which are not sufficiently smooth and therefore, not differentiable. Subsequently, we introduce the weighted residual, which enables us to find the weak solutions. We discretise our system and introduce a numerical scheme, the Galerkin method, for the calculation of the weak solution. Finally we introduce the finite element method, which is used to increase the efficiency of our numerical scheme.

2.1 Weak formulation

PDEs are used to describe a wide range of physical phenomena and processes. To mathematically model processes that take place in a real system one usually has to use a finite spatial domain, impose boundary conditions and possibly take other constraints into account. Therefore, it is very complicated to make a statement about the existence and uniqueness of a classical solution, which needs to be sufficiently often continuously differentiable and satisfies the PDEs at every point of the considered domain. Although a classical solution might not exist, the PDEs still describe the physical processes in these systems.
Consequently, we introduce the weak solution, which is an extension of the solution concept that does not require the solution to be differentiable in the classical sense.

Consider a second-order partial differential equation

$$\frac{\partial u(x,t)}{\partial t} = f(u,x,t) + \frac{\partial u(x,t)}{\partial x} + \frac{\partial^2 u(x,t)}{\partial x^2}. \tag{2.1}$$

The goal is to find a solution to the PDE (2.1) on the domain D, which fulfils certain boundary conditions. These can either be Dirichlet boundary conditions, that fix the value of the solution at the domain boundary, or Neumann boundary conditions, which fix the normal derivative of the solution along the boundary of the domain to zero.
We reformulate the PDE (2.1) by introducing the operator $\mathfrak{R}$, which corresponds to the left hand side of equation (2.1) subtracted from the right hand side

$$0 = \mathfrak{R}\left[u\right]. \tag{2.2}$$

2.2 Weighted residual

To find the weak solution $u_w(x)$, we introduce the weighted residual

$$r = \int_D \mathfrak{R}\left[u_w\right] \phi^{test}(x)\mathrm{d}x, \tag{2.3}$$

which vanishes for any suitable test function $\phi^{test}(x)$, if u_w fulfils the boundary condition and is a weak solution of the PDE. The requirements on the test function to be suitable for the weighted residual are discussed in the following section.
Taking a closer look at the weighted residual, one can see the idea behind the concept of the weak solution. The only requirement is the existence of the integral in equation (2.3). This broadens the solution concept by allowing the solution u_w to be any kind of function, so that the integral in equation (2.3) exists. Using integration by parts one can shift the spatial derivatives from the solution u_w onto the test function.

2.2.1 Integration by parts

Consider the Laplace part of $\mathfrak{R}$ and perform integration by parts

$$\int_D \frac{\partial^2 u_w(x,t)}{\partial x^2}\phi^{test}(x)\mathrm{d}x = \underbrace{\left[\frac{\partial u_w(x,t)}{\partial x}\phi^{test}(x)\right]_{\partial D}}_{=A} - \int_D \frac{\partial u_w(x,t)}{\partial x}\frac{\partial \phi^{test}(x)}{\partial x}\mathrm{d}x. \tag{2.4}$$

The consequence of a vanishing A are a sparse diagonal Jacobian matrix and numerically efficient schemes. For Dirichlet boundary conditions $u_w(x) = g(x)\ \forall\ x \in \partial D$, the test function needs to vanish on the domain boundary. Therefore, we need to constrain the test functions to all those which fulfil $\phi^{test}(x) = 0\ \forall\ x \in \partial D$. By implying this constraint on the test function space, A vanishes for any boundary condition $g(x)$.
Employing Neumann boundary conditions we do not need to constrain the test function space, since $\frac{\partial u_w(x)}{\partial x} = 0\ \forall\ x \in \partial D$. Neumann boundary conditions are also called natural boundary conditions, since they do not imply extra constraints on the test function space.

2.2.2 Function space

We introduce $u_a(x,t)$ as an ansatz function to approximate the weak solution $u_w(x,t)$ of equation (2.3). In the previous section we discussed the requirements on the test function, such that the integration by parts in equation (2.4) is satisfied. To keep track of the constraints that the test functions are subjected to, it is useful to define a function space, which collects all functions that satisfy the requirements. Some requirements on the test functions are more subtle and also apply to the ansatz function u_a. Therefore, it is useful to introduce an ansatz space as well, which collects all possible functions u_a.
One of the more subtle restrictions on the functions u_a and ϕ^{test} is that the integral (2.4) needs to be well-defined. Therefore, u_a and ϕ^{test} and their first derivatives need to be square-integrable over D. This can be characterised using the Sobolev-spaces $H^i(D)$, which are defined as spaces, in which the function itself as well as its derivatives up to i-th order are square-integrable over the domain D. The $H^1(D)$ is defined as

$$f(x) \in H^1(D) \iff \int_D \left(f^2(x) + \left(\frac{\partial f(x)}{\partial x}\right)^2\right)\mathrm{d}x < \infty. \tag{2.5}$$

To restrict $H^1(D)$ to functions that vanish on the domain boundary we introduce the index 0

$$f(x) \in H_0^1(D) \iff f(x) \in H^1(D) \quad \text{with} \quad f|_{\partial D} = 0. \tag{2.6}$$

The idea is to find the function space with the fewest restrictions on our test- and ansatz-function, but with the necessary requirements for the integral in (2.3) and (2.4) to exist. This usually depends on the considered PDE and mostly affects the differentiability and the behaviour of the function along the domain boundary.
For equation (2.4) with Neumann boundary conditions, we can chose the functions u_a and ϕ^{test} from the Sobolev-space $H^1(D)$. In case of Dirichlet boundary conditions, $\phi^{test} \in H_0^1(D)$ to ensure that A in equation (2.4) vanishes, even if $u_a \in H^1(D)$. Nonetheless, we ensure the existence of the integral in equation (2.4), which enables us to rewrite equation (2.3) to only depend on the first order spatial derivative. This implies that we only need the first spatial derivative of the ansatz function u_a to be square integrable over the domain D.
In the following section we discuss the numerical approximation of the weak solution.

2.3 Numerically calculating the weak solution

In the previous section we have introduced the weak formulation as an alternative solution concept for PDEs. Subsequently, we introduced the weighted residual (2.3), which vanishes for a weak solution $u_w(x,t)$. Next, we numerically approximate the ansatz function $u_a(x,t)$ to the weak solution $u_w(x,t)$.

2.3.1 Galerkin method

The Galerkin method is a numerical method to calculate approximations for the weak solution $u_w(x,t)$. We expand the ansatz function $u_a(x,t)$ using a finite set of M basis functions $\psi_j(x)$ from a defined ansatz space. Similarly we expand the test function ϕ^{test} into M basis functions of a so called test space. To apply the Galerkin method, we choose the ansatz space and the test space to be the same function space. This results in a system of M equations for the weighted residual in equation (2.3) [24]. If the ansatz and the test space cannot be chosen to be the same function space, a Petrov-Galerkin method needs to be applied [25]. In the following we discuss the discretisation of the ansatz and test function, as well as the application of the Galerkin method.

2.3.1.1 Dirichlet boundary conditions

For Dirichlet boundary conditions, we separate the ansatz function into two parts

$$u_a(x,t) = u_h(x,t) + u_p(x), \tag{2.7}$$

where $u_p(x) = g(x) \; \forall \; x \in \partial D$ is an arbitrary function that satisfies the boundary conditions. $u_h(x)$ is the required solution, which we expand into a set of basis functions of the ansatz-space $H_0^1(D)$ and therefore vanishes at the domain boundaries

$$u_h(x,t) = \sum_{j=1}^{M} U_j(t)\psi_j(x) \quad \text{with} \quad \psi_j(x) \in H_0^1(D). \tag{2.8}$$

Here, $U_j(t)$ are unknown time-dependent coefficients, while the spatial dependence is transferred onto the basis function $\psi_j(x)$.
The objective of the Galerkin method is to calculate the coefficients $U_j(t)$ to determine the weak solution. It implies, that the test function $\phi^{test}(x)$ is expanded into the same set of basis functions, as the ansatz function. Hence $H_0^1(D)$ is also the test-space, resulting in

$$\phi^{test}(x) = \sum_{k=1}^{M} \Phi_k \psi_k(x) \quad \text{with} \quad \psi_k(x) \in H_0^1(D). \tag{2.9}$$

Since $\phi^{test}(x)$ vanishes on the domain boundary, we ensure the existence of the integral in equation (2.4) and that it vanishes at the domain boundary.

2.3.1.2 Neumann boundary conditions

For Neumann boundary conditions, it is not necessary to find a function $u_p(x)$ of a certain value along the domain boundary. They are also called natural boundary conditions, since their application is very intuitive using the weak formulation (consider section 2.2.1). We employ the ansatz

$$u_a(x,t) = \sum_{j=1}^{M} U_j(t)\psi_j(x) \quad \text{with} \quad \psi_j(x) \in H^1(D). \tag{2.10}$$

The test function $\phi^{test}(x)$ is expanded into a finite set of basis functions of the same function space

$$\phi^{test}(x) = \sum_{k=1}^{N} \Phi_k \psi_k(x) \quad \text{with} \quad \psi_k(x) \in H^1(D). \tag{2.11}$$

Due to the Neumann boundary conditions $\left(\frac{\partial u(x)}{\partial x} = 0 \; \forall \; x \in \partial D\right)$ the integral in equation (2.4) exists and vanishes at the domain boundary, although the test functions may not vanish on the domain boundary. Therefore, we use $H^1(D)$ as the test and ansatz space for Neumann boundary conditions, instead of $H_0^1(D)$ for Dirichlet boundary conditions.

By inserting the discretisation of the test and ansatz functions into the definition of the weighted residual (2.3) we obtain

$$r = \int_D \Re\left[\sum_{j=1}^{M} U_j(t)\psi_j(x)\right] \cdot \sum_{k=1}^{N} \Phi_k\psi_k(x)\mathrm{d}x = 0. \tag{2.12}$$

The weighted residual needs to vanish for every suitable test function $\phi^{test}(x)$. Since (2.12) is linear with respect to the basis functions $\psi_k(x)$, the weighted residual vanishes for any coefficient Φ_k

$$\sum_{k=1}^{N} \Phi_k \int_D \Re\left[\sum_{j=1}^{M} U_j(t)\psi_j(x)\right] \cdot \psi_k(x)\mathrm{d}x = 0. \tag{2.13}$$

For every Φ_k we get an equation

$$r_k(U_1, U_2, ..., U_M) = \int_D \Re\left[\sum_{j=1}^{M} U_j(t)\psi_j(x)\right] \cdot \psi_k(x)\mathrm{d}x = 0 \tag{2.14}$$

for the M unknown coefficients $U_j(t)$. Hence, we obtain an equation system

$$r_k(U_1, U_2, ...U_M) = 0 \quad \text{where} \quad k = 1, ..., M, \tag{2.15}$$

which allows us to determine the coefficients $U_j(t)$.

2.3.2 Newton's method in higher dimensions

The equation system in (2.15) can be solved using Newton's method, a common iterative numerical method to find roots of an equation or an equation system. The general idea of Newton's method in one dimension is briefly described in the following section.
First, one needs to make an initial guess for the position of the root. Next, the tangent of the function at this point is determined. The position of the tangent's root marks the starting point for the next iteration and the process is repeated until a certain accuracy is reached [26].

Similar to the one-dimensional case, Newton's Method for higher dimensional functions is also based on a linear approximation of that function. Consider the system

$$\mathbf{f}(\mathbf{x}) = \mathbf{0} \quad \text{with} \quad \mathbf{f} = (f_1, ..., f_N) \quad \text{and} \quad \mathbf{x} = (x_1, ..., x_N) \tag{2.16}$$

with a solution $\mathbf{x_0}$. We perform a Taylor expansion around our initial guess $\mathbf{x_1}$

$$T_1\mathbf{f}(\mathbf{x}; \mathbf{x_1}) = \mathbf{f}(\mathbf{x_1}) + \mathbf{A}(\mathbf{x_1}) \cdot (\mathbf{x} - \mathbf{x_1}) \tag{2.17}$$

where $A_{ij}(\mathbf{x_1}) = \frac{\partial f_i(\mathbf{x_1})}{\partial x_j}$ represents a matrix of first order partial derivatives. It is also referred to as the Jacobian matrix. To find the solution, we set (2.17) to zero and solve for $\mathbf{x}$, yielding

$$\mathbf{x} = \mathbf{x_1} - \mathbf{A}(\mathbf{x_1})^{-1}\mathbf{f}(\mathbf{x_1}). \tag{2.18}$$

The approximate solution $\mathbf{x}$ will then be used as a new initial guess for the Taylor expansion in (2.17). The algorithm can be written as

$$\mathbf{x_{n+1}} = \mathbf{x_n} - \mathbf{A}(\mathbf{x_n})^{-1}\mathbf{f}(\mathbf{x_n}). \tag{2.19}$$

In practice one also implements a criterion to stop the algorithm, when the solution has reached a desired accuracy [26].

2.3.3 Newton's method incorporating the Galerkin method

Applying Newton's method to the Galerkin procedure, we need to find the root of the weighted residual r_k defined in (2.15). In the first iteration step ($n = 1$), we make an initial guess for the unknown coefficients $U_j^{(n)}$ with $j = 1, ..., M$. Then, we calculate the residual vector

$$r_k^{(n)} = r_k \left(U_1^{(n)}, U_2^{(n)}, ..., U_M^{(n)} \right) \quad \text{with} \quad k = 1, ..., M. \tag{2.20}$$

We also need to find and calculate a suitable norm of the residual vector. The norm is required to find a criterion to stop iterations when the solution has reached a desired accuracy. In the following, we employ the maximum norm, if not stated otherwise

$$||u||_{\max} = \max \left(|u| \right). \tag{2.21}$$

To obtain an approximation for the next iteration step, we calculate the Jacobian matrix

$$J_{kl}^{(n)} = \frac{\partial r_k}{\partial U_l} \Bigg|_{\left(U_1^{(n)}, U_2^{(n)}, ..., U_M^{(n)} \right)} \quad \text{with} \quad k, l = 1, ..., M. \tag{2.22}$$

Inserting it into (2.17) yields

$$0 = r_k^{(n)} + J_{kl}^{(n)} \left(U_l^{(n+1)} - U_l^{(n)} \right). \tag{2.23}$$

Next, we calculate the correction

$$\delta U_l = U_l^{(n+1)} - U_l^{(n)} \tag{2.24}$$

to our initial solution $U_l^{(n)}$, by solving the linear equation system

$$\sum_{l=1}^{M} J_{kl} \delta U_l = -r_k^{(n)} \quad \text{for} \quad \delta U_l \quad \text{with} \quad l, k = 1, ..., M. \tag{2.25}$$

To solve the linear equation system for δU_l, we use an algorithm that does not calculate the inverse of the Jacobian matrix (e.g. LU decomposition [26]), since matrix inversions involve high numerical costs. From this we calculate the new solution

$$U_l^{(n+1)} = U_l^{(n)} + \delta U_l \quad \text{with} \quad l = 1, ..., M. \tag{2.26}$$

$U_l^{(n+1)}$ is the starting point for the next iteration step of Newton's Method. This process will be repeated until the norm $||u||_{\max}$ reaches a certain threshold.

2.4 Linear stability analysis

For the description of the behaviour of PDEs it is useful to find its steady states, i.e. solutions that do not change over time. To find out whether a steady solution is stable against small perturbations, we perform a linear stability analysis. Therefore, we rewrite (2.1) into

$$\frac{\partial u(x,t)}{\partial t} \underbrace{- f(u,x,t) - \frac{\partial u(x,t)}{\partial x} - \frac{\partial^2 u(x,t)}{\partial x^2}}_{\mathcal{N}(u,x,t)} = 0 \tag{2.27}$$

and define the operator $\mathcal{N}(u,x,t)$, which might be nonlinear depending on $f(u,x,t)$. We consider a steady state u_s with $\frac{\partial u_s}{\partial t} = 0$ and determine the system's behaviour when applying a small perturbation $\varepsilon \hat{u}$

$$u = u_s + \varepsilon \hat{u} \quad \text{with} \quad |\varepsilon| \ll 1. \tag{2.28}$$

Inserting (2.28) into (2.27) and performing a Taylor expansion of $\mathcal{N}(u, x, t)$ up to linear order in ε we obtain

$$\varepsilon \frac{\partial \hat{u}}{\partial t} - \underbrace{\mathcal{N}(u_s)}_{=0} - \underbrace{\left.\frac{\partial \mathcal{N}}{\partial u}\right|_{u=u_s}}_{\mathcal{L}(u_s)} \varepsilon \hat{u} - \mathcal{O}(\varepsilon^2) = 0. \tag{2.29}$$

Since equation (2.29) is linear in $\hat{u}$, this step is called linearisation and we can refer to the linear operator $\mathcal{L}(u)$ as the linear approximation of $\mathcal{N}(u)$. u_s is a steady state, hence $\mathcal{N}(u_s) = 0$ vanishes. Considering only terms up to linear order of ε in (2.29) we obtain an evolution equation for perturbations $\hat{u}$

$$\frac{\partial \hat{u}}{\partial t} - \mathcal{L}(u_s)\hat{u} = 0. \tag{2.30}$$

To find the temporal behaviour of the perturbation, we employ an exponential ansatz for the perturbation $\hat{u} = ve^{\lambda t}$ and insert it into (2.30), yielding

$$\left(\mathcal{L}(u_s) - \lambda\right) v = 0. \tag{2.31}$$

This results in an eigenvalue problem for the eigenvalues λ and the eigenfunctions v of operator $\mathcal{L}$ on a domain D, which satisfies the boundary conditions. The real part of the eigenvalues determines whether the perturbations $\hat{u}$ decay or grow. If $\mathrm{Re}(\lambda) < 0$, the perturbation decays over time and the steady state is linearly stable. For $\mathrm{Re}(\lambda) > 0$, the steady state is unstable, since perturbations grow exponentially.

2.4.1 Numerical linear stability analysis

We rewrite the weighted residual in (2.3) to

$$r = \int_D \mathfrak{R}\left[u_w\right] \phi^{test}(x)\mathrm{d}x = \int_D \left(\frac{\partial u_w}{\partial t} - \mathcal{N}(u_w)\right) \phi^{test}(x)\mathrm{d}x. \tag{2.32}$$

Similar to section 2.3.1 we discretise the solution according to the Galerkin method. Using the notation from above we can rewrite (2.14) to

$$r_k(U_1, ..., U_M) = \int_D \sum_j^M \frac{\partial U_j(t)}{\partial t} \psi_j \psi_k \mathrm{d}x - \int_D \mathcal{N}\left(u_w\right) \psi_k \mathrm{d}x. \tag{2.33}$$

The nonlinear operator $\mathcal{N}$ can also depend on higher derivatives of u_w, but for simplicity this dependence is omitted in (2.33). Analogously to section 2.4, we consider a small perturbation $\hat{u}$ about a steady state u_s and linearise $\mathcal{N}$ in equation (2.33)

$$0 = \int_D \sum_j^M \frac{\partial \hat{U}_j(t)}{\partial t} \psi_j \psi_k \mathrm{d}x - \int_D \mathcal{L}\left(u_s\right) \sum_j^M \hat{U}_j \psi_j \psi_k \mathrm{d}x. \tag{2.34}$$

Since we consider the weighted residual $r_k(U_1, ..., U_M)$ at the steady state u_s, it vanishes in (2.34). For the coefficients of the perturbation $\hat{U}_j(t)$ we assume an exponential ansatz function $\hat{U}_j(t) = V_j e^{\lambda t}$. Inserting this into (2.34), yields

$$\int_D \mathcal{L}\left(u_s\right) \sum_j^M V_j \psi_j \psi_k \mathrm{d}x = \lambda \int_D \sum_j^M V_j \psi_j \psi_k \mathrm{d}x \quad \text{where} \quad k = 1, ..., M. \tag{2.35}$$

Analogously to section 2.4, we obtain our eigenvalue problem for the eigenvalues λ. We call

$$M_{jk} = \int_D \psi_j \psi_k \mathrm{d}x \tag{2.36}$$

the mass matrix. Due to the orthogonality of the basis functions ψ_i the mass matrix usually corresponds to the identity. Note that the right hand side of (2.35) originates from the time derivative of the field u_w. In case of a PDE of a field with no time derivative, it will be 0, as well as the corresponding entry in the mass matrix.

The Jacobian matrix in section 2.3.2 corresponds to the derivative of $\mathfrak{R}[u_w]$ with respect to u_w, while $\mathcal{L}(u_w)$ is the derivative of $\mathcal{N}(u_w)$ with respect to u_w. Considering (2.32) we see that $\mathcal{L}(u_w)$ is exactly the same, except for the time derivative term.

2.5 Finite element method

In section 2.3, we established a numerical scheme to calculate the weak solution. Next, we introduce the finite element method, which ensures that we fulfil all the requirements of our ansatz and test functions as well as create a numerically efficient scheme. The main idea is to find basis functions for our test and ansatz space, that have finite support over the domain. Using these basis functions results in a sparse Jacobian matrix with only few nonzero entries in the vicinity of the diagonal.
To find these basis functions, we group neighbouring nodes into so called elements. Within these elements we define basis functions with compact support. Additionally we find a local representation of the basis functions and the nodes in the element. Using this local representation, we calculate the residual vector and Jacobian matrix. Following a mapping between the local representation of the element and a global representation, we can assemble the contribution of each element to the residual vector and Jacobian matrix of the global system.

2.5.1 Hat functions

The most basic basis functions are so called "hat functions". They are piecewise linear functions between the nodes of the domain. To illustrate them, we consider an example in one dimension and divide the domain $x \in [a, b]$ in N not necessarily equally-sized intervals separated by N nodes. The position of the nodes are given by $X_j \in [a, b]$ with $j = 0, ..., N$. Then, we define our basis functions as piecewise linear interpolations between the corresponding nodes and their direct neighbours

$$\psi_j(x) = \begin{cases} \frac{x - X_{j-1}}{X_j - X_{j-1}} & \text{for} \quad X_{j-1} < x \leq X_j \\ \frac{X_{j+1} - x}{X_{j+1} - X_j} & \text{for} \quad X_j < x < X_{j+1} \\ 0 & \text{else.} \end{cases} \tag{2.37}$$

Fig. 2 shows the nodes, a hat function and a corresponding piecewise linear solution. By choosing

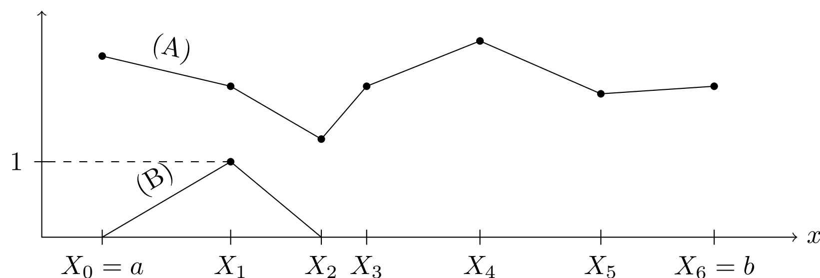

Figure 2: (A) Piecewise linear solution, (B) "hat" shaped basis function $\psi_1(x)$[25].

smaller distances h between neighbouring nodes in regions, where the solution function changes rapidly, we can increase the accuracy of our approximate solution. This is known as mesh refinement or "h-refinement". The basis functions must fulfil the interpolation condition $\psi_j(X_i) = \delta_{ij}$. It ensures that the coefficient's value $U_j(t)$ corresponds to the solution's value at X_j.

2.5.2 Higher order basis functions

We can also increase the accuracy of our interpolation of the solution function by choosing basis functions of higher order. This increases the dimension of our test and ansatz space. Defining quadratic or higher order functions as basis functions is called “p-refinement”, since the order of the polynomials representing the solution is increased.
When using higher order polynomials as basis functions it is important to maintain the compact support and to make sure that they still fulfil the interpolation condition $\psi_j(X_i) = \delta_{ij}$. Fig. 3 shows an illustrative example of quadratic basis functions in the domain $x \in [a, b]$ with equally-spaced nodes $X_j \in [a, b]$.

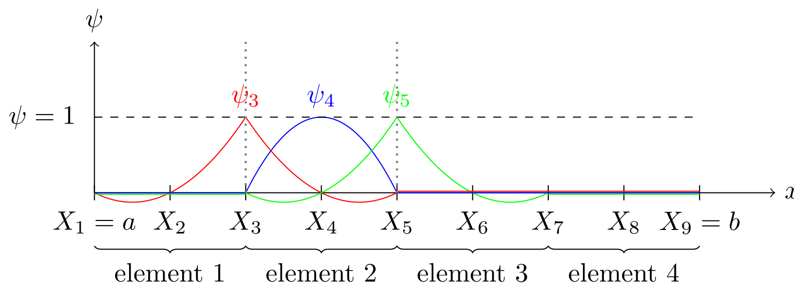

Figure 3: Piecewise quadratic basis functions for a finite element in $x \in [X_3, X_5]$

Notice that in Fig. 3, we use two different quadratic basis functions. One basis function is used for the nodes that lie on the boundary of the element (i.e. ψ_3 and ψ_5) and the other one for a non-boundary node (i.e. ψ_4).

2.5.3 Facilitating elements

In Fig. 3, we have introduced elements, which divide the domain into smaller subdomains. The finite element method facilitates the usage of those elements and their basis functions with compact support to make the calculation of the residual vector and the Jacobian matrix more efficient. Mathematically, it is not necessary to divide the domain into elements, but it makes the implementation easier.
To determine the solution of our problem, we calculate the residual vector of length N, where N represents the amount of nodes we use to spatially discretise the domain D. For each node, we calculate the residual vector entry r_k (2.14), consisting of an integral over the whole domain D. Since we use basis functions with compact support, the integral is nonzero only in the vicinity of the regarded node. The same holds for the contribution of each node to the Jacobian matrix. In the Jacobian matrix, each node is represented by a row, but only the entries close to the diagonal and minor diagonals are nonzero.
With this in mind we implement an algorithm, which takes advantage of these properties. In `oomph-lib`, a scheme is developed that loops over all elements and calculates the contribution of the element to the residual vector and the Jacobian matrix locally (only within the current element). After the local calculation, the respective contributions of the elements are added to the global residual vector and Jacobian matrix. This scheme ensures that the integrals of the residual and Jacobian are only calculated within a region where they are nonzero.

To implement a local calculation of the residual vector and Jacobian matrix, we need to introduce local coordinates, as well as local and global node and equation numbers. We assign a unique global node number to every node, as well as a unique global equation number to every degree of freedom $U_j(t)$ (also referred to as unknown). Similarly every element assigns a local node number to its nodes, which is only unique within the element. Analogously, it assigns a local equation number to its unknown values. We establish a mapping $\mathfrak{J}(j_{local}, e) = j_{global}$ between the local node number j_{local} of the corresponding element e to the global node number j_{global}, as well as a mapping $\hat{\varepsilon}(i_{local}, e) = i_{global}$ between the local and global equation number. The global node and

equation numbers, as well as their local counterparts, are shown in Fig. 4 for a one dimensional mesh with three-node elements.

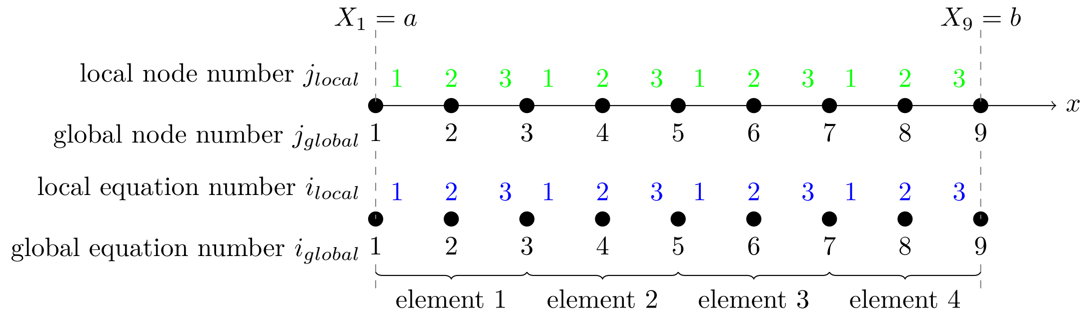

Figure 4: Local and global node and equation numbers of a one dimensional grid with four elements containing three nodes. Note that a node can have multiple local node or equation numbers. This happens when the node is a boundary node and is part of two elements. Nonetheless, the local number within the element needs to be unique, while only the global numbers are unique for every node.

Using the local coordinates we calculate the residual vector and the Jacobian matrix for each element locally. After the calculation, we inverse the mappings $\mathfrak{J}$ and $\hat{\varepsilon}$ to assemble the global residual vector and global Jacobian matrix. Therefore, we fill the local contributions of the residual vector and Jacobian matrix of each element into the according places of their global counterparts. This algorithm divides the numerical cost into many smaller calculations and assembles them to the final result afterwards. Additionally, this algorithm ensures numerical efficiency, by only calculating nonzero parts of the Jacobian matrix.

3 Gradient dynamics

To describe the dynamical behaviour of Newtonian fluids, the Navier-Stokes equations are used. Together with the continuity equation they allow the mathematical description of fluid mechanics. In the following, we are considering liquid films on a solid substrate. Hence, the governing equations are accompanied by suitable boundary conditions. At the solid substrate, no-slip and no-penetration boundary conditions are assumed, i.e. the tangential and normal velocities are zero. At the free liquid surface, normal and tangential stress conditions and a kinematic condition are employed.
The thin-film or lubrication approximation is used to reduce the mathematical complexity of the Navier-Stokes equations. It uses the greatly different length scales of the lateral and vertical directions of the liquid film, to derive an evolution equation for the film height h, the thin-film equation.
The structure of the thin-film equation is a continuity equation for the film height $\partial_t h = \vec{\nabla}\vec{j}$ where the flux $\vec{j}$ itself can be expressed as the product of a mobility function Q and the gradient of a pressure written as the variation of a free energy functional $\mathcal{F}$ with respect to the liquid film height h [27, 28].
To model the behaviour of complex fluids, we simply add the corresponding energetic contributions to the free energy functional and extend the gradient dynamics formulation to multiple order parameter fields. We need to ensure that the order parameter fields are independent variables, such that the variation of $\mathcal{F}$ with respect to one field, does not influence the other [29, 30, 31].
In the following a similar representation of the evolution equations for two order parameter fields is used for the general two-field model.

3.1 General two-field model

The general two-field model is describes a relaxational system including two conserved order parameter fields [29]. It can be used to describe systems with two-layer films, a one-layer film with an insoluble surfactant or a one-layer film of a mixture. In a two-layer film system the two order parameter fields correspond to the film heights of both films, while in a one-layer film system the order parameter fields correspond to the film height and to the projection of the surfactant concentration on the free film surface onto the cartesian substrate plane [32, 31] or to the local amount of solute in the liquid [33].
Equations (3.1)-(3.5) show the general two-field model for the conserved quantities u_1 and u_2.

$$\partial_t u_1 = \nabla \cdot \left[Q_{11} \nabla \frac{\delta \mathcal{F}}{\delta u_1} + Q_{12} \nabla \frac{\delta \mathcal{F}}{\delta u_2} \right] \tag{3.1}$$

$$\partial_t u_2 = \nabla \cdot \left[Q_{21} \nabla \frac{\delta \mathcal{F}}{\delta u_1} + Q_{22} \nabla \frac{\delta \mathcal{F}}{\delta u_2} \right] \tag{3.2}$$

The specific models are defined by the mobility functions $\boldsymbol{Q}$

$$\boldsymbol{Q} = \begin{pmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{pmatrix} \tag{3.3}$$

and the variations of the free energy functional $\mathcal{F}$. We restrict the mobility functions Q_{ij} to functions, that only depend on the order parameter fields, but not on their spatial derivatives, i.e. $Q_{ij} = Q_{ij}(u_1, u_2)$. Additionally, we impose a restriction on the free energy functional $\mathcal{F}$. The variations of the free energy functional must be a function G_i, whose highest order of spatial derivative does not exceed two, e.g. terms like $\nabla u_i \Delta u_j$ or $(\nabla u_i)^3$ are not allowed.

$$w_1 = \frac{\delta \mathcal{F}}{\delta u_1} = G_1 \left(u_i, \nabla u_i, \Delta u_i \right) \tag{3.4}$$

$$w_2 = \frac{\delta \mathcal{F}}{\delta u_2} = G_2 \left(u_i, \nabla u_i, \Delta u_i \right) \tag{3.5}$$

By imposing these restrictions, we are able to implement a numerically efficient algorithm to simulate the evolution equations, while we still cover a large group of hydrodynamic thin-film systems.

3.1.1 Calculating the weak solution

For the calculation of the evolution equations we apply a finite element method to equations (3.1) and (3.2). To take full advantage of the finite element method, the equations must not include spatial derivatives of order greater than two. Therefore we introduce two auxiliary order parameter fields w_1 and w_2, which correspond to the variations of the free energy functional ((3.4) and (3.5)).

$$\partial_t u_1 = \nabla \left[Q_{11} \nabla w_1 + Q_{12} \nabla w_2 \right] \tag{3.6}$$
$$\partial_t u_2 = \nabla \left[Q_{21} \nabla w_1 + Q_{22} \nabla w_2 \right] \tag{3.7}$$
$$w_1 = G_1(u_i, ...) \tag{3.8}$$
$$w_2 = G_2(u_i, ...) \tag{3.9}$$

This means that the original evolution equations (3.1) and (3.2) can incorporate spatial derivatives up to fourth order. In case of higher orders, one would need to introduce another pair of auxiliary order parameter fields. The systems regarded here do not exceed fourth order and therefore the introduction of one pair is sufficient.

