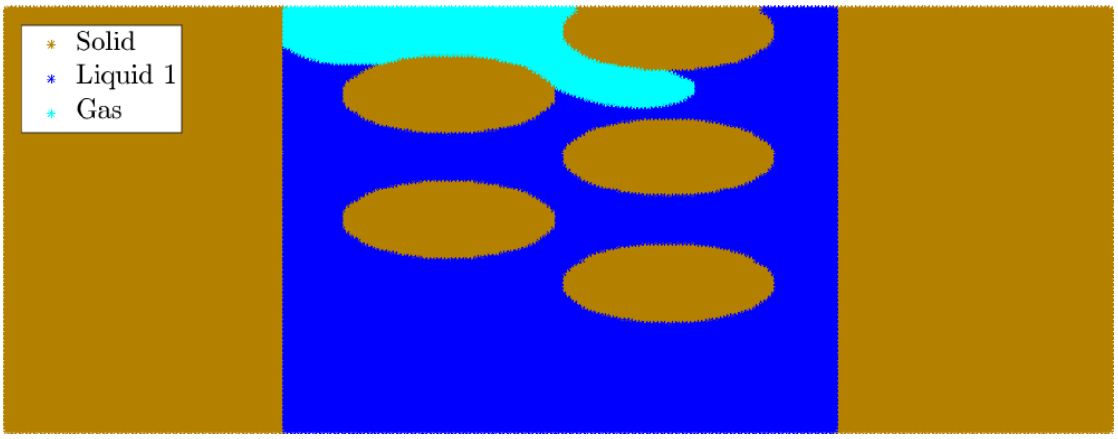
CHALMERS



Curvature flows with junctions as model for capillary flows in complicated domains

Numerical solution of the Ginzburg-Landau model of phase transistion for three phases

MVEX03 Master's Thesis in Engineering Mathematics and Computational Science, MPENM

JOHANNES BORGQVIST

Department of Mathematics
Applied mathematics
Chalmers University of Technology
Gothenburg, Sweden 2015
Master's Thesis in 2015:

Abstract

The focus of the thesis is to model capillary flow for a three phase system in complex domains with a model involving curvature flow in combination with triple junctions.

The first part of the report involves the implementation of a proposed solution algorithm by Ruuth[16; 17]. The solution algorithm involves two steps, namely a diffusion step that models the curvature flow, and a sharpening step that corresponds to the evolution of the three phase system at the *triple junctions*, that is, the points where the three phases meet. The diffusion step involves solving the heat equation for a short period of time $\Delta\tau$, and a finite difference approach is implemented in order to solve the heat equation. The sharpening step is conducted by implementing a so called *projection triangle algorithm*, and this step is also implemented using a finite difference approach. The accuracy of the implementation of the diffusion step is calculated by modelling a two phase system known as "the shrinking sphere" and the projection triangle algorithm is implemented for two different domains. The results of the projection triangle algorithm are compared to the results generated by Ruuth[16].

The second part of the thesis justifies the validity of the projection triangle algorithm in modelling capillary flow. A physical condition concerning the conservation of the contact angle is imposed, and a convolution thresholding calculation is presented using the suggested condition. A numerical validation of the convolution thresholding scheme is presented and compared to the results of the projection triangle algorithm. The convolution thresholding scheme validates the result from the projection triangle algorithm.

The third part of the report includes the numerical simulations of the implemented algorithm. The capillary flow for two different liquids are simulated, and the results indicate that a small contact angle corresponds to a faster capillary flow. Furthermore, the capillary flow for the faster of the two liquids is simulated in two different domain, one in $\mathbb{R}^2$ and one in $\mathbb{R}^3$. The conclusion is that the algorithm proposed by Ruuth[16] can be applied in order to model capillary flow for a three phase system.

Acknowledgements

I would like to thank my supervisor Alexei Heintz for his guidance during the project. Furthermore, I would like to express my gratitude to my friend Toni Matinen who assisted me with his computer in order to conduct certain simulations.

Johannes Borgqvist, Gothenburg $^{26}/_{5} - 2015$

Contents

List of Figures

1

Introduction

In material sciences, it is of interest to study the motion of an interface between a gas-, liquid- and solid phase. An application of such a three phase system could potentially arise when a water column is confined in a narrow pore of a cellulose fiber in for example paper. If the pore is narrow enough, the water surface will rise due to the fact that the water molecules will form hydrogen bonds with the solid fiber[2], and the motion of such a system is called *capillary flow*. Another application in which capillary forces play a vital role, is in the fruit juice industry[13]. In order to concentrate for example orange juice, a Goretex membrane made of Teflon which binds to vapour can draw out water, in vaporous form, from the fruit juice which thus results in a higher fruit concentration. Therefore, during the production of for example paper products such as diapers where capillary flows plays an important role, it could potentially save both time and resources if certain experiments on the products could be simulated in the computer as oppose to conducting these experiments in reality. To implement a valid simulation of such flows requires a chemical and mathematical understanding of the physical phenomena in question, and subsequently follows the underlying theory behind capillary flows.

Consequently, the introduction is divided into five parts. Initially, the chemical and mathematical background in order to explain and model capillary flow is provided. Thereafter follows the purpose and objective of the project and after this the limitations of the project is described. Then, a section on previous articles on capillary flow is described and finally a guide trough the text for the reader is provided.

1.1 Background

The fundamental reason behind capillary flow is that liquids will minimize their surface area[13]. A way of of minimizing the surface area of the liquid can be achieved by *wetting* the solid wall, in other words the liquid will form a chemical bond with the solid surface,

and such forces are often referred to as *adhesive forces*[2; 13]. The molecules in the liquid will also bind to each other in the bulk part of the liquid, and these forces are called *cohesive forces*[2]. The relative strength of the cohesive and adhesive forces will form a *meniscus* and consequently the liquid will bind to the solid with a certain angle called the *contact angle*[2; 13], see figure 1.1. A famous example is the meniscus of water in a glass capillary that will be curved upward at the edges due to the fact that the adhesive forces between the glass and the water is strong on account of the hydroxyl groups in the glass molecules which enables hydrogen bonding with the water. For high energy solids where hydrogen bonding is possible, for example clean glass for water, the water will wet the glass completely which result in a very small contact angle. On the other hand for low energy surfaces such as hydro carbons for water where no hydrogen bonding is possible, the contact angle will be larger and in conclusion a small contact angle is indicative of large adhesive forces relative to the cohesive forces. Furthermore the surface energy, denoted by γ, is related to the contact angle by the so called *Young equation* given by $\cos(\theta) = \frac{\gamma_{\text{Gas,Solid}} - \gamma_{\text{Liquid,Solid}}}{\gamma_{\text{Liquid,Gas}}}$.

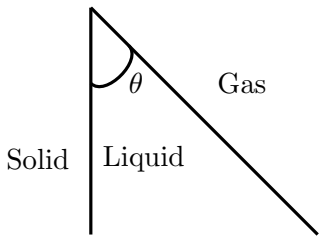


Figure 1.1: Contact Angle

The height of a rising liquid column is determined by the balance of an upward force induced by the reduction of surface energy of the liquid on the one hand, and a downward force applied by gravity[13]. A small contact angle will therefore result in a higher rise of the liquid surface. The aim of this article is therefore to simulate such capillary flows numerically in various domains in two and three dimensions and to analyze such a three phase chemical system mathematically.

The evolution of the three phase interface can be described by the Ginzburg-Landau model for phase transition illustrated in equation 1.1[15]. The model in question consists of a vector valued nonlinear partial differential equation which belongs to the class *reaction diffusion equations*, and this family of equations consists of two parts namely a reaction and a diffusion part.

$$\begin{cases} \dfrac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = \epsilon\Delta\mathbf{u} - \dfrac{1}{\epsilon}V_u(\mathbf{u}), & \mathrm{x} \in \Omega, t \in \mathbb{R}_+ \quad \text{(PDE)} \\ \dfrac{\partial\mathbf{u}(x,\cdot)}{\partial\mathbf{n}} = 0, & x \in \partial\Omega, t \in \mathbb{R}_+ \quad \text{(BC)} \\ \mathbf{u}(x,0) = g(x), & x \in \Omega \quad \text{(IC)} \end{cases} \tag{1.1}$$

In equation 1.1, the solution $\mathbf{u}$ denotes the concentration of the liquid species, the gas species and the solid species. The functional V is an energy potential which physically can be interpreted as the contribution of surface energy at the interface between the various phases. The function g is a continuous function that determines the initial condition of the system in question. Furthermore, the boundary conditions in equation 1.1 is of Neumann type, which physically is interpreted as no material leave the domain. Previously, short time existence of solutions for the three phase problem formulated in equation 1.1 has been proven by Bronsard and Reitich[3]. A solution algorithm for the three phase system in equation 1.1 in the limit $\epsilon \to 0$ has been developed by Ruuth[16; 17] and this algorithm will be applied in order to simulate capillary flow.

1.2 Objective

The objective of the thesis is to implement and analyze the validity of the algorithm proposed by Ruuth in order to solve equation 1.1 in the limit $\epsilon \to 0$ numerically.

The goals of the project can therefore be listed as follows.

- Implement the proposed solution algorithm for equation 1.1 in the limit $\epsilon \to 0$.
- Using a so called *convolution thresholding* analysis, determine the physical validity of the implemented algorithm.
- Simulate capillary flow for a three phase system in arbitrary domains in $\mathbb{R}^2$ and $\mathbb{R}^3$.

The solution algorithm should fulfill at least two conditions. Initially, it should be relatively simple to implement in terms of computer programming, but at the same the algorithm should be able to simulate capillary flow in arbitrarily complicated domains in both two and three dimensions. Secondly, the implemented algorithm should be physically accurate in the sense that each step of the algorithm can be interpreted physically. Therefore, a mathematical analysis will be provided in order to justify the mathematical validity of the implemented algorithm.

1.3 Limitations

The limitations of the project are listed below.

- In order to efficiently simulate capillary flows, fast computer implementations are required. The emphasis on this project however is to implement a solution algorithm and analyze its validity. Consequently, the implementation step will confine itself to a finite difference implementation with a static grid as oppose to faster implementations.

- Moreover, the mathematical analysis of the validity of the algorithm is limited to a *convolution thresholding analysis*. In other words, the mathematical analysis will be confined to various polynomial expansions around certain points, and thus the analysis does *not* involve a more rigid mathematical framework involving measure theory.
- Furthermore, the analysis is limited to study the behavior of the solutions to equation 1.1 in proximity of the so called *triple junction*, i.e. where the three phases meet. In addition to this, the boundary conditions for the simulations of capillary flow will be implemented on square domains in two dimensions and cubic domains in three dimensions, in order to simplify the implementation of appropriate boundary conditions.
- The solid phase will remain stationary through the simulations. Thus the solid phase will effect the motion of the other two phases but will not move itself.
- The solution algorithm will be implemented in C++ and the resulting simulations will be graphically represented in Matlab. These scripts are attached at the end of the report in the Appendix.

1.4 Related litterature

There are extensive literature in order to analyze equation 1.1 mathematically. As mentioned before, short time existence of solutions for the three phase problem formulated in equation 1.1 has been proven by Bronsard and Reitich[3]. Furthermore, the asymptotic expansion of a two phase and a three phase system has also been analyzed by Rubinstein previously in [12; 14; 15].

Previous work concerning solution algorithms consist of the article by Merriman, Bence and Osher[10] and Ruuth[16; 17]. The work by Merriman, Bence and Osher resulted in a solution algorithm for a two phase system. The work by Ruuth developed the algorithm by Merriman, Bence and Osher in order to simulate a three phases system. Finally, the three phase algorithm developed by Ruuth was mathematically analyzed using a so called *convolution thresholding* and a similar analysis have been conducted by Heintz and Grzhibovskis[5], Evans[4] and Ishii[6]. The paper by Grzibovskid and Heintz is rigorous and it includes global existence and uniqueness results. This paper follows the earlier papers by Evans and Ishii.

1.5 Reading guide through the report

The subsequent report consists of five chapters. Chapter 2 includes the theory of the report, and is divided into two sections. The first section concerns the asymptotic behavior of solutions to equation 1.1. This part is rather detailed and it is included in the report in order to give a mathematical understanding of the evolution of the three phase system. However this part is not essential for the rest of the report and can

therefore be skipped. The second part of the second chapter includes the mathematical motivation for developing a solution algorithm to equation 1.1.

In Chapter 3 the implementation of a solution algorithm to equation 1.1 is described. The implementation of the algorithm is based on the second part of the theory chapter, and the description of each step of the implementation is rather detailed. The implementation step involves two different steps, namely a *diffusion* and a *sharpening* step. The main interest of this report, is to understand and implement the sharpening step for a three phase system.

Consequently, chapter 4 involves the analysis of the sharpening step for a three phase system. This analysis will be based on a so called *convolution thresholding* analysis. Using the analysis in question, an analytical formula for the evolution of the three phase system in proximity to the *triple junction* will be derived. The formula will then be tested numerically and compared to the implemented sharpening step in chapter 3.

Chapter 5 includes the results of the simulations generated by the implemented solution algorithm developed in chapter 3. Chapter 5 consists of two parts, namely simulations in $\mathbb{R}^2$ and $\mathbb{R}^3$. The evolution of the three phase system in equation 1.1 are simulated for three different domains in $\mathbb{R}^2$ and one domain is simulated in $\mathbb{R}^3$.

Finally, chapter 6 contains the discussion of the conclusions that can be drawn from the results presented in chapter 5. Also a section on future work is presented.

2

Theory

The theory chapter consists of two parts, namely an asymptotic analysis of equation 1.1 and a mathematical motivation for a solution algorithm. The first part of the theory chapter, called asymptotic analysis, is rather detailed, and is included in order to describe the long term evolution of a three phase system. The second part of the theory chapter, called solution algorithm, provides a mathematical framework for a solution algorithm of equation 1.1.

The asymptotic analysis of the solutions to equation 1.1 has two main properties that will be derived in the subsequent chapter called asymptotic analysis. The first property is that the interface between two phases will evolve according to its mean curvature, which is stated in equation 2.20 on page 16. The second property is that at the *triple junction*, where the three phases meet, the Young's equation, which was stated in the introduction, is satisfied. Therefore, it is essential that a solution algorithm to equation 1.1 has two properties. The first property is that the algorithm generates solutions which evolves according to its mean curvature, and the second is that it takes the angle configuration at the triple junction into account. In order to understand the evolution of interfaces of phases, the term mean curvature requires a definition.

The mean curvature of a curve $\phi(t)$ at a point $\mathbf{x}_0$ measures how much the curve "bends" at the point in question. For instance, if $t \geq 0$ is the time, let $(x(t),y(t),z(t)) \in \mathbb{R}^3$ be a position given a time t and let the curve $\phi(t)$ be the curve defined by the set of points given by $r(t) = x(t) \cdot \mathbf{i} + y(t) \cdot \mathbf{j} + z(t) \cdot \mathbf{k}$ where $\mathbf{i}$, $\mathbf{j}$ and $\mathbf{k}$ are the unit vectors in the x, y and z directions respectively. Then the unit tangent vector, denoted T, is defined by equation 2.1 below and it provides the direction of the curve at a certain point[1; 19].

$$T(t) = \frac{\frac{\mathrm{d}r}{\mathrm{d}t}}{\left|\frac{\mathrm{d}r}{\mathrm{d}t}\right|} \tag{2.1}$$

The curvature, denoted $\kappa(s)$, of the curve $\phi(t)$ at the point $r(s)$ will be given by equation

2.2. In words, when travelling along a curve, the curvature is the rate of change of the tangent at a certain point.

$$\kappa(s) = \left| \frac{\mathrm{d}T}{\mathrm{d}s} \right| \tag{2.2}$$

In general, there will be two *principal curves*[1] , denoted κ_1 and κ_2, and the *mean curvature* will be the mean value of the principal curvatures as stated in equation 2.3[8].

$$H = \frac{\kappa_1(s) + \kappa_2(s)}{2} \tag{2.3}$$

By letting an interface $\Gamma(t)$ at a time t between various phases be described by the *level sets* of a continuous function $\phi : \Omega \subset \mathbb{R}^n \to \mathbb{R}$, that is $\Gamma(t) := \{\mathbf{x} \in \Omega : \phi(\mathbf{x},t) = 0\}$ the mean curvature of the interface will be given by equation 2.4[10; 14].

$$H = \operatorname{div}\left(\frac{\nabla \phi}{|\nabla \phi|} \right) \tag{2.4}$$

In equation 2.4, the operator $\nabla = \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z} \right)^T$ is the gradient and $\operatorname{div} = \frac{\partial}{\partial x} + \frac{\partial}{\partial y} + \frac{\partial}{\partial z}$ denotes the divergence operator where $\operatorname{div}(\nabla \phi) = \Delta \phi$ where $\Delta = \frac{\partial}{\partial x^2} + \frac{\partial}{\partial y^2} + \frac{\partial}{\partial z^2}$ is the Laplace operator.

With these definitions in mind an asymptotic analysis of equation 1.1 will be presented and then follows the theory behind a solution algorithm to equation 1.1.

2.1 Asymptotic analysis

Initially, the various terms in equation 1.1 will be explained in detail. Assume therfore that the domain, denoted $\Omega \subset \mathbb{R}^d : d \in \{2,3\}$, is divided into three subdomains denoted Ω_1, Ω_2 and Ω_3 and let the three interfaces between the three domains be denoted Γ_{12}, Γ_{13} and Γ_{23} respectively. A sketch of such a domain in $\mathbb{R}^2$ is illustrated in figure 2.1 below. Subsequently, Ω_1 will denote the liquid phase, Ω_2 will denote the gas phase and Ω_3 will denote the solid phase.

In equation 1.1, $\epsilon \Delta \mathbf{u}$ is the *diffusion term* and $\frac{1}{\epsilon} \mathbf{V}_u(\mathbf{u})$ is the *reaction term* where V is a functional acting as an energy minimizer. The solution vector $\mathbf{u}$ is defined as $\mathbf{u}(\mathbf{x},t) = \begin{pmatrix} u_1(\mathbf{x},t) \\ u_2(\mathbf{x},t) \\ u_3(\mathbf{x},t) \end{pmatrix}$ where $u_i : \Omega \times \mathbb{R}_+ \to [0,1]$, $i \in \{1,2,3\}$, is the concentration of species i where the value $u_i(\mathbf{x},\cdot) = 1$ corresponds to a position $\mathbf{x} \in \Omega$ *within* phase i and the value $u_i(\mathbf{x},\cdot) = 0$ corresponds to a position $\mathbf{x} \in \Omega$ *outside* phase i. The term $\mathbf{V}_u(\mathbf{u})$ is the

[1] Without introducing any new notation, the principal curvatures κ_1 and κ_2 are the eigenvalues of the Weingarten endomorphism which is described by López in [8].

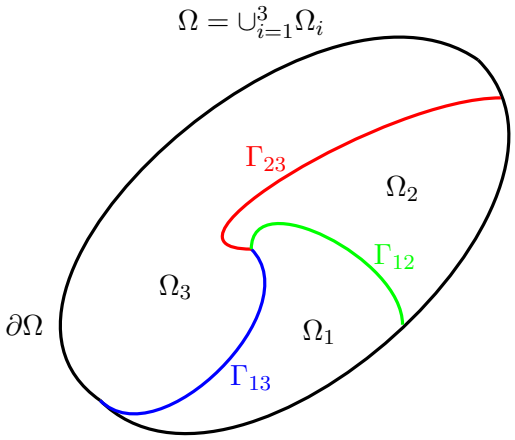


Figure 2.1: Region $\Omega \subset \mathbb{R}^2$ for three phases. Each phase i is divided into a subdomain Ω_i for $i \in \{1,2,3\}$ where the three interfaces are denoted Γ_{12} (in green), Γ_{13} (in blue) and Γ_{23} (in red).

gradient of the energy potential such that there is a *stable minima* in each phase Ω_1, Ω_2 and Ω_3[3; 10]. The boundary conditions, denoted BC, will in the subsequent simulations be of Neumann or periodic type. The initial conditions, denoted IC, are defined by the function $g(\mathbf{x})$ which is a vector function defined in the following way, $g(\mathbf{x}) = \begin{pmatrix} \chi_1(\mathbf{x}) \\ \chi_2(\mathbf{x}) \\ \chi_3(\mathbf{x}) \end{pmatrix}$, where $\chi_i(\mathbf{x}) = \begin{cases} 1, & \text{if } \mathbf{x} \in \Omega_i \\ 0, & \text{if } \mathbf{x} \notin \Omega_i \end{cases}$ for $i \in \{1,2,3\}$ is the characteristic function for a certain domain. Furthermore, the following mass conservation law formulated in equation 2.5 holds for all positions $\mathbf{x} \in \Omega$ and all times $t \in \mathbb{R}_+$.

$$\sum_{i=1}^{3} u_i(\mathbf{x},t) = 1, \quad \forall \mathbf{x} \in \Omega \forall t \in \mathbb{R}_+ \tag{2.5}$$

Therefore, the aim of this section is to study the asymptotic of equation 1.1. In order to study the dynamics of the system analytically, the system will be analyzed by means of various perturbation expansions in **three** different neighbourhoods, namely in a certain phase far from an interface, in proximity to an interface between two phases and close to the *triple junction* i.e. where the three phases meet. The first two expansions are based on the article by Rubinstein, Sternberg and Keller, see [14], and the analysis

near the triple junction is based on the analysis by Bronsard and Reitich, see [3], and by the analysis by Owen, Rubinstein and Sternberg, see [12].

The analysis in this section is rather detailed, and can be skipped if the reader is interested in the numerical simulations of equation 1.1.

2.1.1 In the phase far from an interface

Initially, assume that a position $\mathbf{x} \in \Omega_i$ far away from either the boundary $\partial\Omega$ or the interface Γ_{ij}. Furthermore, assume that the function $V_u(u)$ has a minima a in this phase, where the analysis is conducted for one of the three phases such that $V_u(u)$ can be treated as a continuous function of *one* variable u as oppose to a gradient vector. Given these conditions, the dynamics in the phase is rather slow, and therefore it is appropriate to change the time scale. This can be achieved by the change of variables $\tau = \frac{t}{\epsilon}$. Furthermore, the perturbation series in equation 2.6 introduced in [14] can be established by considering u as a function of ϵ.

$$u(\mathbf{x},t,\epsilon) = v(\mathbf{x},\tau,\epsilon) = v_0(\mathbf{x},\tau,\epsilon) + \epsilon^2 v_1(\mathbf{x},\tau,\epsilon) + O(\epsilon^4) \tag{2.6}$$

Differentiation of the left hand side of equation 2.6 with respect to t yields the following expression.

$$\frac{\mathrm{d}u}{\mathrm{d}t} = \frac{\mathrm{d}v}{\mathrm{d}\tau}\underbrace{\frac{\mathrm{d}\tau}{\mathrm{d}t}}_{=\epsilon^{-1}} = \frac{1}{\epsilon}\frac{\mathrm{d}v}{\mathrm{d}t} \Longrightarrow \boxed{\frac{\mathrm{d}v}{\mathrm{d}t} = \epsilon\frac{\mathrm{d}u}{\mathrm{d}t}}$$

Thus, by plugging in the perturbation series in equation 2.6 into the PDE in equation 1.1 the following expression is obtained.

$$\frac{\mathrm{d}v_0}{\mathrm{d}\tau} + \epsilon^2\frac{\mathrm{d}v_1}{\mathrm{d}\tau} = \epsilon^2\Delta v_0 + \epsilon^3\Delta v_1 - V_u(v_0 + \epsilon^2 v_1) + O(\epsilon^4)$$

Furthermore, assume that the functional V_u is smooth, that is it has a continuous derivative, then it is possible to use the following Taylor expansion around the point $v_1 = 0$.

$$\begin{aligned} V_u(v_0 + \epsilon^2 v_1) &= V_u(v_0) + \left\{ \frac{\mathrm{d}V_u(v_0 + \epsilon^2 v_1)}{\mathrm{d}u}\underbrace{\frac{\mathrm{d}u}{\mathrm{d}v_1}}_{=\epsilon^2} \right\}\Bigg|_{v_1=0} v_1 + O(\epsilon^4) \\ &= V_u(v_0) + V_{uu}(v_0)\epsilon^2 v_1 + O(\epsilon^4) \end{aligned}$$

Consequently, the Taylor expansion of the functional V_u is given by equation 2.7.

$$V_u(v_0 + \epsilon^2 v_1) = V_u(v_0) + V_{uu}(v_0)\epsilon^2 v_1 + O(\epsilon^4) \tag{2.7}$$

Now, plugging in equation 2.7 into the equation above and arranging the terms into coefficients of ϵ yields the following expression.

$$\epsilon^0\left(\frac{\mathrm{d}v_0}{\mathrm{d}\tau}+V_u(v_0)\right)+\epsilon^2\left(\frac{\mathrm{d}v_1}{\mathrm{d}\tau}-\Delta v_0+V_{uu}(v_0)v_1\right)+\epsilon^3\Delta v_1=0$$

When it comes to the initial condition, it follows that the following holds

$$\epsilon^0\cdot v_0(\mathbf{x},0)+\epsilon^2\cdot v_1(\mathbf{x},0)+O(\epsilon^4)=\epsilon\cdot g(\mathbf{x})+\epsilon^2\cdot 0+0\cdot\sum_{i=1}^{\infty}\epsilon^{2i}$$

and equating the coefficients of ϵ^0 and ϵ^2 to zero result in equation 2.8 and 2.9 respectively.

$$\begin{cases}\dfrac{\mathrm{d}v_0}{\mathrm{d}\tau} & = -V_u(v_0), \quad \mathbf{x}\in\Omega,t\in\mathbb{R}_+\\ v_0(\mathbf{x},0) & = g(\mathbf{x}), \quad \mathbf{x}\in\Omega\\ \nabla v_0\cdot\mathbf{n} & = 0, \quad \mathbf{x}\in\partial\Omega,t\in\mathbb{R}_+\end{cases} \tag{2.8}$$

$$\begin{cases}\dfrac{\mathrm{d}v_1}{\mathrm{d}\tau} & = \Delta v_0 - V_{uu}(v_0)v_1, \quad \mathbf{x}\in\Omega,t\in\mathbb{R}_+\\ v_1(\mathbf{x},0) & = 0, \quad \mathbf{x}\in\Omega\\ \nabla v_1\cdot\mathbf{n} & = 0, \quad \mathbf{x}\in\partial\Omega,t\in\mathbb{R}_+\end{cases} \tag{2.9}$$

Equation 2.8 is an ordinary differential equation with respect to τ and to check that this equation satisfies the boundary condition it is possible to take the normal derivative of v_0 in order to get the following equation,

$$\begin{aligned}\frac{\mathrm{d}\left(\nabla v_0\cdot\mathbf{n}\right)}{\mathrm{d}\tau} &= -\nabla\left(V_u(v_0)\right)\cdot\mathbf{n} = -V_{uu}(v_0)\left(\nabla v_0\cdot\mathbf{n}\right)\\ \left(\nabla v_0(\mathbf{x},0)\cdot\mathbf{n}\right) &= \left(\nabla g(\mathbf{x})\cdot\mathbf{n}\right)=0\end{aligned}$$

where $(\nabla g(\mathbf{x})\cdot\mathbf{n})=0$. Thus it follows that v_0 satisfies the boundary condition, and thus there is a solution v_0 that is entirely determined by the functional V_u on the τ-scale. If the normal derivatives are taken in problem 2.9, no solution satisfies the boundary condition, and hence it is only the leading term in equation 2.6 that is valid up to the boundary $\partial\Omega$. However, it is possible using another perturbation series as in [14] to get a function w_1 that is valid up to the boundary $\partial\Omega_i$. In fact, the following limits must hold for the functions v_0, v_1 and w_1 respectively where a is the minima for the functional V_u corresponding to the domain Ω_i for a position $\mathbf{x}\in\Omega_i$.

$$\lim_{\tau\to\infty}v_0(\mathbf{x},\tau)=a \quad , \quad \lim_{\tau\to\infty}v_1(\mathbf{x},\tau)=0 \quad , \quad \text{and} \quad \lim_{\tau\to\infty}w_1(\mathbf{x},\tau)=0$$

Hence, provided a position $\mathbf{x}\in\Omega_i$, the solution u will approach the minima for the respective phase Ω_i. On the other hand, for a position $\mathbf{x}\in\Gamma_{ij}:=\Omega_i\cap\Omega_j$ a different perturbation series is required.

2.1.2 Close to an interface between two phases

To analyze the behaviour of the system near the interface, Γ_{ij}, between two domains Ω_i and Ω_j respectively the level set approach introduced by Osher will be used[10]. Let $\phi(\mathbf{x},t) : \mathbb{R}^3 \times \mathbb{R}_+ \to \mathbb{R}$ be a continuous function that is twice differentiable such that the interface Γ_{ij} between phase Ω_i and Ω_j can be written as in equation 2.10.

$$\Gamma_{ij}(t) := \{\mathbf{x} \in \Omega_i \cap \Omega_j : \phi(\mathbf{x},t) = 0\} \tag{2.10}$$

The aim of the analysis is to predict the dynamics of the interface in equation 2.10 on a long and a short time scale. Let us introduce the variables $\eta = \epsilon \cdot t$ and $\tau = \frac{t}{\epsilon}$ where η is the variable on the short time scale, and τ is the variable on the larger time scale. Then the function ϕ will be a function of η but can be seen as stationary on the τ scale, in other words it holds that ϕ can be written as $\phi(\mathbf{x},t,\eta)$.

Furthermore, it is convenient to represent the distance from a point $\mathbf{x}$ to the interface $\Gamma_{ij}(t)$ at a given time t by the variable $z = \frac{\phi(\mathbf{x},t)}{\epsilon}$. To proceed the perturbation series in equation 2.11 proposed by Rubinstein, Keller and Sternberg[14] can be used.

$$u(\mathbf{x},t,\epsilon) = u^0(\mathbf{x},t,\tau,z,\eta,\epsilon) + \epsilon \cdot u^1(\mathbf{x},t,\tau,z,\eta,\epsilon) + O(\epsilon^2) \tag{2.11}$$

The next step is to plug in equation 2.11 into the PDE in equation 1.1 in order to split the original equation into numerous subequations. Using the multivariate chain rule, the time derivative in the left hand side of equation 1.1 can be written as follows.

$$\begin{aligned}
\frac{\mathrm{d}u}{\mathrm{d}t} &= \frac{\mathrm{d}u^0}{\mathrm{d}t} + \frac{\mathrm{d}u^0}{\mathrm{d}\tau}\underbrace{\frac{\mathrm{d}\tau}{\mathrm{d}t}}_{=\epsilon^{-1}} + \frac{\mathrm{d}u^0}{\mathrm{d}z}\underbrace{\frac{\mathrm{d}z}{\mathrm{d}t}}_{=\epsilon^{-1}\phi_t(\mathbf{x},t)} + \frac{\mathrm{d}u^0}{\mathrm{d}z}\underbrace{\frac{\mathrm{d}z}{\mathrm{d}\eta}}_{=\epsilon^{-1}\phi_\eta(\mathbf{x},t)}\underbrace{\frac{\mathrm{d}\eta}{\mathrm{d}t}}_{=\epsilon} \\
&+ \epsilon\left(\frac{\mathrm{d}u^1}{\mathrm{d}t} + \frac{\mathrm{d}u^1}{\mathrm{d}\tau}\underbrace{\frac{\mathrm{d}\tau}{\mathrm{d}t}}_{=\epsilon^{-1}} + \frac{\mathrm{d}u^1}{\mathrm{d}z}\underbrace{\frac{\mathrm{d}z}{\mathrm{d}t}}_{=\epsilon^{-1}\phi_t(\mathbf{x},t)} + \frac{\mathrm{d}u^1}{\mathrm{d}z}\underbrace{\frac{\mathrm{d}z}{\mathrm{d}\eta}}_{=\epsilon^{-1}\phi_\eta(\mathbf{x},t)}\underbrace{\frac{\mathrm{d}\eta}{\mathrm{d}t}}_{=\epsilon}\right) + O(\epsilon^2) \\
&= \frac{1}{\epsilon}\left(u^0_\tau + u^0_z\phi_t\right) + \epsilon^0\left(u^0_t + u^0_z\phi_\eta + u^1_\tau + u^1_z\phi_t\right) + \epsilon^1\left(u^1_t + u^1_z\phi_\eta\right) + O(\epsilon^2)
\end{aligned}$$

Thus, arranging the coefficients of ϵ, the time derivative can be written as in equation 2.12 below.

$$\frac{\mathrm{d}u}{\mathrm{d}t} = \frac{1}{\epsilon}\left(u^0_\tau + u^0_z\phi_t\right) + \epsilon^0\left(u^0_t + u^0_z\phi_\eta + u^1_\tau + u^1_z\phi_t\right) + \epsilon^1\left(u^1_t + u^1_z\phi_\eta\right) + O(\epsilon^2) \tag{2.12}$$

When it comes to the Laplace operator, do the differentiation of the $x-$variable contained in the $\mathbf{x} = \begin{pmatrix} x & y & z \end{pmatrix}^T$ vector first. From this it follows that

$$\begin{aligned}\frac{\partial u}{\partial x} &= \frac{\partial u^0}{\partial x} + \frac{\partial u^0}{\partial z}\frac{\partial z}{\partial x} + \epsilon\frac{\partial u^1}{\partial x} + \epsilon\frac{\partial u^1}{\partial z}\frac{\partial z}{\partial x} + O(\epsilon^2) \\ &= \frac{1}{\epsilon}u_z^0\phi_x + \epsilon^0\left(u_x^0 + u_z^1\phi_x\right) + \epsilon^1 u_x^1 + O(\epsilon^2)\end{aligned}$$

and differentiating with respect to x a second times yields the following calculation.

$$\begin{aligned}\frac{\partial^2 u}{\partial x} &= \frac{1}{\epsilon}u_{zx}^0\phi_x + \frac{1}{\epsilon^2}u_{zz}^0\phi_x^2 + \frac{1}{\epsilon}u_z^0\phi_{xx} \\ &\quad + u_{xx}^0 + \frac{1}{\epsilon}u_{zx}^0\phi_x + u_{zx}^1\phi_x + \frac{1}{\epsilon}u_{zz}^1\phi_x^2 \\ &\quad + \epsilon u_{xx}^1 + u_{xz}^1\phi_x + O(\epsilon^2)\end{aligned}$$

Arranging the terms and multiplying with a factor ϵ yields the following result.

$$\epsilon\frac{\partial^2 u}{\partial x^2} = \frac{1}{\epsilon}u_{zz}^0\phi_x^2 + \epsilon^0\left(2u_{xz}^0\phi_x + u_z^0\phi_{xx} + u_{zz}^1\phi_x^2\right) + \epsilon^1\left(u_{zx}^1\phi_x + u_{xz}^1\phi_x\right) + \epsilon^2 u_{xx}^1 + O(\epsilon^3)$$

The result for the other three spatial variables are identical, and hence the expression for the Laplace operator is given in equation 2.13.

$$\epsilon\Delta u = \frac{1}{\epsilon}u_{zz}^0\nabla\phi^2 + \epsilon^0\left(2\nabla u_z^0\nabla\phi + u_z^0\Delta\phi + u_{zz}^1\nabla\phi^2\right) + 2\epsilon^1\nabla u_z^1\nabla\phi + \epsilon^2\Delta u^1 + O(\epsilon^3) \quad (2.13)$$

When it comes to the operator $V_u(u)$, equation 2.7 will be used. Thus equating the coefficients of ϵ^{-1} and ϵ^0 equation 2.14 and 2.15 are obtained respectively.

$$u_\tau^0 + u_z^0\phi_t - u_{zz}^0\nabla\phi^2 + V_u(u^0) = 0 \quad (2.14)$$

$$u_t^0 + u_z^0\phi_\eta + u_\tau^1 + u_z^1\phi_t - 2\nabla u_z^0\nabla\phi - u_z^0\Delta\phi - u_{zz}^1\nabla\phi^2 + V_{uu}(u^0)u^1 = 0 \quad (2.15)$$

There are two observations to be made from these equations. The first observation is that in equation 2.14 the only two variables that u^0 depend on are z and τ, and thus equation 2.14 describes the evolution of ϕ on the τ scale. In equation 2.15, u^1 is also dependent on merely z and τ but it contains the term $u_z^0\phi_\eta$ and thus solving this equation will result in a description of ϕ on the η scale. Consequently, by analyzing equation 2.14 and 2.15, the evolution of Γ can be studied on the τ and η time scale. How this analysis is conducted will be described in the subsequent two subsections.

Evolution of Γ on the τ-time scale

In order to analyze the dynamics of the system on the τ-time scale it is possible to assume that the dynamics of the interface $\Gamma_{ij}(t)$ can be described by a *travelling wave*. Let the speed of the wave be denoted c, and let the wave front be described by the function Q, then it is possible to approximate u^0 as follows.

$$u^0(\mathbf{x},t,\tau,z,\eta) \sim Q(z - c\tau,\mathbf{x},t,\eta) \quad \text{as} \quad \tau \to \infty$$

Observe that the following limits holds for the function Q, where u_i and u_j are the respective *bulk* concentrations on each side of the interface Γ_{ij}.

$$\lim_{z\to\infty} Q(z - c\tau,\mathbf{x},t,\eta) = u_i$$
$$\lim_{z\to-\infty} Q(z - c\tau,\mathbf{x},t,\eta) = u_j$$

Hence, plugging in the above expression in equation 2.14, results in the following expression

$$\begin{aligned} -cQ' + Q'\phi_t - Q''\nabla\phi^2 + V_u(Q) &= 0 \\ \Longrightarrow \boxed{-Q''\nabla\phi^2 + Q'(\phi_t - c) + V_u(Q) = 0} & \end{aligned}$$

which is summarized in equation 2.16.

$$-Q''\nabla\phi^2 + Q'(\phi_t - c) + V_u(Q) = 0 \tag{2.16}$$

Proceed by multiplying the above equation by Q' and integrate the expression from $z = -\infty$ to $z = \infty$.

$$\begin{aligned} -\nabla\phi^2 \int_{-\infty}^{\infty} Q'' \cdot Q' \,\mathrm{d}z + (\phi_t - c) \int_{-\infty}^{\infty} (Q')^2 \,\mathrm{d}z + \int_{-\infty}^{\infty} V_u(Q)Q' \,\mathrm{d}z = 0 \\ \Leftrightarrow \boxed{-\nabla\phi^2 I_1 + (\phi_t - c)I_2 + I_3 = 0} \end{aligned}$$

Subsequently the two integrals I_1 and I_3 will be calculated subsequently. Note that the integrand in integral I_1 can be written in the following manner

$$Q'' \cdot Q' = \frac{1}{2}\frac{\mathrm{d}}{\mathrm{d}z}\left\{(Q')^2\right\}$$

and hence the integral I_1 has the value

$$
\begin{aligned}
I_1 &= \int_{-\infty}^{\infty} Q'' \cdot Q' \, \mathrm{d}z \\
&= \frac{1}{2} \int_{-\infty}^{\infty} \frac{\mathrm{d}}{\mathrm{d}z} \left\{ (Q')^2 \right\} \, \mathrm{d}z \\
&= \frac{1}{2} \left[\underbrace{\frac{\mathrm{d}u_i}{\mathrm{d}z}}_{=0} - \underbrace{\frac{\mathrm{d}u_j}{\mathrm{d}z}}_{=0} \right] = 0
\end{aligned}
$$

where the fact that u_i and u_j are indendent of z was used.

The integral I_3 has the following value

$$
\begin{aligned}
I_3 &= \int_{-\infty}^{\infty} V_u(Q) Q' \, \mathrm{d}z \\
&\left\{ \text{Let } \; y = Q \Longrightarrow \mathrm{d}y = Q' \mathrm{d}z \right\} \\
&= \int_{u_j}^{u_i} V_u(y) \mathrm{d}y = V(u_i) - V(u_j) = [V]
\end{aligned}
$$

where $[V]$ denotes the jump in the energy potential between phase i and j. Thus it follows that equation the following hold

$$
\phi_t - c = \frac{[V]}{I_2}
$$

and in fact by using the change of variables

$$
s = \frac{\phi_t - c}{|\Delta \phi|}, \quad R(s) = Q(z - c\tau, x, t, \eta)
$$

and denoting $I_2^\star = \int_{-\infty}^{\infty} R_s^2 \, \mathrm{d}s$ the following expression describes the evolution of ϕ at the τ time scale.

$$
\frac{\phi_t - c}{|\nabla \phi|} = \frac{[V]}{I_2^\star} \tag{2.17}
$$

By introducing the function $\Phi(\mathbf{x}, \eta, t) = \phi(\mathbf{x}, \eta, t) - ct$, the left hand side of equation 2.17 can be written as $\dfrac{\Phi_t}{|\nabla \Phi|}$ which is the *normal velocity* of a level set of Φ. Hence, each level set of Φ moves along the normal direction with constant speed at the τ time scale. When it comes to the dynamics of Γ_{ij} on the η time scale, a similar analysis that just was conducted can be applied to equation 2.15 on page 12.

Evolution of Γ on the η-time scale

In a similar way, assume that u^1 tends to a standing wave denoted by P. In this case the following holds

$$u^1(x,t,\tau,\eta,z) \sim P(z - c\tau,\eta,t,x) \quad \text{as} \quad \tau \to \infty$$

with the two limits as the minima in each adjacent phase.

$$\lim_{z\to\infty} P(z - c\tau,\mathbf{x},t,\eta) = u_i$$
$$\lim_{z\to-\infty} P(z - c\tau,\mathbf{x},t,\eta) = u_j$$

Furthermore, it is possible to use the following assumptions. On the η-time scale it follows that $Q_t = 0$ and furthermore that $\phi_t - c = 0$. Then plugging in the function P into equation 2.15 yields the following expression

$$Q'\phi_\eta - cP' + P'\phi_t - 2\nabla Q'\nabla\phi - Q'\Delta\phi - P''\nabla\phi^2 + V_{uu}(Q)P = 0$$

which simplifies to equation 2.18.

$$-\nabla\phi^2 P'' + V_{uu}(Q)P = \left(\Delta\phi + 2\nabla\phi\cdot\nabla\right)Q' - \phi_\eta Q' \tag{2.18}$$

In order to simplify equation 2.18, an additional expression for a useful identity can be obtained by differentiating equation 2.18 with respect to z, and this identity is shown in equation 2.19 below. Note that in equation 2.19, the assumption $\phi_t - c = 0$ have been used.

$$-\nabla\phi^2 Q''' + V_{uu}(Q)Q' = 0 \tag{2.19}$$

The subsequent step is to multiply equation 2.18 with Q' and integrate with respect to z. The left hand side of equation 2.18 results in the following calculation,

$$\begin{aligned}\int_{-\infty}^{\infty}\left(-\nabla\phi^2 P'' + V_{uu}(Q)P\right)Q'\,\mathrm{d}z &= \underbrace{\left[-\nabla\phi^2 P'Q'\right]_{-\infty}^{\infty}}_{=0} + \int_{-\infty}^{\infty} -\nabla\phi^2 P'Q'' + V_{uu}(Q)Q'P\,\mathrm{d}z \\ &= \underbrace{\left[-\nabla\phi^2 PQ''\right]_{-\infty}^{\infty}}_{=0} + \int_{-\infty}^{\infty} -\nabla\phi^2 PQ''' + V_{uu}(Q)Q'P\,\mathrm{d}z \\ &= \int_{-\infty}^{\infty}\underbrace{\left(-\nabla\phi^2 Q''' + V_{uu}(Q)Q'\right)}_{=0 \text{ by eq. } 2.19}P\,\mathrm{d}z = 0\end{aligned}$$

where integration by parts in combination with the fact that each standing wave, i.e. Q and P, have a stable minima on each side of the interface was used. Thus, multiplying

the right hand side of equation 2.18 with Q' and integrating with respect to z yields the following calculation.

$$\begin{aligned} 0 &= \int_{-\infty}^{\infty} \left(\Delta\phi + 2\nabla\phi\cdot\nabla\right)(Q')^2 - \phi_\eta (Q')^2 \,\mathrm{d}z \\ &= \left(\Delta\phi + 2\nabla\phi\cdot\nabla\right)\int_{-\infty}^{\infty}(Q')^2\,\mathrm{d}z - \phi_\eta\int_{-\infty}^{\infty}(Q')^2\,\mathrm{d}z \\ \implies \phi_\eta &= \Delta\phi + \frac{\left(2\nabla\phi\cdot\nabla\right)\int_{-\infty}^{\infty}(Q')^2\,\mathrm{d}z}{\int_{-\infty}^{\infty}(Q')^2\,\mathrm{d}z} \end{aligned}$$

Now, using the fact that $Q' = \dfrac{R_s}{|\nabla\phi|} \implies \int_{-\infty}^{\infty}(Q')^2\,\mathrm{d}z = |\nabla\phi|\int_{-\infty}^{\infty}(R_s)^2\,\mathrm{d}s$ yields the following calculation

$$\begin{aligned} \phi_\eta &= \Delta\phi + \frac{\left(2\nabla\phi\cdot\nabla\right)\int_{-\infty}^{\infty}(Q')^2\,\mathrm{d}z}{\int_{-\infty}^{\infty}(Q')^2\,\mathrm{d}z} \\ &= \Delta\phi + \frac{\left(2\nabla\phi\cdot\nabla\right)|\nabla\phi|\int_{-\infty}^{\infty}(R_s)^2\,\mathrm{d}s}{|\nabla\phi|\int_{-\infty}^{\infty}(R_s)^2\,\mathrm{d}s} \\ &= \Delta\phi + \frac{\left(2\nabla\phi\cdot\nabla\right)|\nabla\phi|}{|\nabla\phi|} \\ &= |\nabla\phi|\mathrm{div}\left(\frac{\nabla\phi}{|\nabla\phi|}\right) \end{aligned}$$

and this result is summarized in equation 2.20.

$$\frac{\phi_\eta}{|\nabla\phi|} = \mathrm{div}\left(\frac{\nabla\phi}{|\nabla\phi|}\right) = H \tag{2.20}$$

The conclusion from equation 2.20, is that the level set ϕ at the η-time scale will evolve with a speed proportional to its *mean curvature*, denoted k_ϕ which is a well known identity, first derived in [14]. Consequently, the evolution of an interface between two phases is called *mean curvature flow* and next the evolution of the interface at the triple junction will be conducted.

2.1.3 Close to the triple junction

To evaluate the evolution of the triple junction, a formal analysis for a function $u_i : \Omega \subset \mathbb{R}^2 \times \mathbb{R}_+ \to [0,1]$ for $i \in \{1,2,3\}$ with one liquid phase for $i = 1$, one gas phase for $i = 2$ and one solid phase for $i = 3$. In the analysis, the contact angle between the solid phase and the other phases is π, in other words the solid phase will be *stationary*. In order to conduct a formal analysis of the system at hand, introduce a factor $\hat{\epsilon}$ such that the system to be analyzed can be formulated as in equation 2.21 below. The factor 2 in front of $\hat{\epsilon}$ will be of convenience when the system at hand is analyzed.

$$\begin{cases} \dfrac{\mathrm{d}u_1}{\mathrm{d}t} = 2\hat{\epsilon}\Delta u_1 - \dfrac{1}{\hat{\epsilon}}V_u(u), & \mathrm{x} \in \Omega, t \in \mathbb{R}_+ \quad \text{(PDE)} \\ \nabla u_1(x,t) \cdot \mathbf{n} = 0, & x \in \partial\Omega, t \in \mathbb{R}_+ \quad \text{(BC)} \\ u_1(x,0) = \chi_1(x), & x \in \Omega \quad \text{(IC)} \end{cases} \tag{2.21}$$

Furthermore, let the three phases meet at a single point called x_0 as illustrated in figure 2.2 below. Assume furthermore that close to the triple junction, the system evolves at the slow time scale denoted by $\hat{\sigma} = \hat{\epsilon}t$ and that the spatial coordinates of the system can be denoted by $\eta = (\eta_1, \eta_2)$ where η_1 is tangent to the solid wall, η_2 is orthogonal to η_1 and where η is related to x by the equation $\eta = \dfrac{x}{\hat{\epsilon}}$. In this case, η are stretched coordinates centered at x_0 as proposed by Rubinstein and Sternberg[12]. Using these time- and spatial variables, an expansion of the form

$$u(x,t) = u^0(\eta,\hat{\sigma}) + \hat{\epsilon}u^1(\eta,\hat{\sigma}) + \dots \tag{2.22}$$

for the liquid phase, i.e. $u = u_1$, around $\eta = 0$ can be applied in similarity with the calculations conducted in the previous section close to the interface between the two phases. Then using the calculations that resulted in equation 2.14 on page 12 in combination with the assumptions that the function ϕ is normalized, i.e. $|\nabla\phi| = 1$ and that the system is *stationary* on the τ time scale equation 2.23 can be assumed to hold for u_1.

$$2u^0_{zz} = V_u(u^0) \tag{2.23}$$

Plugging in equation 2.22 in the (PDE) in equation 2.21 results in the following calculations

$$\begin{aligned} \frac{\mathrm{d}u^0}{\mathrm{d}t} + \hat{\epsilon}\frac{\mathrm{d}u^1}{\mathrm{d}t} &= 2\hat{\epsilon}\Delta u^0 + 2\hat{\epsilon}^2\Delta u^1 - \frac{1}{\hat{\epsilon}}V_u(u^0) - V_{uu}(u^0)u^1 \\ &= \frac{2\hat{\epsilon}\Delta_\eta u^0}{\hat{\epsilon}^2} + \frac{2\hat{\epsilon}^2\Delta_\eta u^1}{\hat{\epsilon}^2} - \frac{1}{\hat{\epsilon}}V_u(u^0) - V_{uu}(u^0)u^1 \\ &= \frac{1}{\hat{\epsilon}}\left(2\Delta_\eta u^0 - V_u(u^0)\right) + \hat{\epsilon}^0\left(2\Delta_\eta u^1 - V_{uu}(u^0)u^1\right) \end{aligned}$$

where $\Delta_\eta = \dfrac{\partial^2}{\partial\eta_1^2} + \dfrac{\partial^2}{\partial\eta_2^2}$ is the Laplacian with respect to η. Equating the coefficient of $\dfrac{1}{\hat{\epsilon}}$ to zero in the above equation results in equation 2.24.

$$2\Delta_\eta u^0 = V_u(u^0) \tag{2.24}$$

At the triple junction, x_0, the following condition holds.

$$u^0(\eta_1, \eta_2 = 0) = u^0(x_0)$$

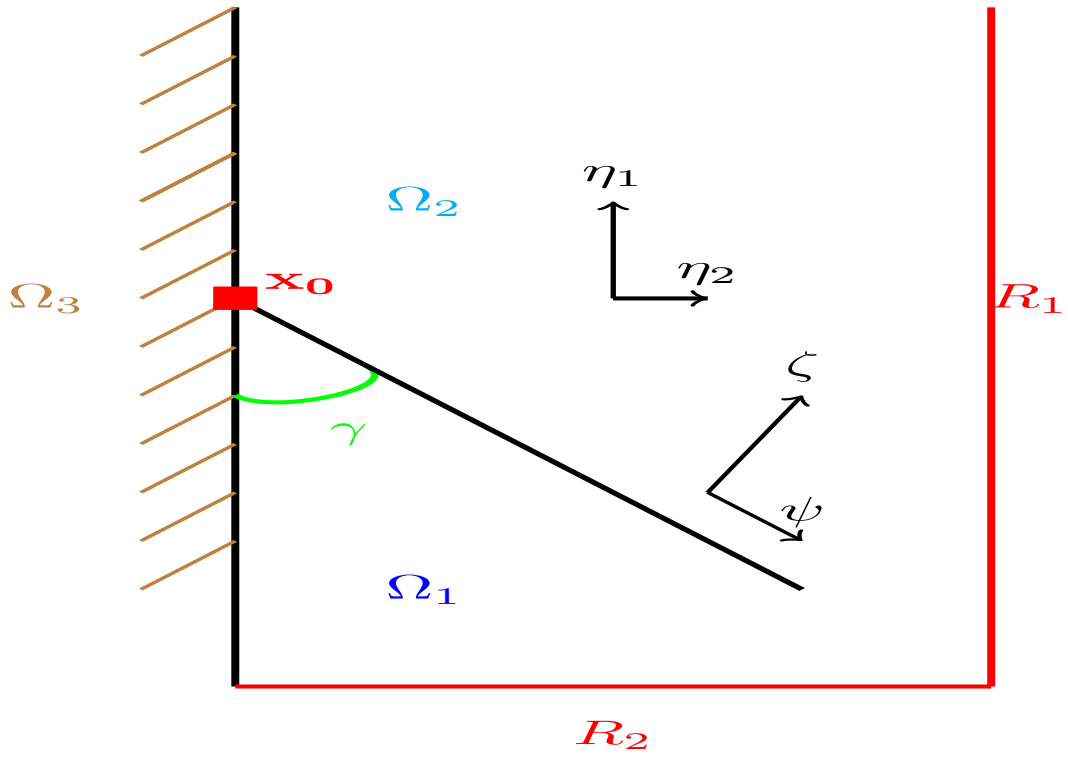


Figure 2.2: Triple junction in $\mathbb{R}^2$ for stationary phase Ω_3

Since the solid phase Ω_3 is stationary, let us denote the *boundary layer* close to the wall Ω_3 and the *transition layer* further from the wall along the ϕ-axis. Both the boundary layer solution denoted u^{bl} and the transition layer solution denoted u^{tl} satisfies equation 2.23. In this case, the so called Van Dyke conditions hold for the solution at hand.

$$\lim_{|\eta_1|\to\infty} u^0(\eta_1,\eta_2) = \lim_{y\to y(x_0)} u_0^{bl}(y,\eta_2) \tag{2.25}$$

$$\lim{}^{\star} u^0(\eta_1,\eta_2) = \lim_{y\to y(x_0)} u_0^{tl}(\psi) \tag{2.26}$$

where $\lim^{\star}$ indicates that $|\eta_1|$ and η_2 tends to infinity along a line at a distance ψ from the interface between the liquid and gas phase. In order to simplify the calculations, it is possible to transform the coordinates of the system from η_1 and η_2 to ζ and ψ by using equation 2.27 and 2.28 respectively.

$$\zeta = \eta_1\cos(\gamma) + \eta_2\sin(\gamma) \tag{2.27}$$

$$\psi = \eta_1\sin(\gamma) - \eta_2\cos(\gamma) \tag{2.28}$$

With these equations in mind, multiply equation 2.24 by $\dfrac{\partial u^0}{\partial \eta_1}$ and integrate over the box D with sides R_1 and R_2 as depicted in figure 2.2.

$$\int_0^{R_2}\int_{-R_1/2}^{R_1/2} V_u(u^0)\frac{\partial u^0}{\partial \eta_1}\,\partial\eta_1\partial\eta_2 = 2\int_0^{R_2}\int_{-R_1/2}^{R_1/2}\left(\frac{\partial^2 u^0}{\partial \eta_1^2}\frac{\partial u^0}{\partial \eta_1}+\frac{\partial^2 u^0}{\partial \eta_2^2}\frac{\partial u^0}{\partial \eta_1}\right)\,\partial\eta_1\partial\eta_2$$