3.1.2 Weighted residuals

For the calculation of the weak solution we multiply (3.6) and (3.7) by the test functions ϕ_{u_i} and integrate over the complete domain D. Using integration by parts we shift the Nabla operator onto the test function. Hereby, we exploit the test functions property that it vanishes on the boundaries of the domain (see section 2.2.1)

$$r_{u_1} = -\int_D \partial_t u_1 \phi_{u_1} \mathrm{d}\vec{x} + \int_D \nabla \cdot \left[Q_{11} \nabla w_1 + Q_{12} \nabla w_2 \right] \phi_{u_1} \mathrm{d}\vec{x} \tag{3.10}$$
$$= -\int_D \partial_t u_1 \phi_{u_1} \mathrm{d}\vec{x} - \int_D \left[Q_{11} \nabla w_1 + Q_{12} \nabla w_2 \right] \cdot \nabla \phi_{u_1} \mathrm{d}\vec{x} + \underbrace{\int_{\partial D} \left[Q_{11} \nabla w_1 + Q_{12} \nabla w_2 \right] \phi_{u_1} \cdot \vec{n} \mathrm{d}S}_{=0} \tag{3.11}$$
$$r_{u_2} = -\int_D \partial_t u_2 \phi_{u_2} \mathrm{d}\vec{x} + \int_D \left[Q_{21} \nabla w_1 + Q_{22} \nabla w_2 \right] \cdot \nabla \phi_{u_2} \mathrm{d}\vec{x}. \tag{3.12}$$

In case of the auxiliary order parameter fields, we multiply the equations (3.8) and (3.9) by the test functions ϕ_{w_i} and integrate over the domain D

$$r_{w_1} = -\int_D w_1 \phi_{w_1} \mathrm{d}\vec{x} + \int_D G_1(u_i, ...) \phi_{w_1} \mathrm{d}\vec{x} \tag{3.13}$$
$$= -\int_D w_1 \phi_{w_1} \mathrm{d}\vec{x} + \int_D \left[g_{11}(u_i, ...) - \nabla \vec{g}_{12}(u_i, ...) \right] \phi_{w_1} \mathrm{d}\vec{x} \tag{3.14}$$
$$= -\int_D w_1 \phi_{w_1} \mathrm{d}\vec{x} + \int_D g_{11}(u_i, ...) \phi_{w_1} \mathrm{d}\vec{x} + \int_D \vec{g}_{12}(u_i, ...) \nabla \phi_{w_1} \mathrm{d}\vec{x} \tag{3.15}$$
$$r_{w_2} = -\int_D w_2 \phi_{w_2} \mathrm{d}\vec{x} + \int_D g_{21}(u_i, ...) \phi_{w_2} \mathrm{d}\vec{x} + \int_D \vec{g}_{22}(u_i, ...) \nabla \phi_{w_2} \mathrm{d}\vec{x}. \tag{3.16}$$

When calculating the weak formulation of (3.8) we split the function G_1 into two parts g_{11} and $-\nabla \vec{g}_{12}$. The function g_{11} contains all the parts of G_1, which are of spatial derivative order one or lower, while $\nabla \vec{g}_{12}$ is of spatial derivative order two. We perform integration by parts with $\nabla \vec{g}_{12}$ and shift the spatial derivative onto the test function ϕ_{w_1}. The same procedure is applied to (3.9) resulting in (3.15) and (3.16).

3.1.3 Galerkin method

We discretise our test and ansatz functions as described in section 2.3.1.

$$u_1(\vec{x},t) = \sum_j U_{1,j}(t)\psi_j(\vec{x}) \qquad \phi_{u_1}(\vec{x}) = \sum_l \Phi_{u_1,l}\psi_l(\vec{x}) \tag{3.17}$$

$$u_2(\vec{x},t) = \sum_j U_{2,j}(t)\psi_j(\vec{x}) \qquad \phi_{u_2}(\vec{x}) = \sum_l \Phi_{u_2,l}\psi_l(\vec{x}) \tag{3.18}$$

$$w_1(\vec{x},t) = \sum_j W_{1,j}\psi_j(\vec{x}) \qquad \phi_{w_1}(\vec{x}) = \sum_l \Phi_{w_1,l}\psi_l(\vec{x}) \tag{3.19}$$

$$w_2(\vec{x},t) = \sum_j W_{2,j}\psi_j(\vec{x}) \qquad \phi_{w_2}(\vec{x}) = \sum_l \Phi_{w_2,l}\psi_l(\vec{x}) \tag{3.20}$$

These equations are inserted into the weighted residuals (3.11), (3.12), (3.15) and (3.16). Similar to section 2.3.1, we arrive at a linear equation system for each of the weighted residuals $r_{u_1,l}$, $r_{u_2,l}$, $r_{w_1,l}$ and $r_{w_2,l}$. Hereby, the first index denotes the residual of the field we are currently regarding, while the second index indicates the test function, that the residual is multiplied by. Equation (3.21) displays $r_{u_1,l}$ exemplarily, using the previously defined discretisation

$$\begin{aligned} r_{u_1,l} = &-\int_D \partial_t \sum_j U_{1,j}(t)\psi_j(\vec{x})\psi_l(\vec{x})\mathrm{d}\vec{x} \\ &-\int_D \left[Q_{11}\nabla\cdot\left(\sum_j W_{1,j}\psi_j(\vec{x})\right) + Q_{12}\nabla\cdot\left(\sum_j W_{2,j}\psi_j(\vec{x})\right)\right]\nabla\psi_l(\vec{x})\mathrm{d}\vec{x}. \end{aligned} \tag{3.21}$$

3.1.4 Calculating the Jacobian matrix analytically

Using the above mentioned discretisation we can calculate the Jacobian matrix analytically. Therefore, we need to calculate the derivative of every weighted residual with respect to every order parameter field. This results in a Jacobian matrix, which consists of 16 parts. Here $J_{u_1;l,u_2;l2}$ corresponds to the derivative of the weighted residual $r_{u_1,l}$ with respect to $U_{2,l2}$

$$J_{u_1;l,u_2;l2} = \left.\frac{\partial r_{u_1,l}}{\partial U_{2,l2}}\right|_{(U_{1,1},U_{1,1},\ldots,U_{2,1},U_{2,2},\ldots,W_{1,1},W_{1,2},\ldots,W_{2,1},W_{2,2},\ldots)}. \tag{3.22}$$

Equations (3.23) - (3.38) show the analytical representation of the Jacobian matrix. In the following we omit the indices l and $l2$ of J, since they do not provide additional information.

$$J_{u_1,u_1} = -\int_D \partial_{U_{1,l2}}(\partial_t u_1)\psi_l\mathrm{d}\vec{x} - \int_D [(\partial_{u_1}Q_{11})\psi_{l2}\nabla w_1 + (\partial_{u_1}Q_{12})\psi_{l2}\nabla w_2]\cdot\nabla\psi_l\mathrm{d}\vec{x} \tag{3.23}$$

$$\text{with} \quad \frac{\partial Q_{11}}{\partial U_{1,l2}} = \frac{\partial Q_{11}(u_1,u_2)}{\partial u_1}\frac{\partial u_1}{\partial U_{1,l2}} = (\partial_{u_1}Q_{11})\psi_{l2}$$

$$J_{u_1,w_1} = -\int_D Q_{11}\nabla\psi_{l2}\cdot\nabla\psi_l\mathrm{d}\vec{x} \tag{3.24}$$

$$J_{u_1,u_2} = -\int_D [(\partial_{u_2}Q_{11})\psi_{l2}\nabla w_1 + (\partial_{u_2}Q_{12})\psi_{l2}\nabla w_2]\cdot\nabla\psi_l\mathrm{d}\vec{x} \tag{3.25}$$

$$J_{u_1,w_2} = -\int_D Q_{12}\nabla\psi_{l2}\cdot\nabla\psi_l\mathrm{d}\vec{x} \tag{3.26}$$

$$J_{w_1,u_1} = \int_D \left(\partial_{U_{1,l2}}g_{11}\right)\psi_l\mathrm{d}\vec{x} + \int_D \left(\partial_{U_{1,l2}}g_{12}\right)\nabla\psi_l\mathrm{d}\vec{x} \tag{3.27}$$

$$J_{w_1,w_1} = -\int_D \psi_{l2}\psi_l\mathrm{d}\vec{x} \tag{3.28}$$

$$J_{w_1,u_2} = \int_D \left(\partial_{U_{2,l2}}g_{11}\right)\psi_l\mathrm{d}\vec{x} + \int_D \left(\partial_{U_{2,l2}}g_{12}\right)\nabla\psi_l\mathrm{d}\vec{x} \tag{3.29}$$

$$J_{w_1,w_2} = 0 \tag{3.30}$$

$$J_{u_2,u_1} = -\int_D [(\partial_{u_1} Q_{21})\psi_{l2}\nabla w_1 + (\partial_{u_1} Q_{22})\psi_{l2}\nabla w_2] \cdot \nabla\psi_l \mathrm{d}\vec{x} \tag{3.31}$$

$$J_{u_2,w_1} = -\int_D Q_{21}\nabla\psi_{l2} \cdot \nabla\psi_l \mathrm{d}\vec{x} \tag{3.32}$$

$$J_{u_2,u_2} = -\int_D \partial_{U_{2,l2}}(\partial_t u_2)\psi_l \mathrm{d}\vec{x} - \int_D [(\partial_{u_2} Q_{21})\psi_{l2}\nabla w_1 + (\partial_{u_2} Q_{22})\psi_{l2}\nabla w_2] \cdot \nabla\psi_l \mathrm{d}\vec{x} \tag{3.33}$$

$$J_{u_2,w_2} = -\int_D Q_{22}\nabla\psi_{l2} \cdot \nabla\psi_l \mathrm{d}\vec{x} \tag{3.34}$$

$$J_{w_2,u_1} = \int_D \left(\partial_{U_{1,l2}} g_{21}\right) \psi_l \mathrm{d}\vec{x} + \int_D \left(\partial_{U_{1,l2}} g_{22}\right) \nabla\psi_l \mathrm{d}\vec{x} \tag{3.35}$$

$$J_{w_2,w_1} = 0 \tag{3.36}$$

$$J_{w_2,u_2} = \int_D \left(\partial_{U_{2,l2}} g_{21}\right) \psi_l \mathrm{d}\vec{x} + \int_D \left(\partial_{U_{2,l2}} g_{22}\right) \nabla\psi_l \mathrm{d}\vec{x} \tag{3.37}$$

$$J_{w_2,w_2} = -\int_D \psi_{l2}\psi_l \mathrm{d}\vec{x} \tag{3.38}$$

3.2 The `general2field` library

We implement a set of oomph-lib element classes, that represent the general two-field model as shown in (3.6) - (3.9). The purpose is to create a library that represents the problem in a very general way, while also being easy to use. It tries to strictly separate between the code necessary for every general two-field model and methods for the specific model (e.g. the mobility functions $\mathbf{Q}$). The library ensures that the user does not need to implement or change the calculation of the residual vector or the Jacobian matrix. He only needs to make sure to set the function pointers to the corresponding functions of his specific problem.
These function pointers are set in a so called "driver code". The driver code contains the parts of code, that are individual for every system and it inherits general methods and functionality from the above mentioned library. It usually initialises an object of the `oomph::Problem` class and controls the steps during and along the simulation. Therefore the element classes in the library will also provide a set of useful functions for analysis during or after the simulation.
The following chapter discusses the implementation of the library `general2field` in details. In appendix A.1 a class diagram, visualising the inheritance hierarchy of the library, is given. For the implementation of an actual system see the driver code in section 4.3. To use the library, we need to place the header file `G2FM_Single_File_unc.h` and the definition file `G2FM_Single_File_unc.cc` in a subfolder of the `user_src` directory of the local `oomph-lib` installation and create an according `makefile` [1].

3.2.1 General structure

The base class is `General2FieldEquationsBase`, which inherits from `FiniteElement`. It is independent of the spatial dimension of the considered system and creates the `typedef` definitions of the function pointers, as well as provides virtual access methods to the function pointers.
Most of the magic happens in `General2FieldEquations<DIM>`, which inherits from the above mentioned `General2FieldEquationsBase` class. It implements the calculation of the residual vector, Jacobian matrix and mass matrix. Therefore, it also handles the access to the function pointers, which are used to specify the mobility functions and the functions that make up the weak formulation of the free energy variation. In contrast to its parent class `General2FieldEquationsBase`, the class is templated by the `DIM` parameter, which represents the spatial dimension of the problem.
The element class itself is `General2FieldQElements<DIM, NNODE_1D>`, which inherits from `General2FieldEquations<DIM>` and `QElement<DIM,NNODE_1D>`. It is templated by the spatial dimension `DIM` and the number of nodes along one spatial direction `NNODE_1D`. The class holds the `Initial_Nvalue` attribute, which defines the number of free variables stored at each node. For the general two-field model this needs to be set to 4, for the values of u_1, w_1, u_2 and w_2. More

[1] `http://oomph-lib.maths.man.ac.uk/doc/the_distribution/html/index.html#add_src`

importantly, this class is used to define the shape functions that are used for the finite element method. The shape functions that are used for the assembly of the residual vector and the Jacobian matrix in its parent class `General2FieldEquations<DIM>` are only virtual functions.
To include mesh adaption during the simulation we create the class `RefineableGeneral2FieldEquations<DIM>`, which inherits from `General2FieldEquations<DIM>`, `RefineableElement` and `ElementWithZ2ErrorEstimator`. It includes the functionality of a spatial error estimator for the automatic mesh adaption process. The class overloads the method for the calculation of the residual vector, Jacobian matrix and mass matrix from its parent class, to fit the needs of a mesh adaption routine. Similar to its parent class it is only templated by the spatial dimension parameter to keep the code general.
The element that needs to be used in simulations with mesh refinement is `RefineableGeneral2FieldQElement<DIM, NNODE_1D>`. It inherits from `General2FieldQElements<DIM, NNODE_1D>` and `RefineableGeneral2FieldEquations<DIM>`.

3.2.2 General2FieldEquationsBase

This is the base class of the library. It holds the `typedef` definitions of the function pointers. They are used to provide the mobility functions Q_{ij} and the functions g_{ij} from the weak formulation of the variation of the free energy functional $\mathcal{F}$. The concept behind this strategy is, that each user creates his individual mobility functions and sets the function pointers to point to his individual methods. In the following, the usage of function pointers is demonstrated and explained by the example of the mobility function Q_{11}.

```
69 || typedef double (*General2Field_Q11)(const double& u1, const double& u2);
```

Listing 1: `typedef` definition of the mobility function $Q_{11}(u_1, u_2)$. It assigns the name `General2Field_Q11` to the type: method, that depends on two arguments of type `double` with the name `u1` and `u2`. public of General2FieldEquationsBase[2]

Lst. 1 shows the definition of the function pointer for the mobility function $Q_{11}(u_1, u_2)$. It is defined as a method of the two arguments `u1` and `u2`. We create a function pointer for each of the remaining functions, keeping in mind that the functions g_{ij} can also depend on the spatial derivatives of the field u_i. Since the functions g_{i2} are multiplied by $\nabla\phi_{w_i}$, the return type needs to be a vector of `double` that matches the spatial dimension (it is not necessary to specify the length of the vector yet).
The class also provides virtual access methods for those function pointers.

```
109 || virtual General2Field_Q11& general2field_q11() = 0;
```

Listing 2: Virtual access for a function pointer `General2Field_Q11`. It will be overloaded in an inheriting class (see Lst. 4). public of General2FieldEquationsBase[3]

Since this class has no attributes, these access functions are virtual and need to be overloaded in an inheriting classes.

3.2.3 General2FieldEquations<unsigned DIM>

This class contains the methods for the assembly of the residual vector, Jacobian matrix and mass matrix. It inherits the `typedef` definitions from its parent class `General2FieldEquationsBase`. Additionally, it holds methods to use the functionality of the above mentioned function pointers.

3.2.3.1 Attributes

The class possesses member data for each `typedef`, that was defined in its parent class.

```
767 || General2Field_Q11 General2field_q11;
```

Listing 3: This line declares `General2field_q11` as member data of the type `General2Field_Q11`. The corresponding type definition is shown in Lst. 1. protected of General2FieldEquations<DIM>[4]

[2]File: ../oomph_Codes/user_src/G2FM_Single_File/G2FM_Single_File_unc.h
[3]File: ../oomph_Codes/user_src/G2FM_Single_File/G2FM_Single_File_unc.h

3.2.3.2 Methods

The constructor of the class initiates all the function pointers to zero using an initialisation list. This means that the function pointers specifically need to be set after the initialisation of the object. Since the function pointer attributes are protected data, we need to provide an access function for each function pointer. Therefore, we overload the virtual functions from the parent class.

```
168 General2Field_Q11& general2field_q11()
169 {
170  return General2field_q11;
171 }
```

Listing 4: Access function for the function pointer `General2field_q11`. public of General2FieldEquations<DIM>[5]

The code excerpt displayed in Lst. 4 allows the user to access the function pointer after the object initialisation.
During the assembly of the residual vector and the Jacobian matrix we need to check, whether the function pointer was set by the user. If the function pointer was set, we need to call the corresponding method, otherwise we return 0. Lst. 5 shows a method, that facilitates the behaviour described above for the mobility function Q_{11}.

```
275 // Q11
276 virtual double general2field_q11_fct(const double &u1, const double &u2) const
277 {
278  double q11 = 0.0;
279 
280  if (General2field_q11 != 0)
281  {
282   q11 = (*General2field_q11)(u1, u2);
283  }
284  return q11;
285 }
```

Listing 5: This method is called during the assembly of the residual vector. It returns 0 if the function pointer is not set, else it will give the return value of the function that is pointed to by `General2field_q11`. public of General2FieldEquations<DIM>[6]

3.2.3.3 fill_in_generic_residual_contribution_g2fm

This is the most important and complex method of this class and it assembles the residual vector, the Jacobian matrix and the mass matrix. Its declaration is shown in Lst. 6. Due to its complexity parts of the definition will be shown and discussed in more detail in the following pages.

```
807 /// \short actual function that calculates the residuals, jacobian and mass
      matrix
808 virtual void fill_in_generic_residual_contribution_g2fm(Vector<double> &
      residuals, DenseMatrix<double> &jacobian, DenseMatrix<double> &mass_matrix,
      unsigned flag);
```

Listing 6: Declaration of the function, that calculates the residual vector, Jacobian matrix and mass matrix. The last argument `flag` is used to distinguish, which components are supposed to be calculated (residual Vector, Jacobian matrix or mass matrix). protected of General2FieldEquations<DIM>[7]

The method definition can be split into two parts. In the first part the we set up the shape functions, their spatial derivatives and allocate memory for local variables. We also calculate the interpolated values of the element to insert them into the methods we specified above, e.g. Q_{11} (Lst. 7).

[4]File: ../oomph_Codes/user_src/G2FM_Single_File/G2FM_Single_File_unc.h
[5]File: ../oomph_Codes/user_src/G2FM_Single_File/G2FM_Single_File_unc.h
[6]File: ../oomph_Codes/user_src/G2FM_Single_File/G2FM_Single_File_unc.h
[7]File: ../oomph_Codes/user_src/G2FM_Single_File/G2FM_Single_File_unc.h

```
155    double q11_fct_value = general2field_q11_fct(interpolated_u1, interpolated_u2);
```

Listing 7: Calling the method displayed in Lst. 5 using the previously calculated values of u_1 and u_2 as arguments.[8]

The second part of the method assembles the residual vector, Jacobian matrix and mass matrix. It is the mathematical foundation of the program. Basically, it consists out of multiple loops over the test functions $\psi_l(\vec{x})$. Lst. 8 displays the assembly of the residual vector part $r_{u_1,l}$ and the Jacobian matrix bit J_{u_1,u_1} (3.23). The displayed code is used as an example and needs to be applied to the remaining parts of the residual vector and the Jacobian matrix.

```
192    // Loop over the test functions (nodes)
193    for (unsigned l = 0; l < n_node; l++)
194    {
195     /// u1_index +++++++++++++++++++++++++++++++++
196     local_eqn = nodal_local_eqn(l, u1_index);
197
198     // IF it's not a boundary condition
199     if (local_eqn >= 0)
200     {
201      // Add time derivative bit
202      residuals[local_eqn] += -1.0 * du1_dt * test(l) * W;
203
204      // Add the laplace bit, loop over spatial directions
205      for (unsigned alpha = 0; alpha < DIM; alpha++)
206      {
207       residuals[local_eqn] += -1.0 * (q11_fct_value * interpolated_dw1_dx[alpha] +
          q12_fct_value * interpolated_dw2_dx[alpha]) * dtestdx(l, alpha) * W;
208      }
209
210      // Calculate the jacobian and mass matrix
211      //---------------------------------------
212      if (flag == 1 or flag == 2)
213      {
214       // Loop over the shape functions again
215       for (unsigned l2 = 0; l2 < n_node; l2++)
216       {
217        // d residual u1 / d U1
218        local_unknown = nodal_local_eqn(l2, u1_index);
219        // IF it's not a boundary condition
220        if (local_unknown >= 0)
221        {
222         /// mass matrix
223         if (flag == 2)
224         {
225          mass_matrix(local_eqn, local_unknown) += test(l) * psi(l2) * W;
226         }
227         /// add jacobian
228         // timestepping bit
229         jacobian(local_eqn, local_unknown) += -1.0 * node_pt(l2)->time_stepper_pt
           ()->weight(1, 0) * psi(l2) * test(l) * W;
230         // mobility functions bit
231         for (unsigned beta = 0; beta < DIM; beta++)
232         {
233          jacobian(local_eqn, local_unknown) += -1.0 * (du1_q11_fct_value * psi(l2)
            * interpolated_dw1_dx[beta] + du1_q12_fct_value * psi(l2) *
            interpolated_dw2_dx[beta]) * dtestdx(l, beta) * W;
234         }
235        }
```

Listing 8: Assembling the residual vector, the Jacobian matrix and the mass matrix.[9]

The first `for` loop iterates over the test functions $\psi_l(\vec{x})$, which correspond to `test(l)` in the source code. To determine the local index in the residual vector of this element we call `nodal_local_eqn(l, u1_index)`. This function returns the local equation number for the test function at node

[8] File: `../oomph_Codes/user_src/G2FM_Single_File/G2FM_Single_File_unc.cc`
[9] File: `../oomph_Codes/user_src/G2FM_Single_File/G2FM_Single_File_unc.cc`

`l` of the data field with the index `u1_index`. In case of a pinned node (e.g. Dirichlet boundary conditions), it will return -1. In `oomph-lib` a negative equation number represents a pinned node, which is not a degree of freedom for the system. Therefore, pinned nodes are disregarded during the assembly of the residual vector and Jacobian matrix.
After this, we fill the residual vector at the position of the local equation number (`local_eqn`). First, we add a term representing the time dependence of u_1. For the helping fields w_1 and w_2 this part will be 0, since they do not possess an internal dynamic.
The second part adds the Laplace part of the weak formulation of the residual. Here, `dtestdx(l, alpha)` corresponds to $\nabla\psi_l(\vec{x})$. It is an instance of the class `DShape`, which represents the spatial derivative of the test function. `DShape` essentially is a two dimensional matrix, which overloads the round bracket operator to access its elements. The first argument refers to the test function with index l, while the second argument refers to the direction of the spatial derivative. These two parts conclude the assembly of the residual vector (compare (3.11)).

For the assembly of the Jacobian matrix, we need to create another loop over the test functions $\psi_{l2}(\vec{x})$ corresponding to `psi(l2)`. First, we need to determine the value of `local_unknown`, which corresponds to the field that we differentiate by. This means that `local_eqn` corresponds to the residual we are considering and we differentiate it with respect to the field that corresponds to `local_unknown`.
Lst. 8 shows the implementation of J_{u_1,u_1} (3.23), which corresponds to the derivative of the residual $r_{u_1,l}$ with respect to $U_{1,l2}$. First, we add the dynamic component, afterwards the Laplace bit. Similar as in the residual vector bit, we need to loop over the dimensions of the problem to fill in the Laplace bit.
This needs to be repeated for the remaining 3 parts of the residual vector as well as the remaining 15 parts of the Jacobian matrix. In general the process is very similar, but needs to be adapted to the specific equations in (3.11) - (3.16) and (3.23) - (3.38).
The only major difference appears, when implementing the derivatives of g_{ij}. The return values of $\partial_{U_{i,l2}} g_{j1}$ are vectors of doubles, since these functions in most cases include the shape functions $\psi_{l2}(\vec{x})$. Therefore the return value needs to be a vector with the same length as the `Shape` object. This will be clearer in the discussion of the driver code.

```
325        // d residual w1 / d U1
326        local_unknown = nodal_local_eqn(l2, u1_index);
327        // IF it's not a boundary condition
328        if (local_unknown >= 0)
329        {
330         if (flag == 2)
331         {
332          mass_matrix(local_eqn, local_unknown) += 0;
333         }
334         /// add jacobian
335         // mobility functions bit
336         jacobian(local_eqn, local_unknown) += du1_g11_fct_value[l2] * test(l) * W;
337         for (unsigned beta = 0; beta < DIM; beta++)
338         {
339          jacobian(local_eqn, local_unknown) += du1_g12_fct_value[l2][beta] *
               dtestdx(l, beta) * W;
340         }
341        }
```

Listing 9: Code excerpt from the assembly of the Jacobian matrix part J_{w_1,u_1} (3.27).[10]

In Lst. 9, we add `du1_g11_fct_value[l2]` to the Jacobian matrix, which is the vector mentioned above.
The return value of the methods that represent $\partial_{U_{i,l2}} g_{j2}$ is a vector of vectors of `double`, in other words a two dimensional matrix. Lst. 9 displays the example of adding `du1_g12_fct_value[l2][beta]` to the Jacobian matrix. It shows that `du1_g12_fct_value` can be interpreted as a two dimensional array. Similar to the situation above, the return value of the function representing $\partial_{U_{i,l2}} g_{j2}$ needs to match the shape of the `DShape` object `dtestdx`. The second dimension

[10]File: `../oomph_Codes/user_src/G2FM_Single_File/G2FM_Single_File_unc.cc`

of the return object is needed to facilitate the possible dependence of the equation on the spatial direction. Systems with two or more spatial directions can include a distinguished direction, which would make use of that feature.

In general it is not needed to implement the analytical Jacobian matrix, since it can be numerically calculated from the residual vector itself using finite differences (see section 4.3.6). Nonetheless it is very advisable to use the analytical implementation of the Jacobian, since it is numerically more efficient and results in shorter computation times.

Since the method in Lst. 6 is `protected`, the class provides a set of wrapper functions. They call the method from Lst. 6 with different values of the argument `flag`, depending on what parts need to be assembled. If the value is 0, it will only compute the residual vector and provide dummy matrices for the Jacobian matrix and mass matrix. In the case of `flag == 1`, the method will assemble the Jacobian matrix as well. For `flag == 2`, it will assemble the mass matrix as well. These wrapper methods are essential for the program to work correctly, since they are called by internal methods of the `oomph-lib` framework.

```
745  /// Compute element residual Vector (wrapper)
746  void fill_in_contribution_to_residuals(Vector<double> &residuals);
747
748  /// compute element residual vector and jacobian matrix
749  /// can be overloaded in problem file to switch between numerical and analytical
       jacobian matrix
750  void fill_in_contribution_to_jacobian(Vector<double> &residuals, DenseMatrix<
       double> &jacobian);
751
752  /// also get the mass matrix. this is needed for continuation
753  void fill_in_contribution_to_jacobian_and_mass_matrix(Vector<double> &residuals,
       DenseMatrix<double> &jacobian, DenseMatrix<double> &mass_matrix);
754
755  /// another function for the calculation of the mass matrix
756  void fill_in_contribution_to_mass_matrix(Vector<double> &residuals, DenseMatrix<
       double> &mass_matrix);
```

Listing 10: Wrapper methods for the assembly of the residual vector and Jacobian matrix method. They call the `protected` method `fill_in_generic_residual_contribution_g2fm` with different values of the argument `flag`, depending on the parts the method wants to calculate. If the wrapper method only wants to assemble the residual vector, it uses dummy matrices as the arguments for the Jacobian and mass matrix. public of General2FieldEquations<DIM>[11]

The remaining methods of the class are rather self explanatory. They provide methods to return the value of the element, as well as its temporal or spatial derivatives. Additionally, the class contains output methods, to output the current solution. Since these methods facilitate a rather general output functionality, the user can and should overload these methods to fit his specific needs.

3.2.4 General2FieldQElements<unsigned DIM, unsigned NNODE_1D>

This class inherits from `General2FieldEquations<DIM>` and `QElement<DIM,NNODE_1D>`. Its main purpose is to combine the mathematical foundation from `General2FieldEquations<DIM>` with the geometric information of the elements from `QElement<DIM,NNODE_1D>`.
Additionally, it sets the amount of fields considered at each node. Since we are using the general two-field model (including two helping fields), we need to set this value to 4. The value is saved in the private `Initial_Nvalue` attribute and can be accessed via the method `required_nvalue`.

```
727  //=====================================================================
728  // Set the data for the number of Variables at each node
729  //=====================================================================
730  template<unsigned DIM, unsigned NNODE_1D>
```

[11] File: `../oomph_Codes/user_src/G2FM_Single_File/G2FM_Single_File_unc.h`

```
731 || const unsigned General2FieldQElements<DIM, NNODE_1D>::Initial_Nvalue = 4;
```

Listing 11: Setting the `Initial_Nvalue` attribute. `private` of `General2FieldQElements<unsigned DIM, unsigned NNODE_1D>`[12]

This class includes a method to compute the integral over the element. The method returns the integral over the element of the field specified in the argument `u_index`. Such a method can be useful to verify or double check an external flux of the system. It can be used in continuation routines as well, when using a volume constraint as the control parameter.

```
879   /// \short compute integral over element
880   /// computes the integral of the specified field of the element
881   double compute_integral_element(const unsigned &u_index);
```

Listing 12: Method to compute the integral of field with index `u_index` over the elements domain. `public` of `General2FieldQElements<unsigned DIM, unsigned NNODE_1D>`[13]

The class holds two methods, which return the Jacobian of the mapping between global and local coordinates. Do not confuse the Jacobian of the mapping with the Jacobian matrix of the parent class, which is used for the Newton solve algorithm. Lst. 13 shows their declaration. One of them returns the Jacobian of the mapping for the element, while the other one returns it for the integration point that is passed to it in the additional argument.

```
887   /// Shape, test functions & derivs. w.r.t. to global coords. Return Jacobian.
888   double dshape_and_dtest_eulerian_g2fm(const Vector<double> &s, Shape &psi,
        DShape &dpsidx, Shape &test, DShape &dtestdx) const;
889
890   /// \short Shape/test functions and derivs w.r.t. to global coords at
891   /// integration point ipt; return  Jacobian of mapping
892   double dshape_and_dtest_eulerian_at_knot_g2fm(const unsigned &ipt, Shape &psi,
        DShape &dpsidx, Shape &test, DShape &dtestdx) const;
```

Listing 13: Methods that return the Jacobian of the mapping between local and global coordinates. Both methods use pass-by-reference to set the `test` and `dtestdx` equal to `psi` and `dpsidx`. `protected` of `General2FieldQElements<unsigned DIM, unsigned NNODE_1D>`[14]

These methods perform a few central tasks for the underlying mathematical foundation. Both set the test function equal to the ansatz function, which is necessary due to the Galerkin method (see section 2.3.1). Additionally these methods ensure, that the element is isoparametric. It means that the element uses the same shape function for the interpolation of the unknown as for the position. Looking into the `fill_in_generic_residual_contribution_g2fm` method, we can see the shape function `psi(l)` being multiplied with the nodal position to calculate the interpolated positional value.

```
141       interpolated_x[j] += raw_nodal_position(l, j) * psi(l);
```

Listing 14: Calculation of the positional value of current element using the shape function `psi(l)`.[15]

Keep in mind that the shape functions used in the `General2FieldEquations<unsigned DIM>` are `virtual` and only defining them using one of these two methods makes this element isoparametric. The shape functions can not be defined in `General2FieldEquations<unsigned DIM>`, since the class is only templated by the spatial dimension parameter and the shape functions depend on the `NNODE_1D` parameter as well. `General2FieldQElements<unsigned DIM, unsigned NNODE_1D>` on the other hand inherits from `QElement<DIM, NNODE_1D>` and is therefore templated by the `NNODE_1D` parameter, which represents the amount of nodes along a line in an element. When initialising an object of this class, the `NNODE_1D` parameter has to be defined and depending on it, the `QElement` object sets the shape functions accordingly.

[12] File: `../oomph_Codes/user_src/G2FM_Single_File/G2FM_Single_File_unc.cc`
[13] File: `../oomph_Codes/user_src/G2FM_Single_File/G2FM_Single_File_unc.h`
[14] File: `../oomph_Codes/user_src/G2FM_Single_File/G2FM_Single_File_unc.h`
[15] File: `../oomph_Codes/user_src/G2FM_Single_File/G2FM_Single_File_unc.cc`

This is a perfect example for code separation in object-oriented programming, the main coding principle of `oomph-lib`. It shows how the mathematical calculation of the residual vector and the Jacobian matrix is kept separate from the geometric information of the element. This allows the user to switch between different geometric elements or even spatial dimensions very easily.

3.2.5 `RefineableGeneral2FieldEquations<unsigned DIM>`

This class is the equivalent of the class presented in section 3.2.3 for simulations using adaptive mesh refinement. It inherits from `General2FieldEquations<unsigned DIM>`, `RefineableElement` and `ElementWithZ2ErrorEstimator`.
Similar to its parent class `fill_in_generic_residual_contribution_g2fm`, is the most important method of this class. To access the method this class provides a few wrapper methods. Before we go into the details of this method, we will briefly discuss the remaining methods of this class.

To find a criterion for the refine- or unrefinement of the mesh, the class includes a so called Z2 error estimator. Using the Z2 error estimator, we need to implement two methods. One returns the amount of flux terms, that are considered, while the second method provides the flux terms. These functions are included in this library, but will throw an error, when called. This feature is implemented to remind the user to overload these methods and adapt them to his specific needs. These methods are very problem specific and cannot be implemented in a general way, that suits all systems this library covers.
Another important method during the mesh refinement process is the `further_build` method. It is called after the refinement of an element and sets the function pointers of the child element to the function pointers of the father element.
This class provides methods to access the interpolated values of all fields. They are included, since they could be called by some interpolation or documentation routines, but provide no new functionality. To get the interpolated values of the element, we can still call the corresponding method from the parent class `General2FieldEquations<DIM>`.

3.2.5.1 `fill_in_generic_residual_contribution_g2fm`

Since the underlying mathematical equations are not different to the ones calculated in the parent class, this method is similar to the one in its parent class. The main differences are due to the fact that we need to handle hanging and master nodes[16].
During the assembly of the residual vector we need to multiply every contribution by a `hang_weight` factor. This factor depends on the master nodes of the current node and can be different for every node. When the current node is a master node itself, the factor simply becomes 1.
Lst. 15 displays the first step of the above mentioned process. The program determines the amount of master nodes of the current node. If itself is a master node, the counter will be set to 1.

```
1172      //If the node is hanging, get the number of master nodes
1173      if (is_node_hanging)
1174      {
1175       hang_info_pt = this->node_pt(l)->hanging_pt();
1176       n_master = hang_info_pt->nmaster();
1177      }
1178      //Otherwise there is just one master node, the node itself
1179      else
1180      {
1181       n_master = 1;
1182      }
```

Listing 15: Checking if current node `node(l)` is a hanging or a master node. In case of a hanging nodes, we calculate the amount of master nodes and save it to `n_master`.[17]

Afterwards, we create a loop over every master node of the current node. Hereby, we determine the hang weight corresponding to each master node. Then, we add the node's contribution to the

[16] `http://oomph-lib.maths.man.ac.uk/doc/the_data_structure/html/index.html#Hanging_Nodes`
[17] File: `../oomph_Codes/user_src/G2FM_Single_File/G2FM_Single_File_unc.cc`

residual vector multiplied by its hang weight. In case of a master node the hang weight will be 1 and the loop will only do a single iteration over the node itself. Lst. 16 shows the calculation of the hang weight, as well as its contribution to the residual vector.

```
1193      //Get the local equation number and hang_weight
1194      //If the node is hanging
1195      if (is_node_hanging)
1196      {
1197       //Read out the local equation from the master node
1198       local_eqn = this->local_hang_eqn(hang_info_pt->master_node_pt(m), u1_index);
1199       //Read out the weight from the master node
1200       hang_weight = hang_info_pt->master_weight(m);
1201      }
1202      //If the node is not hanging
1203      else
1204      {
1205       //The local equation number comes from the node itself
1206       local_eqn = this->nodal_local_eqn(l, u1_index);
1207       //The hang weight is one
1208       hang_weight = 1.0;
1209      }
1210
1211      /*IF it's not a boundary condition*/
1212      if (local_eqn >= 0)
1213      {
1214       // Add time derivative bit
1215       residuals[local_eqn] += -1.0 * du1_dt * test(l) * W * hang_weight;
1216
1217       // Add the laplace bit, loop over spatial directions
1218       for (unsigned alpha = 0; alpha < DIM; alpha++)
1219       {
1220        residuals[local_eqn] += -1.0 * (q11_fct_value * interpolated_dw1_dx[alpha]
          + q12_fct_value * interpolated_dw2_dx[alpha]) * dtestdx(l, alpha) * W *
          hang_weight;
1221       }
```

Listing 16: Calculating the hang_weight for each master node of the current node. We multiply every contribution to the residual vector by its corresponding hang_weight. Apart from the additional factor the assembly of the residual vector remains the same.[18]

The same routine needs to be done for the assembly of the analytical Jacobian. Inside the loop over the shape functions we determine the amount of master nodes of the current node l2 and its master nodes n_master2. For every master node we calculate the corresponding hang weight hang_weight2 and multiply it as a factor to every contribution of the Jacobian matrix. Hereby, we make sure to multiply every contribution to the Jacobian matrix by the factors hang_weight and hang_weight2. Lst. 17 is a code excerpt, which is nested inside a loop of m2 over the amount of master nodes n_master2. It calculates the current hang_weight2 and uses it as an additional factor for the assembly of the Jacobian matrix.

```
1257          //Get the local unknown and weight
1258          //If the node is hanging
1259          if (is_node2_hanging)
1260          {
1261           //Read out the local unknown from the master node
1262           local_unknown = this->local_hang_eqn(hang_info2_pt->master_node_pt(m2),
             u1_index);
1263           //Read out the hanging weight from the master node
1264           hang_weight2 = hang_info2_pt->master_weight(m2);
1265          }
1266          //If the node is not hanging
1267          else
1268          {
1269           //The local unknown number comes from the node
1270           local_unknown = this->nodal_local_eqn(l2, u1_index);
1271           //The hang weight is one
1272           hang_weight2 = 1.0;
```

[18]File: ../oomph_Codes/user_src/G2FM_Single_File/G2FM_Single_File_unc.cc

```
1273          }
1274          //If at a non-zero degree of freedom add in the entry
1275          //If the unknown is not pinned
1276          if (local_unknown >= 0)
1277          {
1278           /// add mass_matrix bit
1279           if (flag == 2)
1280           {
1281            mass_matrix(local_eqn, local_unknown) += test(l) * psi(l2) * W *
                 hang_weight * hang_weight2;
1282           }
1283
1284           /// jacobian
1285           // add timestepping bit
1286           jacobian(local_eqn, local_unknown) += -1.0 * node_pt(l2)->
                time_stepper_pt()->weight(1, 0) * psi(l2) * test(l) * W * hang_weight
                * hang_weight2;
1287           // add mobility functions bit
1288           for (unsigned beta = 0; beta < DIM; beta++)
1289           {
1290            jacobian(local_eqn, local_unknown) += -1.0 * (du1_q11_fct_value * psi(
                 l2) * interpolated_dw1_dx[beta] + du1_q12_fct_value * psi(l2) *
                 interpolated_dw2_dx[beta]) * dtestdx(l, beta) * W * hang_weight *
                 hang_weight2;
1291           }
1292          } // end of jacobian d residual_u1 / d u1
```

Listing 17: Calculating the `hang_weight2` for each master node of the current node. We multiply every contribution to the Jacobian matrix by its corresponding `hang_weight2` and `hang_weight`.[19]

Lst. 17 shows the assembly of the mass matrix. It is of the same size as the Jacobian matrix, but more sparse. In the general two field model the mass matrix is only different from 0 in the parts that would correspond to the J_{u_1,u_1} and J_{u_2,u_2} bit of the Jacobian matrix.

3.2.6 `RefineableGeneral2FieldQElement<unsigned DIM, unsigned NNODE_1D>`

This class inherits from `RefineableGeneral2FieldEquations<DIM>`, `General2FieldQElements<DIM, NNODE_1D>` and `RefineableQElement<DIM>`. A few methods need to be declared, but they are just passed down to one of its parent classes. The other methods have fixed return values that are independent of the system the user considers or are just empty methods.

[19] File: `../oomph_Codes/user_src/G2FM_Single_File/G2FM_Single_File_unc.cc`

4 Delayed coalescence

The Marangoni effect corresponds to a flow driven by a gradient of the interface tension along the interface between two fluids. The gradient in the interface tension can be caused by temperature gradient (thermal Marangoni effect) or a concentration gradient (solutal Marangoni effect) [10, 9]. When investigating the spreading behaviour of complex fluids, these effects need to be taken into account.
In this section we study the spreading and coalescence of two neighbouring droplets covered with an insoluble surfactant. The surfactant molecules alter the surface tension of the liquid film. Therefore, surfactant concentration gradients induce Marangoni flows in the liquid. This leads to interesting dynamical behaviour of two neighbouring droplets. Depending on the surfactant concentration difference and their contact angle when the contact lines meet, they either coalesce immediately or coalesce significantly slower and remain in a temporary non-coalescence state beforehand. In two dimensional systems fingering instabilities have been reported as well [18].

4.1 Mathematical modelling

In the following we consider a single layer of simple liquid, which is covered by an insoluble surfactant. The layer is on a flat solid substrate and surrounded by air. Fig. 5 shows a sketch of the considered system.

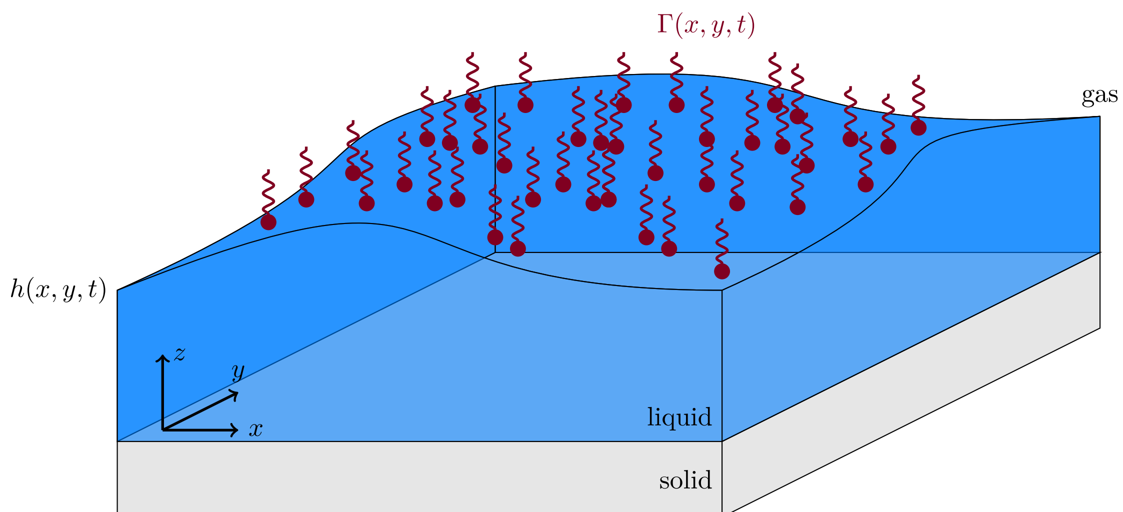

Figure 5: Sketch of a surfactant covered thin-film liquid on a solid substrate. The thin liquid film is described by the height profile $h(x, y, t)$, while the surfactant is described by its concentration on the surface $\Gamma(x, y, t)$. Single surfactant molecules are represented as a combination of a hydrophilic head and a hydrophobic tail. The system is surrounded by gas [27].

We describe it using two fields, the film height h and the concentration Γ of the surfactant. To specify a gradient dynamics formulation as in equation (3.1) and (3.2), we need to find a free energy functional based on independent fields.
At first sight, the variables h and Γ seem independent, but we need to keep in mind that Γ represents a surface concentration and therefore it depends on the curvature of the surface of the liquid film h. Consequently, we project the concentration $\Gamma(s(\vec{x}))$ onto the cartesian substrate plane $\vec{x} = (x, y)$. For a two-dimensional substrate, we consider a surface element at the surface of the liquid film

$$\mathrm{d}s = \sqrt{\mathrm{d}\vec{x}^2 + \mathrm{d}h^2} = \sqrt{1 + (\nabla h)^2}\mathrm{d}\vec{x} = \xi \mathrm{d}\vec{x} \tag{4.1}$$

where ξ is called the metric factor and is used to project the surfactant concentration onto the substrate. $\mathrm{d}s$ represents a surface element of the liquid height profile. The projected surfactant

concentration $\Gamma_p(\vec{x})$ is

$$\Gamma_p(\vec{x}) = \Gamma(s(\vec{x}))\xi, \tag{4.2}$$

which gives the local amount of surfactant independent of h.
The mesoscopic free energy functional in terms of the independent fields h and Γ_p is given by

$$\mathcal{F}_{meso}[h, \Gamma_p] = \int \left[\gamma_{sl} + f_w\left(h, \frac{\Gamma_p}{\xi}\right) + f_s\left(\frac{\Gamma_p}{\xi}\right)\xi - ph - \lambda_\Gamma \Gamma \xi\right] \mathrm{d}\vec{x}. \tag{4.3}$$

$f_w(h, \Gamma)$ represents the wetting energy, while $f_s(\Gamma)$ is the local free energy of the surfactant layer. p and λ_Γ are the Lagrange multipliers for the conservation of the amounts of liquid and surfactant. We consider a system with no wetting energy $f_w = 0$. The local free energy of the surfactant layer has a constant part and an entropic contribution

$$f_s = \gamma^0 + \frac{k_B T}{a^2}\Gamma(\ln(\Gamma) - 1). \tag{4.4}$$

Here, γ^0 corresponds to the mesoscopic interface tension between the liquid and the surrounding gas, while a represents the effective molecular length scale of the surfactant molecules. For small slopes we use the small-gradient or long-wave approximation

$$\xi \approx 1 + \frac{(\nabla h)^2}{2} \tag{4.5}$$

for the metric factor. We derive dynamical equations using the free energy functional (4.3)

$$\partial_t h = \nabla\left[Q_{hh}\nabla\frac{\delta\mathcal{F}}{\delta h} + Q_{h\Gamma}\nabla\frac{\delta\mathcal{F}}{\delta\Gamma_p}\right] \tag{4.6}$$

$$\partial_t \Gamma_p = \nabla\left[Q_{\Gamma h}\nabla\frac{\delta\mathcal{F}}{\delta h} + Q_{\Gamma\Gamma}\nabla\frac{\delta\mathcal{F}}{\delta\Gamma_p}\right]. \tag{4.7}$$

Q represents the mobility matrix [30, 34], which contains the mobility functions Q_{ij}

$$\mathbf{Q} = \begin{pmatrix} Q_{hh} & Q_{h\Gamma} \\ Q_{\Gamma h} & Q_{\Gamma\Gamma} \end{pmatrix} = \begin{pmatrix} \frac{h^3}{3\eta} & \frac{h^2\Gamma}{2\eta} \\ \frac{h^2\Gamma}{2\eta} & \frac{h\Gamma^2}{\eta} + D\Gamma \end{pmatrix}. \tag{4.8}$$

The variations of the free energy functional are

$$\frac{\delta\mathcal{F}}{\delta h} = -\nabla\left(\frac{\nabla h}{\xi}(f_s - \Gamma\partial_\Gamma f_s)\right) \tag{4.9}$$

$$\frac{\delta\mathcal{F}}{\delta\Gamma_p} = \xi\partial_\Gamma f_s. \tag{4.10}$$

We insert (4.9) and (4.10) into (4.6) and (4.7) to yield the dynamical equations

$$\partial_t h = \nabla\left[Q_{hh}\nabla\left(-\nabla\left(\frac{\nabla h}{\xi}(f_s - \Gamma\partial_\Gamma f_s)\right)\right) + Q_{h\Gamma}\nabla(\xi\partial_\Gamma f_s)\right] \tag{4.11}$$

$$\partial_t \Gamma_p = \nabla\left[Q_{\Gamma h}\nabla\left(-\nabla\left(\frac{\nabla h}{\xi}(f_s - \Gamma\partial_\Gamma f_s)\right)\right) + Q_{\Gamma\Gamma}\nabla(\xi\partial_\Gamma f_s)\right] \tag{4.12}$$

in the long-wave limit. The long-wave limit states $|\nabla h| \ll 1$, which results in $\xi \approx 1$ and $\Gamma_p \to \Gamma$. For the implementation of the numerical finite element method we introduce the two additional fields

$$w_1 = \frac{\delta\mathcal{F}}{\delta h} = -\nabla \cdot [(f_s - \Gamma\partial_\Gamma f_s) \cdot \nabla h] \tag{4.13}$$

$$w_2 = \frac{\delta\mathcal{F}}{\delta\Gamma} = \partial_\Gamma f_s. \tag{4.14}$$

Through the introduction of the two fields we have to solve an equation system of four equations with a highest spatial derivative of second order. This allows us to implement a numerically efficient scheme.

4.1.1 Rescaling the model

We rescale the model by introducing the length scale l for the film height, as well as the spatial coordinates and τ for the time scale in (4.6) and (4.7). Only the surfactant concentration remains unscaled

$$x = l\tilde{x} \qquad t = \tau\tilde{t} \qquad h = l\tilde{h} \qquad \Gamma = \tilde{\Gamma}. \tag{4.15}$$

This results in

$$w_1 = \frac{\gamma^0}{l} \underbrace{\left(-\tilde{\nabla} \cdot \left[\left(\tilde{f}_s - \tilde{\Gamma}\partial_{\tilde{\Gamma}}\tilde{f}_s\right) \cdot \tilde{\nabla}\tilde{h}\right]\right)}_{\tilde{w_1}} \qquad w_2 = \gamma^0 \underbrace{\partial_{\tilde{\Gamma}}\tilde{f}_s}_{\tilde{w_2}} \tag{4.16}$$

with

$$\tilde{f}_s = \gamma^0 \left(1 + \epsilon\tilde{\Gamma}(\ln(\tilde{\Gamma}) - 1\right) \tag{4.17}$$

and

$$\begin{pmatrix} Q_{hh} & Q_{h\Gamma} \\ Q_{\Gamma h} & Q_{\Gamma\Gamma} \end{pmatrix} = \underbrace{\begin{pmatrix} \frac{\tilde{h}^3}{3} & \frac{\tilde{h}^2\tilde{\Gamma}}{2} \\ \frac{\tilde{h}^2\tilde{\Gamma}}{2} & \tilde{h}\tilde{\Gamma}^2 + \tilde{D}\tilde{\Gamma} \end{pmatrix}}_{\tilde{\mathbf{Q}}} * \begin{pmatrix} l^3 & l^2 \\ l^2 & l \end{pmatrix} \frac{1}{\eta}. \tag{4.18}$$

The time scale is rescaled by $\tau = \frac{\eta l}{\gamma^0}$. Rescaling the model allows us to describe it by the dimensionless diffusivity $\tilde{D} = \frac{\eta}{l} D$ and $\epsilon = \frac{k_B T}{a^2 \gamma^0}$. This results in the dimensionless form

$$\partial_{\tilde{t}}\tilde{h} = \tilde{\nabla} \cdot \left[Q_{\tilde{h}\tilde{h}}\tilde{\nabla}\tilde{w_1} + Q_{\tilde{h}\tilde{\Gamma}}\tilde{\nabla}\tilde{w_2}\right] \tag{4.19}$$

$$\partial_{\tilde{t}}\tilde{\Gamma} = \tilde{\nabla} \cdot \left[Q_{\tilde{\Gamma}\tilde{h}}\tilde{\nabla}\tilde{w_1} + Q_{\tilde{\Gamma}\tilde{\Gamma}}\tilde{\nabla}\tilde{w_2}\right]. \tag{4.20}$$

4.2 Implementation using the `general2field` library

In the following, we will discuss the implementation of a numerical scheme to calculate the time evolution of the system. We implement the rescaled, dimensionless form (4.19) and (4.20), but omit the tilde in the subsequent sections.

For the correct usage of the `general2field` library we need to find the weighted residuals of the equation system. Since only the free energy functional changes for different systems, we only need to find the weighted residuals of the auxiliary fields w_1 and w_2

$$r_{w_1} = \int_D -w_1\phi_{w_1}\mathrm{d}\vec{x} + \int_D \underbrace{\left[(\nabla h)(f_s - \Gamma\partial_\Gamma f_s)\right]}_{g_{12}} \cdot \nabla\phi_{w_1}\mathrm{d}\vec{x} \tag{4.21}$$

$$r_{w_2} = \int_D -w_2\phi_{w_2}\mathrm{d}\vec{x} + \int_D \underbrace{\partial_\Gamma f_s}_{g_{21}} \phi_{w_2}\mathrm{d}\vec{x}. \tag{4.22}$$

Comparing g_{12} and w_1 we can see they only differ by the factor $-\nabla$ due to the integration by parts. This is in contrast to w_2, which does not contain a spatial derivative of order two and therefore no integration by parts is applied. By deriving (4.21) and (4.22) we find the functions g_{12} and g_{21}, that we need to implement into our program.

To implement the assembly of the analytically derived Jacobian matrix, we derive the weighted residuals to the order parameter fields h and Γ. Keep in mind that we have applied the Galerkin

method as described in section 3.1.3. The contribution to the Jacobian matrix are given by

$$J_{w_1,h} = \int_D \underbrace{[(\nabla\psi_{l2})(f_s - \Gamma\partial_\Gamma f_s)]}_{\partial_{H,l2}g_{12}} \cdot \nabla\psi_l \mathrm{d}\vec{x} \quad (4.23)$$

$$J_{w_1,\Gamma} = \int_D \underbrace{[(\nabla h)\partial_\Gamma(f_s - \Gamma\partial_\Gamma f_s)\psi_{l2}]}_{\partial_{\Gamma,l2}g_{12}} \cdot \nabla\psi_l \mathrm{d}\vec{x} \quad (4.24)$$

$$J_{w_2,h} = 0 \quad (4.25)$$

$$J_{w_2,\Gamma} = \int_D \underbrace{\partial_{\Gamma\Gamma} f_s \psi_{l2}}_{\partial_{\Gamma,l2}g_{21}} \nabla\psi_l \mathrm{d}\vec{x}. \quad (4.26)$$

4.3 The driver code

To solve the above mentioned equation system we implement a driver code. It needs to be placed in a subfolder of the `user_drivers` folder of the local `oomph-lib` installation and compiled accordingly[20].
The driver code initialises an object of the problem class. The corresponding object brings together all the parts necessary for the simulation. It initialises a mesh, using `RefineableGeneral2FieldQElement<unsigned DIM, unsigned NNODE_1D>` objects as the elements. Additionally, it will manage the access of the timestepping routine and ensure that all function pointers are set. The problem object is also responsible for the documentation of the solution and the time stepping process.
In the driver code we define a separate `namespace` as well. It contains variables to store problem dependent parameters (in this case ϵ_1), as well as the methods we need to set the function pointers to. Additionally, it contains a method to calculate the initial conditions of the problem.

4.3.1 `include` statements

In the head of the driver code, we write commands to include the libraries needed for the calculation of the problem. In all cases we need to load the header files for the generic `oomph-lib` routines. Additionally, we can load specific problem-dependent libraries from the `oomph-lib` framework. To calculate the PDEs mentioned above we use the library for the general two field problem discussed in section 3.2. Lst. 18 shows the corresponding `include` command.

```
41 // The library for the problem
42 #include "general2field.h"
```

Listing 18: Including the header file of the library for the calculation of the general two field model.[21]

Furthermore, we include the `oomph-lib` library for rectangular meshes and an external library for the extraction of local extrema of one dimensional functions[22].

4.3.2 `G2FMDelayedCoalescenceFunctionsParameters` namespace

As mentioned above, we define an extra namespace. It will be used to store problem dependent global data (parameters), e.g. the timestepping interval or the grid dimensions. Most of the parameters are self explanatory, while we list the ones that are not below.

- `timestepping_bool`
  boolean value, to switch between timestepping calculations and continuation.
- `Volume_limit_*`
  parameter to hold the current volume of the field. It can be used to store the current volume or as a control parameter for continuation.

[20] `http://oomph-lib.maths.man.ac.uk/doc/the_distribution/html/index.html#add_driver`
[21] File: `../oomph_Codes/user_drivers/Latex_Coalescene/Coalescence_driver_code_unc.cc`
[22] `http://www.csc.kth.se/~weinkauf/notes/persistence1d.html`

The namespace holds a method to define the initial conditions. In this example the method places two droplets in the middle of the domain. The droplets have the shape of an upside down parabola and the complete domain is covered by a thin precursor film. Using *tanh* functions we approximate a smooth step function. The step function is used for the initial conditions of the second field, the surfactant concentration. For the initial conditions the left droplet is covered with a homogenous spread amount of surfactant.

4.3.2.1 Problem specific functions

This namespace also holds the functions that define this specific system of the general 2 field model. Here, we need to define the mobility functions, the g_{ij} functions and the functions necessary to use the analytical Jacobian matrix.
To make the code more readable we define a few auxiliary functions. Namely $f_s(\Gamma)$ (4.4) and its derivatives $\partial_\Gamma f_s$ and $\partial_{\Gamma\Gamma} f_s$.
Lst. 19 displays the definition of the mobility function Q_{11}. Since this is the method, to which the function pointers of the elements in our mesh are pointing, we need to make sure, that its arguments match the arguments of the corresponding function pointer. In this case it means, that our method must have the argument `&u2` although the return value does not depend on it. When the arguments do not match the compiler will raise a conversion error.

```
273 double general2field_q11_fct(const double &u1, const double &u2)
274 {
275  double q11 = pow(u1, 3.0) / 3.0;
276
277  return q11;
278 }
```

Listing 19: Definition of the mobility function $Q_{11}(u_1, u_2)$. The function pointers of the elements will point to this function.[23]

Similarly, we define the function g_{12} (4.21). Hereby, we need to take into account that g_{12} needs to be a vector, whose length corresponds to the spatial dimension of the problem. The definition of the method is shown in Lst. 20.

```
303 Vector<double> general2field_g12_fct(const double& u1, const double& u2, const
      Vector<double>& nabla_u1, const Vector<double>& nabla_u2)
304 {
305  double scalar_factor = fs(u2) - u2 * dfs_dgamma(u2);
306
307  Vector<double> g12(nabla_u1.size(), 0.0);
308  for (oomph::Vector<double>::size_type i = 0; i != g12.size(); i++)
309  {
310   g12[i] = nabla_u1[i] * scalar_factor;
311  }
312  return g12;
313 }
```

Listing 20: Definition of g_{12}. The return value must be a vector with an entry for each spatial dimension of the problem. In the first line of the definition we calculate a scalar factor, which will be multiplied by ∇h. Afterwards we initialise the double vector `g12` and set it to the correct length, by using the argument `nabla_u1`. Then we loop over `g12` and fill it with the correct return values.[24]

After the implementation of the mobility functions, g_{12} and g_{21}, we are able to assemble the residual vector and to use it for a test run. Hereby, we let the internal finite differences routine of `oomph-lib` calculate the Jacobian matrix. This is very useful as a proof of work and can be used to present first results. If the assembly of the residual vector is correctly implemented, we can add the methods for the assembly of the analytically calculated Jacobian matrix.

[23] File: `../oomph_Codes/user_drivers/Latex_Coalescene/Coalescence_driver_code_unc.cc`
[24] File: `../oomph_Codes/user_drivers/Latex_Coalescene/Coalescence_driver_code_unc.cc`

Therefore, we need methods that provide the derivatives of the mobility functions, g_{12} and g_{21} with respect to both fields h and Γ. The implementation of the derivative of the mobility function is straight forward, hence we will focus on the implementation of $\partial_{H,l2}g_{12}$, $\partial_{\Gamma,l2}g_{12}$ and $\partial_{\Gamma,l2}g_{21}$.

We start with the discussion of Lst. 21, which displays the implementation of the method to calculate $\partial_{\Gamma,l2}g_{21}$. It returns a vector of doubles, whose length matches the length of the shape function `psi`, instead of matching the spatial dimension of the problem.

```
444 Vector<double> general2field_du2_g21_fct(const double& u1, const double& u2,
      const Vector<double>& nabla_u1, const Vector<double>& nabla_u2, const Shape&
      psi, const DShape& dpsidx)
445 {
446  Vector<double> du2_g21(psi.nindex1(), 0.0);
447  for (unsigned i = 0; i < psi.nindex1(); i++)
448  {
449   du2_g21[i] += dfs_dgamma_dgamma(u2) * psi(i);
450  }
451  return du2_g21;
452 }
```

Listing 21: Definition of the method representing $\partial_{\Gamma,l2}g_{21}$. In contrast to the method displayed in Lst. 20, the return value of this method needs to be a vector of doubles, whose length matches the one of the `Shape` object `psi`. During the assembly of the Jacobian matrix it will be accessed similarly to `du1_g11_fct_value` displayed in Lst. 9. The method uses the previously defined auxiliary method `dfs_dgamm_dgamma`, which increases the readability of the code and promotes code separation (useful for debugging).[25]

To represent $\partial_{H,l2}g_{12}$ we need to create a two dimensional matrix. The shape of the matrix has to match the shape of the `DShape` object `dpsidx`. It has one index, which represents the shape functions `psi(l2)`, while the other index corresponds to the spatial direction of the respective derivative.

```
385 Vector<Vector<double> > general2field_du1_g12_fct(const double& u1, const double&
       u2, const Vector<double>& nabla_u1, const Vector<double>& nabla_u2, const
      Shape& psi, const DShape& dpsidx)
386 {
387  // initialise to zero
388  Vector<Vector<double> > du1_g12(dpsidx.nindex1());
389  for (unsigned i = 0; i < dpsidx.nindex1(); i++)
390  {
391   du1_g12[i] = Vector<double>(dpsidx.nindex3(), 0.0);
392  }
393
394  // calculate values
395  double scalar_factor = fs(u2) - u2 * dfs_dgamma(u2);
396  for (unsigned l2 = 0; l2 < dpsidx.nindex1(); l2++)
397  {
398   for (unsigned beta = 0; beta < dpsidx.nindex3(); beta++)
399   {
400    du1_g12[l2][beta] += dpsidx(l2, beta) * scalar_factor;
401   }
402  }
403  return du1_g12;
404 }
```

Listing 22: Definition of the method representing $\partial_{H,l2}g_{12}$. Its return value is a vector of vectors of `double`. First, we initialise `du1_g12` to match its shape to the shape of `dpsidx`. In the second step we create two loops to iterate over the two indices `l2` (for the shape functions) and `beta` (the direction of the derivative) to access the individual values. Compare to `du1_g12_fct_value` in Lst. 9.[26]

The definition of the method to represent $\partial_{\Gamma,l2}g_{12}$ is comparable to the one shown in Lst. 22. Arguments regarding the shape of the return value, mentioned in Lst. 22 apply as well.

[25] File: ../oomph_Codes/user_drivers/Latex_Coalescene/Coalescence_driver_code_unc.cc
[26] File: ../oomph_Codes/user_drivers/Latex_Coalescene/Coalescence_driver_code_unc.cc

```
409 Vector<Vector<double> > general2field_du2_g12_fct(const double& u1, const double&
      u2, const Vector<double>& nabla_u1, const Vector<double>& nabla_u2, const
     Shape& psi, const DShape& dpsidx)
410 {
411  // initialise to zero
412  Vector<Vector<double> > du2_g12(dpsidx.nindex1());
413  for (unsigned i = 0; i < dpsidx.nindex1(); i++)
414  {
415   du2_g12[i] = Vector<double>(dpsidx.nindex3(), 0.0);
416  }
417
418  // calculate values
419  for (unsigned l2 = 0; l2 < dpsidx.nindex1(); l2++)
420  {
421   for (unsigned beta = 0; beta < dpsidx.nindex3(); beta++)
422   {
423    du2_g12[l2][beta] += nabla_u1[beta] * dfs_dgamma(u2) * psi(l2);
424    du2_g12[l2][beta] += nabla_u1[beta] * (-1.0 * dfs_dgamma(u2) * psi(l2));
425    du2_g12[l2][beta] += nabla_u1[beta] * (-1.0 * u2 * dfs_dgamma_dgamma(u2) * psi
      (l2));
426   }
427  }
428  return du2_g12;
429 }
```

Listing 23: Definition of the method calculating $\partial_{\Gamma,l2} g_{12}$.[27]

Comparing Lst. 22 and Lst. 23, we can see why it is necessary to pass the shape functions `psi` and `dpsidx` as input parameters to both methods. In Lst. 22 only the shape function `dpsidx` is used, while in Lst. 23 it is only `psi`. It shows, that depending on the problem specific function g_{12} the method needs to use one, the other or both. This contradicts the programming principle, that is applied to all other function pointers, which tries to separate the shape functions from the problem specific mathematics. Unfortunately there is no simple way to separate them, without making the library and its usage artificially complicated.

4.3.3 CustomRefineableGeneral2FieldQElements<unsigned DIM, unsigned NNODE_D>

We implement a custom element class, which inherits from `RefineableGeneral2FieldQElement<DIM,NNODE_1D>`. It is used to overload a few methods of its parent class. As mentioned in section 3.2.5, we need to overload the methods that return the flux values for the Z2 error estimator. The implemented methods will throw an error if they are not overloaded. Additionally, we overload the output method of the element, to output ∇h values of the element as well.

The method `num_Z2_flux_terms()` returns the amount of flux values that will be considered for the Z2 error estimation. The method definition is shown in Lst. 24.

```
120 template<unsigned DIM, unsigned NNODE_1D>
121 unsigned CustomRefineableGeneral2FieldQElements<DIM, NNODE_1D>::num_Z2_flux_terms
     ()
122 {
123  /// for both fields
124  return DIM + DIM;
125 }
```

Listing 24: Definition of the `num_Z2_flux_terms()` method, which returns the amount of flux values that will be considered by the Z2 error estimator. In this case we return twice the spatial dimension of the problem, to take the flux values of both fields into account. public of CustomRefineableGeneral2FieldQElements<unsigned DIM, unsigned NNODE_1D>[28]

We need to provide a method to get the flux terms as well. The `get_Z2_flux(const Vector<double> &s, Vector<double> &flux)` method uses pass-by-reference to provide the vector,

[27] File: `../oomph_Codes/user_drivers/Latex_Coalescene/Coalescence_driver_code_unc.cc`
[28] File: `../oomph_Codes/user_drivers/Latex_Coalescene/G2FM_CustomRefineableElement_unc.cpp`

we need to fill with the appropriate flux terms. The definition of the method is shown in Lst. 25 and it includes an example implementation of a very simple self test.

```
template<unsigned DIM, unsigned NNODE_1D>
void CustomRefineableGeneral2FieldQElements<DIM, NNODE_1D>::get_Z2_flux(const
  Vector<double> &s, Vector<double> &flux)
{
#ifdef PARANOID
 unsigned num_entries = DIM + DIM;
 if (flux.size() < num_entries)
 {
  std::ostringstream error_message;
  error_message << "The flux vector has the wrong number of entries, " << flux.
    size() << ", whereas it should be at least " << num_entries << std::endl;
  throw OomphLibError(error_message.str(), OOMPH_CURRENT_FUNCTION,
    OOMPH_EXCEPTION_LOCATION);
 }
#endif

 //Find out how many nodes there are in the element
 unsigned n_node = this->nnode();

 //Set up memory for the shape and test functions
 Shape psi(n_node);
 DShape dpsidx(n_node, DIM);

 //Call the derivatives of the shape and test functions
 this->dshape_eulerian(s, psi, dpsidx);

 //Initialise to zero
 for (unsigned j = 0; j < flux.size(); j++)
 {
  flux[j] = 0.0;
 }

 // Loop over nodes
 for (unsigned l = 0; l < n_node; l++)
 {
  //Loop over derivative directions
  // h value
  for (unsigned j = 0; j < DIM; j++)
  {
   flux[j] += this->nodal_value(l, G2FMDelayedCoalescenceFunctionsParameters::
     u1_index) * dpsidx(l, j);
  }
  // gamma value
  for (unsigned j = 0; j < DIM; j++)
  {
   flux[j + DIM] += this->nodal_value(l,
     G2FMDelayedCoalescenceFunctionsParameters::u2_index) * dpsidx(l, j) * 0;
  }
 }
}
```

Listing 25: Definition of the `get_Z2_flux` method, which uses pass-by-reference to fill the `flux` vector with the corresponding values. Additionally it features a simple self test, which checks if the length of the provided flux vector matches the expected length. public of CustomRefineableGeneral2FieldQElements<unsigned DIM, unsigned NNODE_1D>[29]

The `oomph-lib` framework features a so called PARANOID compiler flag, which allows the execution of sanity checks during the code execution. If the PARANOID compiler flag is set the statements inside `#ifdef` and `#endif` are executed. In this case it is a simple check, whether the provided flux vector is of the expected length. If the test fails the program will throw an error, which can be customised to provide a helpful error message.

The class also overloads the output method of the element to output the spatial derivative of

[29]File: `../oomph_Codes/user_drivers/Latex_Coalescene/G2FM_CustomRefineableElement_unc.cpp`

the h field. Its implementation is straight forward and will not be discussed in detail[30].
This element class includes a few features, that have not been discussed so far and are useful for continuation routines. These routines extend the functionality of the program and they consist out of methods and attributes to facilitate the continuation of the system including a volume constraint. The usage of the volume constraint during the time evolution simulation is not necessary, since all considered cases of the general two field model only contain conserved quantities and therefore the system fulfils the continuity equation.
Since the only stable state of the currently considered system is the homogenous solution, applying and demonstrating continuation routines is not done here. They will be discussed in another tutorial.

4.3.4 `G2FMUnsteadyProblem`

After declaring and defining the namespace `G2FMDelayedCoalescenceFunctionsParameters` and creating our custom element `CustomRefineableGeneral2FieldQElements<unsigned DIM, unsigned NNODE_D>`, we bring all these pieces of code together using the `G2FMUnsteadyProblem` class. The problem class unites the functionality of the included `oomph-lib` libraries and the user-provided code.