Using the chain rule, the integrands in the above equation can be rewritten. The integrand in the left hand side can be written in the following manner.

$$V_u(u^0)\frac{\partial u^0}{\partial \eta_1} = \frac{\partial}{\partial \eta_1}\left[V(u^0)\right]$$

Similarly the integrand in the right hand side can be rewritten as in the following calculation

$$\begin{aligned}
2\left(\frac{\partial^2 u^0}{\partial \eta_1^2}\frac{\partial u^0}{\partial \eta_1}+\frac{\partial^2 u^0}{\partial \eta_2^2}\frac{\partial u^0}{\partial \eta_1}\right) &= 2\frac{\partial^2 u^0}{\partial \eta_1^2}\frac{\partial u^0}{\partial \eta_1}+2\frac{\partial^2 u^0}{\partial \eta_2^2}\frac{\partial u^0}{\partial \eta_1}\\
&= \left[-2\frac{\partial u^0}{\partial \eta_2}\frac{\partial^2 u^0}{\partial \eta_1\partial \eta_2}+2\frac{\partial^2 u^0}{\partial \eta_1^2}\frac{\partial u^0}{\partial \eta_1}\right]+\left[2\frac{\partial^2 u^0}{\partial \eta_2^2}\frac{\partial u^0}{\partial \eta_1}+2\frac{\partial u^0}{\partial \eta_2}\frac{\partial^2 u^0}{\partial \eta_1\partial \eta_2}\right]\\
&= \frac{\partial}{\partial \eta_1}\left[-\left(\frac{\partial u^0}{\partial \eta_2}\right)^2+\left(\frac{\partial u^0}{\partial \eta_1}\right)^2\right]+2\frac{\partial}{\partial \eta_2}\left[\frac{\partial u^0}{\partial \eta_1}\frac{\partial u^0}{\partial \eta_2}\right]
\end{aligned}$$

and combining these results yield equation 2.29 below.

$$\int_0^{R_2}\int_{-R_1/2}^{R_1/2}\frac{\partial}{\partial \eta_1}\left[V(u^0)+\left(\frac{\partial u^0}{\partial \eta_2}\right)^2-\left(\frac{\partial u^0}{\partial \eta_1}\right)^2\right]\,\mathrm{d}\eta_1\mathrm{d}\eta_2 = 2\int_0^{R_2}\int_{-R_1/2}^{R_1/2}\frac{\partial}{\partial \eta_2}\left[\frac{\partial u^0}{\partial \eta_1}\frac{\partial u^0}{\partial \eta_2}\right]\,\mathrm{d}\eta_1\mathrm{d}\eta_2 \tag{2.29}$$

Performing the integration in both the left and right hand side, results in equation 2.30.

$$\int_0^{R_2}\left[V(u^0)+\left(\frac{\partial u^0}{\partial \eta_2}\right)^2-\left(\frac{\partial u^0}{\partial \eta_1}\right)^2\right]_{-R_1/2}^{R_1/2}\,\partial\eta_2 = 2\int_{-R_1/2}^{R_1/2}\frac{\partial u^0}{\partial \eta_1}\frac{\partial u^0}{\partial \eta_2}\,\partial\eta_1 \tag{2.30}$$

Subsequently, analyze the left hand side and right hand side of equation 2.30 separately. Starting with the left hand side, let the energy minima of V on each side of the interface be denoted a and b respectively, in other words the limits $\lim_{R_1\to-\infty} u^0(\eta_1 = R_1,\eta_2) = u_a^0$ and $\lim_{R_1\to\infty} u^0(\eta_1 = R_1,\eta_2) = u_b^0$ hold. Furthermore, when u^0 attains its minima, the value is stationary so the limit $\lim_{|\eta_1|\to\infty}\dfrac{\partial u^0}{\partial \eta_1} = 0$ must hold, as well as the Van dyke conditions

in equation 2.25 and 2.26. Thus letting $R_1 \to \infty$ and $R_2 \to \infty$ in the left hand side of equation 2.30 results in the following expression.

$$\int_0^\infty V(u_b^0) + \left(\frac{\partial u_b^0(y(x_0),\eta_2)}{\partial \eta_2}\right)^2 \partial\eta_2 - \left(\int_0^\infty V(u_a^0) + \left(\frac{\partial u_a^0(y(x_0),\eta_2)}{\partial \eta_2}\right)^2 \partial\eta_2\right)$$

Now multiplying equation 2.23 by u^0 and integrating with respect to z yields

$$\begin{aligned} 2u_{zz}^0 u_z^0 &= V_u(u^0)u_z^0 \\ \implies \frac{\partial}{\partial z}\left(\left(\frac{\partial u^0}{\partial z}\right)^2\right) &= \frac{\partial V(u^0)}{\partial z} \\ \implies \boxed{\frac{\partial u^0}{\partial z} = \sqrt{V(u^0)}} & \end{aligned}$$

and this results in equation 2.31.

$$\frac{\partial u^0}{\partial z} = \sqrt{V(u^0)} \tag{2.31}$$

Inserting this expression into the previous integral equality yields the following expression.

$$\int_0^\infty V(u_b^0) + V(u_b^0(y(x_0),\eta_2))\ \partial\eta_2 - \left(\int_0^\infty V(u_a^0) + V(u_a^0(y(x_0),\eta_2))\ \partial\eta_2\right)$$

Defining the surface energy between the two phases as $\phi(b) = 2\int_a^b \sqrt{V(t)}\ \mathrm{d}t$ the above expression can be written as equation 2.32[12].

$$-\phi(u^0(x_0)) + (\phi(b) - \phi(u^0(x_0))) \tag{2.32}$$

When it comes to the right hand side of equation 2.30, apply the chain rule in order to change the integration variables to ζ and ψ. The first factor in the integrand can then be written as

$$\begin{aligned} \frac{\partial u^0}{\partial \eta_1} &= \frac{\partial u^0}{\partial \psi}\frac{\partial \psi}{\partial \eta_1} + \frac{\partial u^0}{\partial \zeta}\frac{\partial \zeta}{\partial \eta_1} \\ &= \frac{\partial u^0}{\partial \psi}\sin(\gamma) + \frac{\partial u^0}{\partial \zeta}\cos(\gamma) \end{aligned}$$

and the second factor can be written as

$$\begin{aligned}\frac{\partial u^0}{\partial \eta_2} &= \frac{\partial u^0}{\partial \psi}\frac{\partial \psi}{\partial \eta_2} + \frac{\partial u^0}{\partial \zeta}\frac{\partial \zeta}{\partial \eta_2} \\ &= -\frac{\partial u^0}{\partial \psi}\cos(\gamma) + \frac{\partial u^0}{\partial \zeta}\sin(\gamma)\end{aligned}$$

and consequently the product of the two factors can be calculated as follows.

$$\begin{aligned}\frac{\partial u^0}{\partial \eta_1}\frac{\partial u^0}{\partial \eta_2} &= \left(\frac{\partial u^0}{\partial \psi}\sin(\gamma) + \frac{\partial u^0}{\partial \zeta}\cos(\gamma)\right)\left(-\frac{\partial u^0}{\partial \psi}\cos(\gamma) + \frac{\partial u^0}{\partial \zeta}\sin(\gamma)\right) \\ &= -\left(\frac{\partial u^0}{\partial \psi}\right)^2\cos(\gamma)\sin(\gamma) + \frac{\partial u^0}{\partial \psi}\frac{\partial u^0}{\partial \zeta}\left(\sin^2(\gamma) - \cos^2(\gamma)\right) + \left(\frac{\mathrm{d}u^0}{\mathrm{d}\zeta}\right)^2\sin(\gamma)\cos(\gamma)\end{aligned}$$

Furthermore, when it comes to the integration variable η_1 it follows that

$$\partial \eta_1 = \frac{1}{\sin(\gamma)}\partial \psi$$

and when it comes to the integration limits the following equations hold.

$$\begin{aligned}\psi\left(-\frac{R_1}{2}, R_2\right) &= -\frac{R_1}{2}\sin(\gamma) - R_2\cos(\gamma) \\ \psi\left(\frac{R_1}{2}, R_2\right) &= \frac{R_1}{2}\sin(\gamma) - R_2\cos(\gamma)\end{aligned}$$

Consequently, the right hand side of equation 2.30 in the variables ζ and ψ can be written as in equation 2.33.

$$2\int_{-\frac{R_1}{2}\sin(\gamma)-R_2\cos(\gamma)}^{\frac{R_1}{2}\sin(\gamma)-R_2\cos(\gamma)} -\left(\frac{\partial u^0}{\partial \psi}\right)^2\cos(\gamma) + \frac{\partial u^0}{\partial \psi}\frac{\partial u^0}{\partial \zeta}\left(\sin(\gamma) - \frac{\cos^2(\gamma)}{\sin(\gamma)}\right) + \left(\frac{\mathrm{d}u^0}{\mathrm{d}\zeta}\right)^2\cos(\gamma)\ \ \mathrm{d}\psi \tag{2.33}$$

Taking the limits $\alpha \to \infty$ where $|R_1\sin(\gamma) - R_2\cos(\gamma)| < \alpha$, $R_1 \to \infty$ and $R_2 \to \infty$ in that order. In this case, merely the first term will remain which results in the following calculation

$$\begin{aligned}-\cos(\gamma)\int_{-\infty}^{\infty} 2\left(\frac{\partial u^0}{\partial \psi}\right)^2\ \ \partial\psi &= \{\text{Equation 2.31}\} \\ &= -\cos(\gamma)\phi(b)\end{aligned}$$

And equating this expression with equation 2.32 yields the following expression for the angle γ.

$$\cos(\gamma) = \frac{-\phi(u^0(x_0)) + (\phi(b) - \phi(u^0(x_0)))}{\phi(b)} \tag{2.34}$$

Equation 2.34 is similar to equation Young's equation in capillary flow[12; 13]. The entire evolution of a three phase system with one stationary phase is therefore determined entirely by equation 2.20 and equation 2.34 which is consistent with the analysis by Owen, Rubinstein and Sternberg[12].

Thus, with the above analysis in mind, the framework for a solution algorithm that solves equation 1.1 such that the solution follows equation 2.20 and 2.34 will be described subsequently.

2.2 Solution Algorithm

Subsequently, a solution algorithm to equation 1.1 in the limit $\epsilon \to 0$ will be described. The described algorithm will have two properties. The first property is that the interface between two phases, in for example a two phase system or a three phase system far from the triple junction, should evolve according to its mean curvature as in equation 2.20. Secondly, the algorithm requires a way of dealing with the solutions in proximity of the triple junction. Therefore, an algorithm for generating solutions that evolves according to mean curvature flow will first be presented for a two phase system, and then follows an algorithm for a three phase system where the triple junction is taken into account. The algorithm for the two phase system was originally proposed by Merriman, Bence and Osher[10] and the algorithm describing the evolution of a three phase system was originally introduced by Ruuth[16; 17].

2.2.1 Two phase system

An example of the evolution of the interface between two phases of chemical species, is the evolution of the interface between a droplet of water and the surrounding air as illustrated in figure 2.3. A spherical domain has been chosen due to the fact that there is an analytical solution to the surface evolution of the interface, and this solution can be compared to the implementation of the algorithm in order to determine its validity. With this figure in mind, the problem can be formulated mathematically. In figure 2.3 the set $\Omega \subset \mathbb{R}^3$ is defined in the following way.

$$\Omega := \left\{ \mathbf{x} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \in \mathbb{R}^3 : x \in [-1,1], y \in [-1,1], z \in [-1,1] \right\}$$

Hence, Ω is a cube with side length 2 centered at the origin and in this case the boundary of Ω, denoted $\partial\Omega$, is defined in the following way.

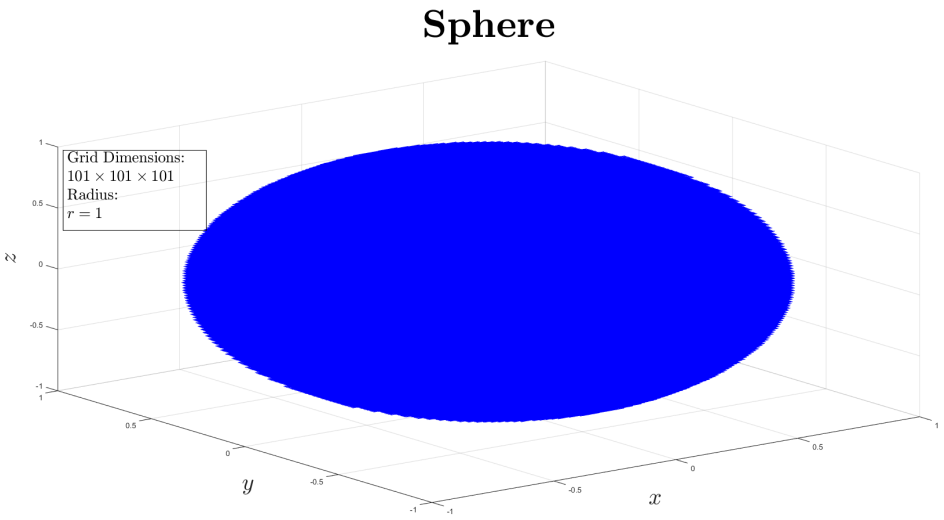


Figure 2.3: Water droplet. The water spherical water droplet has radius 1 and is trapped in a cube with side length 2. Each side is divided 101 times in order to define a three dimensional mesh.

$$\partial\Omega := \{x \in [-1,1], y \in [-1,1], z \in \{-1,1\}\} \cap \{x \in [-1,1], y\,\{-1,1\}\,, z \in [-1,1]\}$$
$$\cap\,\{x \in \{-1,1\}\,, y \in [-1,1], z \in [-1,1]\}$$

For simpicity, assume that the water phase consists of a sphere of radius 1 as defined by set S below.

$$S := \left\{ \mathbf{x} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \in \Omega : x^2 + y^2 + z^2 \leq 1 \right\}$$

Furthermore, let the *characteristic function of a set* $C \subset \mathbb{R}^3$, denoted $\chi_C(\mathbf{x})$, be defined in the following way.

$$\chi_C(\mathbf{x}) = \begin{cases} 1 & , \mathbf{x} \in C \\ 0 & , \mathbf{x} \notin C \end{cases}$$

With these notations in mind it is possible to construct a *reaction diffusion equation*[10] that describes how the water droplet evolves over time. Therefore, define the function $u : \Omega \times \mathbb{R}_+ \to [0,1]$ which the denotes the concentration of water given a position in space, i.e. $\mathbf{x} \in \Omega \subset \mathbb{R}^3$, and a time, i.e. $t \in \mathbb{R}_+$. The value $u(\mathbf{x},\cdot) = 1$ corresponds to a position *within* the water phase, the value $u(\mathbf{x},\cdot) = 0$ corresponds to a position *outside*

the water phase, that is in the other phase, and the value $u(\mathbf{x},\cdot) = \frac{1}{2}$ corresponds to a position on the *interface* between the two phases. In this case boundary value problem describing the evolution of the surface of the water droplet above can be described as in equation 2.35.

$$\begin{cases} \frac{\partial u}{\partial t} & = \epsilon \Delta u - \frac{1}{\epsilon} f(u) & \text{(PDE)} \\ \frac{\partial u(\mathbf{x},t)}{\partial \mathbf{n}} & = 0 \quad ,x \in \partial\Omega,\, t \in \mathbb{R}_+ & \text{(BC)} \\ u(\mathbf{x},0) & = \chi_S(\mathbf{x}) \quad ,x \in \Omega & \text{(IC)} \end{cases} \tag{2.35}$$

In equation 2.35 the abbreviations are "PDE=Partial Differential Equation", "BC=Boundary Conditions" and "IC=Initial Condition". Note that in equation 2.35, Neumann boundary conditions have been implemented, which physically can be interpreted as no material leaves the domain.

In the PDE in equation 2.35, the term $\epsilon\Delta u$ is the so called *diffusion term* and the term $\frac{1}{\epsilon} f(u)$ is the *reaction term* where the function $f : \Omega \times \mathbb{R}_+ \to [0,1]$ in the case of two phases will be defined by equation 2.36[10].

$$f(u) = u\left(u - \frac{1}{2}\right)(u-1) \tag{2.36}$$

The scalar $\epsilon \in (0,\infty)$ is a *small* constant that renders the contribution of the reaction term much more significant than the diffusion term after long time, and thus diffusion plays an important role for merely short time intervals. In order to predict the long term behaviour of the solution to equation 2.35, a linear stability analysis can be conducted.

Linear stability analysis

The solution algorithm for equation 2.35 relies on the fact that the reaction term is substantially larger than the diffusion term, due to the factor $\epsilon > 0$. This fact implies that the overall dynamics of the system can be approximated by the following *ordinary differential equation*, abbreviated ODE, with time $t \in \mathbb{R}_+$ as the only variable.

$$\frac{\mathrm{d}u}{\mathrm{d}t} = -f(u(t)), \quad t \in \mathbb{R}_+, \quad u(0) = C \in (0,1], \quad \text{(IC)} \tag{2.37}$$

Note equation 2.37 is the analogous expression for a two phase system to equation 2.8 on page 10 derived in section 2.1.1 in the case of a three phase system. Now, a classical linear stability analysis will be conducted on equation 2.37. A *fix point*, denoted $u(t^\star) = u_\star$, of equation 2.37 is a point that satisfies the following condition.

$$\frac{\mathrm{d}u_\star}{\mathrm{d}t} = -f(u_\star) = 0 \iff f(u_\star) = 0$$

By equation 2.36, the system has the following three fix points.

$$u_{1\star} = 0$$
$$u_{2\star} = \frac{1}{2}$$
$$u_{3\star} = 1$$

In fact the dynamics of the system in equation 2.37 will be determined by the stability of these fix points. In order to determine the stability of these fix points, a first order Taylor expansion of $f(u)$ in the vicinity of a fix point $u_\star$ can be calculated

$$\begin{aligned} f(u) &\approx \underbrace{f(u_\star)}_{=0} + \left(\left. \frac{\mathrm{d}f}{\mathrm{d}u} \right|_{u=u_\star} \right) (u - u_\star) + \text{H.O.T} \\ &\{\text{Linearization} \Rightarrow \text{H.O.T}=\text{"Higher Order Terms" are neglected}\} \\ &\approx \left(\left. \frac{\mathrm{d}f}{\mathrm{d}u} \right|_{u=u_\star} \right) (u - u_\star) \end{aligned}$$

and thus the linearization of $f(u)$ around $u_\star$ is given by the following equation.

$$f(u) \approx \left(\left. \frac{\mathrm{d}f}{\mathrm{d}u} \right|_{u=u_\star} \right) (u - u_\star)$$

In order to proceed, it is possible to define $u(t) = u_\star + \delta u(t)$ where $\delta u(t) > 0$ is a small perturbation of the fix point $u_\star$. Plugging this expression for u into the above approximation of $f(u)$ yields.

$$\begin{aligned} f(u) = f(u^\star + \delta u) &\approx \left(\left. \frac{\mathrm{d}f}{\mathrm{d}u} \right|_{u=u_\star} \right) (u_\star + \delta u - u_\star) \\ &= \left(\left. \frac{\mathrm{d}f}{\mathrm{d}u} \right|_{u=u_\star} \right) \delta u \\ \Longrightarrow & \boxed{f(u^\star + \delta u) \approx \left(\left. \frac{\mathrm{d}f}{\mathrm{d}u} \right|_{u=u_\star} \right) \delta u} \end{aligned}$$

Further, using equation 2.37 yields the following result

$$
\begin{aligned}
f(u^\star + \delta u) = -\frac{\mathrm{d}\left(u_\star + \delta u\right)}{\mathrm{d}t} &= -\left(\underbrace{\frac{\mathrm{d}u_\star}{\mathrm{d}t}}_{=0} + \frac{\mathrm{d}\delta u}{\mathrm{d}t}\right) \\
&= -\frac{\mathrm{d}\delta u}{\mathrm{d}t} = f(\delta u) \\
\Longrightarrow &\boxed{f(u^\star + \delta u) = f(\delta u)}
\end{aligned}
$$

and thus the following *linearization* of $f(u)$ holds in the vicinity of a fix point $u_\star$.

$$
f(\delta u) \approx \left(\left.\frac{\mathrm{d}f}{\mathrm{d}u}\right|_{u=u_\star}\right)\delta u \tag{2.38}
$$

Combining equation 2.37 and equation 2.38 yields the following approximation of the ODE in the vicinity of $u_\star$, i.e. where $u \approx u_\star$.

$$
\frac{\mathrm{d}\delta u}{\mathrm{d}t} \approx -\left(\left.\frac{\mathrm{d}f}{\mathrm{d}u}\right|_{u=u_\star}\right)\delta u \tag{2.39}
$$

The solution of equation 2.39 is given by

$$
\delta u(t) = \delta u(0) \cdot \exp\left(-\left.\frac{\mathrm{d}f}{\mathrm{d}u}\right|_{u=u_\star} \cdot t\right) \tag{2.40}
$$

A *stable* fix point is a point where the perturbation around a fix point disappears and hence satisfies the following condition.

$$
\lim_{t\to\infty} \delta u(t) = 0
$$

An *unstable* fix point is a point where the perturbation around a fix point blows up and hence satisfies the following condition.

$$
\lim_{t\to\infty} \delta u(t) = \infty
$$

Translating these stability conditions using equation 4.2 yields the following equations.

$$
\begin{aligned}
\left.\frac{\mathrm{d}f}{\mathrm{d}u}\right|_{u=u_\star} &> 0 \Longrightarrow \ u_\star \text{ is } \textit{stable} \\
\left.\frac{\mathrm{d}f}{\mathrm{d}u}\right|_{u=u_\star} &< 0 \Longrightarrow \ u_\star \text{ is } \textit{unstable}
\end{aligned}
$$

Hence for the definition of $f(u)$ for two phases, see equation 2.36, it follows that the derivative is given by

$$\frac{\mathrm{d}f}{\mathrm{d}u} = \left(u - \frac{1}{2}\right)(u-1) + u(u-1) + u\left(u - \frac{1}{2}\right)$$

and thus the following results follows.

$$\begin{aligned}
\left.\frac{\mathrm{d}f}{\mathrm{d}u}\right|_{u=0} &= \left(-\frac{1}{2}\right)(-1) = \frac{1}{2} > 0 \Longrightarrow \text{stable} \\
\left.\frac{\mathrm{d}f}{\mathrm{d}u}\right|_{u=\frac{1}{2}} &= \frac{1}{2}\left(-\frac{1}{2}\right) = -\frac{1}{4} < 0 \Longrightarrow \text{unstable} \\
\left.\frac{\mathrm{d}f}{\mathrm{d}u}\right|_{u=1} &= \frac{1}{2} > 0 \Longrightarrow \text{stable}
\end{aligned}$$

Hence, the system has two stable fix points at $u_{1\star} = 0$ and $u_{3\star} = 1$ and one unstable fix point at $u_{2\star} = \frac{1}{2}$. This implies that after a certain time when the reaction term determines the dynamics of the system, solution u will move towards the stable fix points and away from the unstable fix point. Thus, given a certain position $\mathbf{x} \in \Omega$ and a sufficiently large time $t \in \mathbb{R}_+$ the solution $u(\mathbf{x},t)$ will quickly approach the value 1 if $u(\mathbf{x},t) > \frac{1}{2}$ and conversely $u(\mathbf{x},t)$ will quickly approach the value 0 if $u(\mathbf{x},t) < \frac{1}{2}$. Two stable fix points with an intermediate unstable fix point is indicative of a *travelling wave*, and in fact this observation agree with the results derived in section 2.1.2 on page 11. Furthermore, the motion of interface between two phases in equation 2.20 is described by *mean curvature* of that interface. This equation holds also for a two phase system, and therefore a solution algorithm for 2.35 should generate an interface which motion is described by the mean curvature of the interface.

In the table below, the solution algorithm for a two phase system proposed by Merriman, Bence and Osher[10] is stated.The algorithm relies on the fact that the system in equation 2.35 has a *sharp* phase transition, and can be splitted into two steps, namely a *diffusion* and a *sharpening* step. Therefore, the algorithm in question can be referred to as a *splitting* algorithm and it can be described as follows.

Solution algorithm for two phases	
Initialization step	Initialize the function $u(\mathbf{x},t)$. In the case of a sphere let $u(\mathbf{x},0) = \chi_S(\mathbf{x})$ as in equation 2.35.
Diffusion step	For a certain time $\Delta\tau$, solve the diffusion equation $\frac{\mathrm{d}u}{\mathrm{d}t} = \Delta u$ with the given boundary conditions.
Sharpening step	After a time $\Delta\tau$, "sharpen" the diffused function by setting $u(\mathbf{x},t) = \begin{cases} 1, & \text{if } u(\mathbf{x},t) > \frac{1}{2} \\ 0, & \text{if } u(\mathbf{x},t) < \frac{1}{2} \end{cases}$. Go back to the diffusion step.

Furthermore, the solution algorithm for a two phase system generates an interface which evolves according to its mean curvature. In fact, the speed of the interface at a point x_0 is given by $v = H + O(\sqrt{t})$ as $t \to \infty$ where H is the mean curvature of the interface at the point x_0. This fact has been definitely proven by Evans, see theorem 4.1 on page 546, which can be found in [4]. An illustrative example of mean curvature flow is the evolution of the interface between the two phases in equation 2.35, and this situation is commonly known as the "Shrinking sphere".

Shrinking sphere

Suppose the evolution of the interface between the two phases in equation 2.35 on page 24 is to be derived. The interface $\Gamma(t) := \{\mathbf{x} \in S : \phi(\mathbf{x},t) = 0\}$ is described by the level sets of the function $\phi(\mathbf{x},t) = x^2+y^2+z^2-r(t)^2$. By the definition of the mean curvature H, see equation 2.4 on page 7, the mean curvature is given by the following equation

$$H = \operatorname{div}\left(\frac{\nabla\phi}{|\nabla\phi|}\right)$$

where $\nabla\phi = \Big(2x, 2y, 2z\Big)^T$ and $|\nabla\phi| = 2\sqrt{x^2+y^2+z^2}$. Then, it follows that

$$H = \frac{\partial}{\partial x}\left(\frac{x}{\sqrt{x^2+y^2+z^2}}\right) + \frac{\partial}{\partial y}\left(\frac{y}{\sqrt{x^2+y^2+z^2}}\right) + \frac{\partial}{\partial z}\left(\frac{z}{\sqrt{x^2+y^2+z^2}}\right)$$

and by the product rule it follows that the first term is given by

$$\begin{aligned}\frac{\partial}{\partial x}\left(\frac{x}{\sqrt{x^2+y^2+z^2}}\right) &= \frac{\partial}{\partial x}\left(x\cdot(x^2+y^2+z^2)^{-\frac{1}{2}}\right)\\ &= \frac{1}{\sqrt{x^2+y^2+z^2}} - \frac{x^2}{\left(\sqrt{x^2+y^2+z^2}\right)^3}\\ &= \frac{x^2+y^2}{(\sqrt{x^2+y^2+z^2})^3}\end{aligned}$$

and the other two terms are calculated analogously. Thus the mean curvature is given by the following calculation.

$$\begin{aligned}H &= \frac{2(x^2+y^2+z^2)}{(\sqrt{x^2+y^2+z^2})^3}\\ &= \frac{2}{\sqrt{x^2+y^2+z^2}}\\ &= \frac{2}{r(t)}\end{aligned}$$

It follows that for an n-dimensional sphere, the surface of the sphere has the following mean curvature.

$$H = \frac{n-1}{r(t)}$$

The radius of the surface of the sphere will decrease with a speed of the same size as the mean curvature. Hence the evolution of the radius satisfies the following ordinary differential equation[18].

$$\begin{cases} \dfrac{\mathrm{d}r}{\mathrm{d}t} = -\dfrac{(n-1)}{r(t)} & ,(ODE) \\ r(0) = r_0 & ,(IC) \end{cases} \tag{2.41}$$

Hence the solution is obtained by this simple calculation,

$$\begin{aligned} \frac{\mathrm{d}r}{\mathrm{d}t} &= -\frac{n-1}{r} \\ r\mathrm{d}r &= -(n-1)\mathrm{d}t \\ \int_{r_0}^{r(t)} r\mathrm{d}r &= -(n-1)\int_0^t \mathrm{d}t \\ \frac{1}{2}\left(r(t)^2 - r_0^2\right) &= -(n-1)t \\ \implies r(t) &= \sqrt{r_0^2 - 2(n-1)t} \end{aligned}$$

and the evolution of the radius of an n-dimensional sphere is therefore given by equation 2.42.

$$r(t) = \sqrt{r_0^2 - 2(n-1)t} \tag{2.42}$$

Equation 2.42 can be used in order to obtain an error estimate of an implemented numerical solution. By using the solution algorithm on page 28 in order to solve equation 2.35 on page 24 numerically, the numerical radius of the shrinking sphere at a time t can be found by identifying the interface $\Gamma(t) := \left\{\mathbf{x}_0 \in S : u(\mathbf{x_0},t) = \frac{1}{2}\right\}$ using linear interpolation and then calculating the radius by calculating the mean radius denoted $\overline{r}(t)$ of all radii given by $r_0(t) = |\mathbf{x}_0(t)| \forall \mathbf{x}_0(t) \in \Gamma(t)$. The mean radius $\overline{r}(t)$ can then be compared to the analytical value, $r(t)$ in equation 2.42, in order to obtain an error estimate, given by $e(t) = r(t) - \overline{r}(t)$, for the numerical solution.

Subsequently, the theory for a three phase system is described.

2.2.2 Three Phase system

When it comes to a three phase system, *the sharpening step* in the solution algorithm on page 28 differs slightly while the diffusion step is identical. However, using the

chemical condition in equation 2.5 on page 8, a second algorithm can be derived in order to conduct the sharpening step for a three phase system. Hence here follows a short motivation behind the so called *projection triangle algorithm* which is applied in order to conduct the sharpening step for a three phase system.

Projection triangle

A graphical representation of the solution $\mathbf{u}(\mathbf{x},t)$ to equation 1.1 is illustrated in figure 2.4. Due to the mass conservation condition in equation 2.5 on page 8, the phase portrait of the solutions to equation 1.1 on page 2 are confined to a triangle which is trivially represented in $\mathbb{R}^3$.

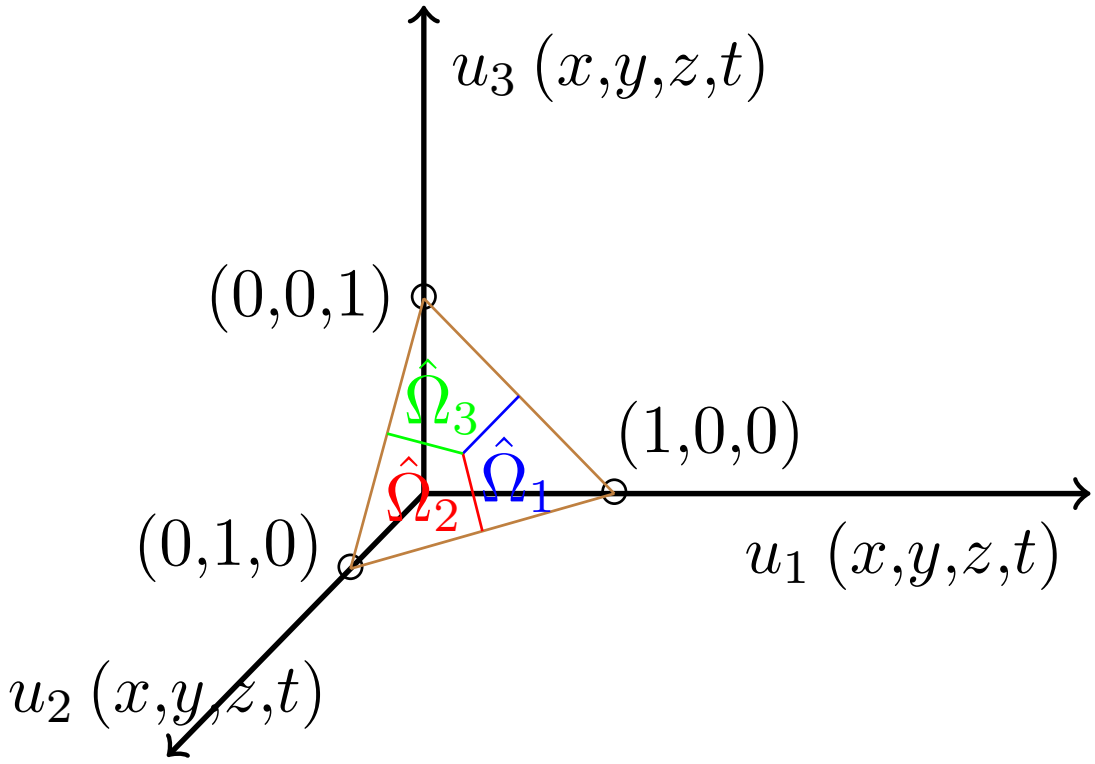


Figure 2.4: Sharpening triangle for three phases in $\mathbb{R}^3$.

Hence, the chemical condition in equation 2.5 on page 8 enables a simple way of representing solutions to equation 1.1 graphically. Since the diffusion step is identical for both a two phase and a three phase possible, it is possible to project all solutions for a three phase system to the heat equation after one time step $\Delta\tau$ onto a projection triangle. This projection triangle can be illustrated in $\mathbb{R}^2$, which is done in figure 2.5. The projection triangle is used in order to conduct the sharpening step for a three phase system, and physically it corresponds to the role of the gradient of the energy potential denoted V_u in equation 1.1 on page 2.

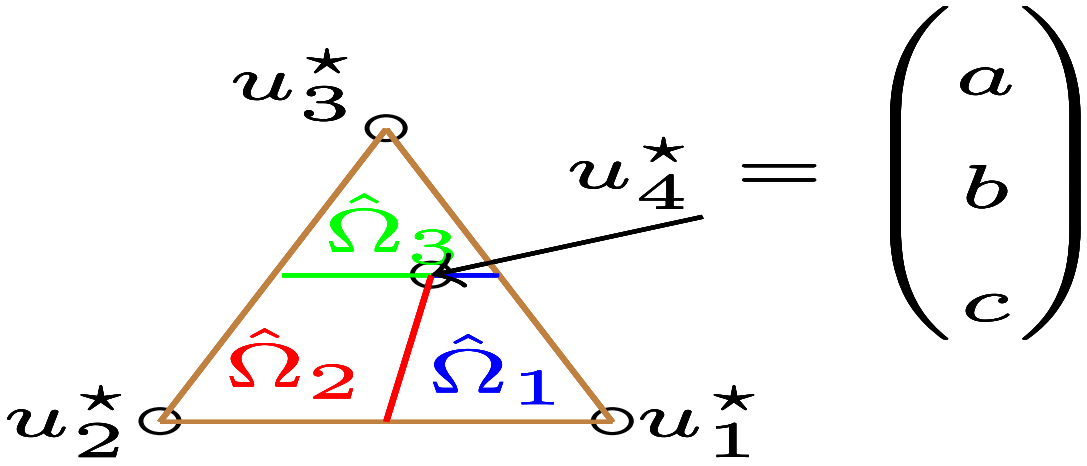


Figure 2.5: Sharpening triangle for three phases in $\mathbb{R}^2$

The projection triangle determines the sharpening step for a three phase system in the following way. If the solution to the heat equation $\mathbf{u}(\mathbf{x},t)$ after a time $\Delta\tau$ given a position $\mathbf{x} \in \Omega$ and time $t \in \mathbb{R}_+$, has a value that falls in the domain $\hat{\Omega}_3$ in the projection triangle, see the green domain in figure 2.5, the value of $\mathbf{u}(\mathbf{x},t)$ is set to $\mathbf{u}_3^\star = \left(0,0,1\right)^T$. Similarly, if $\mathbf{u}(\mathbf{x},t) \in \hat{\Omega}_2$ after a time $\Delta\tau$ the value of $\mathbf{u}(\mathbf{x},t)$ is set to $\mathbf{u}_2^\star = \left(0,1,0\right)^T$ and if $\mathbf{u}(\mathbf{x},t) \in \hat{\Omega}_1$ after a time $\Delta\tau$ the value of $\mathbf{u}(\mathbf{x},t)$ is set to $\mathbf{u}_1^\star = \left(1,0,0\right)^T$. Hence the key step is to find the interfaces between the three phases denoted $\hat{\Gamma}_{12}$ represented by the red line in figure 2.5, $\hat{\Gamma}_{13}$ represented by the blue line in figure 2.5 and $\hat{\Gamma}_{23}$ represented by the green line in figure 2.5. These lines need to fulfill two conditions, which are listed below. The first condition implies that for positions far from the triple junction, the evolution of the three phase system is reduced to the two phase system. Hence, in these cases, the interfaces Γ_{12}, Γ_{13} and Γ_{23} will evolve with a speed proportional to their mean curvature. The second condition will be derived later in chapter 4 called "Convolution thresholding".

Condition 1 In the case where $\mathbf{u}_i = 0$ for $i \in \{1,2,3\}$ the sharpening step is reduced to the two phase case. In other words, the red line $\hat{\Gamma}_{12}$ goes through the point $\left(\frac{1}{2},\frac{1}{2},0\right)^T$, the green line $\hat{\Gamma}_{23}$ goes through the point $\left(0,\frac{1}{2},\frac{1}{2}\right)^T$ and the blue line $\hat{\Gamma}_{13}$ goes through the point $\left(\frac{1}{2},0,\frac{1}{2}\right)^T$.

Condition 2 The three lines $\hat{\Gamma}_{12}$, $\hat{\Gamma}_{13}$ and $\hat{\Gamma}_{23}$ all pass through the point $u_4^\star$ which is *called* the *triple junction*. In particular when the solid phase is stationary, and has a contact angle of π the triple junction will be given by $u_4^\star = \left(\frac{\alpha}{2\pi}, \frac{\pi-\alpha}{2\pi}, \frac{1}{2}\right)^T$ where α is the contact angle between the liquid phase and the solid phase.

The so called *projection triangle algorithm* proposed by Ruuth in [16] can be used in order to generate a projection triangle as depicted in figure 2.5.

Projection Triangle Algorithm	
Find position for interfaces	Let Ω_3 denote the solid phase, let Ω_2 denote the gas phase and let Ω_1 denote the liquid phase. Begin by defining the sets $\Gamma_{12} := \{\mathbf{x} \in \Omega : \mathbf{x} \in \Omega_1 \cap \Omega_2\}$, $\Gamma_{13} := \{\mathbf{x} \in \Omega : \mathbf{x} \in \Omega_1 \cap \Omega_3\}$ and $\Gamma_{23} := \{\mathbf{x} \in \Omega : \mathbf{x} \in \Omega_2 \cap \Omega_3\}$.
Initialisation step	Initialize the the functions in the following way: $u_1(\mathbf{x},0) = \chi_1(\mathbf{x})$, $u_2(\mathbf{x},0) = \chi_2(\mathbf{x})$ and $u_3(\mathbf{x},0) = \chi_3(\mathbf{x})$.
Diffusion step	For a certain time $\Delta\tau$, solve the diffusion equation $\frac{\mathrm{d}u_i}{\mathrm{d}t} = \Delta u_i$ for each phase $i \in \{1,2,3\}$ with the given boundary conditions.
Projection triangle	After a time $\Delta\tau$, define the lines $\hat{\Gamma}_{12} := \left\{\mathbf{u}(\mathbf{x},\Delta\tau) = \left(u_1(\mathbf{x},\Delta\tau), u_2(\mathbf{x},\Delta\tau), u_3(\mathbf{x},\Delta\tau)\right)^T, \mathbf{x} \in \Gamma_{12}\right\}$, $\hat{\Gamma}_{13} := \left\{\mathbf{u}(\mathbf{x},\Delta\tau) = \left(u_1(\mathbf{x},\Delta\tau), u_2(\mathbf{x},\Delta\tau), u_3(\mathbf{x},\Delta\tau)\right)^T, \mathbf{x} \in \Gamma_{13}\right\}$ and $\hat{\Gamma}_{23} := \left\{\mathbf{u}(\mathbf{x},\Delta\tau) = \left(u_1(\mathbf{x},\Delta\tau), u_2(\mathbf{x},\Delta\tau), u_3(\mathbf{x},\Delta\tau)\right)^T, \mathbf{x} \in \Gamma_{23}\right\}$. The lines $\hat{\Gamma}_{12}$, $\hat{\Gamma}_{13}$ and $\hat{\Gamma}_{23}$ define the red, the blue and the green lines in figure 2.5.

Thus, the lines $\hat{\Gamma}_{12}$, $\hat{\Gamma}_{13}$ and $\hat{\Gamma}_{23}$ in the projection triangle algorithm corresponds to the function values of the solutions u_1, u_2 and u_3 to the heat equation after a time $\Delta\tau$ at the positions $\mathbf{x} \in \Gamma_{12}$, $\mathbf{x} \in \Gamma_{13}$ and $\mathbf{x} \in \Gamma_{23}$ respectively. Moreover, in the case where the solid phase Ω_3 has an angle of π with the other phases, the lines $\hat{\Gamma}_{13}$ and $\hat{\Gamma}_{23}$ will

form a T-shape in the projection triangle[16], which in the case of a straight wall will preserve the solid phase throughout the simulations. Combining the projection triangle algorithm on page 33 and the solution algorithm for two phases on page 28, a general algorithm for simulating capillary flow for a three phase system can be formulated.

Solution algorithm for a three phase system	
Projection triangle	Given a contact angle as depicted in for example figure 3.11, use the projection triangle algorithm on page 33 in order to construct a projection triangle.
Initialisation step	Given a new domain $\Omega = \Omega_1 \cup \Omega_2 \cup \Omega_3$, initialize the the functions in the following way: $u_1(\mathbf{x},0) = \chi_1(\mathbf{x})$, $u_2(\mathbf{x},0) = \chi_2(\mathbf{x})$ and $u_3(\mathbf{x},0) = \chi_3(\mathbf{x})$.
Diffusion step	For a certain time $\Delta\tau$, solve the diffusion equation $\frac{\mathrm{d}u_i}{\mathrm{d}t} = \Delta u_i$ for each phase $i \in \{1,2,3\}$ with the given boundary conditions.
Sharpening step	"Sharpen" the diffused region by using the projection triangle generated in the projection triangle step. Go back to the diffusion step using the reinitialized functions u_1, u_2 and u_3 obtained from the projection triangle.

Using the solution algorithm above, capillary flow for a three phase system can be simulated. Note however that it is *not* evident that the projection triangle algorithm generates a physically valid sharpening step. Therefore, a so called convolution thresholding analysis will be conducted so that a sharpening step that generates physically valid results can be implemented. Such a convolution thresholding calculation is presented in chapter 4. First however, follows an implementation of the solution algorithm for a three phase system presented above.

3

Implementation of algorithm

The implementation of the solution algorithm on page 34 is divided into two parts. The first parts involves implementing the *diffusion step* and the second part includes implementing the *sharpening step*. In order to implement the diffusion step, the two phase system known as the "shrinking sphere" will be simulated using the two phase algorithm formulated on page 28. The sharpening step will be implemented using the projection triangle algorithm on page 33.

3.1 Diffusion step

In order to implement the two phase solution algorithm, a finite difference approach is employed. One of the reasons why a finite difference approach is employed with a stationary mesh, is due to the fact that it is fairly easy to implement as oppose to using for example an adaptive mesh as in the the thesis by Ruuth[17]. Consequently, the following subsection contains a derivation of the finite difference implementation in order to solve the heat equation numerically. Then follows to subsections with implementations of the algorithm in two examples using two and three phases respectively.

In order to solve differential equations numerically, derivatives can be approximated by differences. In order to conduct this approximation, the continuous time line into *discrete* time points and the continuous domain $\Omega \subset \mathbb{R}^d$, $d \in \{2,3\}$ is divided into discrete spatial points by constructing a *grid*. The illustration of the discretization of the time line and the spatial domain is presented in figure 3.1. A high grid size results in a small distance between the nodes which yields a better approximation of the derivatives.

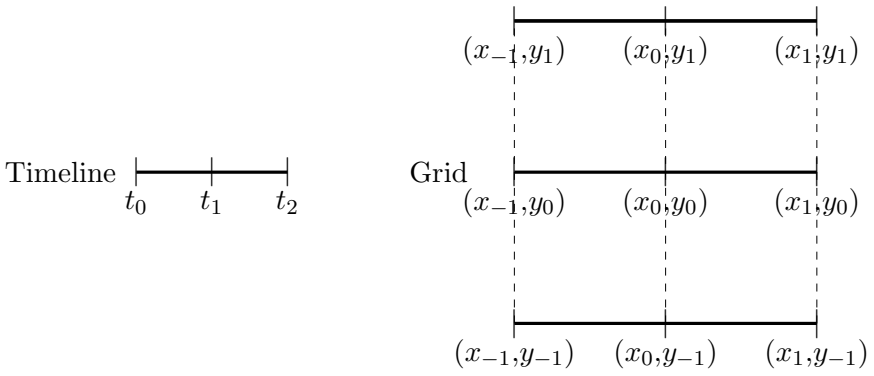


Figure 3.1: Discretization. The left figure shows a discretization of a continuous time line while the right figure shows a discretization of a part of a domain $\Omega \subset \mathbb{R}^2$ by constructing a *grid*.

Consequently, using the finite difference approach, differential equations are approximated by difference equations. In the case of the heat equation this discretization concerns rewriting the time derivative on the one hand and the Laplace operator on the other hand. A detailed explanation of the discretization of the time and spatial derivatives respectively is provided next.

Time discretization

In order to solve the partial differential equation $\dfrac{\partial u}{\partial t} = \Delta u$ numerically the time derivative in the left hand side and the Laplace operator in the right hand side requires to be approximated using finite differences. The time derivative will be approximated using an Euler forward scheme, and how both finite differences are approximated is described in more detail next.

Let $u(\mathbf{x},t)$ be the solution to the equation $\dfrac{\partial u}{\partial t} = \Delta u$. Then the Laplace operator in the right hand side is given by

$$\frac{\mathrm{d}u}{\mathrm{d}t} = \Delta u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \tag{3.1}$$

and the time derivative in the left hand side can be discretized using the following equation.

$$\frac{\mathrm{d}u}{\mathrm{d}t} \approx \frac{u(\mathbf{x},t+\Delta t) - u(\mathbf{x},t)}{\Delta t}$$

Using this notation and plugging this expression into the equation $\dfrac{\partial u}{\partial t} = \Delta u$ yields the following expression.

$$u(x,y,z,t+\Delta t) \approx u(x,y,z,t) + \Delta t \cdot \left(\left.\frac{\partial^2 u}{\partial x^2}\right|_{\mathbf{x},t} + \left.\frac{\partial^2 u}{\partial y^2}\right|_{\mathbf{x},t} + \left.\frac{\partial^2 u}{\partial z^2}\right|_{\mathbf{x},t} \right)$$

Now, a discretization of each second deriviative is required. The discretezation of a certain derivative at a point $\mathbf{x}$ and a time t is calculated below.

$$\begin{aligned}
\left.\frac{\partial^2 u}{\partial x^2}\right|_{\mathbf{x},t} &\approx \frac{\left(\left.\frac{\partial u}{\partial x}\right|_{x+\Delta x,y,z,t} - \left.\frac{\partial u}{\partial x}\right|_{x,y,z,t} \right)}{\Delta x} \\
&\approx \frac{\left(\frac{u(x+\Delta x,y,z,t)-u(x,y,z,t)}{\Delta x} - \left(\frac{u(x,y,z,t)-u(x-\Delta x,y,z,t)}{\Delta x} \right) \right)}{\Delta x} \\
&= \frac{u(x+\Delta x,y,z,t) - 2u(x,y,z,t) + u(x-\Delta x,y,z,t)}{(\Delta x)^2}
\end{aligned}$$

In summary, the three second derivatives can be summarized in the following three expressions.

$$\begin{aligned}
\left.\frac{\partial^2 u}{\partial x^2}\right|_{\mathbf{x},t} &\approx \frac{u(x+\Delta x,y,z,t) - 2u(x,y,z,t) + u(x-\Delta x,y,z,t)}{(\Delta x)^2} \\
\left.\frac{\partial^2 u}{\partial y^2}\right|_{\mathbf{x},t} &\approx \frac{u(x,y+\Delta y,z,t) - 2u(x,y,z,t) + u(x,y-\Delta y,z,t)}{(\Delta y)^2} \\
\left.\frac{\partial^2 u}{\partial z^2}\right|_{\mathbf{x},t} &\approx \frac{u(x,y,z+\Delta z,t) - 2u(x,y,z,t) + u(x,y,z-\Delta z,t)}{(\Delta z)^2}
\end{aligned}$$

Proceed by defining a uniform mesh with step size $h = \Delta x = \Delta y = \Delta z$. Further define the *discrete Laplacian of* u, denoted $\hat{\Delta} u$ as in equation 3.2.

$$\begin{aligned}
\hat{\Delta} u(x,y,z,t) \approx\ & u(x+h,y,z,t) + u(x-h,y,z,t) \\
& + u(x,y+h,z,t) + u(x,y-h,z,t) \\
& + u(x,y,z+h,t) + u(x,y,z-h,t) \\
& - 6u(x,y,z,t)
\end{aligned} \tag{3.2}$$

Using the discrete Laplace operator defined in equation 3.2 the following Euler forward method is obtained for solving the diffusion equation $\dfrac{\partial u}{\partial t} = \Delta u$.

$$u(x,y,z,t+\Delta t) = u(x,y,z,t) + \frac{\Delta t}{h^2} \cdot \hat{\Delta} u \tag{3.3}$$

The discretized Laplace operator $\hat{\Delta}u$ above is referred to as a *stencil*, and there are various stencils in order to approximate the Laplace operator numerically which are explained later in more detail. Subsequently, the value of the ratio $\frac{\Delta t}{\Delta h^2}$ is tested numerically.

Test of appropriate time step In order to obtain reliable results for the finite difference implementation, the ratio $\frac{\Delta t}{h^2}$ in equation 3.3 must be appropriately chosen. Therefore, numerical simulations of the heat equation in $\mathbb{R}^2$ for a circular domain and for a sphere in $\mathbb{R}^3$ using the discrete Laplace operator in equation 3.2 and the corresponding Laplace operator in two dimensions were conducted.

The results of the simulations in two dimensions are illustrated in figure 3.2 and 3.3 below. The diffusion step works correctly if the concentration profile is spread or *diffused* smoothly at the edges of the circle.

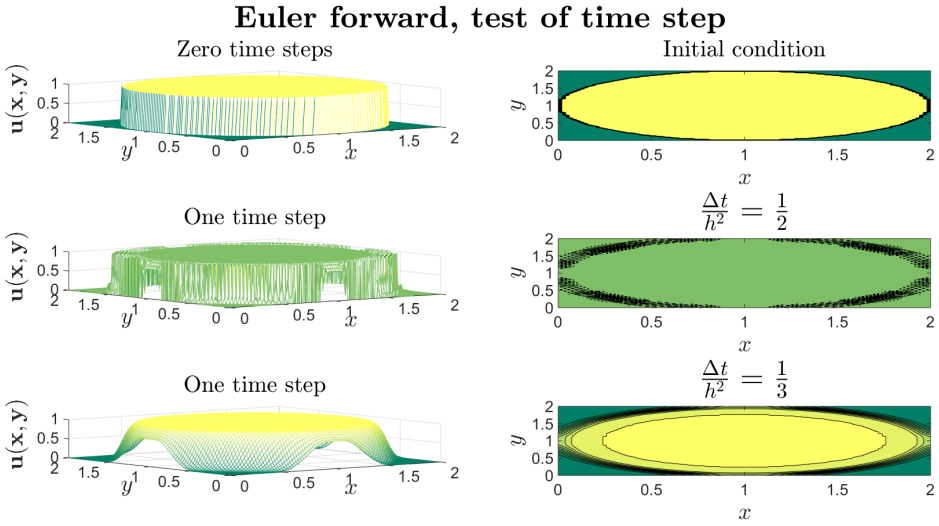


Figure 3.2: Test of time step using ratio $\frac{\Delta t}{h^2} = \frac{1}{2}$ & $\frac{1}{3}$. The grid size for the above simulations are 100×100 and the time interval that the heat equation was conducted during is $\Delta\tau = 10 \cdot \Delta t = \frac{10 \cdot h^2}{n}$ where $n = 2$ in the middle figure and $n = 3$ in the bottom figure. The diffusion works for the quotient $\frac{\Delta t}{h^2} = \frac{1}{3}$ but does not work for the quotient $\frac{\Delta t}{h^2} = \frac{1}{2}$.

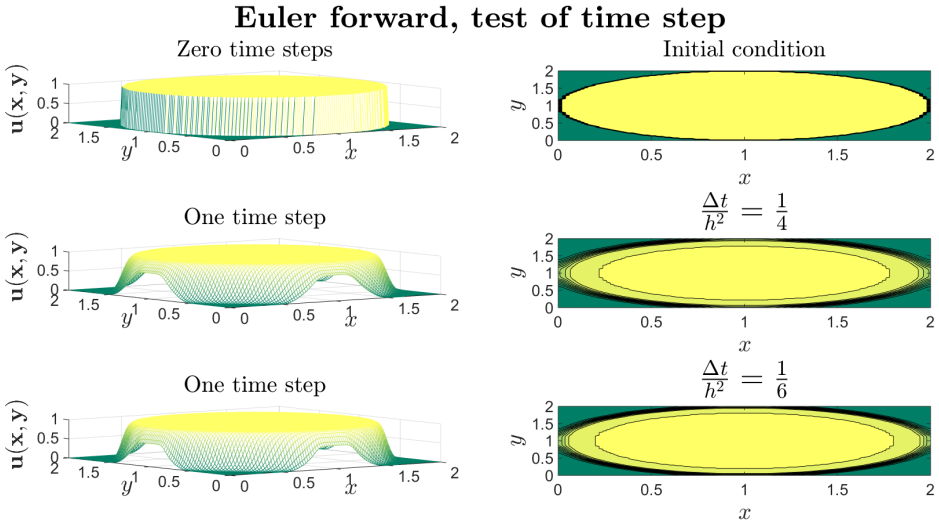


Figure 3.3: Test of time step using ratio $\frac{\Delta t}{h^2} = \frac{1}{4}$ & $\frac{1}{6}$. The grid size for the above simulations are 100×100 and the time interval that the heat equation was conducted during is $\Delta\tau = 10 \cdot \Delta t = \frac{10 \cdot h^2}{n}$ where $n = 4$ in the middle figure and $n = 6$ in the bottom figure. The diffusion works for both the quotient $\frac{\Delta t}{h^2} = \frac{1}{4}$ as well as for the quotient $\frac{\Delta t}{h^2} = \frac{1}{6}$.