The `G2FMUnsteadyProblem` class inherits from the `Problem` class of the `oomph-lib` framework and is templated by the `ELEMENT` parameter. It handles the solution process of the PDEs, i.e. it coordinates the solving of the PDEs, documentation and analysis during and after the solution process, mesh refinement in between time or continuation steps and the usage of different linear solvers. Additionally, it provides access to global data, i.e. the current simulation time, as well as the degrees of freedom of each node. This class can easily be extended to fit the needs of the user.

In the following, we discuss the most important methods for simulating the temporal evolution of the system. We omit functionality that is used for continuation routines, since this will be covered in another section.

4.3.4.1 Attributes

The class has 4 attributes. It contains two output info objects of the class `DocInfo`. They are used to store output information, such as the path to an output folder or a file number. It also stores a pointer `Bulk_mesh_pt` to a `Mesh` class object. The pointer is needed to access the mesh and its elements. It contains another mesh pointer, which is used for the introduction of a volume constraint. All attributes of this class are `protected` and therefore the class holds `public` access methods as well.

4.3.5 `G2FMUnsteadyProblem<ELEMENT>` methods

The problem class contains methods, that are necessary during the solution process, as well as methods, which are convenient tools of analysis. In the following I will discuss the necessary methods in detail, while just briefly elaborating on the convenient tools (here called “handy” methods).

4.3.5.1 `G2FMUnsteadyProblem<ELEMENT>()`, constructor

The structure of every constructor of an `oomph-lib` problem is, despite a few nuances, more or less the same. It can be divided into a few steps, that need to be done.

1. **adding a timestepper** - necessary
   At first, we need to add a timestepper to the problem. In this case we use `BDF` timestepper, which corresponds to a second order backwards differentiation formula. Adding the timestepper to the problem, will allocate memory to store the previous timesteps.

[30] `http://oomph-lib.maths.man.ac.uk/doc/FAQ/html/index.html#cust_opt`

2. *setting a linear solver* - optional
   We can specify the linear solver, that will be used for Newton's method. If we do not set the `linear_solver_pt`, it will use the `oomph-lib` default, the `SuperLU` solver. This option can be useful for debugging purposes, when implementing the usage of the analytical Jacobian matrix. By setting the linear solver to the `FD_LU` solver, we can use an internal finite differences scheme to calculate the Jacobian numerically from the residual vector. Comparing the numerical and analytical Jacobian matrix usually leads to good hints about the origin of bugs.

3. **building the mesh** - necessary
   We initialise a new mesh object with elements of the template parameter type, by calling the corresponding constructor. Hereby we pass the corresponding dimensions of the mesh, as well as its initial amount of elements. It is important to pass a pointer to the current timestepper of the problem to the mesh constructor as well.

4. **creating a Z2 error estimator** - necessary
   This step is only necessary, when using mesh refinement. We create a pointer to an error estimator and we set the `refinable_mesh_pt()` of our current mesh to it. This only works, if the mesh that was created in a previous step inherits from the `RefineableMeshBase` class.

5. **setting the function pointers** - necessary
   To ensure that the program calculates the solution to the considered PDEs we set the function pointers of each element to the corresponding methods in our namespace `G2FMDelayed CoalescenceFunctionsParameters` (see section 4.3.2). Therefore, we call the handy method `prepare_mesh(Mesh* Mesh_pointer)` and pass our current `Bulk_mesh_pt` as the argument. The method loops over the elements of the mesh pointed to by the `Mesh_pointer` and sets the necessary function pointers accordingly. Since the `element_pt(const unsigned long &e)` method returns a pointer to a `GeneralisedElement`, we need to upcast the element pointer by using the C++ `dynamic_cast` conversion. This allows us to cast pointers up the inheritance hierarchy and cast the pointer to the type of the template parameter.

6. **adding the sub mesh** - necessary
   To introduce the `Bulk_mesh_pt` mesh into the global mesh we call the `add_sub_mesh` method. This method adds sub meshes to the global mesh. In this case the global mesh consists only of one mesh, nonetheless this call is necessary. There are many scenarios, in which it is necessary to create multiple submeshes (e.g. using a volume constraint, using non-Neumann flux boundary conditions). They can be introduced into the global mesh by subsequent calls of this method.

7. **building the global mesh** - necessary
   After adding the submeshes to the global mesh we build the global mesh. This method combines all submeshes into one global mesh.

8. **assigning equation numbers** - necessary
   The last call of the constructor must to be the equation numbering. `assign_eqn_numbers()` returns the number of degrees of freedom in this problem, but more importantly it assigns the equation numbers. This is necessary for the solution process. The method includes global data, as well as the degrees of freedom of the elements. If the equation numbering is not done properly, the solver will fail, due to an improper assembly of the residual vector and Jacobian matrix.

4.3.5.2 `void set_initial_condition()`

This method is called to apply the initial conditions we defined in our namespace to the current problem. Additionally, to setting the initial conditions for the time $t = 0$ we also set the initial conditions for the "previous" timesteps. This ensures that the timestepper does not try to access undefined values in the first steps.
The basic structure of the method consists of two nested loops. The outer loop iterates over the amount of timesteps, that the timestepper allocated memory for, while the inner loop iterates

over the nodes in the mesh. Inside of both loops we acquire the nodal position of the current node and call the `get_initial_conditions` method to get the values for the current node using pass-by-reference.

4.3.5.3 `doc_solution_timestepping`

These methods are used to document the current solution. They are called in between time steps and can be used to display the temporal evolution of the system.
This class contains two methods of this name, which differ only in the amount of arguments they take. For the method that takes three arguments, the last argument is a boolean value, which decides whether the output should be for a uniform refined mesh. Outputting the solution on an uniform mesh can be very useful in the post processing of the data, especially when dealing with 2D systems.

The method taking two arguments is the method, that actually outputs the solution. It specifies the path of the output file using the information stored in the `DocInfo` object and also writes the first few lines of the output file. These lines include a legend to interpret the output file, as well as global information about the current state of the system (e.g. the current time step, the integral over a field). Afterwards the method calls the output function of the `Bulk_mesh_pt`, which loops over the mesh's elements and calls the output function of each element, which arrives at the output function we overloaded in section 4.3.3.

To output the solution on an uniform mesh we create a copy of the current problem. The mesh of the copy must be refined in the same way, as the current mesh. When the refinement pattern of both meshes are matched we can copy the data values from the original mesh to the copy. Then we search for the most refined element and refine every other element until they have the same refinement level. This ensures that we have a uniformly refined mesh. Afterwards, we call the above mentioned output method from the refined mesh. We delete the copy and continue the simulation using the original mesh after saving the output.
By applying the refinement and output routines only to the copy, we make sure that the calculations will be done on the non-uniformly refined original mesh.

4.3.5.4 other methods

The following list mentions other methods of this class and will explain their purpose and how they work in a rather compact way.

- `void actions_after_adapt()`
  Action function that is called after the adaption process. In simple time evolution procedures this remains empty, but needs some adjustment when using a volume constraint in continuation routines.

- `void introduce_volume_constraint()`
  Method used to introduce a volume constraint for continuation routines.

- `void delete_volume_constraint(Mesh* const &volume_mesh_pt)`
  By calling this method we can delete the mesh volume_mesh_pt, which is connected to the volume constraint. This and the methods above will be discussed in detail in a tutorial featuring continuation routines.

- `double global_temporal_error_norm()`
  This method is needed for using time evolution with an adaptive step size.

- `double compute_integral(const unsigned &u_index)`
  Method that computes the integral of field `u_index` over the complete domain.

- `Problem* make_copy()`
  It initialises an object of this problem class and returns the pointer to the copy. For continuation routines it also introduces a volume constraint to the copy.

- `RefineableMeshBase* refinable_mesh_pt()`
  This returns a pointer to the `Bulk_mesh_pt` attribute, which is upcast to `RefineableMesh-Base` class pointer.

- `TreeBasedRefineableMeshBase* tree_based_refineable_mesh_pt()`
  Returns a pointer to a `TreeBasedRefineableMeshBase` class from the upcast `Bulk_mesh_pt`.

4.3.6 Assembling the Jacobian matrix

There are multiple ways to calculate the Jacobian matrix. The most efficient option is to assemble the Jacobian matrix on the basis of analytical calculations. Therefore, we have to take the derivatives of the residual vector with respect to the fields of our system and supply the results into the assembly of the Jacobian matrix. Unfortunately, this is also the most difficult option, since there are many ways for bugs to sneak into the code (most prominently sign errors).

The other two ways numerically calculate the Jacobian matrix from the residual vector. The `FD_LU` linear solver calculates the Jacobian matrix from the global residual vector. This is very slow, but includes all global interactions between elements. To use the `FD_LU` linear solver, we set the `linear_solver_pt` of the problem in the constructor of the problem (see section 4.3.5.1).
A numerically more efficient way to calculate the Jacobian matrix, is to calculate it on an element-wise basis. Instead of calculating it from the global residual vector, we take the local residual vector and numerically calculate the local Jacobian matrix from it. Afterwards, we assemble the local Jacobian matrices to a global Jacobian matrix. This can be done using the default `oomph-lib` linear solver, `SuperLU`. Lst. 26 shows the method definition, that needs to be overloaded in the driver code.

```
515 template<unsigned DIM>
516 void RefineableGeneral2FieldEquations<DIM>::fill_in_contribution_to_jacobian(
      Vector<double> &residuals, DenseMatrix<double> &jacobian)
```

Listing 26: Overloading the function for the assembly of the Jacobian matrix. Here, we can switch between using the assembly of the analytical Jacobian matrix and calculating the Jacobian matrix numerically using finite differences. Since this method is executed on the element, it will not include global interactions between elements. The method is very useful for debugging the implementation of the analytical Jacobian matrix.[31]

During the implementation of a new system, this method is very useful, since it allows the user to compare the analytical implementation of the Jacobian matrix to the numerically calculated one. The comparison of both matrices often indicates the source of a possible mistake in the implementation. Additionally, it allows the user to double check that his implementation of the Jacobian matrix is correct. Comparing the numerical to the analytical Jacobian matrix on an element-wise basis is clearer than analysing the global Jacobian matrix.

4.3.7 `int main(int argc, char* argv[])`, the actual driver code

This is the main method of every C++ code. It marks the start of the driver code.
The method is called with two arguments. `argc` represents the amount of arguments, that have been passed to it, when executing the program, while `argv` represents the values of these arguments. Keep in mind that the call to execute the program itself is the first argument. Calling the program `./Coalescence hello`, would result in `argc`=2, `argv[0]`=`./Coalescence` and `argv[1]`=`hello`.

We use this to set two problem parameters on execution. This allows us to use the same code for runs with different parameter values without recompiling the program. Including such a feature makes the program more suitable for parameter scans. In this case, we set the parameter ϵ and c. While ϵ is a parameter of the equation system itself (4.4), c represents the height of the droplets in the initial condition. By controlling the height of the droplets we control the contact angle of the droplets.

[31] File: `../oomph_Codes/user_drivers/Latex_Coalescene/Coalescence_driver_code_unc.cc`

In the following we name the statements of the `int main(int argc, char* argv[])` method and discuss them.
The first call of the method is to initialise a problem object of the class discussed in section 4.3.4. Lst. 27 displays the constructor call for the `problem` object. It will be used in the subsequent calls to control the timestepping process and output the solutions.

```
553 G2FMUnsteadyProblem<CustomRefineableGeneral2FieldQElements<1, 4> > problem;
```

Listing 27: Initialising the object `problem` to be of the type `G2FMUnsteadyProblem <CustomRefineableGeneral2FieldQElements<1,4> >`. The template parameter is a one dimensional, four node `CustomRefineableGeneral2FieldQElements`. By only editing this line we can change the amount of nodes along a line in each element. When we change the spatial dimension of the element, we need to make sure to use an appropriate mesh in the problem constructor as well.[32]

The lines following the code excerpt in Lst. 27 set a few properties of the `problem` object. They are explained briefly in the following list.

- Specifying the output directories for the documentation of the solutions. The user needs to make sure, that the specified output directories exist. If they do not exist `oomph-lib` will issue a warning, but will not interrupt the program.
- Setting the error limits of the Z2 spatial error estimator.
- Initialising the timestepper parameter `dt`, which corresponds to the length of a timestep. In the case of an adaptive timestepping scheme, we can also set a `maximum_dt`, which corresponds to a maximum timestep size.
- We set the tolerances for the linear solver, as well as the maximum number of Newton iterations. It is advisable to set them higher for the first timestep and reset them for consecutive steps.

After setting the problem properties, we check the arguments passed via the command line. In case of three arguments, we set the values of ϵ and c accordingly, while for one argument the simulation is started using the default values. The structure of the subsequent code is very similar, independent of the amount of arguments passed. Nonetheless, the following refers to the case of three passed arguments.
At first we set the boolean value, representing the first time step, to `true`. Then we use multiple calls of the `refine_uniformly()` method. It forces the mesh of the problem to refine to the next higher refinement level. The idea behind this procedure is to start with a very coarse mesh in the initialisation of the mesh in the problem constructor, since this represents the coarsest possible mesh. After the initialisation we force the mesh to refine multiple times and then set the initial conditions on an appropriately dense mesh.
After setting the initial conditions, we prepare the timestepping scheme. Therefore, we initialise a target error for the adaptive timestepping method, as well as variables to control the output frequency during the simulation. Since we are applying an adaptive timestepping scheme, we use a `while` loop as the enclosing iteration procedure. The method `doubly_adaptive_unsteady_newton_solve(dt, epsilon_t, max_adapt, first)` performs a timestep of the problem. It tries to advance the problem by the time `dt`, while allowing `max_adapt` spatial adaption steps. The method will adjust the timestep size `dt`, to satisfy the global temporal error norm within the tolerance of `epsilon_t`. Spatial adaption is performed after the adaptive timestep. The method returns a suggestion for the next timestep size `dt_next`.
Before and after the timestep we perform a few checks. At first we check, if it is necessary to output the current solution. Before executing the adaptive timestep, we also check whether the timestep suggestion has exceeded a certain threshold. After the first timestep, we readjust the number of possible adaption steps, as well as the maximum allowed residuals for the linear solver.

[32] File: `../oomph_Codes/user_drivers/Latex_Coalescene/Coalescence_driver_code_unc.cc`

As soon as the problem advanced to a time greater than t_max, the loop terminates. After the timestepping process we create another output file, which saves the parameters of the problem. This is particularly useful, when we want to rerun the simulation to reproduce the results. In case the user wants to rerun the simulation, continuing from this timestep, he would need to call the dump(const std::string &dump_file_name) method at this point. The method saves the current problem to the file specified in dump_file_name. The driver code discussed in this section does not feature restarts and needs to be extended to support them[33].

[33] see http://oomph-lib.maths.man.ac.uk/doc/unsteady_heat/two_d_unsteady_heat2/html/index.html

4.4 Spreading and coalescence of neighbouring droplets

The previously discussed implementation is used to calculate the time evolution of the model presented in section 4.1 (equations (4.19) and (4.20)).
First, we investigate the spreading of a single surfactant-covered droplet on a solid substrate. Since no wetting energy is induced, the liquid ideally wets the substrate and in consequence only one steady state, the homogenous flat-film state, exists. It follows that the droplet spreads until the complete domain is covered by a homogenous liquid film with a constant surfactant concentration. The initial condition of the parabolic droplet is modelled by

$$h_{init} = a(x-b)^2 + c \tag{4.27}$$

with initial volume

$$V_{init} = \frac{2}{3}(h_a - c)\sqrt{\frac{h_a - c}{a}}. \tag{4.28}$$

$h_a = 1$ corresponds to the height of the adsorption layer. To model a homogenous surfactant concentration on the droplet we approximate a step function using smooth hyperbolic functions. The initial surfactant concentration is given by

$$\Gamma_{init} = a_\Gamma \frac{\tanh\left(c_\Gamma(x-b+r)\right) + \tanh\left(c_\Gamma(-(x-b)+r)\right)}{2} \tag{4.29}$$

with a_Γ determining the homogenous surfactant concentration on the droplet and c_Γ describing the steepness of the step. To increase the numerical stability the complete domain is covered by a low surfactant concentration of $\Gamma_a = 0.02$.
Fig. 6 displays snapshots for the spreading of a single droplet initially covered by a homogenous surfactant concentration that sits on an equilibrium adsorption layer without surfactant.

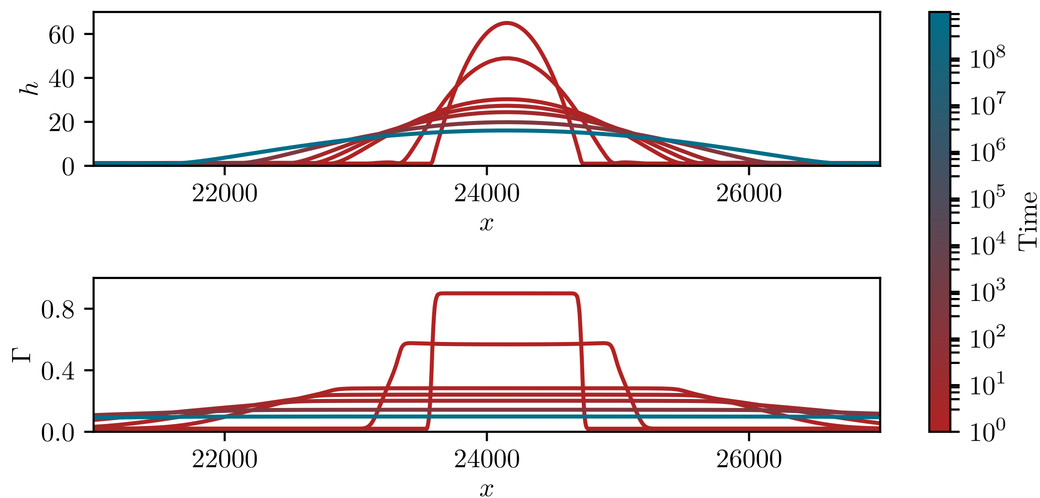

Figure 6: Shown are snapshots from the time evolution of a single surfactant covered droplet (top: liquid height profile, bottom: surfactant concentration profile) at different times (colour coded). The initial condition is a parabolic droplet, which is covered by a constant surfactant concentration. The remaining domain is covered by a thin liquid precursor film. We observe that the droplet is spreading symmetrically and approaches the homogenous solution. The simulation uses an adaptive timestepping routine and a spatially refineable 1D mesh for a domain size $l_x = 50000$ (only part shown). The remaining parameters are $D = 0.1$ and $\epsilon = 0.05$. The displayed profiles are at times $t = [0,\ 8.4 \times 10^5,\ 2.3 \times 10^7,\ 4.3 \times 10^7,\ 8.4 \times 10^7,\ 2.8 \times 10^8,\ 9.9 \times 10^8]$.[34]

[34] Initial conditions for droplet: $a = -0.00019528$, $c = 65$, $V_{init} = 50000$, $a_\Gamma = 0.9$ and $c_\Gamma = 0.05$

Next, we introduce a second initial droplet into the system, which is not covered by surfactant. Then, the Marangoni flow, induced by the surface tension gradient due to the surfactant concentration difference, can delay the coalescence of the droplets. Fig. 7 shows a sketch of the initial conditions for the following simulations of two droplets. It displays the surfactant coverage, as well as the direction of the Marangoni flow due to the surface tension gradient. It shows the initial conditions described in equations (4.27) and (4.29) as well.

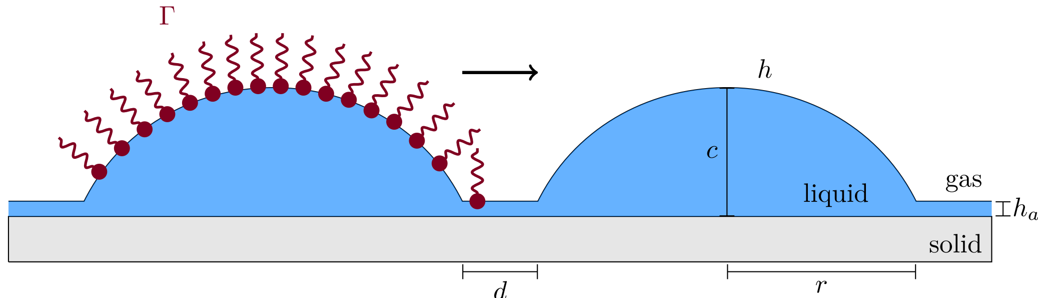

Figure 7: Sketch of two neighbouring droplets. The left droplet is covered by surfactant molecules and has a lower surface tension than the right droplet. The arrow indicates the direction of the Marangoni flow.

In Fig. 8, we show snapshots of the time evolution of two droplets. The left droplet is covered by a homogenous surfactant concentration, while the right droplet is bare. Therefore, the surface tension gradient between the two droplets induces a Marangoni flow towards the right. From the initial conditions, both parabolic profiles, fast relax into droplets smoothly connected to the adsorption layer. Then, both droplets spread until they get into contact. After contact, their coalescence can be delayed, due to the induced Marangoni flow, and both droplets keep to spread separately, pushing their centre of mass away from each other.

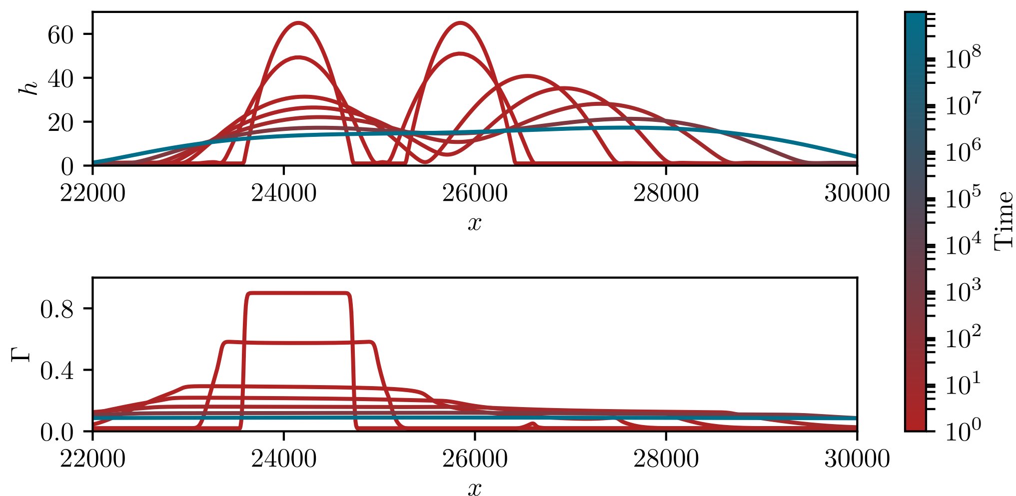

Figure 8: Shown are snapshots from the time evolution of two droplets next to each other (top: liquid height profile, bottom: surfactant concentration profile) at different times (colour coded). The left droplet is initially covered by a homogenous surfactant concentration (as the droplet in Fig. 6), while the right droplet is nearly bare. The droplets spread until their contact lines get in contact. Due to the surfactant-gradient induced Marangoni flow, the droplets do not coalescence, but keep spreading separately. Therefore, their coalescence is delayed until they merge at $t_{coalesce} \approx 0.7 \times 10^9$. The profiles are taken at times $t = [0, 7.9 \times 10^5, 1.5 \times 10^7, 3.5 \times 10^7, 8.8 \times 10^7, 2.9 \times 10^8, 9.9 \times 10^8]$ indicated by the colour-bar, while the remaining parameters are as in Fig. 6.[35]

[35] Initial conditions for droplets: $a = -0.00019528$, $c = 65$, $V_{init} = 50000$, $a_\Gamma = 0.9$, $c_\Gamma = 0.05$ and $d = 550$.

To show that the surface tension gradient is responsible for the delayed coalescence of the droplets, we remove the surfactant coverage of the left droplet. Now the entire system is only covered by a small residual homogenous surfactant concentration (guaranteeing numerical stability) and no surface tension gradient between the droplets exists. Fig. 9 shows the time evolution for the system. We can see that the droplets have already coalesced in the third displayed profile.

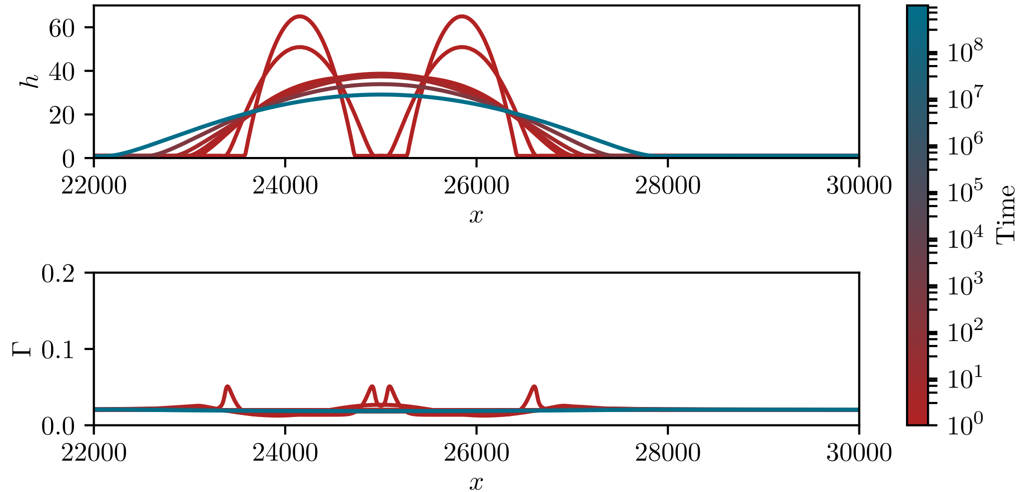

Figure 9: Shown is the fast coalescence of two droplets (top: liquid height profile, bottom: surfactant concentration profile) at different times (colour coded). The surfactant is only present in form of a small, homogenous concentration. Consequently, there is no Marangoni flow present and the droplets coalesce significantly faster than in Fig. 8. Already the third displayed profile shows coalesced droplets, which means that the droplets coalesce after a time of $t_{coalesce} \approx 10^7$, which is two orders of magnitude smaller than in Fig. 8. The displayed profiles are taken at times: $t = [0, 7.8 \times 10^5, 1.5 \times 10^7, 3.4 \times 10^7, 8.8 \times 10^7, 2.9 \times 10^8, 9.9 \times 10^8]$, the remaining parameters are as in Fig. 6.[36]

In Fig. 10, we show the position of the local maxima of both droplets, as well as of the minimum between them. We compare the coalescence time $t_{coalesce}$ and investigate the motion of the droplets.

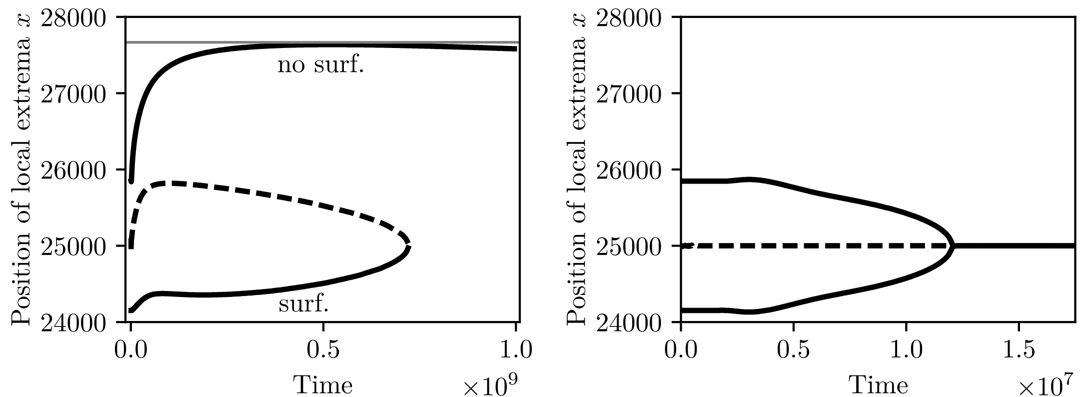

Figure 10: Shown is the dependence of the positions of the droplet maxima (solid lines) and the minimum (dashed line) on time, for the cases of delayed coalescence (left, corresponds to Fig. 8) and fast coalescence (right, corresponds to Fig. 9). The coalescence times differ by two orders of magnitude. For a detailed discussion see main text. The right panel displays the simulation results for times $t = [0, 1.75 \times 10^7]$, since the maximum position does not change afterwards. As in all other cases, the time evolution is calculated for $t = [0, 10^9]$.

[36] Initial conditions for droplets: $a = -0.00019528$, $c = 65$, $V_{init} = 50000$, $d = 550$ and $\Gamma = \Gamma_a$.

Considering the delayed coalescence in Fig. 10 (left), we note that the initial fast relaxation is barely visible. We observe that at the beginning of the time-simulation the droplet not covered by surfactant moves rapidly. This is due to the spreading of the droplets in conjunction with the Marangoni flow, which prevents the droplets from merging. Consequently, the droplets spread and push themselves away from each other. The spreading motion is followed by a delayed coalescence of the droplets. Using the grey line as a guide to the eye, we observe that the maximum of the droplet without surfactant is slowly retreating after it has reached its maximal x value, while the other maximum slowly approaches the minimum. Once the minimum and maximum meet at time $t_{coalesce} \approx 10^8$, the droplets have coalesced.
We see a fundamentally different behaviour of the droplets in the absence of surfactant, i.e. without a surface tension gradient (Fig. 10, right). The droplets start to spread after $t = 0$ as well, but they do not push each other away. Due to the absence of a surface tension gradient, there is no Marangoni flow to prevent the coalescence of the droplets. Therefore, they begin to merge immediately. We observe that the coalescence of the droplets is complete after $t_{coalesce} \approx 10^7$, which is significantly shorter compared to the delayed coalescence.
The immediate or delayed coalescence is influenced by the contact angle of the two droplets as well. Its effect on the coalescence of the two droplets is demonstrated by the time evolution shown in Fig. 11. Compared to Fig. 8, the initial height of the droplets and the initial distance between them is different. The different initial conditions account for a steeper initial contact angle θ_{init} and the droplets coalesce rapidly.

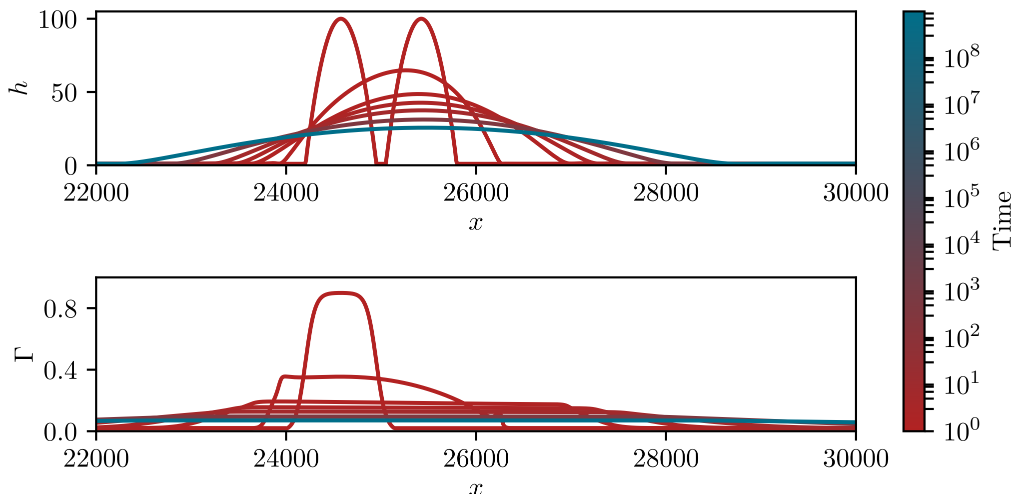

Figure 11: Shown are snapshots from the time evolution of the height profile (top) and the surfactant concentration profile (bottom) of two droplets next to each other. The left droplet is initially covered by a homogenous surfactant concentration, while the right droplet is nearly bare, introducing a Marangoni flow. Despite the Marangoni flow, the droplets coalesce rapidly at $t_{\text{coalesce}} \approx 10^6$. They are placed closer to each other and their initial height is increased as compared to Fig. 8. Consequently their initial contact angle $\theta_{\text{init}} \approx 0.48$ is steeper. The snapshots are taken at times: $t = [0, 8.2 \times 10^5, 1.3 \times 10^7, 3.6 \times 10^7, 8.8 \times 10^7, 2.9 \times 10^8, 9.9 \times 10^8]$, the remaining parameters are the same as in Fig. 6.[37]

Fig. 11 displays immediate coalescence of two droplets, due to a steep initial contact angle θ_{init} despite a Marangoni flow. To determine the value of θ_{init} we let the droplets relax until they are smoothly connected to the adsorption layer and take a snapshot of the system at $t \approx 10^6$. In this snapshot we calculate the tangent at the right inflection point of the left droplet and vice versa. Using the slope of the tangents we determine the corresponding contact angles and take their mean.

The local free energy contribution of the surfactant layer is given by $f_s = 1 + \epsilon\Gamma(\ln(\Gamma) - 1)$ in the rescaled model, with $\epsilon = \frac{k_B T}{a^2 \gamma^0}$ describing the ratio between the energetic contribution of the

[37]Initial conditions for droplets: $a = -0.000711153$, $c = 100$, $V_{init} = 50000$, $a_\Gamma = 0.9$, $c_\Gamma = 0.01$ and $d = 100$.

surfactant and the interfacial tension without surfactant. The surface tension gradients increase for high values of ϵ, which results in a stronger Marangoni effect.
We determine the influence of the initial contact angle θ_{init} on the coalescence of the two droplets, by performing a parameter scan in the $(\epsilon, \theta_{\text{init}})$-plane. Therefore, we calculate the time evolution of the system similar to Fig. 11 for various initial heights of the droplets, while their volume stays constant. The distance between the droplets remains constant as well. This results in different contact angles θ_{init} of the droplets. The calculations are repeated for different values of ϵ. To distinguish between a system of coalesced or separated droplets, we analyse the amount of local maxima in the system. As soon as there is only one local maximum in the system, the droplets did coalesce. Fig. 12 shows the time of coalescence.