The conclusion from the simulations in figure 3.2 and figure 3.3 is that in order for the finite difference implementation of the heat equation to work at least the quotient $\frac{\Delta t}{h^2}$ cannot be set to the value $\frac{1}{2}$ but can be set to the value $\frac{1}{3}$ in $\mathbb{R}^2$. The same simulations were conducted for a sphere in $\mathbb{R}^3$ and the similar condition in $\mathbb{R}^3$ is that the quotient $\frac{\Delta t}{h^2}$ cannot take the value $\frac{1}{4}$ but the diffusion step works for the value $\frac{1}{8}$. Subsequently, the value $\frac{\Delta t}{h^2} = \frac{1}{10}$ will be used in simuations in both three and two dimensions.

Using an appropriate time step, the heat equation can be solved numerically. In fact there are several ways of solving the heat equation in both $\mathbb{R}^2$ and $\mathbb{R}^3$ using finite differences, and in the next paragraph these methods are described.

Discretization of Laplace operator

The various stencils used to approximate the Laplace operator differ depending on the number of adjacent points that the stencil takes into account. In order to vizualize the situation at hand, it is possible to think of a cube where centered at a position (x,y,z) as illustrated in figure 3.4.

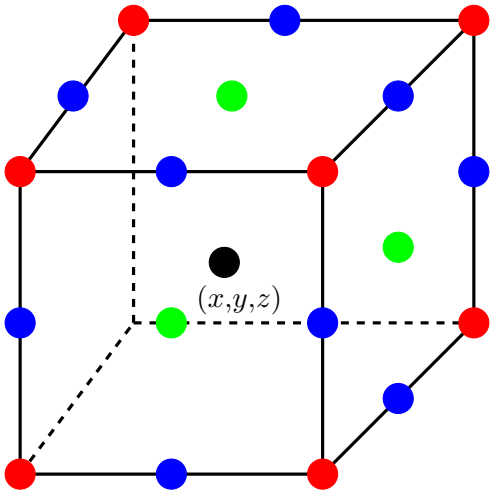


Figure 3.4: Cube for determining Laplace stencils

In figure 3.4, the cube consists of 27 points, that is 9 points at each plane. The center point, that is the black point at position (x,y,z), has therefore 26 neighbours and it is possible to characterize these adajacent points using three different categories. The first category consists of the *face* points which are the points at a center of a face of the cube ilustrated by the green points in figure 3.4. The second category consists of the *edge* points which are the points at the center of an edge of the cube ilustrated by the blue points in figure 3.4. The third category consists of the *corner* points which are the points at the corners of the cube ilustrated by the red points in figure 3.4. Note that

there are 6 points at the faces of each cube, there are 12 points at the edges and 8 points at the corners of the cube.

Using the above notation, it is possible to define three different stencils for the Laplace operator. The three stencils are called the *7 point stencil*, the *19 point stencil* and the *27 point stencil*[11]. The difference between the three stencils depends on the number of adjacent points that the stencils takes into account.

The 7 point stencil only takes the face points into account, and is exactly the expression derived in equation 3.2. A more compact way of denoting this stencil is to introduce three new sets of indices, denoted N_f, N_e and N_c respectively, which contain the indices for the faces, edges and corners of the cube. Furthermore, denote the value of u at the center point by u_i. Then the 7 point stencil of the discrete Laplacian, denoted $\hat{\Delta}u_f$, is summarized in equation 3.4.

$$\hat{\Delta}u_f = \sum_{j \in N_f} u_j - 6u_i \tag{3.4}$$

The 19 point stencil is based on the faces and the edges of the cube. The 19 point stencil of the discrete Laplacian, denoted $\hat{\Delta}u_{fe}$ is shown in equation 3.5[11].

$$\hat{\Delta}u_{fe} = \frac{1}{6}\left(2\sum_{j \in N_f} u_j + \sum_{j \in N_e} u_j - 24u_i\right) \tag{3.5}$$

Finally, the 27 point stencil is based on the faces, edges and corners of the cube. The 27 point stencil of the discrete Laplacian, denoted $\hat{\Delta}u_{fec}$ is shown in equation 3.6[11].

$$\hat{\Delta}u_{fec} = \frac{3}{13}\left(\sum_{j \in N_f} u_j + \frac{1}{2}\sum_{j \in N_e} u_j + \frac{1}{3}\sum_{j \in N_c} -\frac{44}{3}u_i\right) \tag{3.6}$$

In $\mathbb{R}^2$ the corresponding discretization to the 7 point in $\mathbb{R}^2$ is called the five point stencil. However, there is a slightly more complicated stencil, called the *nine point stencil* derived in [9], and it is presented in equation 3.7 below.

$$\hat{\Delta}u_9 = \frac{2}{3}\sum_{j \in N_f} u_j + \frac{1}{6}\sum_{j \in N_c} u_j - \frac{10 \cdot u_i}{3} \tag{3.7}$$

Subsequently, the choice of the Laplace stencil in $\mathbb{R}^3$ depends on the results from numerical simulations for a famous example of the phase transition between two phases, namely the shrinking sphere. For the simulations in $\mathbb{R}^2$ the nine point Laplacian in equation 3.7 will be applied.

Shrinking sphere

The results from the finite difference implementation for two phases are generated from a spherical domain Ω_1, i.e. the problem formulated in section 2.2.1 on page 29. In figure 3.5, the evolution of the shrinking sphere is depicted. Using the method described

in section 2.2.1 on page 29 the accuracy of the finite difference implementation can be estimated by tracking the evolution of the error defined as $e(t) = r(t) - \overline{r}(t)$ where $r(t)$ is the analytical radius of the sphere given by equation 2.42 on page 30 and $\overline{r}(t)$ is the numerical radius generated dy the simulation.

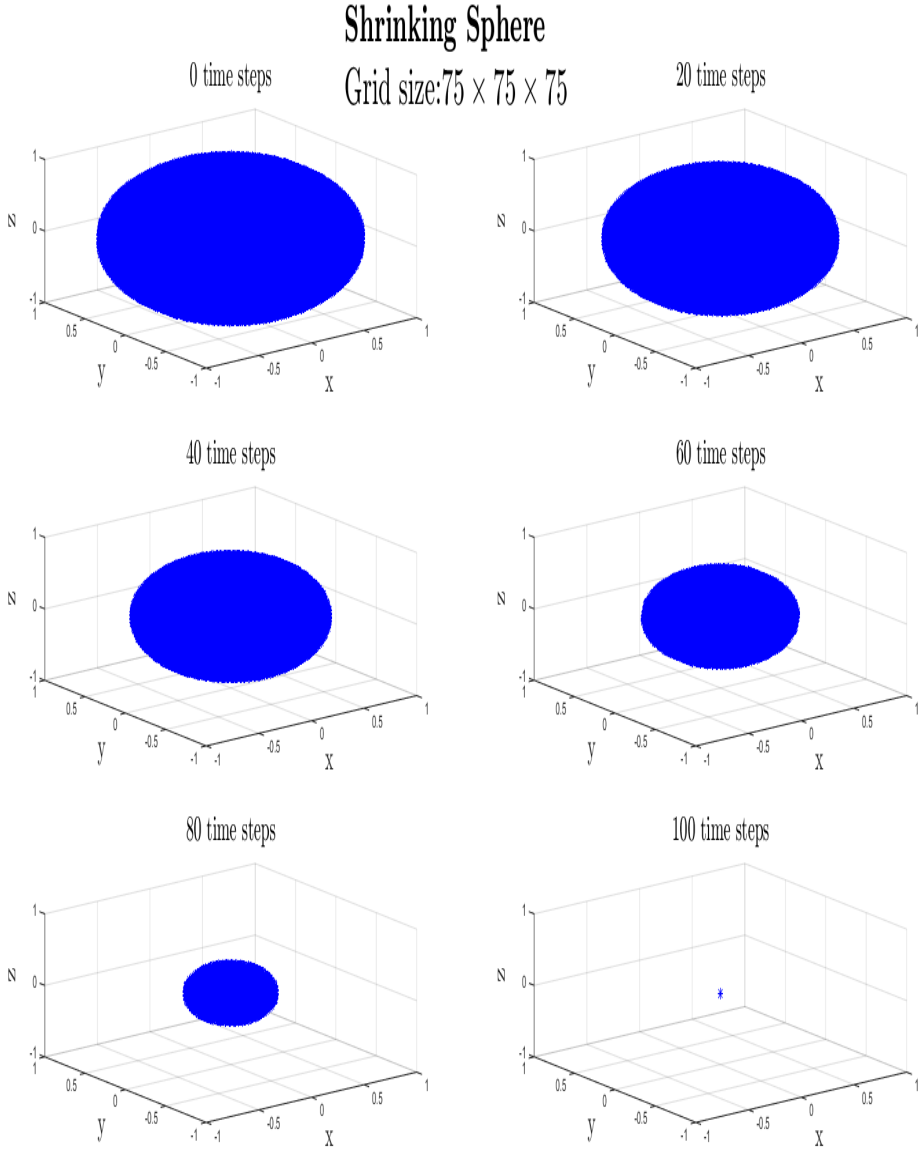
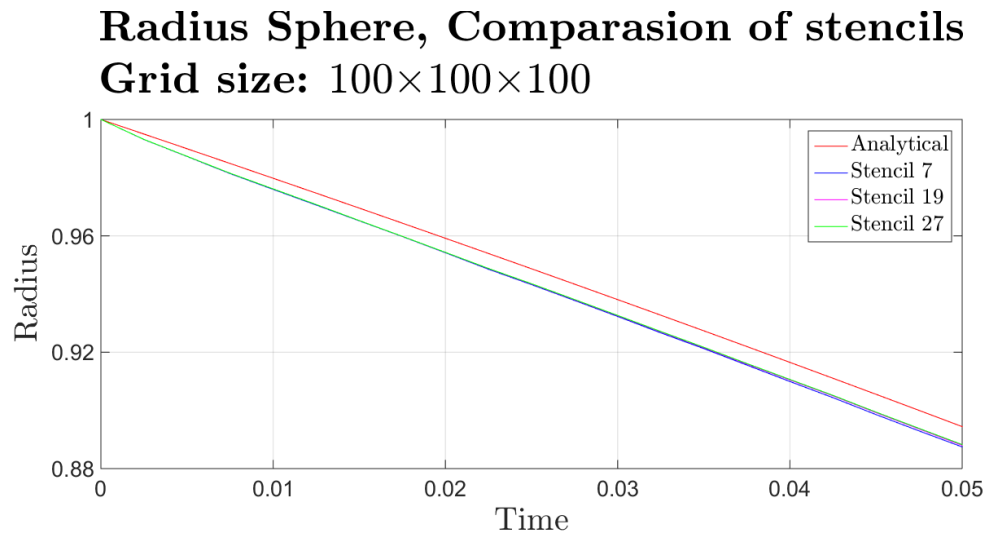


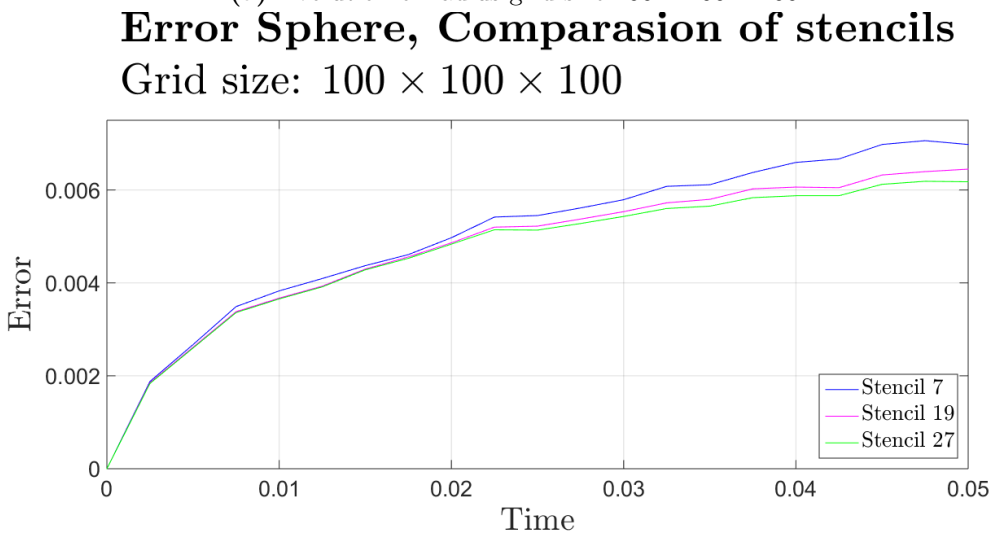
Figure 3.5: Shrinking Sphere

The results from the error analysis is illustrated in figure 3.6b to 3.10. Two different

grid sizes, namely $100 \times 100 \times 100$ and $175 \times 175 \times 175$, were used in the simulations. In addition to the grid size, the choice of Laplace stencil is investigated in figure 3.6 and 3.7 respectively. In figure 3.6, the grid size was set to $100 \times 100 \times 100$ and in figure 3.7 the grid size was set to $175 \times 175 \times 175$.



(a) Evolution of radius grid size $100 \times 100 \times 100$.

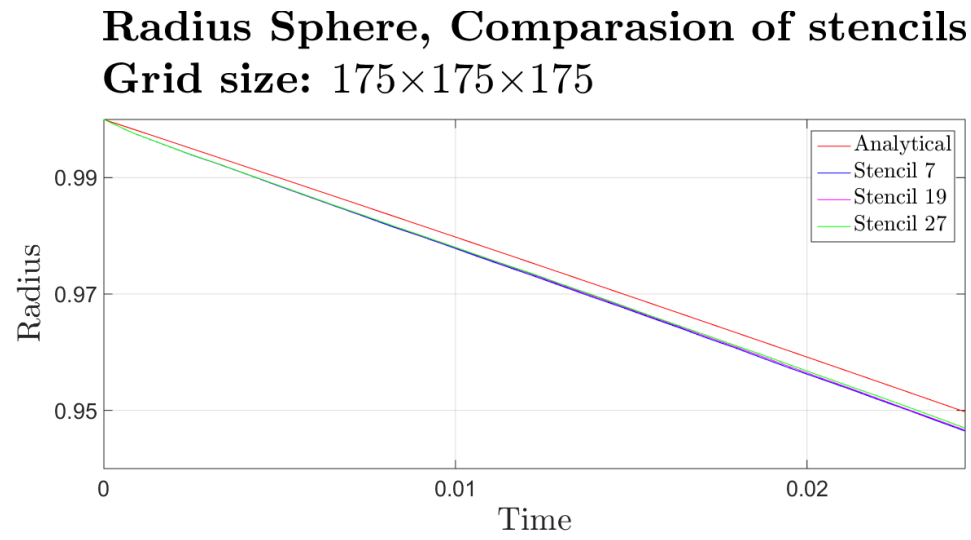


(b) Evolution of error grid size $100 \times 100 \times 100$.

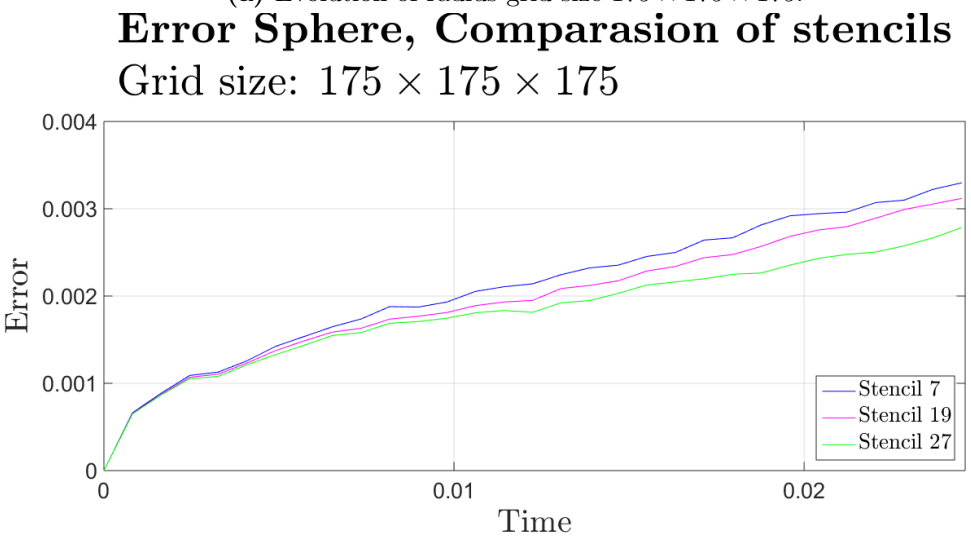
Figure 3.6: Comparasion of stencils, grid size $100 \times 100 \times 100$.

From figure 3.6b the expected result, namely that the error for the 27 point stencil is smaller than the 19 point stencil which is smaller than the 7 point stencil, is confirmed. However, for the grid size $100 \times 100 \times 100$, the difference in grid size is not particularly large and this difference will be quantified later on. In figure 3.7, the same analysis has

been conducted for a grid size of $175 \times 175 \times 175$.



(a) Evolution of radius grid size $175 \times 175 \times 175$.

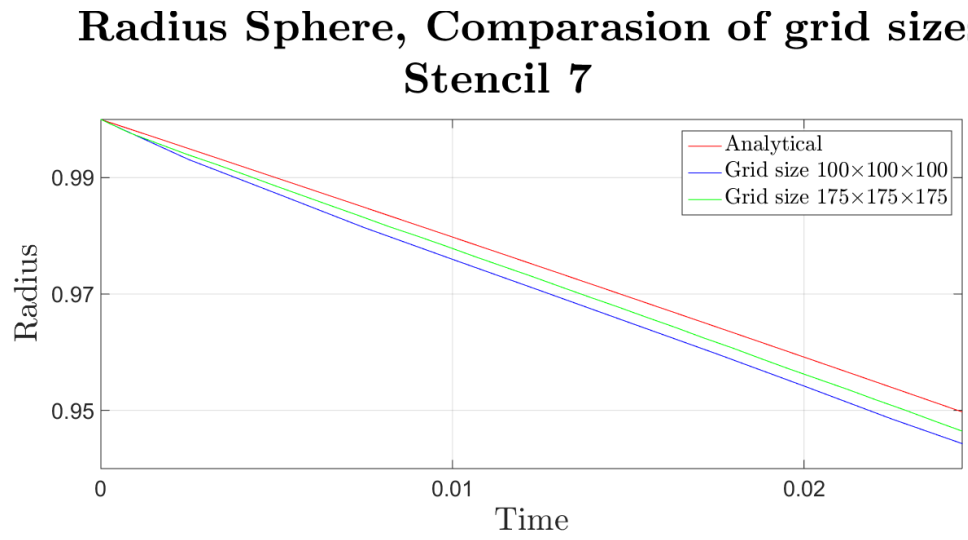


(b) Evolution of error grid size $175 \times 175 \times 175$.

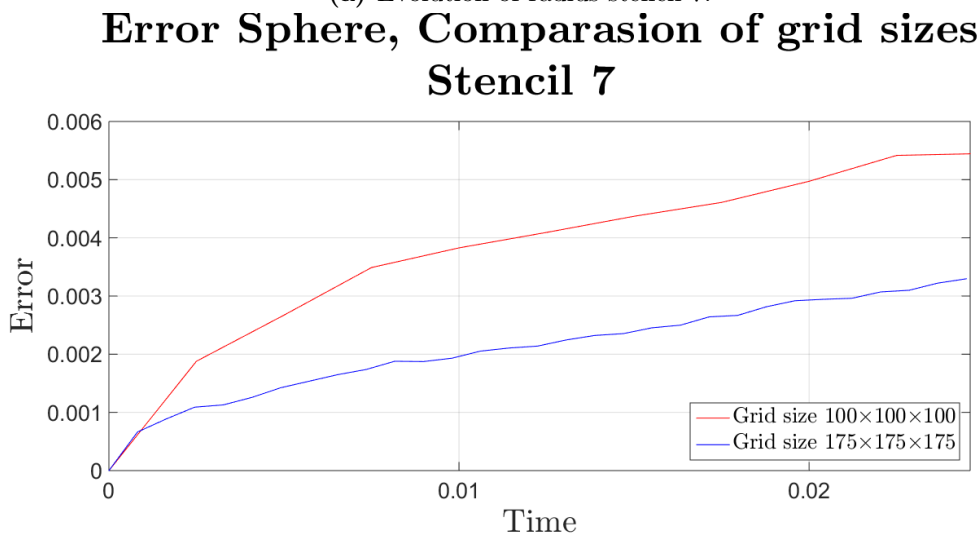
Figure 3.7: Comparasion of stencils, grid size $175 \times 175 \times 175$.

From figure 3.7b the expected result, namely that the error for the 27 point stencil is smaller than the 19 point stencil which is smaller than the 7 point stencil, is confirmed. Furthermore, for the grid size $175 \times 175 \times 175$, the difference in error between the 27 point and the 19 point stencils is larger than in figure 3.6 which indicates that for larger grid sizes the choice of stencil is of larger significance. In order to quantify the accuracy of the finite difference implementation for the different grid sizes and the various stencils, the effect on grid size for the 7 point stencil, 19 point stencil and 27 point stencil is

compared in figure 3.8, 3.9 and 3.10 below.



(a) Evolution of radius stencil 7.



(b) Evolution of error stencil 7.

Figure 3.8: Comparasion of grid sizes stencil 7.

In order to quantify the convergence of the error in the time interval $t \in (0.01,0.02)$ assume that both the red graph and the blue graph in figure 3.8b is linear in the interval in question. Then it follows that the following calculation holds for the red graph,

$$\begin{cases} r_7^{100}(0.01) &= 0.9760 \\ r_7^{100}(0.02) &= 0.9542 \end{cases} \Longrightarrow \begin{cases} e_7^{100}(0.01) &= 0.0038 \\ e_7^{100}(0.02) &= 0.0050 \end{cases} \Longrightarrow$$

$$\boxed{k_7^{100} = \frac{0.0050 - 0.0038}{0.01} = 0.12 \ \left({}^{\text{length unit}}/_{\text{time unit}}\right)}$$

and similarly for the blue graph the following calculation holds.

$$\begin{cases} r_7^{175}(0.0098) &= 0.9783 \\ r_7^{175}(0.0204) &= 0.9554 \end{cases} \Longrightarrow \begin{cases} e_7^{175}(0.0098) &= 0.0019 \\ e_7^{175}(0.0204) &= 0.0029 \end{cases} \Longrightarrow$$

$$\boxed{k_7^{175} = \frac{0.0029 - 0.0019}{0.0106} = 0.0943 \ \left({}^{\text{length unit}}/_{\text{time unit}}\right)}$$

Furthermore, the mean error in the interval $t \in (0.01,0.02)$ for the grid size $100 \times 100 \times 100$ is

$$\overline{e_7^{100}} = \frac{0.0038 + 0.0050}{2} = 0.0044$$

and the mean error in the interval $t \in (0.0098,0.0204) \approx (0.01,0.02)$ for the grid size $100 \times 100 \times 100$ is given by the expression below.

$$\overline{e_7^{175}} = \frac{0.0019 + 0.0029}{2} = 0.0024$$

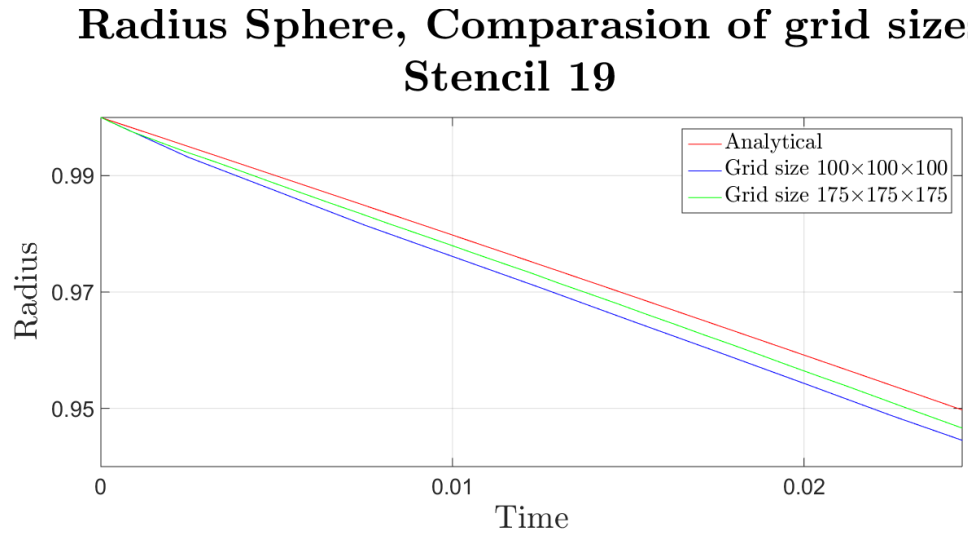
Thus the mean error is decreased by $1 - \dfrac{0.0024}{0.0044} \approx 1 - 0.5455 \approx 45\%$ in the time interval $t \in (0.01,0.02)$ when the grid size is increased from $100 \times 100 \times 100$ to $175 \times 175 \times 175$.

Next the same calculations are conucted for the 19 point stencil, and the results from these simulations are illustrated in figure 3.9.
In order to quantify the convergence of the error in the time interval $t \in (0.01,0.02)$ assume that both the red graph and the blue graph in figure 3.9b is linear in the interval in question. Then it follows that the following calculation holds for the red graph,

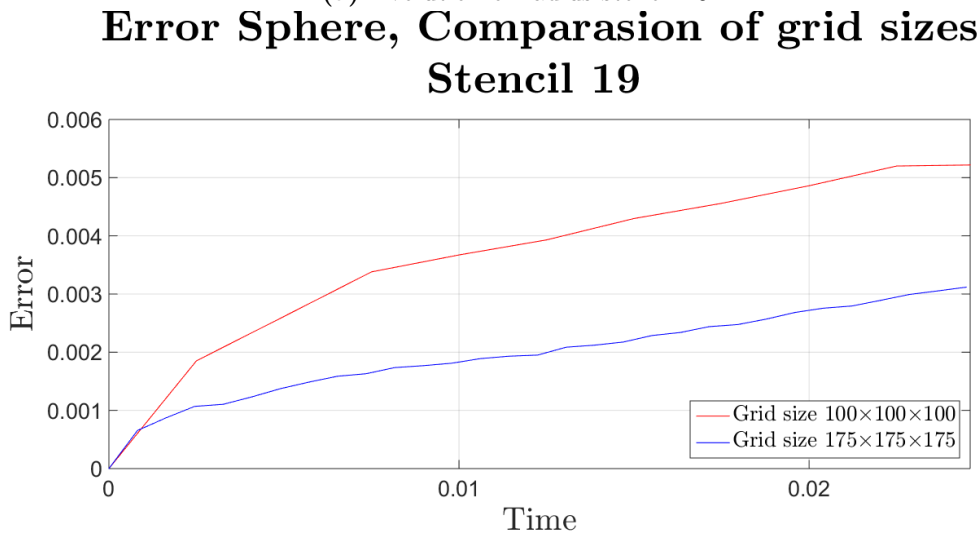
$$\begin{cases} r_{19}^{100}(0.01) &= 0.9761 \\ r_{19}^{100}(0.02) &= 0.9543 \end{cases} \Longrightarrow \begin{cases} e_{19}^{100}(0.01) &= 0.0037 \\ e_{19}^{100}(0.02) &= 0.0049 \end{cases} \Longrightarrow$$

$$\boxed{k_{19}^{100} = \frac{0.0049 - 0.0037}{0.01} = 0.12 \ \left({}^{\text{length unit}}/_{\text{time unit}}\right)}$$

and similarly for the blue graph the following calculation holds.



(a) Evolution of radius stencil 19.



(b) Evolution of error stencil 19.

Figure 3.9: Comparasion of grid sizes stencil 19.

$$\begin{cases} r_{19}^{175}(0.0098) & = 0.9784 \\ r_{19}^{175}(0.0204) & = 0.9556 \end{cases} \Longrightarrow \begin{cases} e_{19}^{175}(0.0098) & = 0.0018 \\ e_{19}^{175}(0.0204) & = 0.0027 \end{cases} \Longrightarrow$$

$$\boxed{k_{19}^{175} = \frac{0.0027 - 0.0018}{0.0106} = 0.0849 \ \left({}^{\text{length unit}}/_{\text{time unit}}\right)}$$

Furthermore, the mean error in the interval $t \in (0.01,0.02)$ for the grid size $100 \times 100 \times 100$

is

$$\overline{e_{19}^{100}} = \frac{0.0037 + 0.0049}{2} = 0.0043$$

and the mean error in the interval $t \in (0.0098,0.0204) \approx (0.01,0.02)$ for the grid size $100 \times 100 \times 100$ is given by the expression below.

$$\overline{e_{19}^{175}} = \frac{0.0018 + 0.0027}{2} = 0.0023$$

Thus the mean error is decreased by $1 - \dfrac{0.0023}{0.0043} \approx 1 - 0.5233 \approx 48\%$ in the time interval $t \in (0.01,0.02)$ when the grid size is increased from $100 \times 100 \times 100$ to $175 \times 175 \times 175$.

Next the same calculations are conucted for the 27 point stencil, and the results from these simulations are illustrated in figure 3.10.
In order to quantify the convergence of the error in the time interval $t \in (0.01,0.02)$ assume that both the red graph and the blue graph in figure 3.10b is linear in the interval in question. Then it follows that the following calculation holds for the red graph,

$$\begin{cases} r_{27}^{100}(0.01) &= 0.9761 \\ r_{27}^{100}(0.02) &= 0.9543 \end{cases} \Longrightarrow \begin{cases} e_{27}^{100}(0.01) &= 0.0037 \\ e_{27}^{100}(0.02) &= 0.0049 \end{cases} \Longrightarrow$$

$$\boxed{k_{27}^{100} = \frac{0.0049 - 0.0037}{0.01} = 0.12 \ \left({}^{\text{length unit}}/_{\text{time unit}}\right)}$$

and similarly for the blue graph the following calculation holds.

$$\begin{cases} r_{27}^{175}(0.0098) &= 0.9785 \\ r_{27}^{175}(0.0204) &= 0.9559 \end{cases} \Longrightarrow \begin{cases} e_{27}^{175}(0.0098) &= 0.0017 \\ e_{27}^{175}(0.0204) &= 0.0024 \end{cases} \Longrightarrow$$

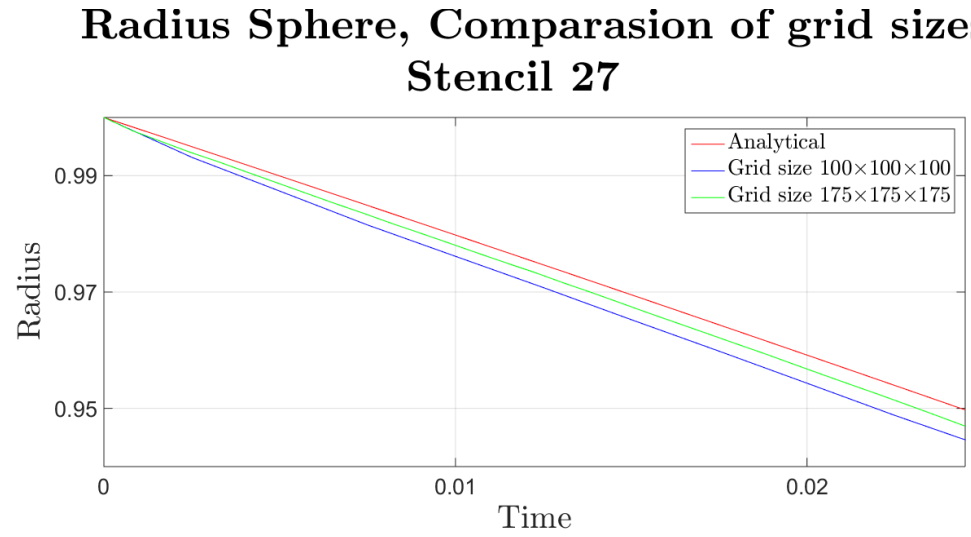
$$\boxed{k_{27}^{175} = \frac{0.0024 - 0.0017}{0.0106} = 0.0660 \ \left({}^{\text{length unit}}/_{\text{time unit}}\right)}$$

Furthermore, the mean error in the interval $t \in (0.01,0.02)$ for the grid size $100 \times 100 \times 100$ is

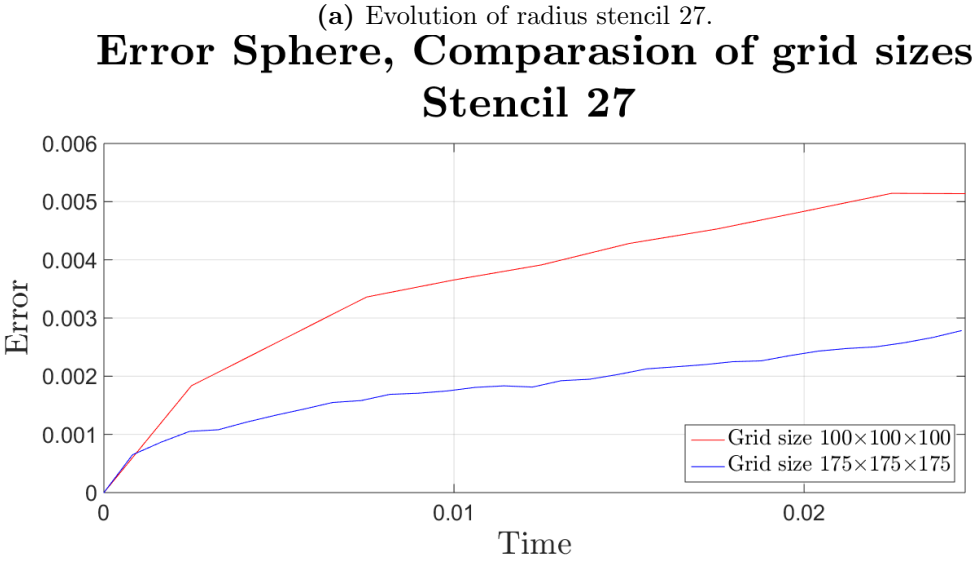
$$\overline{e_{27}^{100}} = \frac{0.0037 + 0.0049}{2} = 0.0043$$

and the mean error in the interval $t \in (0.0098,0.0204) \approx (0.01,0.02)$ for the grid size $100 \times 100 \times 100$ is given by the expression below.

$$\overline{e_{27}^{175}} = \frac{0.0017 + 0.0024}{2} = 0.0020$$



(a) Evolution of radius stencil 27.



(b) Evolution of error stencil 27.

Figure 3.10: Comparasion of grid sizes stencil 27.

Thus the mean error is decreased by $1 - \frac{0.0020}{0.0043} \approx 1 - 0.4651 \approx 53\%$ in the time interval $t \in (0.01,0.02)$ when the grid size is increased from $100 \times 100 \times 100$ to $175 \times 175 \times 175$.

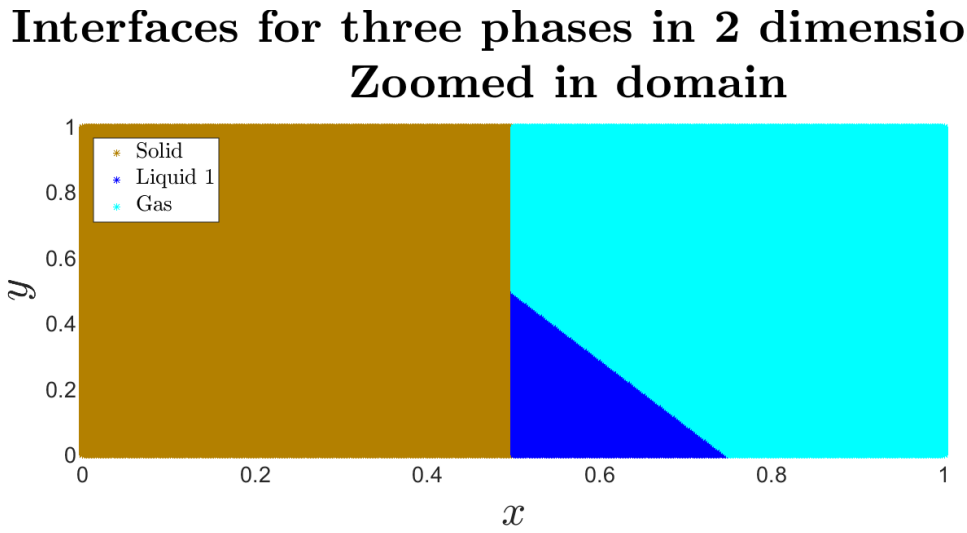
In conclusion, two observations can be made concerning the choice of Laplace stencil. The increase of error over time is lowest for the 27 point Laplace stencil, approximately $k_{27}^{175} = 0.0660$ $\left({}^{\text{length unit}}/_{\text{time unit}}\right)$ in the time interval $t \in (0.0098,0.0204)$, it is second lowest for the 19 point stencil, approximately $k_{19}^{175} = 0.0849$ $\left({}^{\text{length unit}}/_{\text{time unit}}\right)$ in the time interval $t \in (0.0098,0.0204)$, and it is highest for the 7 point stencil, namely $k_{7}^{175} = 0.0943$ $\left({}^{\text{length unit}}/_{\text{time unit}}\right)$ in the time interval $t \in (0.0098,0.0204)$. Furthermore,

the increase in accuracy namely 53% for the 27 point stencil, 48% for the 19 point stencil and 45% for the 7 point stencil when increasing the grid size from $100 \times 100 \times 100$ to $175 \times 175 \times 175$ in the time interval $t \in (0.01,0.02)$ indicates that for higher grid sizes the 27 point stencil will result in a highly accurate result. Consequently, the 27 point stencil will be used when conducting simulations in $\mathbb{R}^3$ and the corresponding 9 point stencil will be used when conducting simulations in $\mathbb{R}^2$.

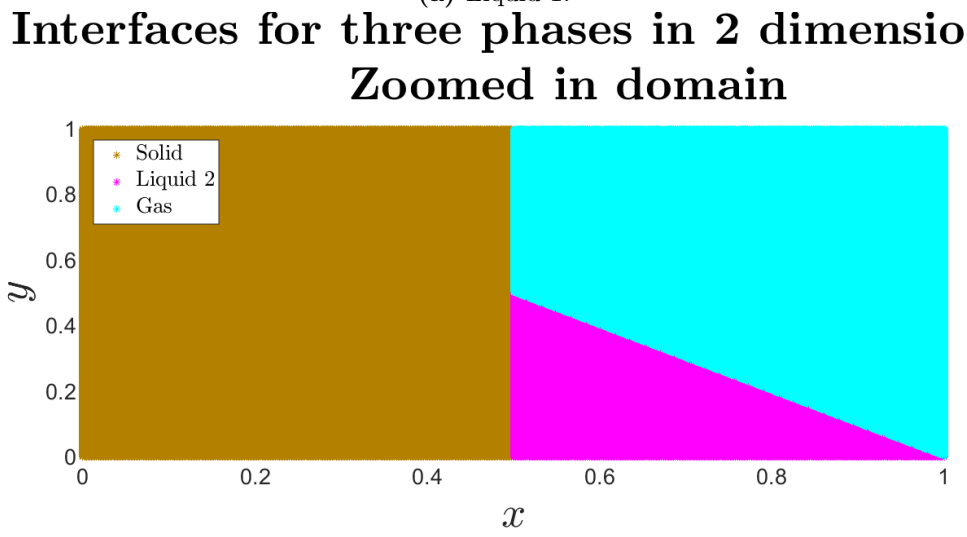
When it comes to simulating a three phase system numerically, the projection triangle algorithm on page 33 requires implementation.

3.2 Sharpening step

Subsequently, the projection triangle algorithm on page 33 is tested numerically for two different liquids which are depicted in figure 3.11. What differs between the liquid in figure 3.11a and the liquid in figure 3.11b, is the contact angle between the solid phase and the liquid phase. Thus, using the two conditions listed in section 2.2.2 on page 30, the appearence of the projection triangle for the two liquids in figure 3.11 can be predicted.



(a) Liquid 1.



(b) Liquid 2.

Figure 3.11: Contact angle for two liquids.

The water surface for the first liquid is determined by the function $\Gamma_{12}(x) = \frac{3-4x}{2}$ and the water surface for the second liquid is determined by $\Gamma_{12}(x) = 1 - x$. Thus for the first liquid it follows that the contact angle α is given by

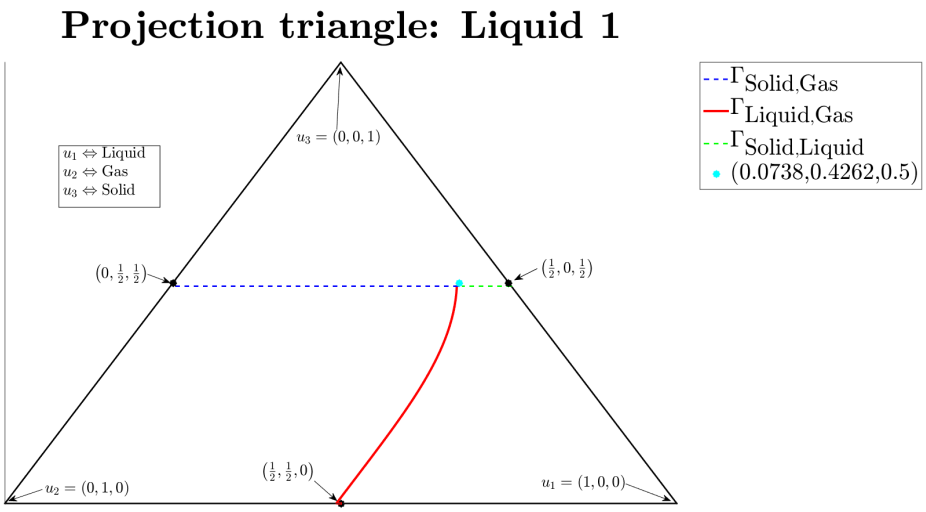
$$\alpha = \tan^{-1}\left(\frac{1}{2}\right) \approx 0.4636$$

and it follows that $\frac{\alpha}{2\pi} \approx \frac{0.4636}{2\pi} = 0.0738$. Hence the triple junction in the projection triangle for the first liquid is given by $\left(\frac{\alpha}{2\pi}, \frac{\pi-\alpha}{2\pi}, \frac{1}{2}\right) = (0.0738, 0.4262, 0.5)$. Using a similar calculation for the second liquid it follows that the triple junction should be located at $\left(\frac{\alpha}{2\pi}, \frac{\pi-\alpha}{2\pi}, \frac{1}{2}\right) = (0.1250, 0.3750, 0.5)$.

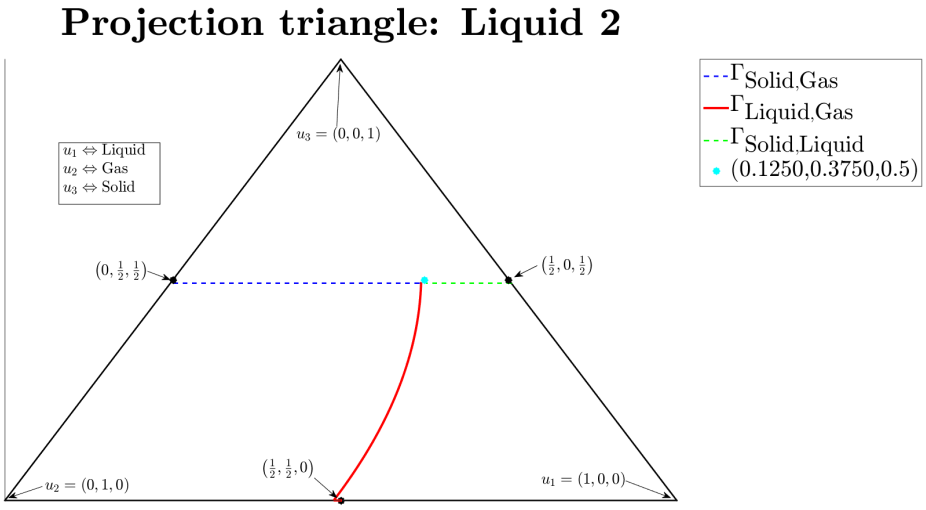
The results from the implementation of the projection triangle algorithm for the two liquid in figure 3.11 are illustrated in figure 3.12. As can be seen, the red line meets the cyan point indicating the location of the triple junction at one end and meets the black point $\left(\frac{1}{2}, \frac{1}{2}, 0\right)$ which indicates mean curvature flow between the liquid and air phase. This fact corresponds to the theory describing the triple junction algorithm in section 2.2.2 on page 30.

Furthermore, the interfaces $\Gamma_{\text{Solid,Gas}}$ and $\Gamma_{\text{Solid,Liquid}}$ form a T-junction which is consistent with a straight solid phase, see section 2.2.2 on page 30. Both these lines meet at the points $\left(\frac{1}{2}, 0, \frac{1}{2}\right)$ and $\left(0, \frac{1}{2}, \frac{1}{2}\right)$ respectively at one of the two ends, which corresponds to mean curvature flow far from the triple junction, and both lines meet at the triple junction on the other end.

The results in figure 3.12 were generated using the following settings. The grid size was set 500×500 which results in a step length of $h = \frac{1}{500} = 0.002$. The time step in the Euler forward time discretization was set to $\Delta t = \frac{h^2}{10} = 4 \cdot 10^{-7}$, and the diffusion step was conducted during the time interval $\Delta\tau = 1500 \cdot \Delta t = 0.0014$.



(a) Liquid 1.



(b) Liquid 2.

Figure 3.12: Projection triangle.

Accordingly, the results in figure 3.12 indicate that the projection triangle on page 33 was succesfully implemented for the two liquids depicted in figure 3.11. However, no mathematical justification for the fact that the projection triangle algorithm, first proposed by Ruuth in [16], produces reliable results in simulating capillary flows is provided in [16] by Ruuth. In order to justify the implementation of the projection triangle algorithm in simulating capillary flows, a convolution thresholding calculation is performed in the subsequent chapter.

4

Convolution thresholding

In order to simulate capillary flow for a three phase system using the solution algorithm on page 34, the projection triangle algorithm requires a mathematical justification. Therefore, the assymptotic behaviour of the solutions close to the triple junction generated by this algorithm after a time $\Delta\tau$ is investigated in both $\mathbb{R}^2$ and $\mathbb{R}^3$. The applied analysis is often referred to as a *convolution thresholding scheme* and a similar analysis has previously been conducted by Heintz and Grzhibovskis[5] and Ruuth in his thesis[16; 17]. In order to establish a convolution thresholding scheme, the various solutions to the heat equation are required and they depend on the so called *Greene's function*[7].

The projection algorithm for the three phase system on page 33 boils down to solving the heat equation for a short time $\Delta\tau$, for each domain denoted by the index $i \in \{1,2,3\}$ and the set of equations are formulated in equation 4.1 .

$$\begin{cases} \dfrac{\mathrm{d}u_i}{\mathrm{d}t} = \Delta u_i(\mathbf{x},t) & ,\mathbf{x} \in \Omega, t \in [0,\Delta\tau] \quad (PDE) \\ u_i(\mathbf{x},t) = 0 & ,\mathbf{x} \in \partial\Omega, t \in [0,\Delta\tau] \quad (BC) \\ u_i(\mathbf{x},0) = \chi_i(\mathbf{x}), & ,\mathbf{x} \in \Omega \quad (IC) \end{cases} \tag{4.1}$$

The solution to equation 4.1 with the characteristic function as initial condition is given by equation 4.2[7], where $k(\mathbf{x}) = \exp\left(\dfrac{-|\mathbf{x}_0 - \mathbf{x}|^2}{4\Delta\tau}\right)$ is the Gaussian *kernel* and d is the dimension of the domain $\Omega \subset \mathbb{R}^d$.

$$u_i(\mathbf{x}_0,\Delta\tau) = \frac{1}{(4\pi\Delta\tau)^{\frac{d}{2}}} \int_{\mathbb{R}^d} \chi_i(\mathbf{x}) \exp\left(\frac{-|\mathbf{x}_0 - \mathbf{x}|^2}{4\Delta\tau}\right) \mathrm{d}\mathbf{x} = \frac{(\chi_i \star k)(\mathbf{x}_0)}{(4\pi\Delta\tau)^{\frac{d}{2}}}, \quad i \in \{1,2,3\} \tag{4.2}$$

Thus the solution of the solution algorithm on page 34 depends on the *convolution* of the Gaussian kernel and the characteristic function for a certain domain denoted $(\chi_i \star k)(\mathbf{x}_0)$. Furthermore, the sharpening step which is also called the *thresholding step* in the projection triangle algorithm on page 33 depends on this convolution, and therefore the solution algorithm on page 34 is often referred to as a convolution thresholding scheme. A formal analysis of the sharpening step can therefore be conducted by solving the integral in the right hand side of equation 4.2. However, for arbitrary domains, this task can be daunting and therefore an expansion of the gaussian kernel around the triple junction will be used in order to approximate the solutions to equation 4.1. Consequently, the subsequent two subsections present the convolution thresholding analysis concerning capillary flow in $\mathbb{R}^2$ ans $\mathbb{R}^3$ respectively.

4.1 Calculations in $\mathbb{R}^2$

Imagine a circular domain in $\mathbb{R}^2$ as depicted in figure 4.1. In this figure, the stationary phase is called Ω_3 and the other two phases are called Ω_1 and Ω_2 respectively. The angles between the three phases are denoted α_1, α_2 and α_3 respectively and the sum of these angles are $\alpha_1 + \alpha_2 + \alpha_3 = 2\pi$. The domains Ω_1, Ω_2 and Ω_3 and their respective angles are depicted below in figure 4.1.

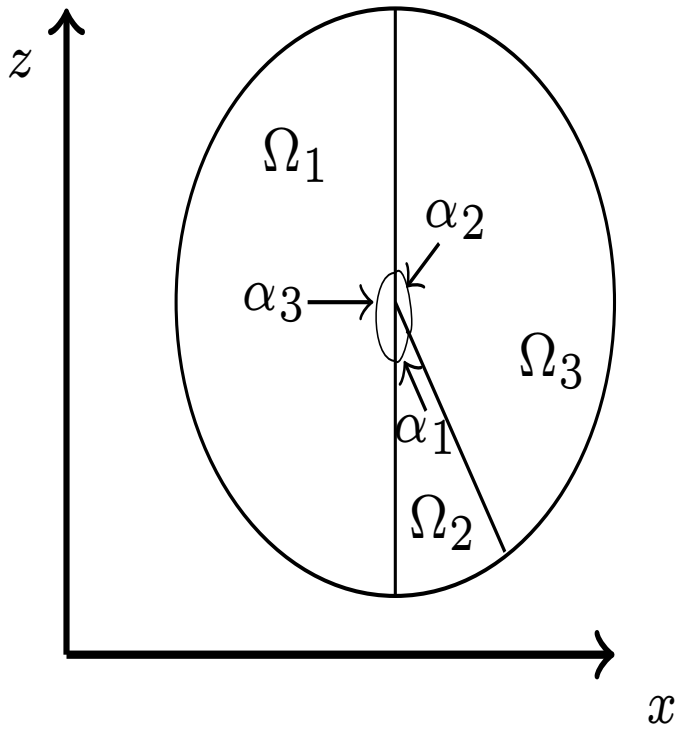


Figure 4.1: Example of triple junction in $\mathbb{R}^2$ with one phase at rest, that is $\alpha_3 = \pi$.

In order to approximate the solutions to equation 4.2 in proximity to the triple junction, which will for simplicity will be centered at the origin in the calculation, the system will be transformed to spherical coordinates. Let $r \in [0,R]$ where $R \leq \infty$ be the radius and $\theta \in [0,2\pi]$ be the angle in the *xy-plane*. Then the various interfaces, denoted Γ_{ij} between phase i and j, the domains denoted Ω_i and the boundary $\partial\Omega$ can be described in the following way.

$$\begin{aligned}
\Omega &:= \left\{ \begin{pmatrix} r \\ \theta \end{pmatrix} \in \mathbb{R}^2 : r \in [0,R],\, \theta \in [0,2\pi] \right\} \\
\partial\Omega &:= \left\{ \begin{pmatrix} r \\ \theta \end{pmatrix} \in \Omega : r = R \right\} \\
\Omega_1 &:= \left\{ \begin{pmatrix} r \\ \theta \end{pmatrix} \in \Omega : \theta \in \left[\frac{3\pi}{2}, \frac{3\pi}{2} + \alpha_1 \right] \right\} \\
\Omega_2 &:= \left\{ \begin{pmatrix} r \\ \theta \end{pmatrix} \in \Omega : \theta \in \left[\frac{3\pi}{2} + \alpha_1, 2\pi \right] \cap \left[0, \frac{\pi}{2} \right] \right\} \\
\Omega_3 &:= \left\{ \begin{pmatrix} r \\ \theta \end{pmatrix} \in \Omega : \theta \in \left[\frac{\pi}{2}, \frac{3\pi}{2} \right] \right\} \\
\Gamma_{12} &= \left\{ \begin{pmatrix} r \\ \theta \end{pmatrix} \in \Omega : \theta = \frac{3\pi}{2} \right\} \\
\Gamma_{13} &= \left\{ \begin{pmatrix} r \\ \theta \end{pmatrix} \in \Omega : \theta = \frac{\pi}{2} \right\} \\
\Gamma_{23} &= \left\{ \begin{pmatrix} r \\ \theta \end{pmatrix} \in \Omega : \theta = \frac{3\pi}{2} + \alpha_1 \right\}
\end{aligned}$$

Furthermore, the interfaces Γ_{12}, Γ_{23} and Γ_{13} can be slightly curved. Consequently, the angles α_1, α_2 and α_3 can be expanded as functions of the radius r around the origin. For example, the angle $\alpha_1(r)$ can be expanded as follows

$$\alpha_1(r) = \underbrace{\alpha_1(r=0)}_{=\alpha_0} + \underbrace{\left.\frac{\partial \alpha_1}{\partial r}\right|_{r=0}}_{\kappa_1} \cdot r + O(r^2) = \alpha_0 + \kappa_{12} r + O(r^2)$$

and therefore the expansion of the three angles α_1, α_2 and α_3 are represented in equation 4.3, 4.4 and 4.5 below. Note that κ_1 is the curvature of the water-air interface and κ_2 is the curvature of the solid wall.

$$\alpha_1(r) = \alpha_0 + \frac{3\pi}{2} + \kappa_1 r + O(r^2) \tag{4.3}$$

$$\alpha_2(r) = \frac{\pi}{2} + \kappa_2 r + O(r^2) \tag{4.4}$$

$$\alpha_3(r) = \frac{3\pi}{2} + \kappa_2 r + O(r^2) \tag{4.5}$$

Using trigonemetric identities, the cosine and sine of the angles in equation 4.3 to 4.5 can be expanded as in lemma 1.