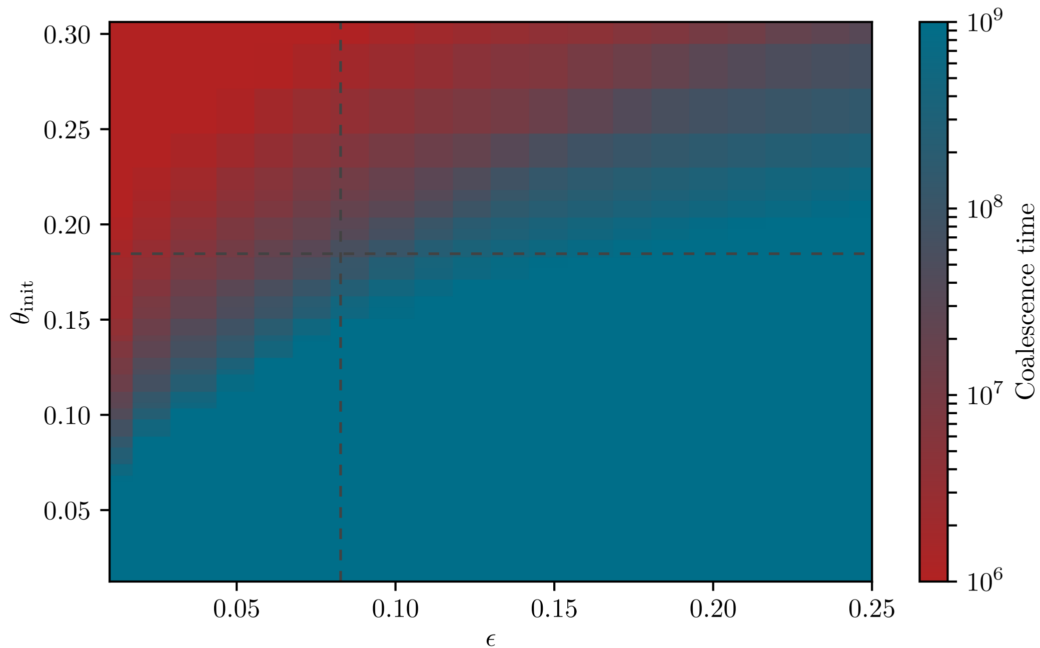

Figure 12: Shown is the coalescence time for two droplets in the domain $D = [0, l_x]$, on the $(\epsilon, \theta_{\text{init}})$-plane. Consequently the red areas correspond to fast coalescing droplets, while the blue areas represent droplets, which did not merge until the end of the simulation $t_{\text{end}} = 10^9$. The simulation parameters are the same as in Fig. 11. The distance $d = 100$ between droplets is kept constant. The displayed tiles are obtained using the coalescence time of the simulation, which is the nearest in the parameter space $(\epsilon, \theta_{\text{init}})$[38]. The dashed lines represent cuts through the parameter space, which are displayed in Fig. 13.

In Fig. 12, we see that we can suppress fast coalescence with large values of ϵ. Only for very steep contact angles between the droplets, we observe fast coalescence times despite high ϵ values. The results presented in Fig. 12 also show that there is a minimum contact angle required for fast coalescence.

[38]Initial conditions for droplets: $V_{init} = 86596.7$, $a_\Gamma = 0.9$, $c_\Gamma = 0.01$ and $d = 100$. The figure represents 960 time simulations with 48 distinct parameters $c \in [20, 101]$ and 20 distinct values of $\epsilon \in [0.01, 0.25]$. According to (4.28), we adapt a so that the initial volume of each droplet is V_{init}.

Fig. 13 shows the coalescence time of two droplets for a fixed value of θ_{init} (left panel) and a fixed value of ϵ (right panel). The left panel shows that the coalescence time increases for higher values of epsilon, which correspond to increasing surface tension gradients. For increasing contact angles the coalescence time decreases rapidly. The logarithmic scaling reveals the strong dependence of the coalescence time on the system parameter ϵ and the influence of the initial contact angle. Within the considered parameter range the coalescence time changes by multiple orders of magnitude.
In contrast to Fig. 12, a cubic spline interpolation method is used to display the results.

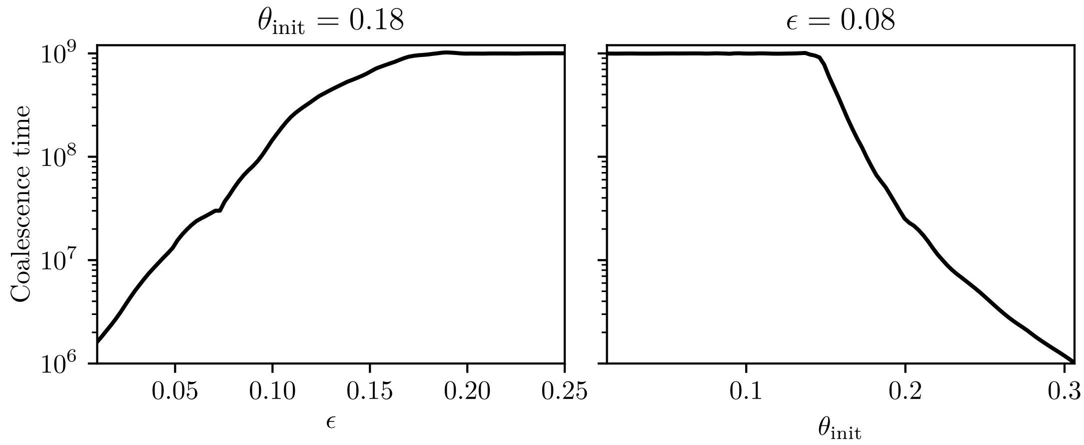

Figure 13: Shown is the coalescence time in dependence of ϵ with constant contact angle $\theta_{init} = 0.18$ (left panel) and the coalescence time in dependence of the contact angle with $\epsilon = 0.08$ (right panel). The horizontal (vertical) dashed line in Fig. 12 corresponds to the left (right) panel in this figure. The displayed results show a cubic spline interpolation of the coalescence times.

5 Thin-film equation

In the following section we consider a partially wetting simple liquid. Similar to section 3 we can describe the evolution equation in the context of gradient dynamics. We only consider one order parameter field, the liquid film height h, with the evolution equation

$$\partial_t h = \nabla \cdot \left[Q(h) \nabla \frac{\delta \mathcal{F}[h]}{\delta h} \right]. \tag{5.1}$$

The free energy functional is given by

$$\mathcal{F}[h] = \int_D \left[\gamma_{sl} + \frac{1}{2}(\nabla h)^2 + f_w(h) - ph \right] \mathrm{d}x. \tag{5.2}$$

Comparing (4.3) to the energy functional for the thin-film equation (5.2), we have an additional term. $\frac{1}{2}(\nabla h)^2$ corresponds to a term, which represents the Laplace pressure that appears at the interface between the liquid and the surrounding gas. In contrast to the model described in section 4 the wetting energy is given by

$$f_w(h) = -\frac{1}{2h^2} + \frac{1}{5h^5}. \tag{5.3}$$

This results in the so called thin-film equation. It describes the behaviour of a thin liquid film on a horizontal homogenous substrate and gives rise to dewetting phenomena. In contrast to the model in section 4 it has non-trivial steady states. Therefore, we use the following section to implement and demonstrate continuation and bifurcation methods in `oomph-lib`.

5.1 Mathematical modelling

Inserting (5.2) and (5.3) into (5.1) we arrive at the evolution equation for the film height h

$$\partial_t h = -\nabla \cdot \left(Q(h) \nabla \left[\Delta h + \Pi(h) \right] \right). \tag{5.4}$$

Here $Q(h)$ represents the mobility function, while $\Pi(h)$ represents the Derjaguin pressure and Δh corresponds to the Laplace pressure.

$$Q(h) = \frac{h^3}{3} \tag{5.5}$$

$$\Pi(h) = -\frac{1}{h^3} + \frac{1}{h^6} \tag{5.6}$$

5.1.1 Function space

To reduce the requirements on our test- and ansatz function space (see section 2.2.2), we introduce a second field w in (5.4)

$$\partial_t h = -\nabla \cdot \left(Q(h) \nabla \left[w + \Pi(h) \right] \right) \tag{5.7}$$

$$w = \Delta h. \tag{5.8}$$

Instead of a one dimensional equation of order four, we now have a system of two equations of order two. Therefore, the requirements on our test- and ansatz function space only requires square integrable functions up to order one ($H^1(D)$).
For the formulation of the weak form of (5.7)-(5.8) we consider a N dimensional domain D. Throughout the following, we use the notation $f(x_\alpha)$ with $\alpha \in [1, N]$ for $f(x_1, ..., x_N)$ and N the spatial dimension of the problem.
Equations (5.7) and (5.8) in the weak formulation read

$$r_h = -\int_D (\partial_t h)\, \phi_h \mathrm{d}^N x_\alpha - \int_D \nabla \cdot \left(Q(h) \nabla \left[w + \Pi(h) \right] \right) \cdot \phi_h \mathrm{d}^N x_\alpha \tag{5.9}$$

$$r_w = \int_D w \phi_w \mathrm{d}^N x_\alpha - \int_D \Delta h \phi_w \mathrm{d}^N x_\alpha. \tag{5.10}$$

We use integration by parts to shift a spatial derivative onto the test function

$$-\int_D \nabla \cdot (Q(h)\nabla [w+\Pi(h)])\,\phi_h \mathrm{d}^N x_\alpha = \underbrace{-\,(Q(h)\nabla [w+\Pi(h)])\,\phi_h|_{\partial D}}_{=0} + \int_D Q(h)\nabla [w+\Pi(h)]\cdot \nabla\phi_h \mathrm{d}^N x_\alpha \tag{5.11}$$

$$-\int_D \Delta h \phi_w \mathrm{d}^N x_\alpha = \underbrace{-\nabla h\phi_w|_{\partial D}}_{=0} + \int_D \nabla h\cdot\nabla\phi_w \mathrm{d}^N x_\alpha. \tag{5.12}$$

The boundary integrals in (5.11) and (5.12) vanish, since $\phi_h|_{\partial D} = 0$ and $\phi_w|_{\partial D} = 0$ and we obtain for the weak formulation

$$r_h = -\int_D (\partial_t h)\,\phi_h \mathrm{d}^N x_\alpha + \int_D Q(h)\nabla [w+\Pi(h)]\cdot\nabla\phi_h \mathrm{d}^N x_\alpha \tag{5.13}$$

$$r_w = \int_D w\phi_w \mathrm{d}^N x_\alpha + \int_D \nabla h\cdot\nabla\phi_w \mathrm{d}^N x_\alpha. \tag{5.14}$$

The general test functions are represented by ϕ_h and ϕ_w. Since we consider Neumann boundary conditions

$$\nabla_\perp h(x_\alpha) = 0 \text{ and } \nabla_\perp w(x_\alpha) = 0 \;\forall\; x_\alpha \in \partial D. \tag{5.15}$$

Hereby, $\nabla_\perp$ corresponds to the spatial derivative projected onto the normal vector of the domain boundary. Our test and ansatz functions belong to the Sobolev function space (see section 2.2.2)

$$h(x_\alpha), w(x_\alpha) \in H^1(D) \quad \text{and} \quad \phi_h(x_\alpha), \phi_w(x_\alpha) \in H^1(D). \tag{5.16}$$

5.1.2 Discretisation

To calculate our solution using the Galerkin method we discretise our ansatz and test functions into M nodes as described in section 2.3.1, equation (2.10) and (2.11)

$$h(x_\alpha, t) = \sum_{j=1}^{M} H_j(t)\psi_j(x_\alpha) \tag{5.17}$$

$$w(x_\alpha) = \sum_{j=1}^{M} W_j\psi_j(x_\alpha) \quad \text{with} \quad \psi_j(x_\alpha) \in H^1(D) \tag{5.18}$$

$$\phi_h(x_\alpha) = \sum_{k=1}^{M} \Phi_{h,l}\psi_l(x_\alpha) \tag{5.19}$$

$$\phi_w(x_\alpha) = \sum_{k=1}^{M} \Phi_{w,l}\psi_l(x_\alpha) \quad \text{with} \quad \psi_l(x_\alpha) \in H^1(D) \tag{5.20}$$

In order to use Newton's method for the calculation of the solution we need to insert our discretisations (5.17)-(5.20) into the weak formulation (5.13)-(5.14)

$$r_h = \sum_{l=1}^{M} \Phi_{h,l} \left(-\int_D \sum_{j=1}^{M} \partial_t H_j(t)\psi_j(x_\alpha)\psi_l(x_\alpha)\mathrm{d}^N x_\alpha + \int_D Q(h)\nabla\left(\sum_{j=1}^{M} W_j\psi_j(x_\alpha) + \Pi(h)\right)\cdot\nabla\psi_l(x_\alpha)\mathrm{d}^N x_\alpha \right) \tag{5.21}$$

$$r_w = \sum_{l=1}^{M} \Phi_{w,l} \left(\int_D \sum_{j=1}^{M} W_j\psi_j(x_\alpha)\psi_l(x_\alpha)\mathrm{d}^N x_\alpha + \sum_{j=1}^{M} H_j(t)\nabla\psi_j(x_\alpha)\cdot\nabla\psi_l(x_\alpha)\mathrm{d}^N x_\alpha \right). \tag{5.22}$$

Since the residuals r_h and r_w are supposed to vanish for a weak solution and any suitable test function, we can rewrite the equations above into $2M$ linear independent equations $r_{h,l}$ and $r_{w,l}$ with $l = 1, ..., M$

$$r_{h,l} = -\int_D \sum_{j=1}^{M} \partial_t H_j(t)\psi_j(x_\alpha)\psi_l(x_\alpha)\mathrm{d}^N x_\alpha + \int_D Q(h)\nabla\left(\sum_{j=1}^{M} W_j\psi_j(x_\alpha) + \Pi(h)\right)\cdot\nabla\psi_l(x_\alpha)\mathrm{d}^N x_\alpha \tag{5.23}$$

$$r_{w,l} = \int_D \sum_{j=1}^{M} W_j\psi_j(x_\alpha)\psi_l(x_\alpha)\mathrm{d}^N x_\alpha + \int_D \sum_{j=1}^{M} H_j(t)\nabla\psi_j(x_\alpha)\cdot\nabla\psi_l(x_\alpha)\mathrm{d}^N x_\alpha. \tag{5.24}$$

Expressing the nabla operators in terms of individual spatial derivatives yields

$$\begin{aligned} r_{h,l} = &-\int_D \sum_{j=1}^{M} \partial_t H_j(t)\psi_j(x_\alpha)\psi_l(x_\alpha)\mathrm{d}^N x_\alpha \\ &+\int_D Q(h)\left(\sum_{\beta=1}^{N}\left(\sum_{j=1}^{M} W_j\frac{\mathrm{d}\psi_j(x_\alpha)}{\mathrm{d}x_\beta} + \partial_h\Pi(h)\sum_{j=1}^{M} H_j(t)\frac{\mathrm{d}\psi_j(x_\alpha)}{\mathrm{d}x_\beta}\right)\frac{\mathrm{d}\psi_l(x_\alpha)}{\mathrm{d}x_\beta}\right)\mathrm{d}^N x_\alpha \end{aligned} \tag{5.25}$$

$$r_{w,l} = \int_D \sum_{j=1}^{M} W_j\psi_j(x_\alpha)\psi_l(x_\alpha)\mathrm{d}^N x_\alpha + \int_D \sum_{\beta=1}^{N}\sum_{j=1}^{M} H_j(t)\frac{\mathrm{d}\psi_j(x_\alpha)}{\mathrm{d}x_\beta}\frac{\mathrm{d}\psi_l(x_\alpha)}{\mathrm{d}x_\beta}\mathrm{d}^N x_\alpha. \tag{5.26}$$

Equations (5.25) and (5.26) represent the discretised weighted residual.

5.1.3 Jacobian matrix

We calculate the Jacobian matrix analytically. Although, the Jacobian of our system can be calculated numerically by `oomph-lib` using a finite differences scheme, it is advised to implement the assembly of the Jacobian matrix analytically. This greatly reduces the numerical expense and therefore increases the efficiency and speed of our program.
In the following we calculate the contribution to the Jacobian matrix for the residual vector $r_{h,l}$ (5.25)

$$J_{h;l,h;l2} = \left.\frac{\partial r_{h,l}}{\partial H_{l2}}\right|_{(H_1,...,H_M,W_1,...,W_M)} =$$

$$-\int_D \partial_{H_{l2}}(\partial_t h)\mathrm{d}^N x_\alpha \tag{5.27}$$

$$+\int_D \partial_h Q(h)\psi_l\left(\partial_{x_\alpha} w + \partial_h\Pi(h)\partial_{x_\alpha} h\right)\partial_{x_\alpha}\psi_{l2}\mathrm{d}^N x_\alpha \tag{5.28}$$

$$+\int_D Q(h)\left(\partial_{hh}\Pi(h)\psi_l\partial_{x_\alpha} h + \partial_h\Pi(h)\partial_{x_\alpha}\psi_l\right)\partial_{x_\alpha}\psi_{l2}\mathrm{d}^N x_\alpha \tag{5.29}$$

$$J_{h;l,w;l2} = \frac{\partial r_{h,l}}{\partial W_{l2}}|_{(H_1,...,H_M,W_1,...,W_M)} = \int_D Q(h)\partial_{x_\alpha}\psi_l\partial_{x_\alpha}\psi_k\mathrm{d}^N x_\alpha \tag{5.30}$$

and $r_{w,l}$ (5.26)

$$J_{w;l,h;l2} = \frac{\partial r_{w,l}}{\partial H_{l2}}|_{(H_1,...,H_M,W_1,...,W_M)} = \int_D \partial_{x_\alpha}\psi_l\partial_{x_\alpha}\psi_{l2}\mathrm{d}^N x_\alpha \tag{5.31}$$

$$J_{w;l,w;l2} = \frac{\partial r_{w,l}}{\partial W_{l2}}|_{(H_1,...,H_M,W_1,...,W_M)} = \int_D \psi_l\psi_{l2}\mathrm{d}^N x_\alpha. \tag{5.32}$$

5.2 Implementation in oomph-lib

The most basic approach to solving an equation in oomph-lib is to choose an already existing and implemented equation from the list of example codes and tutorials[39], which is similar to the equation we want to solve and adapt the code accordingly. It is the easiest way to get started with oomph-lib and highly recommended for beginners.
In contrast to the procedure described in section 3.2 we do not create a library, which we then use to link our driver codes against. This approach only uses the driver code for the implementation of the needed functionality. In general, this makes it easier in the early stages of the projects development, but more complicated to keep track of the code, as the project is growing.
For the following implementation of the thin-film equation we choose the unsteady heat equation from the above mentioned list[40]. It is an implementation of the unsteady heat equation in two dimensions with the source function $f(x_1, x_2, t)$

$$\sum_{i=1}^{2} \frac{\partial^2 u}{\partial x_i^2} = \frac{\partial u}{\partial t} + f\left(x_1, x_2, t\right). \tag{5.33}$$

The main difference between the unsteady heat equation and the thin-film equation (5.7) and (5.8) is that we need to implement a second field. Minor differences include the calculation of the residual vector and Jacobian matrix, as well as additions for the implementation of an integral constraint.

In the following sections we will discuss those differences and how to adapt the code of the unsteady heat equation in details. We begin with the CustomWettingElement<DIM, NNODE_1D>, which is the basic element used to solve our thin film system. Afterwards we discuss the child class of it, the RefineableCustomWettingElement<DIM, NNODE_1D> class. It represents the elements that are used to solve the thin-film equation on an adaptive mesh. At last, we describe the introduction of a volume constraint element, which will be used in continuation routines. Fig. 22 in appendix A.2 shows an inheritance diagram for the classes discussed in this section.

5.2.1 CustomWettingElement<DIM, NNODE_1D>

The CustomWettingElement<DIM, NNODE_1D> is the basic element for the solution of the thin-film equation.
It inherits from oomph::QUnsteadyHeatElement<DIM, NNODE_1D>, which is a class for isoparametric elements, that solves the unsteady heat equation. Although the example code on the website suggest an implementation in two dimensions, the class itself depends on the template parameter <DIM> that represents the spatial dimension of the element. Therefore, the implementation of the class is rather general and one can use it to solve the unsteady heat equation in one, two and three spatial dimensions. Theoretically, it is also possible to solve it in four or higher spatial dimensions, but then we would need to provide the corresponding basis functions (also called shape functions). For the already implemented dimensions, the element becomes linear, quadrilateral or brick-shaped. The second template parameter <NNODE_1D> corresponds to the amount of nodes along one edge of the element. For example a oomph::QUnsteadyHeatElement<2,3> corresponds to a quadrilateral element with 9 nodes.
By inheriting from oomph::QUnsteadyHeatElement<DIM, NNODE_1D>, we can reuse the geometric functionality, as well as the implemented shape functions. Therefore, we only have to adapt the element to calculate the solution of the thin-film equation, instead of the unsteady heat equation. As a consequence, we add typedefs and function pointers, which will be used for the mobility functions (5.5), the Derjaguin-pressure (5.6) and their derivatives.
We overload the method fill_in_generic_residual_contribution_ust_heat(...), which calculates the residual vector and the Jacobian matrix. The fill_in_contribution_to* methods are also overloaded, which is useful for bug-fixing, when implementing the analytical calculation of the Jacobian matrix.

[39] http://oomph-lib.maths.man.ac.uk/doc/example_code_list/html/index.html
[40] http://oomph-lib.maths.man.ac.uk/doc/unsteady_heat/two_d_unsteady_heat/html/index.html

The `output(...)` method of the element is overloaded to create output that suits our desired plotting program[41]. Overloading the `required_nvalue` function is very important, since the unsteady heat equation requires only one field, while the thin-film equation needs two. After applying the changes mentioned above, the `CustomWettingElement<DIM, NNODE_1D>` is ready to calculate the time evolution.
The following paragraphs will give a detailed insight of the overloaded methods.

5.2.1.1 `unsigned required_nvalue(const unsigned &n) const`

This method returns the amount of fields in the element. Since we use a helping field w (5.7) we need to set this value to 2 (fields h and w). At this point it is quite useful to establish a numbering convention, which associates a field to a value in each `Data` object. Throughout the code for the thin-film equation, the h field will be associated with index 0, while w is associated with index 1.

```
225 /// \short 2 values, index 0 = h, index 1 = w
226 // sets the amount of fields
227 unsigned required_nvalue(const unsigned &n) const
228 {
229  return 2;
230 }
```

Listing 28: Overloading the `required_nvalue(...)` method, which returns the amount of fields stored at each node. This method needs to be overloaded, since the unsteady heat equation only features one field. public of CustomWettingElement<DIM, NNODE_1D>[42]

5.2.1.2 typedefs and function pointers

In `oomph-lib` codes function pointers are used to formulate a problem in a rather general way. For example the `oomph::QUnsteadyHeatElement<DIM, NNODE_1D>` (inherited from `oomph::UnsteadyHeatEquationsBase`) uses a function pointer to implement the source function without knowing its exact definition. It only specifies that the function depends on the spatial coordinates $\vec{x}$ and the time t. When doing an actual calculation, one needs to set the function pointer to a specific method. This enables the code implementation to be rather general and it makes it very easy to calculate the unsteady heat equation with different source functions.

In the `CustomWettingElement<DIM, NNODE_1D>` class, function pointers are used to include the mobility function as well as the Derjaguin-pressure and their derivatives. Lst. 29 shows the implementation of function pointers using the example of the Derjaguin-pressure $\Pi(h)$.

```
140 /// typedef to implement the helping functions in each element
141 typedef double (*Wetting_Pi_FctPt)(const double& h);
142 /// access function for the function pointer (private attribute)
143 Wetting_Pi_FctPt& wetting_Pi_fct_pt()
144 {
145  return Wetting_Pi_fct_pt;
146 }
147 double bigPi_of_h(const double &x) const
148 {
149  double source = 0.0;
150
151  /// if function pointer is not set return 0, else get function value
152  if (Wetting_Pi_fct_pt != 0)
153  {
154   source = (*Wetting_Pi_fct_pt)(x);
155  }
156  return source;
```

[41] http://oomph-lib.maths.man.ac.uk/doc/FAQ/html/index.html#tecplot
[42] File: ../oomph_Codes/user_drivers/Latex_Thinfilm/CustomWettingElement_unc.cpp

```
157 }
```

Listing 29: In the first lines of the above code excerpt we define a `typedef` of a function pointer `(*Wetting_Pi_FctPt)`. It represents a method, which depends on a argument `h` of the type `double` and returns a value of the type `double`. In Lst. 30 we initialise a private attribute `Wetting_Pi_fct_pt` of this `typedef`. Since the attribute is `private` we provide an access method `wetting_Pi_fct_pt()` for it. At last we define a method that evaluates the function. `bigPi_of_h(...)` returns 0 if the function pointer `Wetting_Pi_fct_pt` is not set, otherwise it returns the value of the method, that the function pointer is pointing to. public of CustomWettingElement<DIM, NNODE_1D>[43]

First of all, we need to create a `typedef Wetting_Pi_FctPt` that defines the format of the function pointer. In this case, the function only depends on the film height h. We provide an access method `wetting_Pi_fct_pt()` for the function pointer `Wetting_Pi_fct_pt`, since the pointer is a private attribute of the class. At last we implement the function `bigPi_of_h(const double &x)`, which depends on the same kind of arguments as our `typedef`. This method will be called, when calculating the residual vector and the Jacobian matrix. It returns 0 if the function pointer is not set and the corresponding value if we have set the function pointer.

```
274 /// Function pointer
275 Wetting_Pi_FctPt Wetting_Pi_fct_pt;
```

Listing 30: Initialising the `private` function pointer `Wetting_Pi_fct_pt`. private of CustomWettingElement<DIM, NNODE_1D>[44]

Then we create a function pointer `Wetting_Pi_fct_pt` as a private attribute (see Lst. 30). This should be initialised to 0 in the constructor of the class, otherwise it can cause segmentation faults, when the method `bigPi_of_h(...)` is called.

5.2.1.3 `void output(std::ostream& output_file, const unsigned &n_plot)`

This method overloads the elements output function. It returns the spatial coordinates of the current node, as well as the values of the fields h and w. Lst. 31 shows the overloading of the `output(...)` method[45].

```
748 template<unsigned DIM, unsigned NNODE_1D>
749 void CustomWettingElement<DIM, NNODE_1D>::output(std::ostream& output_file, const
      unsigned &n_plot)
750 {
751  unsigned n_node = QUnsteadyHeatElement<DIM, NNODE_1D>::nnode();
752 
753  for (unsigned n = 0; n < n_node; n++)
754  {
755   for (unsigned i = 0; i < DIM; i++)
756   {
757    output_file << QUnsteadyHeatElement<DIM, NNODE_1D>::nodal_position(n, i) << "\t
        ";
758   }
759   output_file << this->node_pt(n)->value(0) << "\t" << this->node_pt(n)->value(1)
       << std::endl;
760  }
761 }
```

Listing 31: Overloaded `output(...)` method. For each node in the element it writes the spatial coordinates, as well as the nodal values of field h and w to the file `output_file`. The amount of spatial coordinates depends on the spatial dimensions of the problem. The values are separated by a “tab” character, while the values for each node are printed in different lines.[46]

[43] File: `../oomph_Codes/user_drivers/Latex_Thinfilm/CustomWettingElement_unc.cpp`
[44] File: `../oomph_Codes/user_drivers/Latex_Thinfilm/CustomWettingElement_unc.cpp`
[45] `http://oomph-lib.maths.man.ac.uk/doc/FAQ/html/index.html#tecplot`
[46] File: `../oomph_Codes/user_drivers/Latex_Thinfilm/CustomWettingElement_unc.cpp`

5.2.1.4 void fill_in_generic_residual_contribution_ust_heat(Vector<double> &residuals, DenseMatrix<double> &jacobian, DenseMatrix<double> &mass_matrix, unsigned flag)

In this method we calculate the residual vector, the Jacobian matrix and the mass matrix. The following will only highlight and comment things that are different to the unsteady heat equation. For a better understanding of the assembly of the residual vector and the Jacobian matrix have a look into the infamous (Not-So-)Quick Users Guide[47] on the `oomph-lib` homepage.
Since the `CustomWettingElement<DIM, NNODE_1D>` contains two fields, we create variables that hold the corresponding indices. This helps keeping track of the numbering scheme we established. Lst. 32 displays the variable declaration and assignment.

```
344 /// defining indeces
345 unsigned h_index = 0;
346 unsigned w_index = 1;
```

Listing 32: Defining the indices according to our numbering scheme displayed in section 5.2.1.1.[48]

After the calculation of the elements values of h, w and their spatial derivatives, we can calculate the values of the mobility function $Q(h)$, the Derjaguin-pressure $\Pi(h)$ and their derivatives. In Lst. 33 we define variables to hold the values of the functions $\Pi(h)$, $\partial_h\Pi(h)$, $\partial_h\partial_h\Pi(h)$, $Q(h)$ and $\partial_h Q(h)$ for the current element.

```
411   /// function values
412   double pi_func_value = bigPi_of_h(interpolated_h);
413   double pidh_func_value = bigPi_dh(interpolated_h);
414   double pidhdh_func_value = bigPi_dh_dh(interpolated_h);
415   double q_func_value = q_of_h(interpolated_h);
416   double qdh_func_value = q_dh(interpolated_h);
```

Listing 33: Using the h value of the current element, we assign the values of the functions $\Pi(h)$ and $Q(h)$, as well as their derivatives to variables. These variables will be used in the subsequent assembly of the residual vector and the Jacobian matrix. The first method called in this excerpt corresponds to the method defined in Lst. 29.[49]

The calculation of the residual vector $r_{h,l}$ (5.25) is displayed in Lst. 34. Here, the test function $\psi_l(x_\alpha)$ corresponds to `test(l)` in the program code.

```
432     // Add body force/source term and time derivative
433     residuals[local_eqn] += -1.0 * dhdt * test(l) * W;
434
435     // Laplace operator
436     for (unsigned alpha = 0; alpha < DIM; alpha++)
437     {
438      residuals[local_eqn] += q_func_value * (interpolated_dwdx[alpha] +
           pidh_func_value * interpolated_dhdx[alpha]) * dtestdx(l, alpha) * W;
439     }
```

Listing 34: Calculating and assembling the residual vector $r_{h,l}$. The assembly can be split into two parts. While the first part assembles the scalar bit of $r_{h,l}$, which includes the time derivative, the second part corresponds to the Laplace operator. The second part usually includes everything, that is multiplied by $\nabla\psi_l(x_\alpha)$.[50]

Since the variable `local_eqn` is already set to the `h_index`, it is efficient to calculate the Jacobian matrix parts $J_{h;l,w;l2}$ and $J_{h;l,w;l2}$. Therefore, a second variable `l2` is introduced, that loops over the test functions as well. For the implementation of $J_{h;l,w;l2}$ we will set the value of `local_unknown` to the `h_index` as well. Lst. 35 shows the implementation of $J_{h;l,w;l2}$.

```
465         /// add jacobian timestepping bit
```

[47] http://oomph-lib.maths.man.ac.uk/doc/quick_guide/html/index.html
[48] File: ../oomph_Codes/user_drivers/Latex_Thinfilm/CustomWettingElement_unc.cpp
[49] File: ../oomph_Codes/user_drivers/Latex_Thinfilm/CustomWettingElement_unc.cpp
[50] File: ../oomph_Codes/user_drivers/Latex_Thinfilm/CustomWettingElement_unc.cpp

```
466         jacobian(local_eqn, local_unknown) += -1.0 * psi(l2) * QUnsteadyHeatElement
              <DIM, NNODE_1D>::node_pt(l2)->time_stepper_pt()->weight(1, 0) * test(l) *
               W;
467
468         /// Laplace operator bit
469         for (unsigned i = 0; i < DIM; i++)
470         {
471          /// first part
472          jacobian(local_eqn, local_unknown) += qdh_func_value * psi(l2) * (
               interpolated_dwdx[i] + pidh_func_value * interpolated_dhdx[i]) * dtestdx
               (l, i) * W;
473          /// second part
474          jacobian(local_eqn, local_unknown) += q_func_value * (interpolated_dhdx[i]
                * pidhdh_func_value * psi(l2)) * dtestdx(l, i) * W;
475          /// third part
476          jacobian(local_eqn, local_unknown) += q_func_value * dpsidx(l2, i) *
               pidh_func_value * dtestdx(l, i) * W;
477         }
```

Listing 35: Assembling the Jacobian matrix $J_{h;l,w;l2}$. The positions of the contributions within the Jacobian matrix are determined by the equation numbers `local_eqn` and `local_unknown`. In line 466 we add the time derivative (5.27), which is analogous to the implementation in the unsteady heat equation code. The bit of (5.28) is added in line 472, while (5.29) is added in line 474 and 476.[51]

During continuation routines we need to calculate the eigenvalues and eigenfunctions of the system[52]. Therefore, we need to implement the assembly of the mass matrix. The mass matrix can be assembled parallel to the Jacobian matrix, but it has fewer entries. In Lst. 36 we show the only entry to the mass matrix for the thin-film equation.

```
457         // add mass_matrix bit
458         if (flag == 2)
459         {
460          mass_matrix(local_eqn, local_unknown) += test(l) * psi(l2) * W;
461         }
```

Listing 36: Assembling the mass matrix for the thin-film equation. `local_eqn` and `local_unknown` are set to the `h_index`. Therefore, this code excerpt corresponds to the same position of the mass matrix as the Jacobian matrix shown in Lst. 35.[53]

5.2.1.5 `void fill_in_contribution_to_residuals`

The class `oomph::GeneralisedElement` provides interfaces for the computation of the residual vector as well as the Jacobian matrix. These functions have to be overloaded in the specific problem implementation. They all have in common that they call the `fill_in_generic_residual_contribution_ust_heat(...)` method, but with a different value for the `unsigned flag` argument. Lst. 37 shows the method, which overloads `fill_in_contribution_to_residuals(Vector<double> &residuals)` method of `oomph::GeneralisedElement`. It calls the above mentioned method for the calculation of the residual vector, Jacobian matrix and mass matrix with the argument `flag=0`. This forces the method to only assemble the residual vector and therefore dummy matrices are passed as arguments for the Jacobian and mass matrix.

```
769 template<unsigned DIM, unsigned NNODE_1D>
770 void CustomWettingElement<DIM, NNODE_1D>::fill_in_contribution_to_residuals(Vector
      <double> &residuals)
771 {
772  fill_in_generic_residual_contribution_ust_heat(residuals, GeneralisedElement::
      Dummy_matrix, GeneralisedElement::Dummy_matrix, 0);
```

[51] File: `../oomph_Codes/user_drivers/Latex_Thinfilm/CustomWettingElement_unc.cpp`
[52] `http://oomph-lib.maths.man.ac.uk/doc/eigenproblems/harmonic/html/index.html`
[53] File: `../oomph_Codes/user_drivers/Latex_Thinfilm/CustomWettingElement_unc.cpp`

```
773 }
```

Listing 37: The public wrapper method for the protected fill_in_generic_residual_contribution_ust_heat(...) method to only calculate the residual vector. Consequently dummy matrices are passed for the references to the Jacobian and mass matrix. This method needs to be overloaded, since it will be called from the linear solver routines for the assembly of the residual vector. public of CustomWettingElement<unsigned DIM, unsigned NNODE_1D>[54]

Similar to the wrapper method in Lst. 37 for the calculation of the residual vector, we overload the fill_in_contribution_to_jacobian_and_mass_matrix method as well, which calculates and assembles the residual vector, the Jacobian matrix and the mass matrix.

5.2.1.6 void fill_in_contribution_to_jacobian(...)

Similar to the above mentioned wrapper methods, this method is used for the calculation of the Jacobian matrix. In contrast to the other methods, this one plays a rather important role during code development. By overloading this method we decide, whether the Jacobian matrix is calculated from the fill_in_generic_residual_contribution_ust_heat(...) method or if it should be calculated from the residual vector using an internal finite differences method.