Lemma 1. *Let the angles of the interfaces Γ_{12}, Γ_{23} and Γ_{13} be defined by equation 4.3, 4.4 and 4.5 respectively. Then the following trigonemetric identities holds close to the origin at $r \approx 0$.*

(a) $\sin(\alpha_1(r)) = \kappa_1 r \sin(\alpha_0) - \cos(\alpha_0) + O(r^2)$

(b) $\cos(\alpha_1(r)) = \sin(\alpha_0) + \kappa_1 r \cos(\alpha_0) + O(r^2)$

(c) $\sin(2\alpha_1(r)) = -\sin(2\alpha_0) - 2\kappa_1 r \cos(2\alpha_0) + O(r^2)$

(d) $\cos(2\alpha_1(r)) = 2\kappa_1 r \sin(\alpha_0) - \cos(2\alpha_0) + O(r^2)$

(e) $\sin(\alpha_2(r)) = 1 + O(r^2)$

(f) $\cos(\alpha_2(r)) = -\kappa_2 r + O(r^2)$

(g) $\sin(2\alpha_2(r)) = -2\kappa_2 r + O(r^2)$

(h) $\cos(2\alpha_2(r)) = -1 + O(r^2)$

(i) $\sin(\alpha_3(r)) = -1 + O(r^2)$

(j) $\cos(\alpha_3(r)) = \kappa_2 r + O(r^2)$

(k) $\sin(2\alpha_3(r)) = -2\kappa_2 r + O(r^2)$

(l) $\cos(2\alpha_3(r)) = -1 + O(r^2)$

Proof *Only the first identity will be proven here. The following trigonometric rules*

$$\cos(a+b) = \cos(a)\cos(b) - \sin(a)\sin(b)$$
$$\sin(a+b) = \sin(a)\cos(b) + \cos(a)\sin(b)$$

in combination with the following Taylor expansions around $x \approx 0$ will be applied repeatedly.

$$\begin{aligned}\cos(x) &= 1 + O(x^2)\\ \sin(x) &= x + O(x^3)\\ \cos(x^2) &= 1 + O(x^4)\\ \sin(x^2) &= O(x^2)\end{aligned}$$

Then it follows that

$$\begin{aligned}\sin\left(\alpha_1(r)\right) &= \sin\left(\alpha_0 + \frac{3\pi}{2} + \kappa_1 r + O(r^2)\right)\\ &= \sin\left(\alpha_0 + \frac{3\pi}{2} + \kappa_1 r\right)\underbrace{\cos\left(O(r^2)\right)}_{1+O(r^6)} + \cos\left(\alpha_0 + \frac{3\pi}{2} + \kappa_1 r\right)\underbrace{\sin\left(O(r^2)\right)}_{O(r^2)}\\ &= \sin\left(\alpha_0 + \frac{3\pi}{2} + \kappa_1 r\right) + O(r^2)\\ &= \sin\left(\alpha_0 + \frac{3\pi}{2}\right)\cos(\kappa_1 r) + \cos\left(\alpha_0 + \frac{3\pi}{2}\right)\sin\left(\kappa_1 r\right) + O(r^2)\\ &= \sin\left(\alpha_0\right)\cos\left(\frac{3\pi}{2}\right)\cos(\kappa_1 r) + \cos\left(\alpha_0\right)\sin\left(\frac{3\pi}{2}\right)\cos(\kappa_1 r)\\ &+ \cos\left(\alpha_0\right)\cos\left(\frac{3\pi}{2}\right)\sin\left(\kappa_1 r\right) - \sin\left(\alpha_0\right)\sin\left(\frac{3\pi}{2}\right)\sin\left(\kappa_1 r\right) + O(r^2)\\ &= -\cos\left(\alpha_0\right)\cos(\kappa_1 r) + \sin\left(\alpha_0\right)\sin\left(\kappa_1 r\right) + O(r^2)\\ &= \kappa_1 r\sin(\alpha_0) - \cos(\alpha_0) + O(r^2)\end{aligned}$$

which completes the proof.

■

Next the solution of the heat equation in proximity of the triple junction can be calculated.

Analytical solution in proximity of the triple junction

In the particular case in equation 4.1 where $\Omega_i \subset \Omega \subset \mathbb{R}^2$ the solution in equation 4.2 can be described in equation 4.6.

$$u_i(\mathbf{x}_0,\Delta\tau) = \frac{1}{4\pi\Delta\tau}\int_{\Omega_i}\exp\left(\frac{-|\mathbf{x}_0 - \mathbf{x}|^2}{4\Delta\tau}\right)\,\mathrm{d}\mathbf{x} \tag{4.6}$$

In equation 4.6, let the vectors $\mathbf{x}_0 \in \Omega_i$ and $\mathbf{x} \in \Omega_i$ be defined by $\mathbf{x}_0 = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix}$ and $\mathbf{x} = \begin{pmatrix} x \\ y \end{pmatrix}$. Then the following holds

$$\begin{aligned} |\mathbf{x}_0 - \mathbf{x}|^2 &= (x_0 - x)^2 + (y_0 - y)^2 \\ &= \left(x_0^2 + y_0^2\right) - 2\left(x_0 x + y_0 y\right) + \left(x^2 + y^2\right) \end{aligned}$$

and thus it the Gaussian kernel can be written as follows.

$$\exp\left(\frac{-|\mathbf{x}_0 - \mathbf{x}|^2}{4\Delta\tau}\right) = \exp\left(\frac{-\left(x_0^2 + y_0^2\right)}{4\Delta\tau}\right) \cdot \exp\left(\frac{\left(x_0 x + y_0 y\right)}{2\Delta\tau}\right) \cdot \exp\left(\frac{-\left(x^2 + y^2\right)}{4\Delta\tau}\right)$$

Now, converting $\mathbf{x}_0$ and $\mathbf{x}$ to polar coordinates using the two points (r_0,θ_0) and (r,θ), we have that $r_0^2 = x_0^2 + y_0^2$ and $r^2 = x^2 + y^2$. The nominator in the exponent of the middle term above can be simplified using the following calculations.

$$\begin{aligned} x_0 x + y_0 y &= r_0 r \cos(\theta_0)\cos(\theta) + r_0 r \sin(\theta_0)\sin(\theta) \\ &= r r_0 \left\{ \underbrace{\cos(\theta_0)\cos(\theta) + \sin(\theta_0)\sin(\theta)}_{=\cos(\theta_0 - \theta)} \right\} \\ &= r r_0 \cos(\theta_0 - \theta) \end{aligned}$$

Thus, in spherical coordinates the following holds

$$\exp\left(\frac{-|\mathbf{x}_0 - \mathbf{x}|^2}{4\Delta\tau}\right) = \exp\left(\frac{-r_0^2}{4\Delta\tau}\right) \cdot \exp\left(\frac{r r_0 \cos(\theta_0 - \theta)}{2\Delta\tau}\right) \cdot \exp\left(\frac{-r^2}{4\Delta\tau}\right)$$

and thus plugging this into equation 4.6 yields equation 4.7 for the solution to phase 1.

$$u_1(\mathbf{x}_0,\Delta\tau) = \frac{\exp\left(\frac{-r_0^2}{4\Delta\tau}\right)}{4\pi\Delta\tau} \int_0^R \exp\left(\frac{-r^2}{4\Delta\tau}\right) \int_{\alpha_3}^{\alpha_1} \exp\left(\frac{r r_0 \cos(\theta_0 - \theta)}{2\Delta\tau}\right) \mathrm{d}r\mathrm{d}\theta \tag{4.7}$$

The solution to equation 4.7, along with the solution for the other phases are listed below in theorem 1.

Theorem 1. *Suppose $\Omega \subset \mathbb{R}^2$ be a domain divided into three subdomains, namely a solid phase Ω_3, a gas phase Ω_2 and a liquid phase Ω_1. Furthermore, suppose the angle configuration, denoted (α_1,α_2,α_3), of the three phase system satisfies equation 4.3, 4.4 and 4.5 as depicted in figure 4.1. In addition, suppose that the three phases meet at the so called triple junction centered at the origin, where the quotient between the radius, $r_0 = \sqrt{x_0^2 + y_0^2}$ and the time step, $\Delta\tau$, is small, that is $\frac{r_0}{\sqrt{\Delta\tau}} \approx 0$. Then the solution to the system in equation 4.1 close to the triple junction satisfies equation 4.8, 4.9 and 4.10 below, where the distances d_N and d_T are defined as $d_N = r_0\sin(\theta_0 - \alpha_0)$ and $d_T = r_0\cos(\theta_0 - \alpha_0)$ respectively.*

$$\begin{aligned} u_1(x_0,y_0,d_N,d_T,\Delta\tau) &= \frac{\alpha_0}{2\pi} + \frac{(\kappa_1 - \kappa_2)\sqrt{\Delta\tau}}{2\sqrt{\pi}} + \frac{x_0 - d_T}{4\sqrt{\pi}\sqrt{\Delta\tau}} \\ &+ \frac{\kappa_2 y_0 - \kappa_1 d_N}{\pi} + \frac{\alpha_0(x_0^2 + y_0^2) + d_N d_T - x_0 y_0}{8\pi\Delta\tau} \\ &+ \frac{3(\kappa_1 d_N^2 - \kappa_2 y_0^2)}{8\sqrt{\pi}\sqrt{\Delta\tau}} + O\left(\frac{r_0^3}{\sqrt{\Delta\tau}}\right) \end{aligned} \tag{4.8}$$

$$\begin{aligned} u_2(x_0,y_0,d_N,d_T,\Delta\tau) &= \frac{\pi - \alpha_0}{2\pi} + \frac{(\kappa_2 - \kappa_1)\sqrt{\Delta\tau}}{2\sqrt{\pi}} + \frac{x_0 + d_T}{4\sqrt{\pi}\sqrt{\Delta\tau}} \\ &+ \frac{\kappa_2 y_0 + \kappa_1 d_N}{\pi} + \frac{x_0 y_0 - d_N d_T - (\alpha_0 + \pi)(x_0^2 + y_0^2)}{8\pi\Delta\tau} \\ &+ \frac{3(\kappa_2 y_0^2 - \kappa_1 d_N^2)}{8\sqrt{\pi}\sqrt{\Delta\tau}} + O\left(\frac{r_0^3}{\sqrt{\Delta\tau}}\right) \end{aligned} \tag{4.9}$$

$$u_3(x_0,y_0,\Delta\tau) = \frac{1}{2} - \frac{x_0}{2\sqrt{\pi}\sqrt{\Delta\tau}} - \frac{2\kappa_2 y_0}{\pi} + \frac{x_0^2 + y_0^2}{8\Delta\tau} + O\left(\frac{r_0^3}{\sqrt{\Delta\tau}}\right) \tag{4.10}$$

Proof

In the light of equation 4.7, the solution u_1 depends on solving the integral in the right hand side of equation 4.7.

Provided that r_0 is small enough such that the quotient $\frac{r_0}{\sqrt{\Delta\tau}}$ is small, it is possible to use a Taylor expansion of the innermost exponential function around zero in equation 4.7. In general, it holds that

$$exp(x) = 1 + x + \frac{x^2}{2} + O(x^3)$$

where $x \approx 0$. Thus using this approximation on the innermost exponential in equation 4.7 assuming that $\frac{r_0}{\sqrt{\Delta\tau}}$ is small enough such that $\frac{rr_0\left\{\sin(\phi_0)\sin(\phi)\cos(\theta_0 - \theta) + \cos(\phi_0)\cos(\phi)\right\}}{2\Delta\tau} \approx 0$ yields that

$$\exp\left(\frac{rr_0\cos(\theta_0 - \theta)}{2\Delta\tau}\right) = 1 + \frac{rr_0\cos(\theta_0 - \theta)}{2\Delta\tau} + \frac{1}{2}\left(\frac{rr_0\cos(\theta_0 - \theta)}{2\Delta\tau}\right)^2 + O\left(\frac{r_0^3}{\Delta\tau^3}\right)$$

Multiplying the above sum with the factor r and simplifying the square yields the following expression

$$\begin{aligned}
r\exp\left(\frac{rr_0\cos(\theta_0-\theta)}{2\Delta\tau}\right) &= r+\frac{r^2r_0\cos(\theta_0-\theta)}{2\Delta\tau}+\frac{r}{2}\left(\frac{rr_0\cos(\theta_0-\theta)}{2\Delta\tau}\right)^2+O\left(\frac{r_0^3}{\Delta\tau^3}\right)\\
&= r+\frac{r^2r_0\cos(\theta_0-\theta)}{2\Delta\tau}+\frac{r}{2}\left(\frac{r^2r_0^2\cos^2(\theta_0-\theta)}{4\Delta\tau^2}\right)+O\left(\frac{r_0^3}{\Delta\tau^3}\right)\\
&= r+\frac{r^2r_0\cos(\theta_0-\theta)}{2\Delta\tau}+\frac{r^3r_0^2\left(\cos(\theta_0)\cos(\theta)+\sin(\theta_0)\sin(\theta)\right)^2}{8\Delta\tau^2}+O\left(\frac{r_0^3}{\Delta\tau^3}\right)\\
&= r+\frac{r^2r_0\cos(\theta_0-\theta)}{2\Delta\tau}+\frac{r^3r_0^2\cos^2(\theta_0)\cos^2(\theta)}{8\Delta\tau^2}\\
&+\frac{r^3r_0^2\sin^2(\theta_0)\sin^2(\theta)}{8\Delta\tau^2}+\frac{r^3r_0^2\cos(\theta_0)\cos(\theta)\sin(\theta_0)\sin(\theta)}{4\Delta\tau^2}+O\left(\frac{r_0^3}{\Delta\tau^3}\right)\\
&= r+\frac{r^2r_0\cos(\theta_0-\theta)}{2\Delta\tau}+\frac{r^3r_0^2\cos^2(\theta_0)\cos^2(\theta)}{8\Delta\tau^2}\\
&+\frac{r^3r_0^2\sin^2(\theta_0)\sin^2(\theta)}{8\Delta\tau^2}+\frac{r^3r_0^2\sin(2\theta_0)\sin(2\theta)}{16\Delta\tau^2}+O\left(\frac{r_0^3}{\Delta\tau^3}\right)
\end{aligned}$$

which is summarized in equation 4.11 below.

$$\begin{aligned}
r\exp\left(\frac{rr_0\cos(\theta_0-\theta)}{2\Delta\tau}\right) &= r+\frac{r^2r_0\cos(\theta_0-\theta)}{2\Delta\tau}+\frac{r^3r_0^2\cos^2(\theta_0)\cos^2(\theta)}{8\Delta\tau^2}\\
&+\frac{r^3r_0^2\sin^2(\theta_0)\sin^2(\theta)}{8\Delta\tau^2}+\frac{r^3r_0^2\sin(2\theta_0)\sin(2\theta)}{16\Delta\tau^2}+O\left(\frac{r_0^3}{\Delta\tau^3}\right)
\end{aligned}\tag{4.11}$$

Now, each term is integrated with respect for θ. In order to solve these integrals, the appropriate expansions, i.e. (a)-(d) and (i)-(l), formulated in lemma 1 will be used. The first term results in the following integral.

$$\int_{\alpha_3}^{\alpha_1}\mathrm{d}\theta=\alpha_1-\alpha_3=\alpha_0+(\kappa_1-\kappa_2)r+O(r^2)$$

When it comes to the second term the follwing integral

$$\int_{\alpha_3}^{\alpha_1}\cos(\theta_0-\theta)\,\mathrm{d}\theta$$

is to be calculated, which can be done using the following identity.

$$\cos(\theta_0-\theta)=\cos(\theta_0)\cos(\theta)+\sin(\theta_0)\sin(\theta)$$

Thus, the integral in question can be calculated

$$\begin{aligned}\int_{\alpha_3}^{\alpha_1} \cos(\theta_0 - \theta)\,\mathrm{d}\theta &= \cos(\theta_0)\int_{\alpha_3}^{\alpha_1} \cos(\theta)\,\mathrm{d}\theta + \sin(\theta_0)\int_{\alpha_3}^{\alpha_1} \sin(\theta)\,\mathrm{d}\theta \\ &= \cos(\theta_0)\left(\sin\left(\alpha_1\right) - \sin\left(\alpha_3\right)\right) - \sin(\theta_0)\left(\cos\left(\alpha_1\right) - \cos\left(\alpha_3\right)\right) \\ &= \cos(\theta_0)\left(\kappa_1 r\sin\left(\alpha_0\right) - \cos(\alpha_0) + 1\right) \\ &- \sin(\theta_0)\left(\sin(\alpha_0) + \kappa_1 r\cos(\alpha_0) - \kappa_2 r\right) + O(r^2) \\ &= \cos(\theta_0) - \cos(\theta_0 - \alpha_0) - \kappa_1 r\sin(\theta_0 - \alpha_0) + \kappa_2 r\sin(\theta_0) + O(r^2)\end{aligned}$$

and the quadratic term results in the following three integrals.

$$\begin{gathered}\int_{\alpha_3(r)}^{\alpha_1(r)} \cos^2(\theta)\,\mathrm{d}\theta \\ \int_{\alpha_3(r)}^{\alpha_1(r)} \sin^2(\theta)\,\mathrm{d}\theta \\ \int_{\alpha_3(r)}^{\alpha_1(r)} \sin(2\theta)\,\mathrm{d}\theta\end{gathered}$$

The first integral can be calculated using the identity $\cos^2(\theta) = \dfrac{1+\cos(2\theta)}{2}$*, and this gives the following value.*

$$\begin{aligned}\int_{\alpha_3(r)}^{\alpha_1(r)} \cos^2(\theta)\,\mathrm{d}\theta &= \frac{1}{2}\int_{\alpha_3(r)}^{\alpha_1(r)} 1 + \cos(2\theta)\,\mathrm{d}\theta \\ &= \frac{1}{2}\left(\alpha_1(r) - \alpha_3(r) + \frac{\sin(2\alpha_1(r)) - \sin(2\alpha_3(r))}{2}\right) \\ &= \frac{1}{2}\left(\alpha_0 + (\kappa_1 - \kappa_2)r + \frac{2\kappa_2 r - \sin(2\alpha_0) - 2\kappa_1 r\cos(2\alpha_0)}{2}\right) + O(r^2) \\ &= \frac{1}{2}\left(\alpha_0 + \kappa_1(1 - \cos(2\alpha_0))r - \frac{\sin(2\alpha_0)}{2}\right) + O(r^2) \\ &= \frac{\alpha_0}{2} + \kappa_1\sin^2(\alpha_0)r - \frac{\sin(2\alpha_0)}{4} + O(r^2)\end{aligned}$$

Similarly, the second integral can be calculated as follows

$$\begin{aligned}
\int_{\alpha_3(r)}^{\alpha_1(r)} \sin^2(\theta)\,\mathrm{d}\theta &= \int_{\alpha_3(r)}^{\alpha_1(r)} 1-\cos^2(\theta)\,\mathrm{d}\theta \\
&= \frac{1}{2}\int_{\alpha_3(r)}^{\alpha_1(r)} 1-\cos(2\theta)\,\mathrm{d}\theta \\
&= \frac{1}{2}\left(\alpha_1(r)-\alpha_3(r)-\left(\frac{\sin(2\alpha_1(r))-\sin(2\alpha_3(r))}{2}\right)\right) \\
&= \frac{1}{2}\left(\alpha_0+(\kappa_1-\kappa_2)r+\frac{\sin(2\alpha_0)+2\kappa_1 r\cos(2\alpha_0)-2\kappa_2 r}{2}\right)+O(r^2) \\
&= \frac{1}{2}\left(\alpha_0+\kappa_1 r(1+\cos(2\alpha_0))-2\kappa_2 r+\frac{\sin(2\alpha_0)}{2}\right)+O(r^2) \\
&= \frac{\alpha_0}{2}+\kappa_1\cos^2(\alpha_0)r-\kappa_2 r+\frac{\sin(2\alpha_0)}{4}+O(r^2)
\end{aligned}$$

and the third integral is calculated below.

$$\begin{aligned}
\int_{\alpha_3(r)}^{\alpha_1(r)} \sin(2\theta)\,\mathrm{d}\theta &= -\frac{1}{2}\left[\cos(2\theta)\right]_{\alpha_3(r)}^{\alpha_1(r)} = \frac{\cos(2\alpha_3(\mathrm{r}))-\cos(2\alpha_1(\mathrm{r}))}{2} \\
&= \frac{\cos(2\alpha_0)-1-2\kappa_1 r\sin(2\alpha_0)}{2}+O(r^2) \\
&= -\left(\frac{1-\cos(2\alpha_0)}{2}\right)-\kappa_1 r\sin(2\alpha_0)+O(r^2)
\end{aligned}$$

Furthermore, arranging the terms the integral I_1 is obtained.

$$\begin{aligned}
I_1 &= \int_{\alpha_3}^{\alpha_1} r\exp\left(\frac{rr_0\cos(\theta_0-\theta)}{2\Delta\tau}\right)\,\mathrm{d}\theta \\
&= \alpha_0 r+(\kappa_1-\kappa_2)r^2+\frac{r_0 r^2}{2\Delta\tau}\left[\cos(\theta_0)-\cos(\theta_0-\alpha_0)\right]+\frac{r_0 r^3}{2\Delta\tau}\left[\kappa_2\sin(\theta_0)-\kappa_1\sin(\theta_0-\alpha_0)\right] \\
&+\frac{r_0^2 r^3}{16\Delta\tau^2}\left[\alpha_0+\frac{\sin(2(\theta_0-\alpha_0))-\sin(2\theta_0)}{2}\right]+\frac{r_0^2 r^4}{8\Delta\tau^2}\left[\kappa_1\sin^2(\theta_0-\alpha_0)-\kappa_2\sin^2(\theta_0)\right] \\
&+O(r^3)
\end{aligned}$$

The next step is to solve the integral I_2 defined below.

$$I_2 = \int_0^\infty I_1\exp\left(-\frac{r^2}{4\Delta\tau}\right)\,\mathrm{d}r$$

In order to solve this integral, the classical integral

$$\int_0^\infty e^{-s^2}\,\mathrm{d}s = \frac{\sqrt{\pi}}{2}$$

in combination with integration by parts will be used repeatedly. In fact the integral I_3 is composed by four subintegrals I_{3a}, I_{3b}, I_{3c} and I_{3d} which are defined below.

$$\begin{aligned}
I_{2a} &= \int_0^\infty \left[r \cdot \exp\left(-\frac{r^2}{4\Delta\tau}\right)\right]\,\mathrm{d}r \\
I_{2b} &= \int_0^\infty \left[r \cdot \exp\left(-\frac{r^2}{4\Delta\tau}\right)\right] \cdot r\,\mathrm{d}r \\
I_{2c} &= \int_0^\infty \left[r \cdot \exp\left(-\frac{r^2}{4\Delta\tau}\right)\right] \cdot r^2\,\mathrm{d}r \\
I_{2d} &= \int_0^\infty \left[r \cdot \exp\left(-\frac{r^2}{4\Delta\tau}\right)\right] \cdot r^3\,\mathrm{d}r
\end{aligned}$$

Note that if $F(r) = \exp\left(\frac{-r^2}{4\Delta\tau}\right)$ *it follows that* $F'(r) = -\frac{r}{2\Delta\tau}\exp\left(\frac{-r^2}{4\Delta\tau}\right) \Longrightarrow -2\Delta\tau F'(r) = \left[r \cdot \exp\left(-\frac{r^2}{4\Delta\tau}\right)\right]$ *and thus four integrals above can be rewritten as*

$$\begin{aligned}
I_{2a} &= -2\Delta\tau \int_0^\infty F'(r)\,\mathrm{d}r \\
I_{2b} &= -2\Delta\tau \int_0^\infty r \cdot F'(r)\,\mathrm{d}r \\
I_{2c} &= -2\Delta\tau \int_0^\infty r^2 \cdot F'(r)\,\mathrm{d}r \\
I_{2d} &= -2\Delta\tau \int_0^\infty r^3 \cdot F'(r)\,\mathrm{d}r
\end{aligned}$$

and these integrals can be solved using integration by parts. Thus the three integrals equals the following values

$$\begin{aligned}
I_{2a} &= 2\Delta\tau \\
I_{2b} &= 2\sqrt{\pi}\Delta\tau^{\frac{3}{2}} \\
I_{2c} &= 8\Delta\tau^2 \\
I_{2d} &= 12\sqrt{\pi}(\sqrt{\Delta\tau})^5
\end{aligned}$$

and thus the value of I_3 is given below.

$$\begin{aligned} I_2 &= 2\alpha_0\Delta\tau + 2\sqrt{\pi}(\kappa_1-\kappa_2)(\sqrt{\Delta\tau})^3 + r_0\sqrt{\pi}\sqrt{\Delta\tau}\left[\cos(\theta_0)-\cos(\theta_0-\alpha_0)\right] \\ &+ 4r_0\Delta\tau\left[\kappa_2\sin(\theta_0)-\kappa_1\sin(\theta_0-\alpha_0)\right] + \frac{r_0^2}{2}\left[\alpha_0 + \frac{\sin(2(\theta_0-\alpha_0))-\sin(2\theta_0)}{2}\right] \\ &+ 6r_0^2\Delta\tau\left[\kappa_1\sin^2(\theta_0-\alpha_0)-\kappa_2\sin^2(\theta_0)\right] + O\left(\frac{r_0^3}{\Delta\tau^2}\right) \end{aligned}$$

By equation 4.7 the solution is given by the following equation.

$$u_1(r_0,\theta_0,\phi_0,\Delta\tau) = \frac{\exp\left(-\frac{r_0^2}{4\Delta\tau}\right)}{4\pi\Delta\tau}\cdot I_2$$

However, since $\frac{r_0}{\Delta\tau}$ is small, it holds that

$$\exp\left(-\frac{r_0^2}{4\Delta\tau}\right) = 1 + O\left(-\frac{r_0^2}{\Delta\tau}\right)$$

and thus the solution after on time step $\Delta\tau$ in spherical coordinates i given in equation 4.12.

$$\begin{aligned} u_1(r_0,\theta_0,\Delta\tau) &= \frac{\alpha_0}{2\pi} + \frac{(\kappa_1-\kappa_2)\sqrt{\Delta\tau}}{2\sqrt{\pi}} + \frac{r_0}{4\sqrt{\pi}\sqrt{\Delta\tau}}\left[\cos(\theta_0)-\cos(\theta_0-\alpha_0)\right] \\ &+ \frac{r_0}{\pi}\left[\kappa_2\sin(\theta_0)-\kappa_1\sin(\theta_0-\alpha_0)\right] + \frac{r_0^2}{8\pi\Delta\tau}\left[\alpha_0 + \frac{\sin(2(\theta_0-\alpha_0))-\sin(2\theta_0)}{2}\right] \\ &+ \frac{3r_0^2}{8\sqrt{\pi}\sqrt{\Delta\tau}}\left[\kappa_1\sin^2(\theta_0-\alpha_0)-\kappa_2\sin^2(\theta_0)\right] + O\left(\frac{r_0^3}{\sqrt{\Delta\tau}}\right) \end{aligned} \tag{4.12}$$

The corresponding solution in Cartesian coordinates is obtained by using the substitutions $x_0 = r_0\cos(\theta_0)$, $y_0 = r_0\sin(\theta_0)$, $d_x = r_0\cos(\theta_0-\alpha_0)$ and $d_y = r_0\sin(\theta_0-\alpha_0)$. The solution for u_2 can be obtained using the same calculations but integrating from $\alpha_2(r)$ to $\alpha_3(r)$ with respect to θ. Finally, u_2 is obtained by using the conservation law $u_1+u_2+u_3=1$.