During the code development one can switch between the two options to double check the implementation of the analytical Jacobian matrix. Therefore, this method represents a rather important bugfixing tool.

```
299 template<unsigned DIM, unsigned NNODE_1D>
300 void CustomWettingElement<DIM, NNODE_1D>::fill_in_contribution_to_jacobian(Vector<
      double> &residuals, DenseMatrix<double> &jacobian)
```

Listing 38: Definition head of a wrapper method for the protected fill_in_generic_residual_contribution_ust_heat(...) method to calculate the residual vector and Jacobian matrix. By commenting and uncommenting the two code sections of the methods body, we can switch between using the analytically calculated Jacobian matrix and the Jacobian matrix calculated from finite differences. It uses pass-by-reference for the results. public of CustomWettingElement<unsigned DIM, unsigned NNODE_1D>[55]

5.2.2 RefineableCustomWettingElement<DIM, NNODE_1D>

This class facilitates mesh refinement for the current system. It inherits from CustomWettingElement<DIM, NNODE_1D>, RefineableQElement<DIM> and ElementWithZ2ErrorEstimator.

It needs to modify and overload the fill_in_generic_residual_contribution_ust_heat(...) method to handle hanging and master nodes. This can be done analogously to section 3.2.5.

To establish a criterion which triggers the (un-) refinement of an element, we implement the functionality of a Z2 error estimator. Therefore, we overload the methods num_Z2_flux_terms() and get_Z2_flux(...). While the former method returns the length of the flux vector considered by the error estimator, the latter is used to calculate the flux vector itself. For the thin-film equation, we use the gradient of the field h as the criterion for mesh refinement. Consequently the length of the flux vector needs to be equal to the spatial dimension of the system DIM. The implementation of those methods is very similar to the one described in section 3.2.6, but here only one field is considered.

The remaining methods of this class are either empty, self explanatory or used in conjunction with a volume constraint during continuation. The methods needed to facilitate the implementation of a volume constraint are described below.

[54] File: ../oomph_Codes/user_drivers/Latex_Thinfilm/CustomWettingElement_unc.cpp
[55] File: ../oomph_Codes/user_drivers/Latex_Thinfilm/CustomWettingElement_unc.cpp

5.3 Introducing a volume constraint

When simulating the time evolution of the thin-film equation, we ensure that the volume of the droplet is constant over time

$$V_0 = \int_D h \mathrm{d}x_\alpha \quad \text{with} \quad \partial_t V_0 = 0. \tag{5.34}$$

Fortunately, equation (5.4) is a continuity equation, which means that the constraint in (5.34) (also called volume constraint) is automatically satisfied in a time simulation. To show this, we integrate (5.4) over the complete domain

$$\int_D \partial_t h \mathrm{d}x_\alpha = -\int_D \nabla j \cdot \mathrm{d}x_\alpha \quad \text{with} \quad j = Q(h)\nabla\left[\Delta h + \Pi(h)\right]. \tag{5.35}$$

For the left hand side of equation (5.35) we first integrate and then apply the derivative. On the right hand side, we apply the divergence theorem yielding

$$\partial_t V_0 = \oint_{\partial D} \vec{j} \cdot \vec{n} \mathrm{d}s. \tag{5.36}$$

The right hand side of equation (5.36) is equal to 0, since we consider the thin-film equation with Neumann boundary conditions. We follow the same reasoning as in section 5.1.1, where we shift the spatial derivative onto the test function.
Unfortunately this mechanism does not apply in the parameter continuation of equation (5.4). Another method needs to ensure constraint (5.34) in the continuation routines. We modify the form of the weighted residual r_h (5.13) by adding the volume constraint (5.34) to it

$$r_h = -\int_D \partial_t h \phi_h \mathrm{d}x_\alpha + \int_D Q(h)\nabla\left[w + \Pi(h)\right] \cdot \nabla\phi_h \mathrm{d}x_\alpha + \left[\int_D h \mathrm{d}x_\alpha - V_0\right]\lambda \tag{5.37}$$

where λ is a height. To implement equation (5.37) into `oomph-lib`, we need to create a new element, the `VolumeConstraintElement`, which represents the volume constraint and modify the existing elements, `CustomWettingElement<DIM, NNODE_1D>` and `RefineableCustomWettingElement<DIM, NNODE_1D>`. The `VolumeConstraintElement` will add a new degree of freedom to the problem, which overdetermines the problem.
We then "hijack" the internal height degree of freedom of one single node of the mesh and associate it with the equation of the volume constraint. This way we ensure, that we have the same number of degrees of freedom, including the volume constraint element.

5.3.1 `VolumeConstraintElement`

The `VolumeConstraintElement` class inherits from the very basic `GeneralisedElement` class. It stores three private attributes:

- `*Prescribed_Volume_pt`
  A pointer to a variable, which sets the volume V_0 of the system.
- `External_data_index_of_traded_height`
  Index of the external data that contains the traded height. This value corresponds to a node number.
- `Index_of_traded_height`
  Index of the value in the traded height data that corresponds to the traded height value. It is the index of the value we want to trade of the node `External_data_index_of_traded_height`. In this case it should always be 0, since every height value is associated with the index 0.

The constructor of this class has three arguments, which correspond to the above mentioned attributes. Calling the constructor method initialises the attributes. The first argument of the

constructor is a pointer to a variable, which sets the volume of the system. The second argument is a pointer to the node, whose degree of freedom we will trade for the volume constraint, while the final argument is the index, which determines the specific degree of freedom of the node that will be traded. The node, whose degree of freedom is traded, is added to the `VolumeConstraintElement` as external data. Lst. 39 displays the constructor of this class.

```
14  VolumeConstraintElement(double* prescribed_volume_pt, Data* height_traded_data_pt
      , const unsigned& index_of_traded_height)
15  {
16   // Store pointer to prescribed volume
17   Prescribed_Volume_pt = prescribed_volume_pt;
18   // Add as external data and record the index
19   External_data_index_of_traded_height = add_external_data(height_traded_data_pt);
20   // Record index
21   Index_of_traded_height = index_of_traded_height;
22  }
```

Listing 39: Constructor of the `VolumeConstraintElement` class. It assigns the attributes in the above mentioned list. The data pointer passed to it, is added to the element as external data.[56]

Like all element classes, this class needs to overload the wrapper methods `fill_in_contribution_to_residuals`, `fill_in_contribution_to_jacobian` and `fill_in_generic_contribution_to_residuals_volume_constraint`. All wrapper methods call the private `fill_in_generic_contribution_to_residuals_volume_constraint` method.

```
80  void fill_in_generic_contribution_to_residuals_volume_constraint(Vector<double> &
      residuals)
81  {
82   const int local_eqn = this->external_local_eqn(
       External_data_index_of_traded_height, Index_of_traded_height);
83
84   if (local_eqn >= 0)
85   {
86    residuals[local_eqn] -= *Prescribed_Volume_pt;
87   }
88  }
```

Listing 40: Adding the contribution of the volume constraint element to the residual vector. It corresponds to the $-V_0$ term in (5.34), which does not contribute to the Jacobian or mass matrix. This method is called by the mentioned wrapper methods. private of VolumeConstraintElement[57]

5.3.2 CustomWettingElement<DIM, NNODE_1D>

In the following, we discuss the modifications that are made to the `CustomWettingElement<DIM, NNODE_1D>` class to support the introduction of a volume constraint. The modifications applied to the `RefineableCustomWettingElement<DIM, NNODE_1D>` class are analogous.

We implement routines into the element that allow to determine, whether the current element is the one, whose height degree of freedom is being "hijacked". Additionally, we extend the `fill_in_generic_residual_contribution_ust_heat` method. First, we add three `public` attributes to the class.

- `Data_number_of_traded_height`
  If the element contains the node, whose degree of freedom is replaced by the volume constraint, this attribute corresponds to the node number in the element. If this element does not contain the corresponding node number, this attribute adds the "traded" height value of the `VolumeConstraintElement` as external data and this attribute stores the external data number.

- `Index_of_traded_height_value`
  This attribute corresponds to the index of the value of the node `Data_number_of_traded_height`, whose degree of freedom is taken for the volume constraint element. If the current

[56] File: ../oomph_Codes/user_drivers/Latex_Thinfilm/VolumeConstraintElement_unc.cpp
[57] File: ../oomph_Codes/user_drivers/Latex_Thinfilm/VolumeConstraintElement_unc.cpp

element, does not contain the hijacked degree of freedom it points to the index of the data object of the VolumeConstraintElement.

- Traded_height_stored_at_node
  The attribute is a boolean value that indicates, whether the traded height is stored as a nodal value in this element or whether it is stored as external data.

5.3.2.1 set_volume_constraint_element(...)

This method is called after creating the VolumeConstraintElement object, since the argument passed to it is a pointer to the mentioned object. If the current element is the element, whose degree of freedom is traded, it will set the boolean value Traded_height_stored_at_node to true and set the Data_number_of_traded_height and Index_of_traded_height_value to the corresponding values. In case the element does not contain the node, whose height degree of freedom is traded, it will simply add the traded height value as external data.

5.3.2.2 height_traded_local_eqn()

This method is called during the assembly of the residual vector and Jacobian matrix when the element adds its contribution to the volume constraint. It returns the corresponding local equation numbers local_eqn. If the element is the one, whose degree of freedom was traded, it returns the internal equation number, while the external equation number is returned for all other elements. Lst. 41 shows the method declaration and definition of height_traded_local_eqn().

```
78  inline int height_traded_local_eqn()
79  {
80   //Return the appropriate nodal value if required
81   if (Traded_height_stored_at_node)
82   {
83    return this->nodal_local_eqn(Data_number_of_traded_height,
        Index_of_traded_height_value);
84   }
85   else
86   {
87    return this->external_local_eqn(Data_number_of_traded_height,
        Index_of_traded_height_value);
88   }
89  }
```

Listing 41: Returns the local equation number for the elements contribution to the volume constraint. Hereby, the method differs between the internal data of the “traded” node and external data of every other element. public of CustomWettingElement<DIM, NNODE_1D>[58]

5.3.2.3 Modifications to fill_in_generic_residual_contribution_ust_heat(...)

To add the contributions to residual vector and Jacobian matrix, we modify the fill_in_generic_residual_contribution_ust_heat(...) method.
In (5.37) the volume constraint term is not multiplied by the test functions ϕ_h. Therefore, we write the modifications outside the loop over the test functions l. In Lst. 42 we display the extension to the method definition of the fill_in_generic_residual_contribution_ust_heat(...) method.

```
569   if (!ExactSolnForUnsteadyHeat::timestepping_bool)
570   {
571    local_eqn = this->height_traded_local_eqn();
572    if (local_eqn >= 0)
573    {
574     residuals[local_eqn] += interpolated_h * W;
575     if (flag == 1)
576     {
```

[58] File: ../oomph_Codes/user_drivers/Latex_Thinfilm/CustomWettingElement_unc.cpp

```
577        for (unsigned l2 = 0; l2 < n_node; l2++)
578        {
579         local_unknown = this->nodal_local_eqn(l2, h_index);
580         if (local_unknown >= 0)
581         {
582          jacobian(local_eqn, local_unknown) += psi(l2) * W;
583         }
584        }
585       }
586      }
587     }
```

Listing 42: Adding the contributions of the `CustomWettingElement<DIM, NNODE_1D>` objects to the volume constraint of the residual vector and the Jacobian matrix. It corresponds to the $\int_D h \mathrm{d}x_\alpha$ term of (5.37). Notice the sign difference of the contributions compared to Lst. 40. We call the `height_traded_local_eqn()` method to obtain the local equation number. The `if` statement in the first line of this code excerpt is used to determine, whether we are currently calculating timestepping or continuation simulations. It will only execute the following code during continuation procedures.[59]

5.3.3 UnsteadyHeatProblem<ELEMENT>

In this class are quite a few methods that may be modified. Fortunately, they all provide the same functionality. The main concept of these methods is to introduce and set up a volume constraint to the system.
They are supposed to be called, after an sufficient amount of timesteps, when the system has relaxed into a steady state. From there it is possible to use the prescribed volume V_0 as an control parameter for continuation routines.

Lst. 43 displays the definition of the `problem_level_function()` method. It introduces a volume constraint to the system. The first part of the method initialises an `VolumeConstraintElement` object. To create this object, we pass a pointer to the prescribed volume `Volume_limit`, as well as the node number and the index of the traded degree of freedom. In this implementation, we always use the node with global node number 0 and we pass `h_index` to its constructor. To accommodate the volume constraint element we create an extra mesh `Volume_element_mesh_pt`. After the initialisation of the volume constraint element we loop over the `CustomWettingElement<DIM, NNODE_1D>` elements of the bulk mesh `Bulk_mesh_pt` and call the `set_volume_constraint_element` method, passing the volume constraint element.
To finalise the setup we add the `Volume_element_mesh_pt` mesh to the global mesh and rebuild it. After the global mesh is complete, we need to call the `assign_eqn_numbers()` to reassign the equation numbers. This is essential to make sure that the assembly of the residual vector and the Jacobian matrix works as expected and without segmentation faults.

```
167 template<class ELEMENT>
168 void UnsteadyHeatProblem<ELEMENT>::problem_level_function()
169 {
170  /// Create volume constraint element
171  unsigned h_index = 0;
172  VolumeConstraintElement* volume_element_pt = new VolumeConstraintElement(&
       ExactSolnForUnsteadyHeat::Volume_limit, Bulk_mesh_pt->node_pt(0), h_index);
173
174  Volume_element_mesh_pt = new Mesh;
175  /// Add it to the mesh
176  Volume_element_mesh_pt->add_element_pt(volume_element_pt);
177
178  /// Find number of elements in bulk mesh
179  unsigned n_element = Bulk_mesh_pt->nelement();
180
181  /// Loop over the elements to set up element-specific
182  /// things that cannot be handled by constructor
```

[59] File: `../oomph_Codes/user_drivers/Latex_Thinfilm/CustomWettingElement_unc.cpp`

```
183  for (unsigned i = 0; i < n_element; i++)
184  {
185   ELEMENT *el_pt = dynamic_cast<ELEMENT*>(Bulk_mesh_pt->element_pt(i));
186   el_pt->set_volume_constraint_element(volume_element_pt);
187  }
188
189  /// Now we need to combine the meshes
190  add_sub_mesh(Volume_element_mesh_pt);
191
192  /// Rebuild the Problem's global mesh from its various sub-meshes
193  rebuild_global_mesh();
194
195  /// Do equation numbering
196  cout << " Number of equations: " << assign_eqn_numbers() << std::endl;
197 }
```

Listing 43: Method to introduce a volume constraint to the system. The prescribed volume is defined by the `Volume_limit` variable of the `ExactSolnForUnsteadyHeat` namespace. The node, which trades its degree of freedom with the volume constraint element is the one with global node number 0. After calling the `set_volume_constraint_element(...)` method from every element we rebuild the global mesh and reassign the equation numbers. public of UnsteadyHeatProblem<ELEMENT>[60]

5.4 The driver code

Now we write a driver code, which controls the simulation process. The first part of the driver code is used to set up the problem, while the second part governs the calculation routines. The general idea of the subsequently presented driver code is to use timestepping routines to relax onto a steady state. After the system has reached a steady state, we introduce a volume constraint to the system and start continuation. The driver code mainly evolves around an initialisation of the `UnsteadyHeatProblem<ELEMENT>` class.

5.4.1 Defining system parameters and functions

We create a specific namespace `ExactSolnForUnsteadyHeat` to store problem parameters, to define the mobility function, the Derjaguin pressure and their derivatives. Additionally, this namespace includes a method, which is used to apply the initial conditions of the problem. Lst. 44 shows the definition of the problem parameters. It includes the definition of a boolean value, which is used to determine between timestepping and continuation routines. This is necessary, since the now extended method `fill_in_generic_residual_contribution_ust_heat(...)` can only be called when the system includes a volume constraint (see section 5.3.2.3). If the system does not include a volume constraint element, it will cause a segmentation fault. Therefore, we ensure the extension to the method is only called when we are performing continuation routines.

```
58  /// control parameter for continuation
59  double Volume_limit = 1189.95;
60
61  /// domain size and discretisation
62  double lx = 8 * M_PI;
63  double ly = 8 * M_PI;
64  unsigned nx = 5;
65  unsigned ny = 5;
66
67  /// boolean value to switch between timestepping and continuation
68  bool timestepping_bool = true;
```

Listing 44: Defining the domain size l_x and l_y, the mesh's initial discretisation n_x and n_y and the variable `Volume_limit`, which represents the volume V_0 of the volume constraint element. The attribute `Volume_limit` will be used as a control parameter during the continuation routines. The boolean value `timestepping_bool` distinguishes between timestepping and continuation routines. namespace ExactSolnForUnsteadyHeat[61]

[60] File: `../oomph_Codes/user_drivers/Latex_Thinfilm/UnsteadyHeatProblem_unc.cpp`

Furthermore we need to define the mobility function, the Derjaguin pressure and their derivatives. They are defined as methods of the current namespace. In the constructor of the `UnsteadyHeat-Problem<ELEMENT>` class, we set the function pointers of the elements to the methods defined in this `namespace`. In Lst. 45 we define the methods, which correspond to the functions $\Pi(h)$, $\partial_h\Pi(h)$, $\partial_h\partial_h\Pi(h)$, $Q(h)$ and $\partial_h Q(h)$.

```
 73  double bigPi_of_h(const double& h)
 74  {
 75   double Pi = -1.0 / pow(h, 3.0) + 1.0 / pow(h, 6.0);
 76
 77   return Pi;
 78  }
 79
 80  double bigPi_dh(const double& h)
 81  {
 82   double Pidh = 3.0 / pow(h, 4.0) - 6.0 / pow(h, 7.0);
 83
 84   return Pidh;
 85  }
 86
 87  double q_of_h(const double& h)
 88  {
 89   double Q = pow(h, 3.0) / 3.0;
 90
 91   return Q;
 92  }
 93
 94  double q_dh(const double& h)
 95  {
 96   double Qdh = pow(h, 2.0);
 97
 98   return Qdh;
 99  }
100
101  double bigPi_dh_dh(const double& h)
102  {
103   double Pidhdh = -12.0 / pow(h, 5.0) + 42.0 / pow(h, 8.0);
104
105   return Pidhdh;
106  }
```

Listing 45: Definition of the methods, that represent the mobility function $Q(h)$, its derivative $\partial_h Q(h)$, the Derjaguin pressure $\Pi(h)$ and its derivatives $\partial_h\Pi(h)$ and $\partial_h\partial_h\Pi(h)$. `namespace ExactSolnForUnsteadyHeat`[62]

At last, we define a method that sets the initial conditions of the problem. The method `get_initial_conditions(const Vector<double>& x, Vector<double>& soln)` uses pass-by-reference with the `soln` argument, to return the initial conditions. Lst. 46 shows the implementation of the initial conditions for a two dimensional system. It sets the initial conditions to a quarter of a droplet with its maximum at the origin. The quarter of a droplet only fulfils the Neumann boundary conditions if its maximum is at the origin for a domain $D = [0, l_x] \times [0, l_y]$. Using these initial conditions for the simulation of the time evolution, we ensure that the droplet position is fixed on the mesh during relaxation into a steady state. To display a full size droplet we can mirror the solution at the domain boundaries. This results in an equivalent solution of the problem with periodic boundary conditions on a domain of size $[-l_x, l_x] \times [-l_y, l_y]$. Unless stated differently, we use the latter representation in the following to display simulation results.

```
120  void get_initial_conditions(const Vector<double>& x, Vector<double>& soln)
121  {
122   /// Initial Conditions for DIM=2
123   /// Quarter of a droplet, with maximum on a domain corner
124   unsigned h_index = 0;
125   unsigned w_index = 1;
126
```

[61] File: `../oomph_Codes/user_drivers/Latex_Thinfilm/wetting_time_two_d_V21_Lite_unc.cc`
[62] File: `../oomph_Codes/user_drivers/Latex_Thinfilm/wetting_time_two_d_V21_Lite_unc.cc`

```
127   double amplitude = 5.0;
128   double r = sqrt(pow(x[0], 2) + pow(x[1], 2));
129   double r_max = 0.002 * sqrt(2) * 4 * M_PI;
130   double decay_parameter = -1.0 / (pow(2 * M_PI, 2));
131
132   soln[h_index] = amplitude * cos(r_max * r) * exp(decay_parameter * pow(r, 2)) +
        1;
133   soln[w_index] = -amplitude*exp(decay_parameter * pow(r, 2)) * cos(r_max * r) *
        (-1.0 * decay_parameter * pow(2 * r, 2) + pow(r_max, 2) + -2.0 *
        decay_parameter);
134 }
```

Listing 46: Setting the initial conditions for the problem. The current implementation sets the initial conditions to a quarter of a droplet, with its maximum at $(x, y) = (0, 0)$. The droplet has the shape of a Gaussian curve modulate with a cosine function. Instead of approaching a film height of 0 far away from the origin, it approaches a precursor film height of 1. We set the initial conditions for the helping field w according to the initial conditions of the film height h,. Applying the initial conditions we ensure that the initial conditions fulfil the boundary conditions of the problem. For different initial conditions we need to redefine this method. `namespace ExactSolnForUnsteadyHeat`[63]

5.4.2 `UnsteadyHeatProblem<ELEMENT>`

The `UnsteadyHeatProblem<ELEMENT>` class governs the simulation. It creates a mesh, fills it with elements of the corresponding type and provides useful methods during and after the calculations. Additionally, it is used to output the calculated solutions. The class inherits from the `oomph::Problem` class, which already provides many useful methods[64].
In the following we discuss methods, which are central to the class. Since this class is used to govern the complete simulation, it grows rapidly in functionality. Therefore. it contains methods that are not necessary to understand the basic operating principles of the `oomph-lib` framework. These methods will be omitted in the following discussion or discussed rather briefly.

5.4.2.1 The constructor

The definition of the class' constructor is very similar to the one presented in section 4.3.5.1. The following list displays it tasks in order.

1. Changing the linear solver (optional).
2. Allocating memory for a timestepper (necessary).
3. Creating a mesh of the sizes and discretisation specified in `ExactSolnForUnsteadyHeat` namespace (necessary).
4. Initialising a Z2 error estimator for mesh adaption routines (necessary for mesh refinement).
5. Introducing a volume constraint element (optional, can also be done after the object construction by calling the `problem_level_function()` method).
6. Setting the function pointers of the elements to the methods in `ExactSolnForUnsteadyHeat` namespace (necessary).
7. Building the global mesh (necessary).
8. Assigning equation numbers (necessary).

[63]File: `../oomph_Codes/user_drivers/Latex_Thinfilm/wetting_time_two_d_V21_Lite_unc.cc`
[64]`http://oomph-lib.maths.man.ac.uk/doc/the_data_structure/html/classoomph_1_1Problem.html`

5.4.2.2 actions_after_adapt()

This is a method of the so called “action” methods, which are executed during the call of newton_solve(). These “action” methods are used to perform additional tasks during the solution process[65].
The method discussed in this paragraph is called after the mesh adaption. When continuing this method deletes the current volume constraint element and creates a new one. This ensures that the volume constraint is correctly applied after we adapt the element that holds the traded degree of freedom.

5.4.3 int main(int argc, char* argv[])

The main method of this driver code is similar to the one discussed in section 4.3.7. In the following, we will briefly discuss the tasks this method performs, only going into more details for the parts that are fundamentally different to the ones previously described.
The subsequent list gives an overview of the tasks performed by the main(...) method and discusses them briefly.

1. Initialising the problem object using the RefineableCustomWettingElement<2,4> class as the template parameter for the element class.

2. Setting up the output information object doc_info_time to output the solutions of the time evolution calculations.

3. Initialising the maximum calculation time t_max and the timestep width dt.

4. Defining the thresholds for the spatial (un-) refinement of the used mesh.

5. Refining the mesh and setting the initial conditions. Afterwards, resetting the Volume_limit variable of the ExactSolnForUnsteadyHeat namespace. This is important, since the value can change after applying the initial conditions to a finer mesh.

6. Starting the timestepping loop and writing the solution after every timestep into a file for post processing. In contrast to section 4.3.7, the timestepping is done using the method unsteady_newton_solve(...), which advances the system by a fixed timestep dt, instead of an adaptive one.

After the completion of the list we commence the continuation of the system.

We check the time evolution of the system to see, whether it has relaxed into a steady state. Since the dynamic of the system has slowed down significantly during the last few timesteps, we assume that the system is very close to a steady state. By calling the method newton_solve(), we perform a steady Newton solve and double check, whether the system is close to a steady state. If we are close to a steady state, the method will converge the system onto the steady state. For the case that we are too far away from a steady state, Newton’s method will not converge and the program exits with an error message.

After we made sure that the system is in a steady state, we prepare the continuation scheme. Therefore we set the timestepping_bool variable of the ExactSolnForUnsteadyHeat namespace to false. Additionally, we reset the variable Volume_limit of said namespace. To introduce the volume constraint element we call the problem_level_function() method (see section 5.3.3). After the introduction of the volume constraint, we are able to call the continuation_solve_lite() method, which will perform the actual continuation of the system.

[65] http://oomph-lib.maths.man.ac.uk/doc/order_of_action_functions/html/index.html

5.4.3.1 `continuation_solve_lite()`

This method governs the continuation scheme of the system. The method is similar to the `continuation_solve_check_all(...)` method, but has been reduced to its bare essentials making it more comprehensible. In the following, we will discuss the `continuation_solve_lite()` method in details and mention functionality, that has been omitted in the reduction of the `continuation_solve_check_all(...)` method.

At first, we set up a so called `trace_file`, which will keep track of the system during the continuation loop. The `trace_file` holds the information that will be used to display a bifurcation diagram after the continuation routines. Before we start the continuation loop, we reassign the equation numbers and define the step length `ds` for the pseudo arclength method.
The continuation loop calls the method `arc_length_step_solve(...)`, which performs a pseudo arclength step. The first argument passed to the method is a pointer to the continuation parameter. In this case it corresponds to the `Volume_limit` variable of the namespace `ExactSolnForUnsteadyHeat`. `ds` is the second argument and it represents the desired arclength of the step, while the third argument represents the number of spatial adaptions that are performed during the pseudo arclength step. The method itself returns a suggestion for the step length of the next step.
After the pseudo arclength step, we write the new information into the `trace_file`. Therefore, we calculate the new volume of the system, as well as its absolute deviation $|\delta h|$. The absolute deviation of the system is defined as

$$|\delta h| = \frac{1}{D}\int_D |h(\vec{x}) - h_0|\mathrm{d}\vec{x}, \tag{5.38}$$

where h_0 corresponds to the average film height. $|\delta h|$ will be used as a measure for the solution of the system. Before we perform the next pseudo arclength step, we save the current solution of the system.

Prior to the continuation loop we set the value of the global variable `Desired_newton_iterations_ds`. Depending on its value the `arc_length_step_solve(...)` method returns the suggestion for the next step length. After the successful calculation of the pseudo arclength step, the method compares the amount of newton iterations it took to converge to the value of `Desired_newton_iterations_ds`. If more iterations were needed the method suggests a smaller step size, while fewer will increase it.

We set the value of the boolean `Bifurcation_detection` to `true`. This causes the `arc_length_step_solve(...)` method to look for a sign change of the determinant of the Jacobian matrix between arclength steps. In case there is a sign change, the method will write the corresponding steps into the `bifurcation_info` file. A sign change of the determinant of the Jacobian matrix can indicate a bifurcation or turning point.

5.4.3.2 Bifurcation detection

During the continuation runs we want to detect bifurcation points. As previously mentioned we can set the boolean value `Bifurcation_detection` to `true`, which uses the sign change of the determinant of the Jacobian matrix as an indicator for a bifurcation point. This is numerically very efficient, but it does not ensure that we find all bifurcation points. We can construct multiple scenarios in which this method fails. If an even number of eigenvalues crosses the imaginary axis at once, this method will not detect an bifurcation point, since the sign of the determinant does not change.
Unfortunately this method can also falsely detect a bifurcation point, even though there is none. It happens when one (un-) refines the mesh and the number of degrees of freedom changes from even to odd or vice versa. This will cause a sign change of the determinant of the Jacobian matrix and therefore trigger a bifurcation point detection. By using an element type, which adds or subtracts

an even amount of degrees of freedom to the mesh during refinement (in two dimensions a <2,4> element would be sufficient), we can work around this issue.
Considering the above mentioned issues, we cannot solely rely on the sign change of the determinant of the Jacobian matrix for the detection of the bifurcation points. Therefore, we need to implement another way for the bifurcation point detection. To make sure that we find all bifurcation points, we calculate the eigenvalues of the system after each step and compare them to the eigenvalues of the previous step. This scheme comes at a higher numerical cost, but ensures that we only pick up on real bifurcation points and that we find all of them.
In the following we discuss a method, which is used to calculate the eigenvalues of the system.

5.4.3.3 Calculating eigenvalues and eigenfunctions

The method `calculate_eigenvalues_geq_zero(...)` is a wrapper method for the calculation of the eigenvalues and eigenfunctions. At first we define a solver for the eigenproblem (see Lst. 47).

```
474 this->eigen_solver_pt() = new ANASAZI;
```

Listing 47: Defining the `ANASAZI` solver as the eigensolver to solve the eigenproblem.[66]

In the `oomph-lib` framework all eigenproblems are solved using interfaces to third party libraries. The default eigensolver is the `LAPACK_QZ` eigensolver, which is distributed with `oomph-lib`. The `ARPACK` and the `ANASAZI` eigensolvers are extensions to the library, that need to be installed separately to use them[67].
After the definition of the eigensolver, we call the `solve_eigenproblem(...)` method to calculate the eigenvalues and the eigenfunctions. In Lst. 48 we show the method call and discuss its arguments and return values in the caption.

```
483 //Solve the eigenproblem
484 this->solve_eigenproblem(n_eval, eigenvalues, eigenvectors);
```

Listing 48: Calculating the eigenvalues and eigenfunctions. The variable `n_eval` corresponds to the number of eigenvalues we want to calculate, while `eigenvalues` is a vector of complex numbers to store the eigenvalues. `eigenvectors` is used to store the corresponding eigenfunctions. The variables `eigenvalues` and `eigenvectors` are resized internally according to the amount of calculated eigenvalues and the results are returned using pass-by-reference. We can call the method omitting the `eigenvectors` argument, which will cause it to only calculate the eigenvalues.[68]

The calls displayed in Lst. 47 and Lst. 48 build the heart of the `calculate_eigenvalues_geq_zero(...)` method. After the calculation of the eigenvalues and eigenfunctions, the method performs post processing like sorting the eigenvalues according to their real part.

5.4.3.4 Converging onto a bifurcation point

After the detection of a bifurcation point, we can use the `oomph-lib` framework to converge onto the bifurcation point. Therefore, we need to be sufficiently close to the point itself. From there we use an augmented system to converge onto the bifurcation point itself.
If we want to converge onto a fold point, we call the method `activate_fold_tracking(...)` from the `Problem` class. This will set the `Assembly_handler_pt` to a `FoldHandler` and subsequent calls of the `steady_newton_solve()` method will converge onto the fold point. The method `activate_fold_tracking(...)` takes two arguments. The first argument is a pointer to the continuation parameter, here `Volume_limit` of the `ExactSolnForUnsteadyHeat` namespace, while the second argument is a boolean value, which decides whether numerically efficient block factorisation is activated or not.

[66] File: `../oomph_Codes/user_drivers/Latex_Thinfilm/UnsteadyHeatProblem_unc.cpp`
[67] `http://oomph-lib.maths.man.ac.uk/doc/the_distribution/html/index.html#external`
[68] File: `../oomph_Codes/user_drivers/Latex_Thinfilm/UnsteadyHeatProblem_unc.cpp`

The oomph-lib library provides a way to converge onto a pitchfork bifurcation point as well. Therefore, we simply call the activate_pitchfork_tracking(...) method. Similar to the fold tracking method, this method sets the assembly handler to assemble an augmented system, which converges onto a Pitchfork bifurcation point. On a pitchfork bifurcation point a symmetry of the system is broken. The activate_pitchfork_tracking(...) methods second argument corresponds to a symmetry vector, which specifies the symmetry that needs to be broken on the bifurcation point we want to converge onto. The easiest way to provide this symmetry vector is to pass a previously calculated eigenfunction. After activating the pitchfork tracking, we need to call the steady_newton_solve() method to converge onto the bifurcation point. Since the pitchfork bifurcation depends on the symmetries of the systems, it is very useful to make sure that the mesh is in a symmetrical state. Hence, it is very advisable to refine the mesh to a uniform state before the pitchfork tracking.
The oomph-lib framework provides several classes for augmented systems to converge onto other types of bifurcations as well[69]. To find a complete list, consult the inheritance diagram of AssemblyHandler class.
After we converged onto a bifurcation point, we can proceed with regular continuation. Therefore, we need to call the deactivate_bifurcation_tracking() method, which resets the Assembly_handler_pt.

5.4.3.5 Branch switching

To switch onto a different branch we first need to converge onto a bifurcation point. From there we calculate the eigenvalues and the eigenfunctions. Then, we add the eigenfunction of the branch we want to follow to the current solution of our system. The method add_eigenvector_to_dofs(...) allows us to add an eigenvector, which is passed as an method argument, to the current solution. Additionally, we can scale the eigenvector by a factor, which we pass to the method as well. After we pushed our solution into the direction of a particular branch, we start the standard continuation routines. Similar to the pitchfork bifurcation point detection, branch switching works better on symmetrically or uniform refined meshes.

5.5 Numerical results

Using the previously described numerical methods, we solve (5.4) on a two-dimensional domain $[0, 8\pi] \times [0, 8\pi]$ with Neumann boundary conditions. Mirroring the solution at the domain boundaries, it can be expanded onto the domain of size $[-8\pi, 8\pi] \times [-8\pi, 8\pi]$ using periodic boundary conditions. The latter representation is used to display the solutions in the subsequent section.
To create a bifurcation bifurcation diagram, we need to find one stationary solution and use it as a starting solution. Then, we use continuation methods to follow the corresponding solution branch and switch onto other branches of steady solutions. The following list describes three different options to find a starting solution.

- The simplest way is to use a homogeneous solution as the starting solution. To find non-trivial solutions, we then continue along the homogeneous branch and switch onto other branches at bifurcation points.
- Applying non-trivial initial conditions, we call the method steady_newton_solve() to converge the initial conditions onto a stationary solution of the system. Subsequently, we use the stationary solution as the starting solution for the continuation methods. This method only works, if the initial conditions are sufficiently close to a stationary solution.
- If the method does not converge onto a stationary solution, we call the unsteady_newton_solve(const double & dt) method to perform a time evolution of the system. During the time evolution the system will approach a stationary solution. When the system is sufficiently close to the stationary solution, we call steady_newton_solve() to converge onto it. This is necessary to ensure that the system is in a stationary solution and did not approach a ghost.

[69] http://oomph-lib.maths.man.ac.uk/doc/the_data_structure/html/classoomph_1_1AssemblyHandler.html

5.5.1 Approaching a stationary solution using time evolution

To converge onto a non-trivial branch, we simulate the temporal evolution of the system. We start from initial conditions that are close to the shape of a single droplet. During the time evolution, the system relaxes into a steady state (a single droplet on the substrate). Fig. 14 shows the droplets shape at time $t = 100$. At this time we are sufficiently close to the fixpoint, to converge onto it using the `steady_newton_solve()` method.

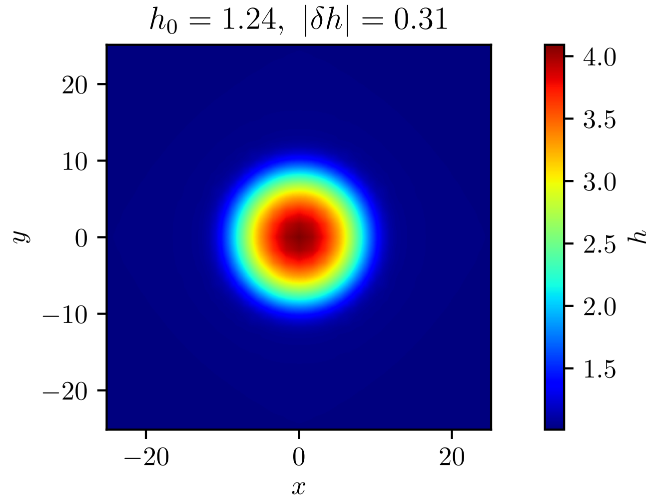

Figure 14: The droplet shape at time $t_{end} = 100$. Applying the initial conditions close to the shape of the steady drop, we use time simulation to relax onto the steady state. The simulation is carried out on a refineable mesh of the size $[0, 8\pi] \times [0, 8\pi]$ using Neumann boundary conditions. Mirroring the solution at the domain boundaries, the figure displays the solution of the same system on a $[-8\pi, 8\pi] \times [-8\pi, 8\pi]$ domain with periodic boundary conditions. The time simulation uses a backward differentiation formula (BDF) of order two with a fixed timestep $t_{step} = 0.2$, starting at $t = 0$.

5.5.2 Continuation

Starting from the stationary solution displayed in Fig. 14, we continue along the corresponding solution branch. Steady states are categorised into two different kinds of solutions, droplet solutions, which will be referred to by the letter Q, and stripe solutions, which will be denoted by the letter S. The letter is followed by a number, which denotes the number of stripes or droplets in the domain. A single droplet (as in Fig. 14) is denoted by $Q1$, while two stripes are referred to as $S2$.

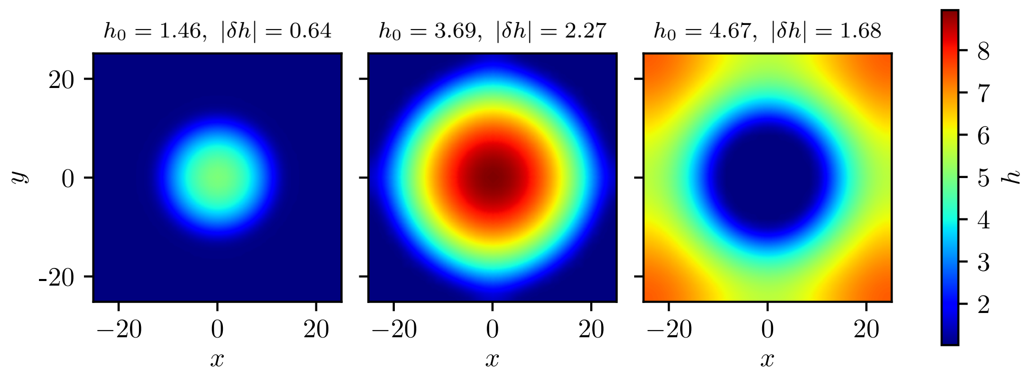

Figure 15: The $Q1$ solution for different control parameters h_0. The left and the middle panel show droplets of different sizes, while the right panel displays a hole solution.

Fig. 15 shows three different $Q1$ solutions and the corresponding values of h_0 and $|\delta h|$. h_0 is the average film height and is used in the bifurcation diagram in Fig. 19 as the control parameter. $|\delta h| = \frac{1}{D}\int_D |h(\vec{x}) - h_0|\mathrm{d}\vec{x}$ corresponds to the absolute deviation (see equation (5.38)) and is used as the solution measure. The $Q1$ solution occurs in form of a droplet or a hole.
Similar to the $Q1$ solution, the stripe solution occurs in two different forms as well. The stripe either appears as an elevated ridge or a depressed trench. Fig. 16 displays three different solutions of the type $S1$. Note that the system is invariant with respect to $\pi/2$-rotations and therefore also supports stripe solutions parallel to the x-axis.

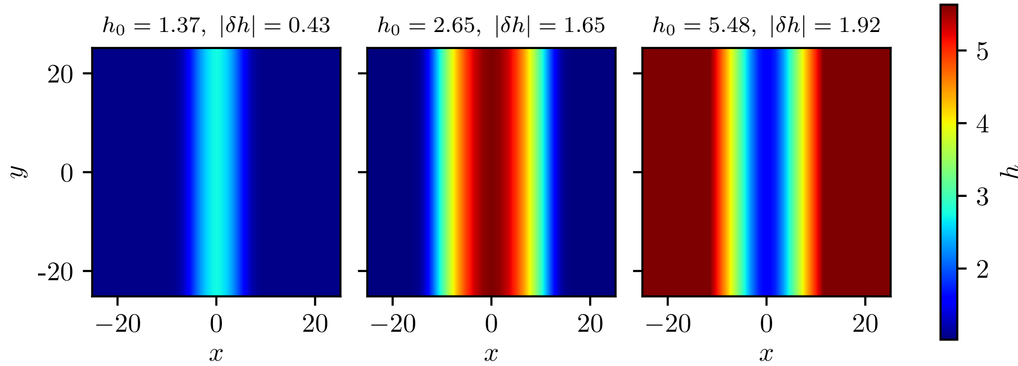

Figure 16: The $S1$ solution for different control parameters h_0. The left and the middle panel show single ridges of different sizes, while the right panel displays a trench. This behaviour is similar to Fig. 15.

The system has steady, but unstable solution branches for two droplets (or holes) as well. They are displayed in Fig. 17.

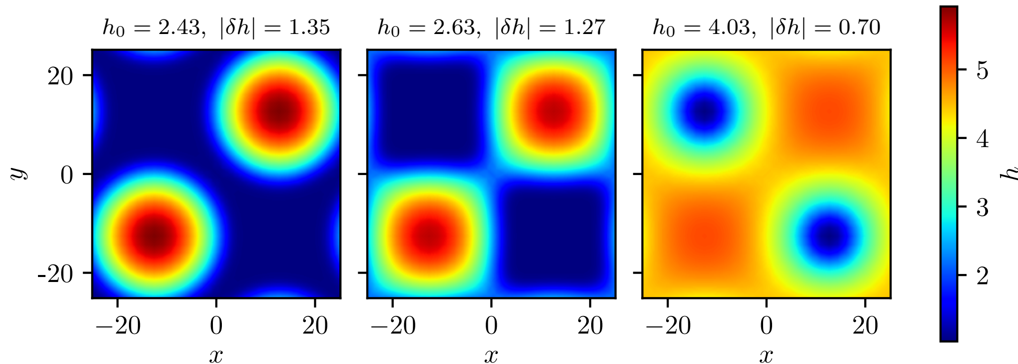

Figure 17: The $Q2$ solution for different control parameters h_0. While the left and middle panel display two droplets, the right panel shows two holes.

It also has a stationary, but unstable solution for four droplets and two stripes. They are shown in Fig. 18.

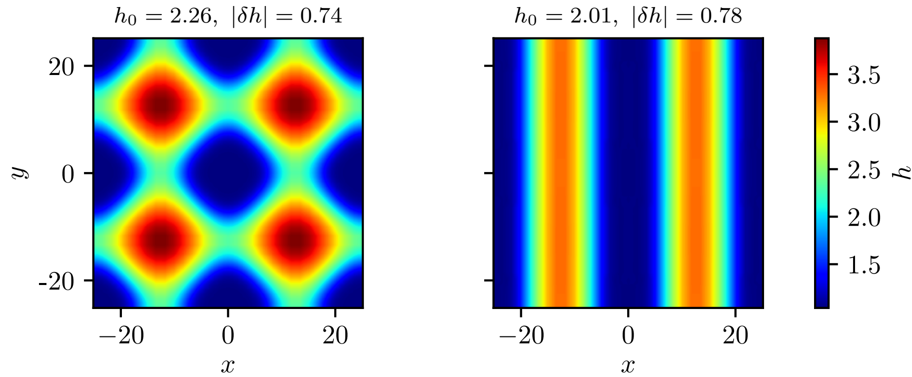

Figure 18: The left panel shows the four-droplet solution $Q4$, while the right panel displays a two stripe solution $S2$.

Figures 15-18 show solutions of the system exemplarily. Using the continuation routines discussed earlier, we create a bifurcation diagram (see Fig. 19). It displays the solution branches of the $Q1$, $Q2$, $Q4$, $S1$ and $S2$ solution, as well as an unstable branch connecting the $Q1$ and $S1$ branch.

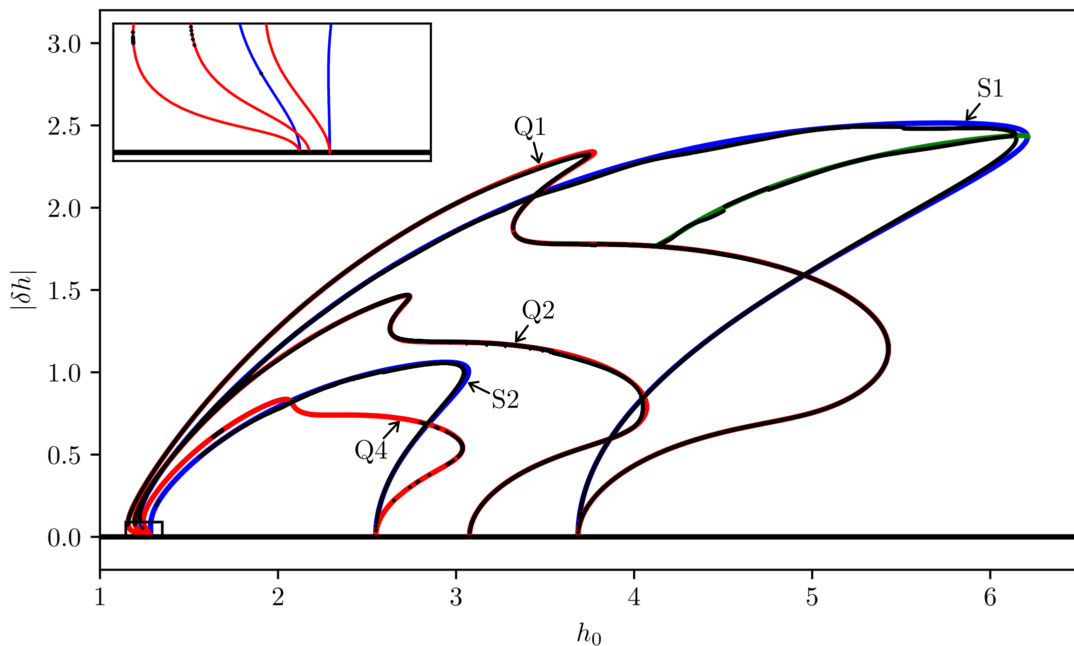

Figure 19: Bifurcation diagram for stationary solutions of the thin-film equation. The black points are obtained by the continuation routines in the `oomph-lib` framework. The underlying red, blue and green lines represent results from Ref. [23] using the `pde2path` package for Matlab. Small jumps along the branches can be attributed to mesh refinement between continuation steps. The zoom in the top left corner of the figure shows pitchfork bifurcations at small h_0. A $[0, 8\pi] \times [0, 8\pi]$ domain is used with Neumann boundary conditions.

Fig. 19 compares the results of the continuation routines using the `oomph-lib` framework with the ones obtained in Ref. [23] using the `pde2path` package for Matlab. Both methods are in quite good agreement.

6 Summary and outlook

In this thesis we have investigated the spreading of simple and complex fluids. To do so, we have used the `oomph-lib` framework to employ finite element methods for the numerical calculation of solutions of the mathematical models.
In the first part of the thesis, we have discussed the mathematical concepts, which underlie the finite element methods. We have introduced the weak solution concept for PDEs, as well as the weighted residual. Using the Galerkin method, we have derived a numerical algorithm to find solutions of the PDEs, as well as calculating their temporal evolution. To increase the efficiency of the calculations, we have introduced the concept of finite elements, which ensure a sparse Jacobian matrix and short computation times.
Subsequently, we have presented a general two-field model for hydrodynamical thin-film systems of simple and complex liquids. It is a general description for a wide range of hydrodynamical systems with two order parameter fields. The model can be used to describe two-layer films or complex fluids including a surfactant or solute concentration. We have developed a finite element method for the numerical calculation of time evolutions for the general two-field model.
Following the object-oriented programming principles, we have implemented a user-friendly library for systems, that can be described in the context of the general two-field model. Implementing adaptive time steps and mesh refinement into the algorithms further increases the numerical efficiency. Hereby, we have discussed the code separation promoted by the `oomph-lib` framework in detail and incorporated it in our library. It has enabled us to implement the general two-field model independent of its spatial dimension. The classes and methods representing the general two-field model have been combined in the `general2field` library.
In the second part of the thesis, we have considered a single layer film covered by an insoluble surfactant. As a first application of the developed method, we have described the system in terms of the general two-field model and we have used the `general2field` library to study it. Surfactant gradients introduce surface tension gradients into the system and result in Marangoni flows. We have studied the effect of different surfactant concentrations on the coalescence behaviour of neighbouring droplets in 1D. Due to the induced Marangoni flows the coalescence of droplets is delayed up to two orders of magnitude. We have determined the initial contact angle of the droplets as another important parameter, influencing the coalescence speed of two droplets. Using the numerically efficient algorithms of the `general2field` library, we have performed a parameter scan of the system varying the contact angle and the ratio between the entropic energetic contribution of the surfactant and the interfacial tension without surfactant.
In the third part, we have investigated the thin-film equation for a partially wetting simple liquid. We have presented the partial differential equations describing the system and have derived a finite element method. After the mathematical foundations have been established, we have used the `oomph-lib` framework for the numerical calculation. We have exploited the object-oriented nature of the library to introduce an integral auxiliary constraint. Using this constraint we have performed continuation routines and created a bifurcation diagram of our two-dimensional system. We have been able to simulate the temporal evolution of the system using the same code.
Summarising the work of this thesis, we have introduced a general two-field model and have developed a finite element method for the numerical calculation of it. Obeying object-oriented coding conventions, we have developed the `general2field` library, which is an user-friendly library to model any system that can be described in the context of the general two-field model. Using this library, we have studied the coalescence behaviour of droplets covered by a concentration of insoluble surfactant in a one-dimensional system. We have showed, that the Marangoni flows induced by the surfactant concentration gradients greatly influence the coalescence behaviour of two neighbouring droplets.
Furthermore, we have developed an efficient numerical scheme for the calculation of the thin-film equation using the `oomph-lib` framework. Due to the object-oriented nature of the library, we have been able to implement a program, which calculates the time evolution of the system, as well as performs continuation routines.
To extend the work of this thesis one could study the coalescence behaviour of the droplets in two spatial dimensions. This may give rise to possibly interesting spreading and coalescence behaviour, since the droplets can spread into more directions. Furthermore, one could try to numerically

simulate fingering instabilities observed in experiments in Ref. [18]. The calculations can be carried out using the `general2field` library. Additionally, one could introduce a general implementation of an integral auxiliary condition to the `general2field` library, to enable the use of continuation routines for the general two-field model.

A Appendix

A.1 general2field library UML class diagram

Fig. 20 and Fig. 21 show an UML class diagram representing the general2field library for the oomph-lib framework, described in section 3.2.

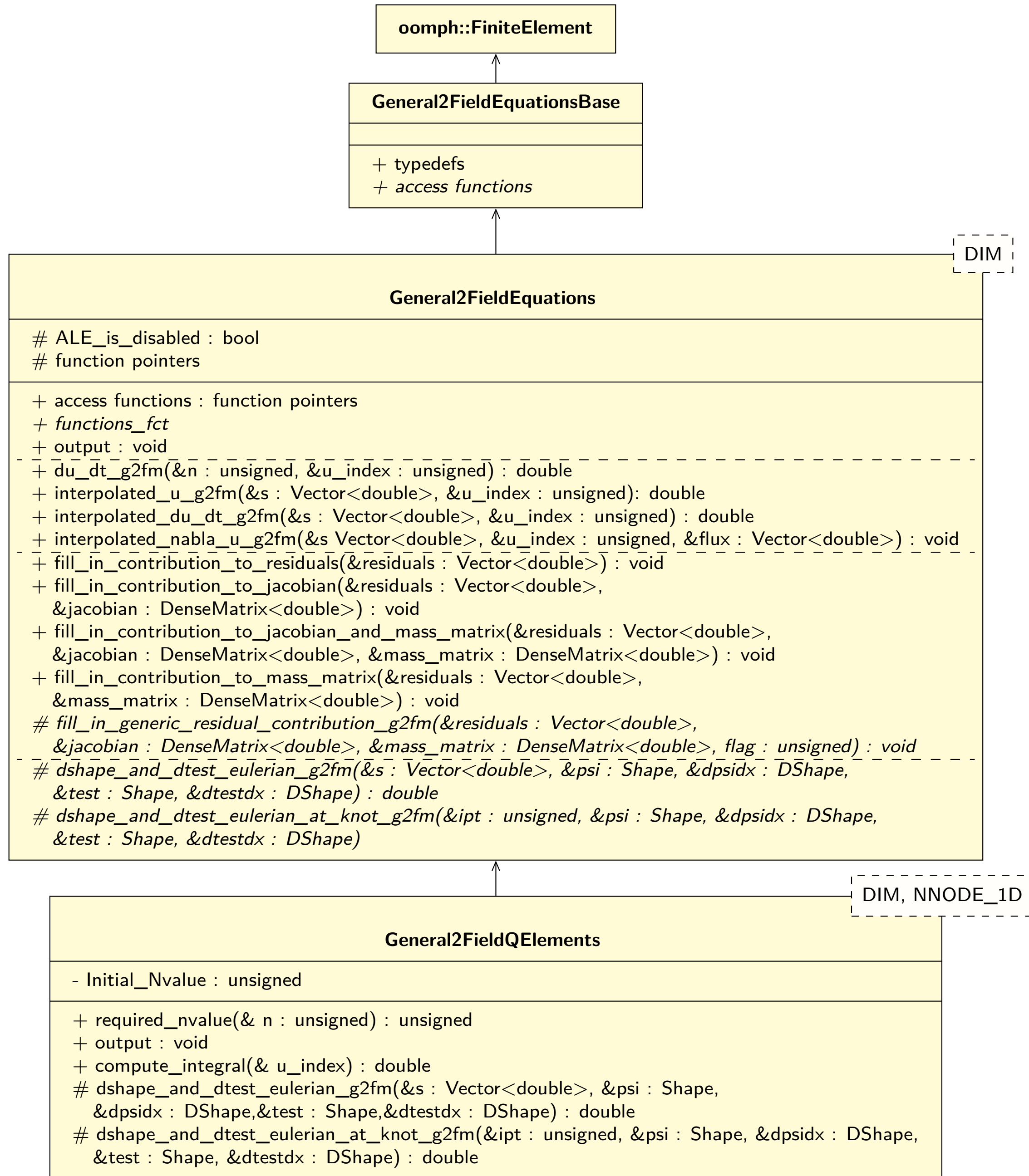

Figure 20: UML class diagram of the general two-field model library general2field in the oomph-lib framework displaying the structure of the General2FieldEquationsBase, General2FieldEquations and General2FieldQElements class. See Fig. 21 for second part of the class diagram.

Fig. 20 shows that the General2FieldEquations class only depends on the spatial dimension template parameter DIM, but not on the NNODE_1D parameter. Therefore, it is implemented independent of the geometric layout of the elements. The NNODE_1D template parameter is only introduced in its child class General2FieldQElements.

Fig. 21 displays the second part of the UML class diagram representing the `general2field` library. It shows the structure of the classes used for mesh refinement `RefineableGeneral2FieldEquations` and `RefineableGeneral2FieldQElement` as well as there inheritance hierarchy. Similar to the structure presented in Fig. 20, it also consists of a class, which is independent of the node geometry and only depends on the spatial dimension. The main purpose of the `RefineableGeneral2FieldEquations` is to calculate the residual vector and the Jacobian matrix for a refineable mesh, which includes hanging and master nodes.

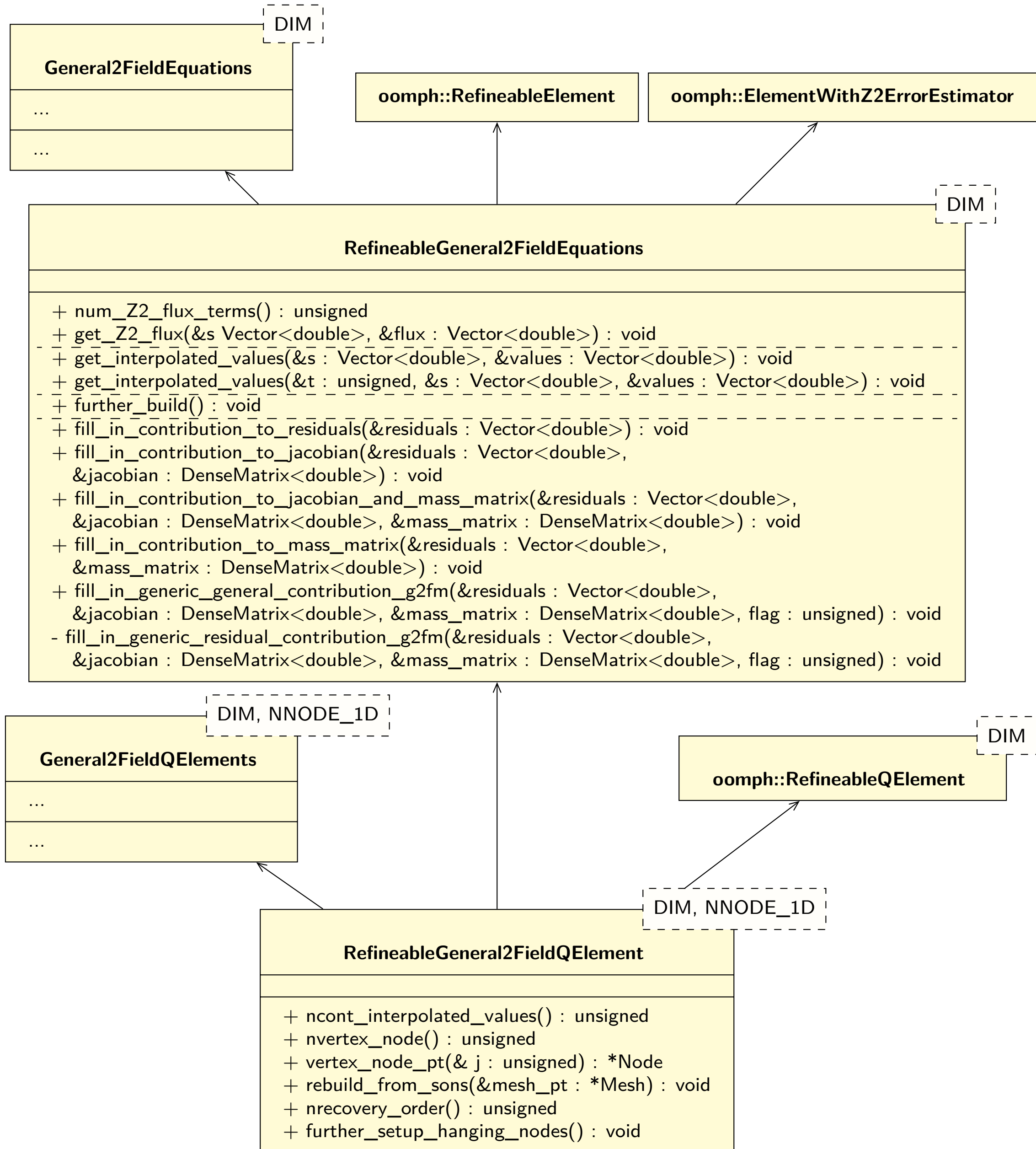

Figure 21: UML class diagram of the general two-field model library `general2field` in the `oomph-lib` framework. See Fig. 20 for first part of class diagram.

In Fig. 20 and subsequent figures displaying UML class diagrams, individual C++ classes are represented using yellow colourized boxes. These boxes are split into three vertically aligned sections, where the first one shows the class name, while the second lists the class' attributes and the third its methods. Class names with the `oomph::` prefix represent classes of the `oomph-lib` framework and are therfore only referenced by their name. `Virtual` member declarations are

written in italics. The “+” symbol denotes `public` members, while “#” is used for `protected` and “-” for `public` members. A colon seperates the method declaration, from its return type. Method declarations without arguments represent overloaded methods with the same functionality. Template parameters of the class are represented by their name framed with dashed lines in the top right corner. The horizontal dashed lines in the method section are a guide to the eye, which is used to group methods with similar functionality. The child class points to its parent classes using a solid arrow to show the inheritance hierarchy.

A.2 UML class diagram for the implementation of the thin-film equation

Fig. 22 shows an UML class diagram, which represents the implementation of the thin-film equation discussed in section 5.2. In contrast to the UML diagram displayed in section A.1 this implementation follows a different approach. Instead of creating a library for the calculation of a model system starting from the `oomph-lib` base classes, this approach uses the already existing implementation of the unsteady heat equation and overloads specific methods.

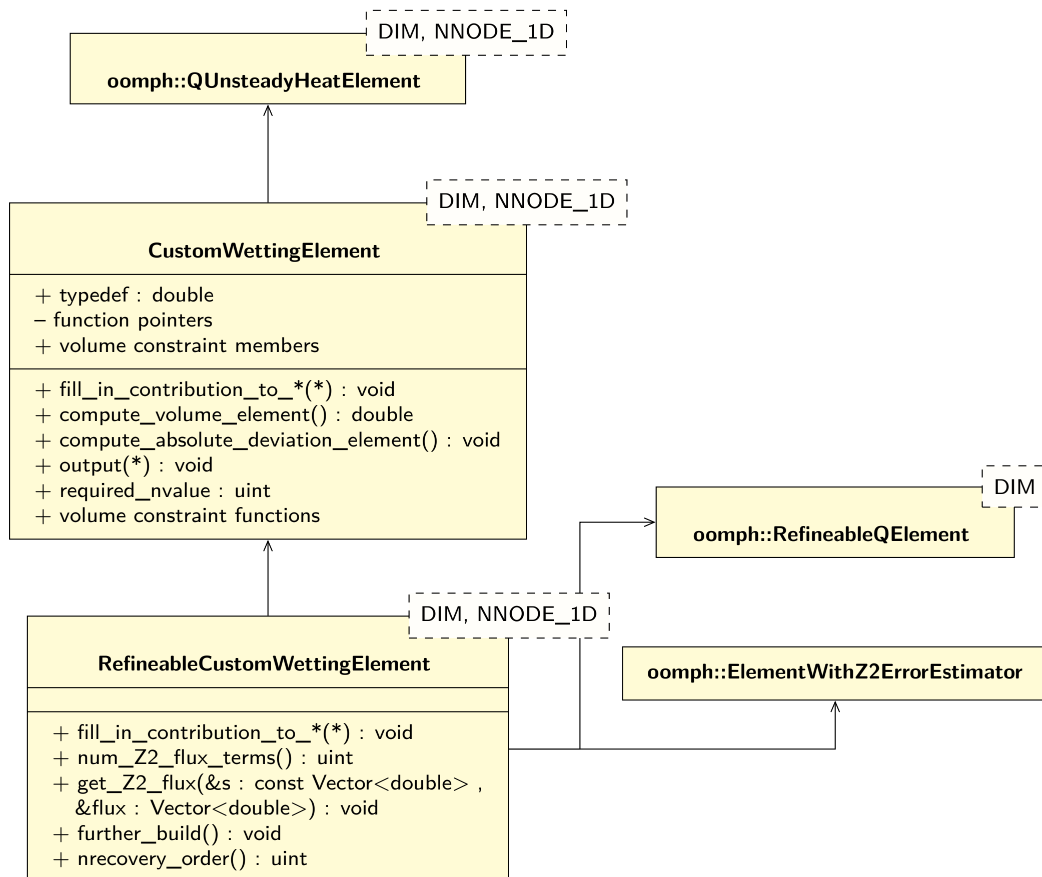

Figure 22: Class diagram of the elements used to calculate the thin-film equation using FEM. The methods for the calculation of the residual vector and Jacobian matrix, as well as other methods of the `oomph::QUnsteadyHeatElement` class are overloaded in the `CustomWettingElement` class.

Fig. 23 displays the UML class diagram for the `VolumeConstraintElement` class. Since it represents an element without a designated spatial coordinate, it inherits from the `oomph::GeneralisedElement` element class instead of the `oomph::FiniteElement` class. Nonetheless, it needs to overload the methods to calculate its contribution to the residual vector and the Jacobian matrix. This class is necessary for the implementation of an integral auxiliary constraint (e.g. volume constraint).
Fig. 24 shows the UML diagram for the `UnsteadyHeatProblem` class, which has been overwritten to solve the thin-film equation. The class stores pointers to initialised objects of the classes shown in Fig. 22 and Fig. 23. A detailed description of the class can be found in section 5.4

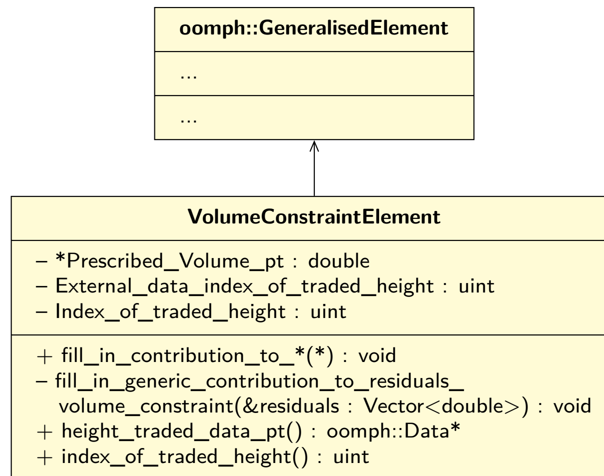

Figure 23: UML class diagram for the `VolumeConstraintElement` class.

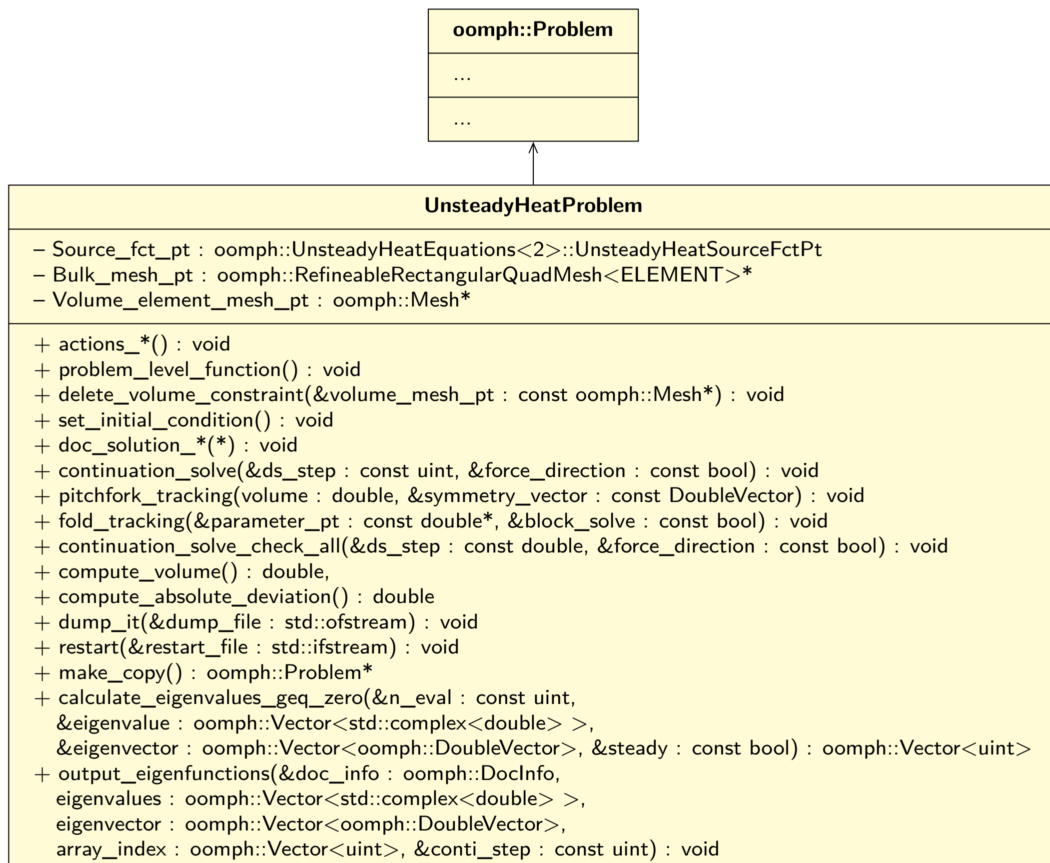

Figure 24: Class diagram of the problem class to calculate the thin-film equation. The problem class governs the calculation process and is part of every driver code.

A.3 The (not-so-)quick users guide

This section focuses on the solution of the "(Not-So-)Quick Users Guide" for the `oomph-lib` library. It can be found under `http://oomph-lib.maths.man.ac.uk/doc/quick_guide/html/index.html`.
A solution to the exercises of the guide can be found in the file `one_d_V7_unc.cc`. Some solutions of the first exercises might have been overwritten by subsequent tasks, but we will comment on this as much as possible.

A.3.1 How to build a `Problem`

A.3.1.1 Outputting the exact solution

The following code is nested in the `actions_after_newton_solve()` method. The "exact" solution is written into the output file with a resolution of 100 sampling points.

```
779 void actions_after_newton_solve()
780 {
781  /// output numerical FEM solution
782  ofstream file("result.dat");
783 
784  mesh_pt()->output(file);
785  file.close();
786 
787  /// output analytical solution
788  ofstream file1("exact.dat");
789  unsigned n_max = 100;
790  for (unsigned n = 0; n <= n_max; n++)
791  {
792   file1 << double(n) / double(n_max) << " " << double(Sign) * ((sin(sqrt(30.0)) -
       1.0) * double(n) / double(n_max) - sin(sqrt(30.0) * double(n) / double(n_max
      ))) << std::endl;
793  }
794  file1.close();
795 }
```

Listing 49: Modified version of the `actions_after_newton_solve()` method, which outputs the analytical solution into the file `exact.dat`. The output is formatted to output the x coordinate first and the exact solution second separated by a blank. Each coordinate is written into a new line of the file.[70]

A.3.1.2 Setting the number of elements in a mesh

The number of elements is passed to the mesh constructor as an argument. In the example solution shown in Lst. 50, the number of elements is set to 10.

```
691 //     Problem::mesh_pt() = new OneDimMesh<TwoNodePoissonElement>(10);
```

Listing 50: Setting the number of elements of the `OneDimMesh` to 10. The line is commented out due to a subsequent exercise.[71]

A.3.1.3 Adjusting the boundary conditions (Dirichlet)

In Lst. 51 we introduce Dirichlet boundary conditions to the system. Therefore, we use the method `boundary_node_pt(const unsigned &b, const unsigned &n)`, which returns a pointer to the node `n` on boundary `b`. The method `pin(const unsigned &i)` marks the `i`-th value of the node as pinned, which means that it has a fixed value. All nodes (the boundary nodes as well) are initialised with the data value 0. To change the value of a node we can use the `set_value(const unsigned &i, const double &value)` method. It sets the `i`-th value of the node to `value`.

[70] File: `../oomph_Codes/user_drivers/Latex_NotSoQuickGuide/one_d_V7_unc.cc`
[71] File: `../oomph_Codes/user_drivers/Latex_NotSoQuickGuide/one_d_V7_unc.cc`

```
706  //Pin the single nodal value at the single node on mesh
707  //boundary 0 (= the left domain boundary at x=0)
708  mesh_pt()->boundary_node_pt(0, 0)->pin(0);
709
710  //Pin the single nodal value at the single node on mesh
711  //boundary 1 (= the right domain boundary at x=1)
712  mesh_pt()->boundary_node_pt(1, 0)->pin(0);
713
714  // All values are initialised to zero. This is consistent with the
715  // boundary condition at x=0 and no further action is required
716  // at that node.
717
718  // Apply the boundary condition at x=1: u(x=1)=-/+1
719  mesh_pt()->boundary_node_pt(1, 0)->set_value(0, -double(Sign));
720  //     mesh_pt()->boundary_node_pt(0,0)->set_value(0,-double(Sign));
```

Listing 51: The code excerpt “pins” the 0-th value of the boundary nodes. Since the considered nodes only store the value for the field u, it is referred to by the 0-th value. At last, we set the value of the node at $x = 1$ to $-$`Sign`. Pinning a value set its equation number to -1 and it will be excluded during the assembly of the residual vector and Jacobian matrix.

When we do not apply a Dirichlet boundary condition on one of the boundaries (i.e. pin the value), we automatically enforce a Neumann boundary condition. This is a result of the integration by parts step during the calculation of the weak form of the problem. By pinning the value on the boundary we demand from the test function $\phi^{test}(x)$ to vanish on that domain boundary, to ensure that the whole integral over the domain boundary vanishes. When the value is not pinned we do not impose this restriction on the test function, but the integral over the domain boundary still vanishes. Therefore, the first spatial derivative of the solution $\frac{\partial u(x)}{\partial x}|_{\partial D}$ must vanish. This means that we have applied a Neumann boundary condition. The Neumann boundary condition is also called natural boundary condition since it is implicitly satisfied by the formulation of the problem.

A.3.1.4 Assignment of equation numbers

It is essential to assign the equation numbers before trying to solve the problem. Since the Jacobian and the residuals are calculated element wise, the equation numbers are needed to assemble the global Jacobian matrix and residual vector. For the assembly of the residual vector and Jacobian matrix only positive equation numbers are considered. Consequently, pinned values are excluded. The assignment of the equation numbers needs to be redone after applying the initial conditions, after changing the mesh's resolution or when introducing an integral constraint.

```
754  cout << "Number of equations is " << assign_eqn_numbers() << std::endl;
```

Listing 52: Assigning the equation numbers by calling the `assign_eqn_numbers()` method, which returns the amount of equation numbers (the amount of degrees of freedom) in the system.[72]

A.3.2 How to build a `Mesh`

A.3.2.1 Modifying the `OneDimMesh`

The tricky part of this exercise is, that we need to move the elements, not just the nodes, according to the scheme. In the example on the website, the elements are disregarded and the loop just covers all nodes. This will work in most cases, but it is not very general, since it can interfere with the code if one changes the amount of nodes per element later. The problem is that we cannot simply assign new coordinates to the elements. We can only set them to the nodes of the elements. Additionally, we have to keep in mind that in 1D elements each boundary node belongs to two elements.

```
608 template<class ELEMENT>
609 class SpecialOneDimMesh : public OneDimMesh<ELEMENT>
```

[71] File: `../oomph_Codes/user_drivers/Latex_NotSoQuickGuide/one_d_V7_unc.cc`
[72] File: `../oomph_Codes/user_drivers/Latex_NotSoQuickGuide/one_d_V7_unc.cc`

```
610 {
611 public:
612  SpecialOneDimMesh(const unsigned& N_0, const unsigned& N_1, const double& x_tilde
       ) : OneDimMesh<ELEMENT>(N_0 + N_1)
613  {
614   unsigned n_node = OneDimMesh<ELEMENT>::nnode();
615   unsigned n_node_per_elemnt = OneDimMesh<ELEMENT>::finite_element_pt(0)->nnode();
616 
617   cout << "Number of nodes: " << n_node << std::endl;
618   cout << "Nodes per element: " << n_node_per_elemnt << std::endl;
619 
620   /// Count for all nodes in Mesh
621   unsigned n_count = 0;
622 
623   /// Node 0 does not need to be moved, but n_count needs to be incremented one
        time (because of inner for loop)
624   Node* nod_pt = OneDimMesh<ELEMENT>::node_pt(n_count);
625   //   double x_old = nod_pt->x(0);
626 
627   double x_new = double(n_count) * x_tilde / double(N_0 * n_node_per_elemnt);
628   nod_pt->x(0) = x_new;
629   //   cout << "___Part 1, Node: " << n_count << "  ,old: " << x_old << "  ,new: "
         << nod_pt->x(0) << std::endl;
630   n_count++;
631 
632   // loops over the first section of elements, which are evenly spaced between 0
        and x_tilde
633   for (unsigned e = 0; e < N_0; e++)
634   {
635    // increments over all nodes of the element but the first node
636    // All elements first nodes are accessed and moved via the last node of the
         previous element
637    for (unsigned i = 1; i < n_node_per_elemnt; i++)
638    {
639     Node* nod_pt = OneDimMesh<ELEMENT>::node_pt(n_count);
640     //      double x_old = nod_pt->x(0);
641     double x_new = double(n_count) * x_tilde / double(N_0 * (n_node_per_elemnt -
          1));
642     nod_pt->x(0) = x_new;
643     //      cout << "Part 1, Node: " << n_count << "  ,old: " << x_old << "  ,new:
          " << nod_pt->x(0) << std::endl;
644     n_count++;
645    }
646   }
647   //   some index shennanigans...
648   unsigned nodes_N1 = n_count - 1;
649   n_count = 1;
650 
651   /// loops over the second section of elements, which are evenly spaced between
        x_tilde and 1
652   for (unsigned e = 0; e < N_1; e++)
653   {
654    // increments over all nodes of the element but the first node
655    for (unsigned i = 1; i < n_node_per_elemnt; i++)
656    {
657     Node* nod_pt = OneDimMesh<ELEMENT>::node_pt(n_count + nodes_N1);
658     //      double x_old = nod_pt->x(0);
659     double x_new = double(n_count) * (1.0 - x_tilde) / double(N_1 * (
          n_node_per_elemnt - 1));
660     nod_pt->x(0) = x_new + x_tilde;
661     //      cout << "Part 2, Node: " << n_count << "  ,old: " << x_old << " x_new:
          " << x_new << "  ,new pos: " << nod_pt->x(0) << std::endl;
662     n_count++;
663    }
664   }
665  }
666 };
```

Listing 53: Creating a one dimensional mesh of the domain $x \in [0,1]$ with `N_0` elements in $[0,\tilde{x}]$ and `N_1` elements in $[\tilde{x},1]$.[73]

A.3.3 How to build a `FiniteElement`

A.3.3.1 Writing: `get_generic_residual_contribution(...)`

The solution can be found in the source code. We will not comment further on this, since it is a more or less straight up copy and paste coding exercise.

A.3.3.2 Improved implementation of the source function

The understanding of the solution to this exercise is rather important, because it explains a very powerful mechanism of C++ and is used in all codes discussed in this thesis.
The following lines of code basically follow the instructions in the task (copy & paste).
First of all, we define a typedef in `TwoNodePoissonElement`. Here, we need to make sure that the definition of the typedef of our function gets the same arguments, as the actual function we want to implement later (in this case just the x-value). We also create an access function, which lets us access the instance `Source_fct_pt` of the typedef we just introduced.

```
217 typedef double (*PoissonSourceFctPt)(const double& x);
218 /// access function for Source_fct_pt
219 PoissonSourceFctPt& source_fct_pt()
220 {
221  return Source_fct_pt;
222 }
```

Listing 54: Defining a `typedef` for the function pointer, as well as providing an access method for the `private` attribute `Source_fct_pt`.[74]

We will add the typedef `Source_fct_pt` to the class definition as a private member. This justifies the access function we implemented beforehand.

```
459 PoissonSourceFctPt Source_fct_pt;
```

Listing 55: Declaring the `private` member attribute of the type `PoissonSourceFctPt`.[75]

Now we can introduce the source function as a member method in `TwoNodePoissonElement`. It returns 0 as long as the function pointer `Source_fct_pt` is not initialised. As soon as we initialise the function pointer, this method will return the corresponding value.

```
226 double f(const double &x) const
227 {
228  double source = 0.0;
229
230  if (Source_fct_pt != 0)
231  {
232   source = (*Source_fct_pt)(x);
233  }
234  return source;
235 }
```

Listing 56: Declaring and defining the method `f(const double &x)`, which represents the source function $f(x)$. If the function pointer is not set, it returns 0.[76]

The actual definition of the source function can be found in the namespace `FishSolnOneDPoisson`. By placing the method in this extra namespace, it is very easy for the user to change the function, without redoing all the maths. It greatly increases the generality of our program.

```
53 double source_function(const double& x)
54 {
55  double number = 30.0;
56
57  return double(Sign) * number * sin(sqrt(number) * x);
```

[73] File: `../oomph_Codes/user_drivers/Latex_NotSoQuickGuide/one_d_V7_unc.cc`
[74] File: `../oomph_Codes/user_drivers/Latex_NotSoQuickGuide/one_d_V7_unc.cc`
[75] File: `../oomph_Codes/user_drivers/Latex_NotSoQuickGuide/one_d_V7_unc.cc`
[76] File: `../oomph_Codes/user_drivers/Latex_NotSoQuickGuide/one_d_V7_unc.cc`

```
58 || }
```

Listing 57: Defining the source function as $f(x) = \texttt{Sign} \cdot 30 \cdot \sin(\sqrt{30}x)$.[77]

Keep in mind that we have to set the function pointers of the elements to the source function, which we defined in `FishSolnOneDPoisson`. This is done in the problem constructor, nested in a loop over all elements.

```
745 ||    specific_element_pt->source_fct_pt() = FishSolnOneDPoisson::source_function;
```

Listing 58: Setting the function pointer `Source_fct_pt` to point to the method defined in Lst. 57.[78]

A.3.3.3 Generalise to TwoNodeSelfAdjointElement

The task is to generalise the `TwoNodePoissonElement` to the `TwoNodeSelfAdjointElement`, which implements the isoparametric discretisation of the self-adjoint ODE

$$\frac{\mathrm{d}}{\mathrm{dx}}\left(a(x)\frac{\mathrm{d}u}{\mathrm{dx}}\right) + b(x)u(x) = f(x). \tag{A.1}$$

This a good exercise to understand the mathematics underlying the Finite-Element method (have a look at `http://oomph-lib.maths.man.ac.uk/doc/intro/html/index.html`).
We need to derive the weak form of the problem.
Calculating the “weighted residual”

$$r = \int_D R(x; u_w(x))\phi^{(test)}(x)\mathrm{d}x \tag{A.2}$$

with R being the PDE in “residual form”.
For the self-adjoint ODE the “weighted residual” is

$$\begin{aligned} r &= \int_D \left[\frac{\mathrm{d}}{\mathrm{dx}}\left(a(x)\frac{\mathrm{d}u}{\mathrm{dx}}\right) + b(x)u(x) - f(x)\right]\phi^{(test)}(x)\mathrm{d}x && \text{(A.3)} \\ &= \left[\frac{\mathrm{d}a(x)}{\mathrm{dx}}\frac{\mathrm{d}u}{\mathrm{dx}} + a(x)\frac{\mathrm{d}^2u(x)}{\mathrm{dx}^2} + b(x)u(x) - f(x)\right]\phi^{(test)}(x)\mathrm{d}x && \text{(A.4)} \end{aligned}$$

Applying integration by parts to the term that includes the second derivative of $u(x)$

$$\int_D \phi^{(test)}(x)a(x)\frac{\mathrm{d}^2u(x)}{\mathrm{dx}^2}\mathrm{dx} = \left[\phi^{(test)}(x)a(x)\frac{\mathrm{d}u(x)}{\mathrm{dx}}\right]_0^1 - \int_D \frac{\mathrm{d}u(x)}{\mathrm{dx}}\frac{\mathrm{d}}{\mathrm{dx}}\left(\phi^{(test)}(x)a(x)\right)\mathrm{d}x. \tag{A.5}$$

Since $\phi^{(test)}(x)$ vanishes at the edges of D (in this case at $x = 0$ and $x = 1$)

$$\left[\phi^{(test)}(x)a(x)\frac{\mathrm{d}u(x)}{\mathrm{dx}}\right]_0^1 = 0 \tag{A.6}$$

and it follows

$$r = \int_D a(x)\frac{\mathrm{d}u(x)}{\mathrm{dx}}\frac{\mathrm{d}\phi^{(test)}(x)}{\mathrm{dx}} + \phi^{(test)}(x)\left(f(x) - b(x)u(x)\right)\mathrm{d}x. \tag{A.7}$$

Having obtained the “weighted residual” we can analytically calculate the Jacobian matrix. Therefore we consider the discrete version of r. Using the discrete version r_k we can now easily obtain the Jacobian matrix J_{kj}

$$J_{kj} = \frac{\partial r_k}{\partial U_j}|_{(U_1,\ldots,U_M)} \quad \text{for} \quad j, k = 1, ..., M \tag{A.8}$$

$$J_{kj} = \int_0^1 a(x)\frac{\mathrm{d}\psi_j(x)}{\mathrm{dx}}\frac{\mathrm{d}\psi_k(x)}{\mathrm{dx}} - b(x)\psi_k(x)\psi_j(x)\mathrm{d}x. \tag{A.9}$$

[77] File: `../oomph_Codes/user_drivers/Latex_NotSoQuickGuide/one_d_V7_unc.cc`
[78] File: `../oomph_Codes/user_drivers/Latex_NotSoQuickGuide/one_d_V7_unc.cc`

The implementation of the residual vector and the Jacobian matrix is done in the `get_generic_residual_contribution(...)` method. See the commented lines for the implementation of the self-adjoint ODE.

```
void get_generic_residual_contribution(Vector<double>& residuals, DenseMatrix<double>& jacobian, bool flag)
{
 unsigned n_dof = GeometricLineElement<NNODE>::ndof();

 /// set jacobian and residuals to zero
 for (unsigned i = 0; i < n_dof; i++)
 {
  residuals[i] = 0.0;
 }
 if (flag)
 {
  for (unsigned i = 0; i < n_dof; i++)
  {
   for (unsigned j = 0; j < n_dof; j++)
   {
    jacobian(i, j) = 0.0;
   }
  }
 }

 unsigned n_node = GeometricLineElement<NNODE>::nnode();

 Shape psi(n_node);
 DShape dpsidx(n_node, 1);
 Vector<double> s(1);

 unsigned n_intpt = GeometricLineElement<NNODE>::integral_pt()->nweight();
 for (unsigned ipt = 0; ipt < n_intpt; ipt++)
 {
  s[0] = GeometricLineElement<NNODE>::integral_pt()->knot(ipt, 0);
  double w = GeometricLineElement<NNODE>::integral_pt()->weight(ipt);
  double J = GeometricLineElement<NNODE>::dshape_eulerian(s, psi, dpsidx);
  double W = w * J;
  double interpolated_x = 0.0, interpolated_u = 0.0, interpolated_dudx = 0.0;
  for (unsigned n = 0; n < n_node; n++)
  {
   interpolated_x += GeometricLineElement<NNODE>::nodal_position(n, 0) * psi[n];
   interpolated_u += u(n) * psi[n];
   interpolated_dudx += u(n) * dpsidx(n, 0);
  }

  //// calculate source function and a(x) and b(x)
  double source = f(interpolated_x);
  //   double func_val_a=a(interpolated_x);
  //   double func_val_b=b(interpolated_x);

  for (unsigned l = 0; l < n_node; l++)
  {
   int local_eqn_number = GeometricLineElement<NNODE>::nodal_local_eqn(l, 0);
   if (local_eqn_number >= 0)
   {
    residuals[local_eqn_number] += source * psi[l] * W;
    residuals[local_eqn_number] += interpolated_dudx * dpsidx(l, 0) * W;
    //// residuals of self adjoint ODE
    //     residuals[local_eqn_number] += (source-func_val_b*interpolated_u)*psi[l]*W;
    //     residuals[local_eqn_number] += func_val_a*interpolated_dudx*dpsidx(l,0)*W;
   }
  }

  //// calculate jacobian as well when flag==True
  if (flag)
  {
```

```
364      for (unsigned l = 0; l < n_node; l++)
365      {
366       int local_eqn_number = GeometricLineElement<NNODE>::nodal_local_eqn(l, 0);
367       if (local_eqn_number >= 0)
368       {
369        for (unsigned l2 = 0; l2 < n_node; l2++)
370        {
371         int local_dof_number = GeometricLineElement<NNODE>::nodal_local_eqn(l2, 0);
372         if (local_dof_number >= 0)
373         {
374          jacobian(local_eqn_number, local_dof_number) += dpsidx(l2, 0) * dpsidx(l,
             0) * W;
375          //// jacobian of self adjoint ODE
376          //         jacobian(local_eqn_number, local_dof_number) += (func_val_a*
             dpsidx(l2,0)*dpsidx(l,0) - func_val_b*psi[l]*psi[l2])*W;
377         }
378        }
379       }
380      }
381     }
382    }
383   }
```

Listing 59: Implementation of the `get_generic_residual_contribution` method, which uses pass-by-reference to return the results. The calculation of the self adjoint ODE can be found in the commented lines, due to a later exercise.[79]

A.3.4 How to build a new geometric element

A.3.4.1 ThreeNodeGeometricElement

First, we switch the argument of the `this->set_n_node(2)` method to 3. This is the only change that needs to be done in the constructor. Also you need to change the return value of the `nnode_1d()` function to 3.
Additionally, we alter the `void shape(const Vector<double> &s, Shape &psi)` and the `void dshape_local(const Vector<double> &s, Shape &psi, DShape &dpsids)` method. These methods represent the shape functions of the nodes expressed in the local coordinate system s of the parent element. In the elements the nodes are equally spaced in $s \in [-1, 1]$, e.g. an element with 3 nodes has nodes at the positions $s = -1, 0, 1$. Now we need to make sure that we have exactly as many `psi[i]` functions as you have nodes. Every function `psi[i]` has to fulfil the following conditions: $\psi_i(s_i) = 1$ and $\psi_i(s_j) = 0$ with $i \neq j$. As soon as the `psi[i]` functions are implemented, we calculate their derivatives in the `dshape_local(const Vector<double> &s, Shape &psi, DShape &dpsids)` method. Due to a subsequent exercise the implementation of this has been overwritten. The code excerpt implementing the shape functions mentioned above can be found in Lst. 63, considering the case of 3 node element.

A.3.4.2 template <unsigned NNODE> class GeometricLineElement

This class is more or less self explanatory. First, we have to define the `GeometricLineElement` class as a template class with the template parameter `<unsigned NNODE>`.

```
96 template <unsigned NNODE>
97 class GeometricLineElement : public FiniteElement
```

Listing 60: Declaration of the `GeometricLineElement` class, which inherits from `FiniteElement`.[80]

The constructor depends on the template parameter and sets the amount of nodes in one element with the method `set_n_node()`. We also set the spatial dimension of the element, which is always 1 for a line element. At last we set the integration scheme.

[79] File: `../oomph_Codes/user_drivers/Latex_NotSoQuickGuide/one_d_V7_unc.cc`
[80] File: `../oomph_Codes/user_drivers/Latex_NotSoQuickGuide/one_d_V7_unc.cc`

```
106  GeometricLineElement()
107  {
108   this->set_n_node(NNODE);
109   this->set_dimension(1);
110   set_integration_scheme(&Default_spatial_integration_scheme);
111  }
```

Listing 61: Constructor of the `GeometricLineElement` class.[81]

Defining the `Default_integration_scheme` is tricky, because it is static member data in a template. Therefore, it needs to be defined somewhere with a consistent template. In this case we set it to a one dimensional Gauss integration scheme.

```
468 // Define the static spatial integration scheme
469 template<unsigned NNODE>
470 Gauss<1, NNODE> GeometricLineElement<NNODE>::Default_spatial_integration_scheme;
```

Listing 62: Defining the default spatial integration scheme. Usually it is sufficient to use the default integration scheme of `oomph-lib`.[82]

The creation of a geometric element allows us to choose the shape functions. In this class they are defined for elements with 2,3 or 4 nodes.

```
115  void shape(const Vector<double> &s, Shape &psi) const
116  {
117   switch (NNODE)
118   {
119   case 2:
120    psi[0] = 0.5 * (1.0 - s[0]);
121    psi[1] = 0.5 * (1.0 + s[0]);
122    break;
123
124   case 3:
125    psi[0] = 0.5 * (s[0] - 1.0) * (s[0] - 0.0);
126    psi[1] = -1.0 * (s[0] + 1.0) * (s[0] - 1.0);
127    psi[2] = 0.5 * (s[0] - 0.0) * (s[0] + 1);
128    break;
129
130   case 4:
131    psi[0] = -9. / 16. * (s[0] - 1. / 3.) * (s[0] + 1. / 3.) * (s[0] - 1.0);
132    psi[1] = 27. / 16. * (s[0] + 1.0) * (s[0] - 1. / 3.) * (s[0] - 1.0);
133    psi[2] = -27. / 16. * (s[0] + 1.0) * (s[0] + 1. / 3.) * (s[0] - 1.0);
134    psi[3] = 9. / 16. * (s[0] - 1. / 3.) * (s[0] + 1. / 3.) * (s[0] + 1.0);
135    break;
136
137   default:
138    std::ostringstream error_message;
139    error_message << "Shape functions for " << NNODE << " Node Elements have not
       been implemented!!!" << std::endl;
140    throw OomphLibError(error_message.str(), OOMPH_CURRENT_FUNCTION,
       OOMPH_EXCEPTION_LOCATION);
141   }
142  }
```

Listing 63: Defining the shape functions for the `GeometricLineElement` class. The shape functions are implemented for 2,3 and 4 node elements. If elements with a different amount of nodes are implemented, this method will throw an error.[83]

When the shape functions are implemented, we need to make sure to also implement the `dshape_local(...)` method, which returns the spatial derivatives of the shape functions.

```
146  void dshape_local(const Vector<double> &s, Shape &psi, DShape &dpsids) const
147  {
148   shape(s, psi);
```

[81] File: `../oomph_Codes/user_drivers/Latex_NotSoQuickGuide/one_d_V7_unc.cc`
[82] File: `../oomph_Codes/user_drivers/Latex_NotSoQuickGuide/one_d_V7_unc.cc`
[83] File: `../oomph_Codes/user_drivers/Latex_NotSoQuickGuide/one_d_V7_unc.cc`

```
149
150    switch (NNODE)
151    {
152    case 2:
153     dpsids(0, 0) = -0.5;
154     dpsids(1, 0) = 0.5;
155     break;
156
157    case 3:
158     dpsids(0, 0) = -0.5 + s[0];
159     dpsids(1, 0) = -2 * s[0];
160     dpsids(2, 0) = 0.5 + s[0];
161     break;
162
163    case 4:
164     dpsids(0, 0) = 1. / 16. * (-27. * s[0] * s[0] + 18. * s[0] + 1.);
165     dpsids(1, 0) = 9. / 16. * (9. * s[0] * s[0] - 2 * s[0] - 3.);
166     dpsids(2, 0) = -9. / 16. * (9. * s[0] * s[0] + 2 * s[0] - 3.);
167     dpsids(3, 0) = 1. / 16. * (27. * s[0] * s[0] + 18. * s[0] - 1.);
168     break;
169
170    default:
171     std::ostringstream error_message;
172     error_message << "Shape functions for " << NNODE << " Node Elements have not
        been implemented!!!" << std::endl;
173     throw OomphLibError(error_message.str(), OOMPH_CURRENT_FUNCTION,
        OOMPH_EXCEPTION_LOCATION);
174    }
175   }
```

Listing 64: Implementing the derivatives of the shape functions specified in Lst. 63.[84]

A.3.4.3 template <unsigned NNODE> class PoissonLineElement

To create the `PoissonLineElement` class we will use the `TwoNodePoissonElement` class and change its name. It is also necessary to define it as a template class and to make sure that it inherits from the `GeometricLineElement<NNODE>` class.

```
206 template <unsigned NNODE>
207 class PoissonLineElement : public GeometricLineElement<NNODE>
```

Listing 65: Class declaration of the `PoissonLineElement` class inheriting from the `GeometricLineElement` class.[85]

This concludes all the changes that need to be done to the `TwoNodePoissonElement` class.
To make use of the newly introduced template class, we need to change a few lines in the problem constructor. It is important that the mesh uses the new `PoissonLineElement` class.

```
695    const unsigned int nodes_per_element = 4;
696
697    Problem::mesh_pt() = new SpecialOneDimMesh<PoissonLineElement<nodes_per_element>
        >(2, 3, 0.2);
```

Listing 66: Changing the problem constructor to use the `PoissonLineElement` class for the mesh construction.[86]

Finally, we need to change the class name in the loop, which sets the source and other function pointers for the elements.

[84] File: ../oomph_Codes/user_drivers/Latex_NotSoQuickGuide/one_d_V7_unc.cc
[85] File: ../oomph_Codes/user_drivers/Latex_NotSoQuickGuide/one_d_V7_unc.cc
[86] File: ../oomph_Codes/user_drivers/Latex_NotSoQuickGuide/one_d_V7_unc.cc

```
739    PoissonLineElement<nodes_per_element> *specific_element_pt = dynamic_cast<
         PoissonLineElement<nodes_per_element>*>(mesh_pt()->element_pt(i));
```

Listing 67: Upcasting the element pointers `element_pt(i)`, which point to a `GeneralisedElement`, to a pointer on a `PoissonLineElement` object.[87]

[87] File: `../oomph_Codes/user_drivers/Latex_NotSoQuickGuide/one_d_V7_unc.cc`

Bibliography

[1] M. Hanyak, A. A. Darhuber, and M. Ren. Surfactant-induced delay of leveling of inkjet-printed patterns. *J. Appl. Phys.*, 109:074905, 2011.

[2] J. Stringer and B. Derby. Formation and stability of lines produced by inkjet printing. *Langmuir*, 26:10365–10372, 2010.

[3] J. Marra and J. A. M. Huethorst. Physical principles of marangoni drying. *Langmuir*, 7:2748–2755, 1991.

[4] O. K. Matar and R. V. Craster. Models for marangoni drying. *Phys. Fluids*, 13:1869–1883, 2001.

[5] D. L. Hu, B. Chan, and J. W. M. Bush. The hydrodynamics of water strider locomotion. *Nature*, 424:663, 2003.

[6] B. Wallmeyer, S. Trinschek, S. Yigit, U. Thiele, and T. Betz. Collective cell migration in embryogenesis follows the laws of wetting. *Biophys. J.*, 114:213–222, 2018.

[7] A. Oron, S. H. Davis, and S. G. Bankoff. Long-scale evolution of thin liquid films. *Rev. Mod. Phys.*, 69:931–980, 1997.

[8] R. V. Craster and O. K. Matar. Dynamics and stability of thin liquid films. *Rev. Mod. Phys.*, 81:1131–1198, 2009.

[9] J. Thomson. On certain curious motions observable at the surface of wine and other alcoholic liquors. *Phil. Mag. Ser. 4*, 10:330–333, 1855.

[10] C. G. Marangoni. Ueber die Ausbreitung der Tropfen einer Flüssigkeit auf der Oberfläche einer anderen. *Ann. Phys. (Poggendorf)*, 143:337–354, 1871.

[11] C. V. Sternling and L. E. Scriven. Interfacial turbulence: Hydrodynamic instability and the Marangoni effect. *A. I. Ch. E. Journal*, 5:514–523, 1959.

[12] L. E. Scriven and C. V. Sternling. Marangoni effects. *Nature*, 187:186–188, 1960.

[13] M. H. A. van Dongen, A. van Loon, R. J. Vrancken, J. P. C. Bernards, and J. F. Dijksman. Uv-mediated coalescence and mixing of inkjet printed drops. *Exp. Fluids*, 55:1744, 2014.

[14] S. Karpitschka and H. Riegler. Sharp transition between coalescence and non-coalescence of sessile drops. *J. Fluid Mech.*, 743:R1, 2014.

[15] S. Karpitschka, C. Hanske, A. Fery, and H. Riegler. Coalescence and noncoalescence of sessile drops: Impact of surface forces. *Langmuir*, 30:6826–6830, 2014.

[16] S. Karpitschka and H. Riegler. Quantitative experimental study on the transition between fast and delayed coalescence of sessile droplets with different but completely miscible liquids. *Langmuir*, 26:11823–11829, 2010.

[17] S. Karpitschka and H. Riegler. Noncoalescence of sessile drops from different but miscible liquids: Hydrodynamic analysis of the twin drop contour as a self-stabilizing traveling wave. *Phys. Rev. Lett.*, 109:066103, 2012.

[18] M. A. Bruning, M. Costalonga, S. Karpitschka, and J. H. Snoeijer. Delayed coalescence of surfactant containing sessile droplets. *Phys. Rev. Fluids*, 3:073605, 2018.

[19] P.-G. de Gennes, F. Brochard-Wyart, D. Quéré, A. Reisinger, and B. Widom. *Capillarity and Wetting Phenomena: Drops, Bubbles, Pearls, Waves.* Springer, New York, 2004.

[20] H. Uecker. Hopf bifurcation and time periodic orbits with pde2path – algorithms and applications. *Commun. Comput. Phys.*, 25, 2017.

[21] M. Blatt, A. Burchardt, A. Dedner, C. Engwer, J. Fahlke, B. Flemisch, C. Gersbacher, C. Gräser, F. Gruber, C. Grüninger, D. Kempf, R. Klöfkorn, T. Malkmus, S. Müthing, M. Nolte, M. Piatkowski, and O. Sander. The distributed and unified numerics environment, version 2.4. *Arch. Num. Soft.*, 4:13–29, 2016.

[22] M. Heil and A. L. Hazel. An object-oriented multi-physics finite-element library. In *Fluid-Structure Interaction*, pages 19–49. Springer (Lect. Notes Comp. Sci.), 2006.

[23] Sebastian Engelnkemper. *Nonlinear Analysis of Physicochemically Driven Dewetting - Static and Dynamics -*. PhD thesis, Westfälischen Wilhelms-Universität Münster, 2017.

[24] E. Bohl. *Finite Modelle gewöhnlicher Randwertaufgaben.* B. G. Teubner Stuttgart, 1981.

[25] W. Hackbusch. *Theorie und Numerik elliptischer Differentialgleichungen.* Springer Spektrum, Wiesbaden, 2017.

[26] T. Pang. *An Introduction to Computational Physics.* Cambridge University Press, 1997.

[27] S. Trinschek. *Thin Film Modelling of Complex Fluids and Bacterial Colonies.* PhD thesis, Westfälische Wilhelms-Universität Münster, 2018.

[28] U. Thiele. Thin film evolution equations from (evaporating) dewetting liquid layers to epitaxial growth. *J. Phys.: Condens. Matter*, 22:084019, 2010.

[29] U. Thiele. Recent advances in and future challenges for mesoscopic hydrodynamic modelling of complex wetting. *Colloid Surface A*, 553:487 − 495, 2018.

[30] U. Thiele, A. J. Archer, and M. Plapp. Thermodynamically consistent description of the hydrodynamics of free surfaces covered by insoluble surfactants of high concentration. *Phys. Fluids*, 24:102107, 2012.

[31] U. Thiele, A. J. Archer, and L. M. Pismen. Gradient dynamics models for liquid films with soluble surfactant. *Phys. Rev. Fluids*, 1:083903, 2016.

[32] U. Thiele, J. H. Snoeijer, S. Trinschek, and K. John. Equilibrium contact angle and adsorption layer properties with surfactants. *Langmuir*, 34:7210–7221, 2018.

[33] U. Thiele, D. V. Todorova, and H. Lopez. Gradient dynamics description for films of mixtures and suspensions: Dewetting triggered by coupled film height and concentration fluctuations. *Phys. Rev. Lett.*, 111:117801, 2013.

[34] M. Wilczek, W. B. H. Tewes, S. V. Gurevich, M. H. Köpf, L. F. Chi, and U. Thiele. Modelling pattern formation in dip-coating experiments. *Math. Model. Nat. Pheno.*, 10:44–60, 2015.

Acknowledgement

I would like to thank

... Prof. Dr. Uwe Thiele for supervising this thesis and his time and patience helping me during my work on it.

... Prof. Dr. Andrew Hazel for his continuous help regarding questions about `oomph-lib`, as well as his research group for welcoming me kindly during my time in Manchester.

... Dr. Sarah Trinschek for her dedicated support during my work on this thesis.

... Simon Hartmann for continuing my `oomph-lib` legacy in this research group.

... Dr. Sarah Trinschek, Simon Hartmann and Fenna Stegemerten for proof-reading this thesis and their helpful critique.

... my office colleagues and friends Fenna Stegemerten and Tobias Frohoff-Hülsmann for the good times spent working on our research and the better times spent avoiding it.

... everybody in the research group for the good atmosphere and the nice times in the mountains.

Declaration of Academic Integrity

I hereby confirm that this thesis on "Spreading of simple and complex liquids - numerical approaches using Finite Element Methods" is solely my own work and that I have used no sources or aids other than the ones stated. All passages in my thesis for which other sources, including electronic media, have been used, be it direct quotes or content references, have been acknowledged as such and the sources cited.

Münster, March 15, 2019

I agree to have my thesis checked in order to rule out potential similarities with other works and to have my thesis stored in a database for this purpose.

Münster, March 15, 2019