■

Note that the convolutions in equation 4.8 to 4.10 in theorem 1 satisfies the second condition of the projection triangle algorithm in section 2.2.2 on page 30. For a straight wall, where the curvature terms are $\kappa_1 = \kappa_2 = 0$, and the position at the triple junction is

$r = 0$ it follows that $u_1 = \frac{\alpha_0}{2\pi}$, $u_2 = \frac{\pi - \alpha_0}{2\pi}$ and $u_1 = \frac{1}{2}$. In addition to this observation, using equation 4.8 to 4.10 it is possible to postulate a convolution thresholding scheme based on the desired properties of the simulated system.

4.1.1 Construction of a convolution thresholding scheme

In figure 4.2, the evolution of the liquid gas interface denoted Γ_{12} after one time step $\Delta\tau$ is illustrated. Therefore, it would be desirable if a thresholding scheme is developed such that the contact angle for the interface $\Gamma_{12}(t = 0)$ and $\Gamma_{12}(t = \Delta\tau)$ respectively, remains α_1 at a certain position P_3 after diffusion has been applied for a time interval $\Delta\tau$. The location of the point P_3 will depend on how long the diffusion step is carried out, however there must be some point for which the angle is conserved. Furthermore, suppose a convex meniscus is desired after one time step $\Delta\tau$. In this case, points closer to the solid wall should have a smaller angle to the point P_3, that is the new triple junction, will points further from the wall should have a larger contact angle with the triple junction. At a very long distance from the solid phase, the interface Γ_{12} should evolve with mean curvature flow. Thus, using figure 4.2 and equation 4.8 and 4.10 in theorem 1, the construction of a convolution thresholding scheme can be attempted.

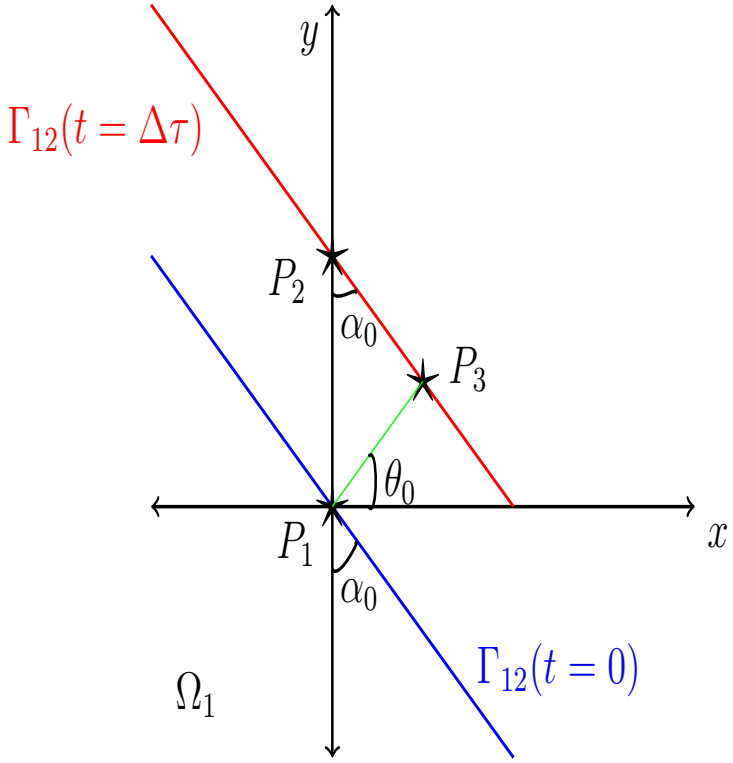


Figure 4.2: Evolution of the water air interface in $\mathbb{R}^2$ after one time step.

Begin by clarifying the notations in figure 4.2. Let $P_1 = (0,0)$ be the position of the initial triple junction, and $P_2 = (0,y_1)$ be the position of the triple junction after a time $t = \Delta\tau$. Let $P_3 = (x_2,y_2) = (r_0,\theta_0)$ be a position located somewhere at the interface $\Gamma_{12}(t = \Delta\tau)$, where r_0 and $\theta_0 = \tan^{-1}\left(\frac{y_0}{x_0}\right)$ is the representation of x_0 and y_0 in polar coordinates.

The desired condition of the convolution thresholding scheme is that the angle α_0 is preserved after a time step $\Delta\tau$. This condition can be formulated as in equation 4.13 below, where y_1 is the y-coordinate of the point P_2 and x_2 is the x-coordinate of the point P_3.

$$\tan(\alpha_0) = \frac{x_2}{y_1 - y_2}, \quad \text{where} \quad x_2 \in (0\infty), y_1 \in (0,\infty) \tag{4.13}$$

The solution for the liquid and the solid phase to the heat equation close to the origin after a time step $\Delta\tau$ is formulated in equation 4.14 and 4.15 respectively. Note that these two equation are equation 4.8 and 4.10 without the curvature term κ_2 in theorem 1, in other words, the solid phase is assumed to be non curved. In equation 4.14 and 4.15 the distances d_T and d_N are defined as $d_T = r_0\cos(\theta_0 - \alpha_0)$ and $d_N = r_0\sin(\theta_0 - \alpha_0)$ respectively. Furthermore, the triple (u_1,u_2,u_3) can be obtained from equaton 4.14 and 4.15 in combination with the mass conservation law $u_1 + u_2 + u_3 = 1$.

$$\begin{aligned} u_1(x_0,y_0,d_N,d_T,\Delta\tau) = {} & \frac{\alpha_0}{2\pi} + \frac{\kappa_1\sqrt{\Delta\tau}}{2\sqrt{\pi}} + \frac{x_0 - d_T}{4\sqrt{\pi}\sqrt{\Delta\tau}} - \frac{\kappa_1 d_N}{\pi} \\ & + \frac{\alpha_0(x_0^2 + y_0^2) + d_N d_T - x_0 y_0}{8\pi\Delta\tau} - \frac{3\kappa_1 d_N^2}{8\sqrt{\pi}\sqrt{\tau}} + O\left(\frac{r_0^3}{\sqrt{\Delta\tau}}\right) \end{aligned} \tag{4.14}$$

$$u_3(x_0,y_0,\Delta\tau) = \frac{1}{2} - \frac{x_0}{2\sqrt{\pi}\sqrt{\Delta\tau}} + \frac{x_0^2 + y_0^2}{8\Delta\tau} + O\left(\frac{r_0^3}{\sqrt{\Delta\tau}}\right) \tag{4.15}$$

The following idea will be used in order to derive a convolution thresholding scheme for a straight solid wall such that the angle α_0 is preserved. Begin by plugging in the point P_3 in equation 4.14 and 4.15, and solve these equations for the coordinate x_2. Use the angle preservation condition stated in equation 4.13 in order to obtain an equation for the behaviour of u_1 and u_3 at the interface after a time $\Delta\tau$. Finally, the obtained equations can be used in order to numerically test the validity of the convolution expansion.

Initially, introduce the following notation in terms of the convolutions in equation 4.14 and 4.15 respectively,

$$\begin{aligned}
\hat{u}_1(x_0,y_0) &= u_1(x_0,y_0,d_N,d_T,\Delta\tau) - O\left(\frac{r_0^3}{\sqrt{\Delta\tau}}\right) \\
\hat{u}_3(x_0,y_0) &= u_3(x_0,y_0,\Delta\tau) - O\left(\frac{r_0^3}{\sqrt{\Delta\tau}}\right) \\
u_1^\star &= \hat{u}_1(x_2,y_2) = \hat{u}_1(P_3) \\
u_3^\star &= \hat{u}_3(x_2,y_2) = \hat{u}_1(P_3)
\end{aligned}$$

and further define the quotient $Q = \dfrac{y_2}{x_2}$ where $Q \in [0,\infty)$ such that the point P_3 is given by $P_3 = (x_2,\, Q \cdot x_2)$. Then, it follows that $\theta_2 = \tan^{-1}(Q)$, and the value of $d_T(P_3)$ is given by the following calculation.

$$\begin{aligned}
d_T(P_3) &= r_0(P_3)\cos(\theta_2 - \alpha_0) \\
&= \left(\sqrt{x_2^2 + \underbrace{y_2^2}_{=Q^2x_2^2}}\right)\cos(\theta_2 - \alpha_0) \\
&= x_2(\sqrt{1+Q^2})\cos(\theta_2 - \alpha_0)
\end{aligned}$$

and similarly it follows that $d_N(P_3) = x_2(\sqrt{1+Q^2})\sin(\theta_2 - \alpha_0)$. Thus plugging in these expressions into equation 4.14 and 4.15 yields the following quadratic equation with respect to x_2.

$$\begin{aligned}
u_1^\star &= \frac{\alpha_0}{2\pi} + \frac{\kappa_1\sqrt{\Delta\tau}}{2\sqrt{\pi}} + \left[\frac{1 - \sqrt{(1+Q^2)}\cos(\theta_2 - \alpha_0)}{4\sqrt{\pi}\sqrt{\Delta\tau}} - \frac{\kappa_1\sqrt{(1+Q^2)}\sin(\theta_2 - \alpha_0)}{\pi}\right] x_2 \\
&\quad + \left[\frac{\alpha_0(1+Q^2) + (1+Q^2)\cos(\theta_2 - \alpha_0)\sin(\theta_2 - \alpha_0) - Q}{8\pi\Delta\tau} - \frac{3\kappa_1(1+Q^2)\sin^2(\theta_2 - \alpha_0)}{8\sqrt{\pi}\sqrt{\tau}}\right] x_2^2 \\
u_3^\star &= \frac{1}{2} - \frac{x_2}{2\sqrt{\pi}\sqrt{\Delta\tau}} + \left(\frac{1+Q^2}{8\Delta\tau}\right) x_2^2
\end{aligned}$$

The solutions to these equations are given in equation 4.16 and 4.17

$$x_2 = -\frac{B}{2A} \pm \sqrt{\frac{C(u_1^\star)}{A} + \frac{B^2}{4A^2}} \tag{4.16}$$

$$x_2 = \frac{D}{2E} \pm \sqrt{\frac{F(u_3^\star)}{E} + \frac{D^2}{4E^2}} \tag{4.17}$$

where the following list of constants holds.

$$A = \frac{\alpha_0(1+Q^2) + (1+Q^2)\cos(\theta_2-\alpha_0)\sin(\theta_2-\alpha_0) - Q}{8\pi\Delta\tau} - \frac{3\kappa_1(1+Q^2)\sin^2(\theta_2-\alpha_0)}{8\sqrt{\pi}\sqrt{\tau}}$$

$$B = \frac{1-\sqrt{(1+Q^2)}\cos(\theta_2-\alpha_0)}{4\sqrt{\pi}\sqrt{\Delta\tau}} - \frac{\kappa_1\sqrt{(1+Q^2)}\sin(\theta_2-\alpha_0)}{\pi}$$

$$C(u_1^\star) = u_1^\star - \frac{\alpha_0}{2\pi} - \frac{\kappa_1\sqrt{\Delta\tau}}{2\sqrt{\pi}}$$

$$D = \frac{1}{2\sqrt{\pi}\sqrt{\Delta\tau}}$$

$$E = \frac{1+Q^2}{8\Delta\tau}$$

$$F(u_3^\star) = u_3^\star - \frac{1}{2}$$

Using the condition, formulated in equation 4.13, in the form $x_2 = \tan(\alpha_0)(y_1 - y_2)$ in equation 4.16 and 4.17 respectively, an expression for $u_1^\star$ and $u_3^\star$ can be obtained.

$$u_1^\star = \frac{\alpha_0}{2\pi} + \frac{\kappa_1\sqrt{\Delta\tau}}{2\sqrt{\pi}} - \frac{B^2}{4A} + \left(A \cdot \left[\tan(\alpha_0)(y_1-y_2) + \frac{B}{2A}\right]^2\right) \tag{4.18}$$

$$u_3^\star = \frac{1}{2} - \frac{D^2}{4E} + \left(E \cdot \left[\tan(\alpha_0)(y_1-y_2) - \frac{D}{2E}\right]^2\right) \tag{4.19}$$

4.1.2 Analytical projection triangle

Here follows an investigation of the validity of equation 4.18 and 4.19. Assuming that after a time interval $\Delta\tau = 1 \cdot 10^{-6}$ the triple junction is located at the height $y_1 = 0.0056569$ given a contact angle $\alpha_0 = \frac{\pi}{4}$ and a curvature $\kappa_1 = 3$. Provided the described situation, the question to answer is, at what point $P_3 = (x_2,y_2)$ should the angle α_0 be conserved in order to obtain an appropriate appearance of the projection triangle? Assuming that the desired shape of the interface between the gas and liquid phase after one time step $\Delta\tau$ is a convex meniscus, points closer to the solid wall than P_3 should have a slightly smaller contact angle with the new triple junction, denoted P_2, than α_0. Similarly, points that is farther away from the solid wall than P_3 should have a larger contact angle with the triple junction P_2.

Thus, the convolutions given in equation 4.18 and 4.19 will be calculated for four different points, and the value of these points will then be connected to the triple junction located at $\left(\frac{\alpha_0}{2\pi}, \frac{\pi-\alpha_0}{2\pi}, \frac{1}{2}\right) = \left(\frac{1}{8}, \frac{3}{8}, \frac{1}{2}\right)$ in the projection triangle and the point $\left(\frac{1}{2}, \frac{1}{2}, 0\right)$

corresponding to mean curvature flow. The four points for which the convolutions will be calculated are depicted in figure 4.3 below.

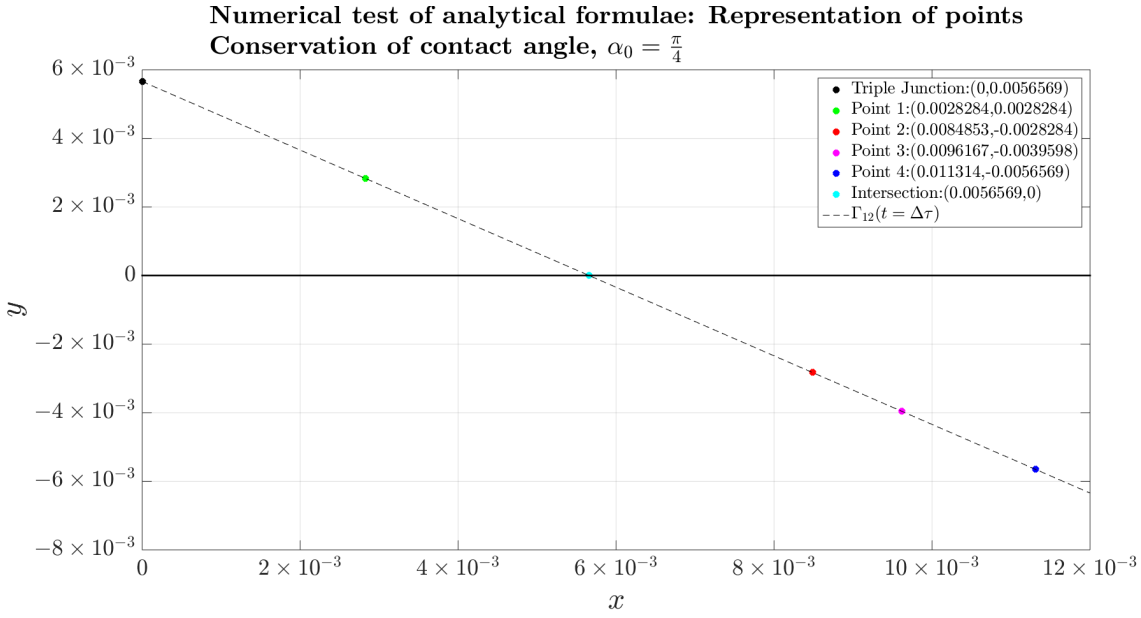


Figure 4.3: Points for conserving contact angle α_0.

The resulting convolutions of the four points in figure 4.3 are illustrated in figure 4.4 below. The expected result is that a point close to is to the solid phase, that is the closer the $x-$coordinate for the points in figure 4.3 is to zero, results in a high value of $u_3^\star$ at that point, since the diffusion from the solid phase results in a high concentration in proximity to the wall. The results in figure 4.4 confirm this, since point 1 has the highest value of $u_3^\star$, namely $u_3^\star \approx 0.45358$, point 2 has the second highest value of $u_3^\star$, namely $u_3^\star \approx 0.38074$, point 3 has the second lowest value of $u_3^\star$, namely $u_3^\star \approx 0.37578$ and point 4 has the lowest value of $u_3^\star$, namely $u_3^\star \approx 0.37432$. By comparing the dashed line in figure 4.4 that has the appearance that mostly resembles the line corresponding to the liquid gas interface in the projection triangle obtained by the projection triangle algorithm, an estimate of the distance at which the contact angle is preserved can be obtained.

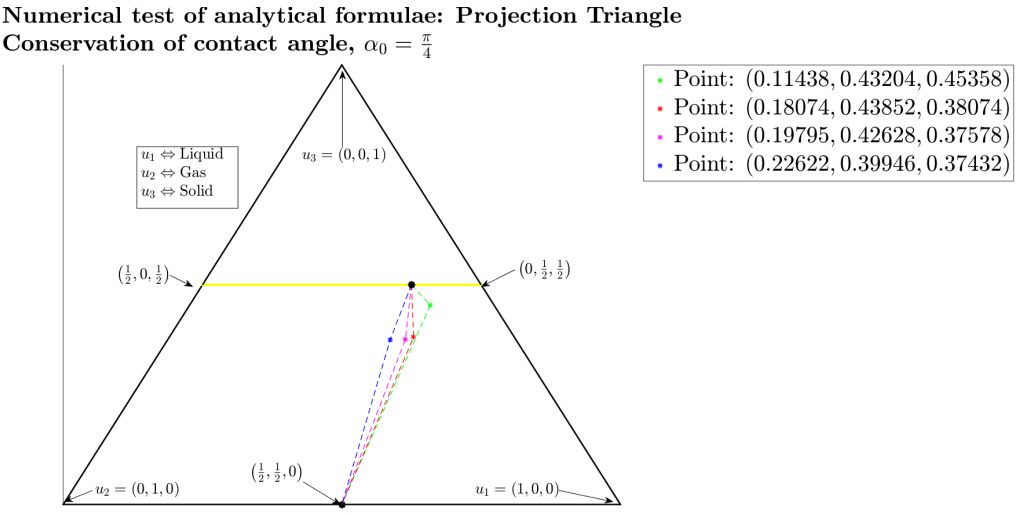
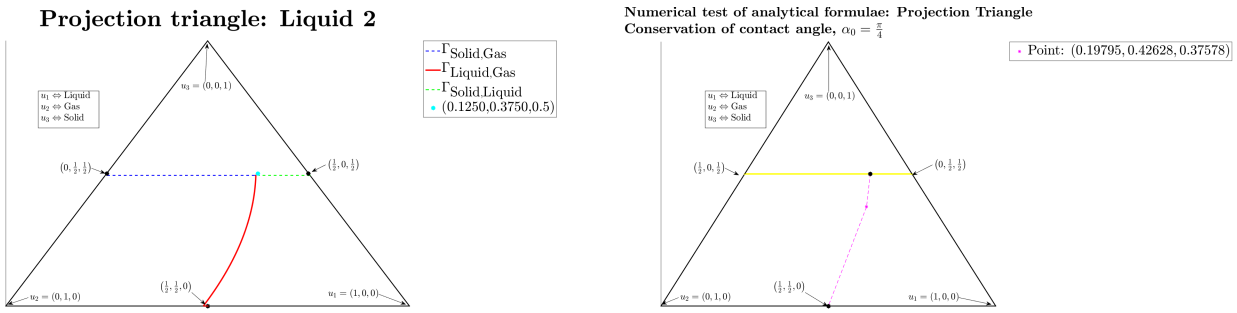


Figure 4.4: Projection triangle for conserving contact angle α_0.

Consequently, the appropriate distance from the wall for the given situation in order to preserve the contact angle is in proximity of point three, which is illustrated in figure 4.5 below. As can be seen, the liquid-gas interface generated from point 4 in figure 4.3, that is the green curve in figure 4.5b, is similar to the interface between the liquid-gas interface generated by the projection triangle algorithm, that is the red curve in figure 4.5a. Thus the conclusion from the numerical experiments in figure 4.4 is that the projection triangle algorithm on page 33 preserves the contact angle for each iteration, which is a necessary condition in order to simulate capillary flow.



(a) Projection triangle algorithm. **(b)** Convolutions at point 4.

Figure 4.5: Comparasion of numerical and analytical approach.

Thus, the projection triangle algorithm can be used in order to simulate capillary flow, such that the contact angle is preserved after each iteration. Similarly, a convolution calculation for a three phase system in proximity to the triple junction in $\mathbb{R}^3$ can be conducted.

4.2 Calculations in $\mathbb{R}^3$

In similarity with the convolution calculation in $\mathbb{R}^2$, the assymptotic expansion will be calculated using spherical coordinates where center of the triple junction is located at the origin as depicted in figure 4.6.

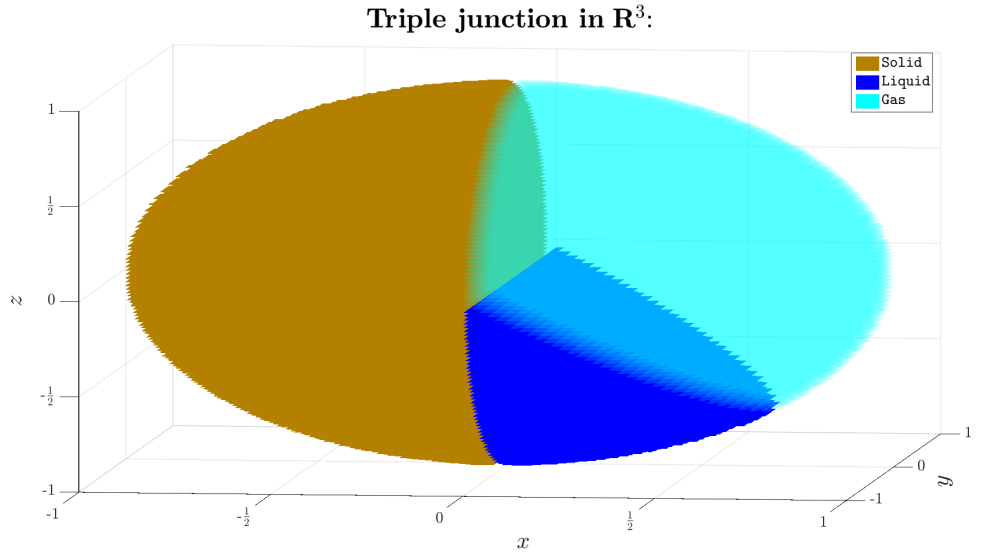


Figure 4.6: Triple junction in $\mathbb{R}^3$.

In this situation, the solution to the heat equation after a time $\Delta\tau$ in $\mathbb{R}^3$ is given by equation 4.20. Note that the spherical coordinates $x = r\cos(\theta)\sin(\phi)$, $z = r\sin(\theta)\sin(\phi)$ and $y = \cos(\phi)$ is used where $\theta \in [0,\pi]$ and $\phi \in [0,\pi]$.

$$u_1(\mathbf{x}_0,\Delta\tau) = \frac{\exp\left(\frac{-r_0^2}{4\Delta\tau}\right)}{8(\sqrt{\pi})^3(\sqrt{\Delta\tau})^3}\int_0^R \exp\left(\frac{-r^2}{4\Delta\tau}\right)\int_{\alpha_3}^{\alpha_1}\int_{\phi_2}^{\phi_1} r^2\exp\left(\frac{rr_0\cos(\theta_0-\theta)}{2\Delta\tau}\right)\sin(\phi)\mathrm{d}\phi\mathrm{d}\theta\mathrm{d}r \tag{4.20}$$

Then the solution to the heat equation after a time $\Delta\tau$ in proximity of the triple junction is given by theorem 2.

Theorem 2. *Suppose $\Omega \subset \mathbb{R}^3$ be a domain divided into three subdomains, namely a solid phase Ω_3, a gas phase Ω_2 and a liquid phase Ω_1. Furthermore, suppose the angle configuration, denoted $(\alpha_1,\alpha_2,\alpha_3)$, of the three phase system satisfies equation 4.3, 4.4 and 4.5 as depicted in figure 4.1. In addition, suppose that the three phases meet at the so called triple junction centered at the origin, where the quotient between the radius, $r_0 = \sqrt{x_0^2+y_0^2+z_0^2}$ and the time step, $\Delta\tau$, is small, that is $\frac{r_0}{\sqrt{\Delta\tau}} \approx 0$. Then the solution to the system in equation 4.1 close to the triple junction satisfies equation 4.21, 4.22 and 4.23 below, where the distances d_N and d_T are defined as $d_N = r_0\sin(\theta_0-\alpha_0)\sin(\phi_0)$ and*

$d_T = r_0\cos(\theta_0 - \alpha_0)\sin(\phi_0)$ *respectively.*

$$\begin{aligned} u_1(x_0,y_0,z_0,d_N,d_T,\Delta\tau) &= \frac{\alpha_0}{2\pi} + \frac{2(\kappa_1-\kappa_2)\sqrt{\Delta\tau}}{(\sqrt{\pi})^3} + \frac{x_0 - d_T}{4\sqrt{\pi}\sqrt{\Delta\tau}} \\ &+ \frac{3(\kappa_2 z_0 - \kappa_1 d_N)}{4} + \frac{\alpha_0(x_0^2+y_0^2+z_0^2) + d_N d_T - x_0 z_0}{4\pi\Delta\tau} \\ &+ \frac{4(\kappa_1 d_N^2 - \kappa_2 d_T^2 + 2\cdot(\kappa_1-\kappa_2)y_0^2)}{3(\sqrt{\pi})^3\sqrt{\Delta\tau}} + O\left(\frac{r_0^3}{\sqrt{\Delta\tau}}\right) \end{aligned} \tag{4.21}$$

$$\begin{aligned} u_2(x_0,y_0,z_0,d_N,d_T,\Delta\tau) &= \frac{\pi - \alpha_0}{2\pi} + \frac{2(\kappa_2-\kappa_1)\sqrt{\Delta\tau}}{(\sqrt{\pi})^3} + \frac{x_0 + d_T}{4\sqrt{\pi}\sqrt{\Delta\tau}} \\ &+ \frac{3(\kappa_2 z_0 + \kappa_1 d_N)}{4} + \frac{x_0 z_0 - d_N d_T - (\alpha_0+\pi)(x_0^2+y_0^2+z_0^2)}{4\pi\Delta\tau} \\ &+ \frac{4(\kappa_2 d_T^2 - \kappa_1 d_N^2 - 2\cdot(\kappa_1-\kappa_2)y_0^2)}{3(\sqrt{\pi})^3\sqrt{\Delta\tau}} + O\left(\frac{r_0^3}{\sqrt{\Delta\tau}}\right) \end{aligned} \tag{4.22}$$

$$u_3(x_0,y_0,z_0,\Delta\tau) = \frac{1}{2} - \frac{x_0}{2\sqrt{\pi}\sqrt{\Delta\tau}} - \frac{2\kappa_2 z_0}{\pi} + \frac{x_0^2+y_0^2+z_0^2}{4\Delta\tau} + O\left(\frac{r_0^3}{\sqrt{\Delta\tau}}\right) \tag{4.23}$$

Proof

In the same manner, introduce the following spherical coordinates, where $\theta \in [0,2\pi]$ *and* $\phi \in [0,\pi]$.

$$\begin{aligned} x_0 &= r_0\cos(\theta)\sin(\phi) \\ z_0 &= r_0\sin(\theta)\sin(\phi) \\ y_0 &= r_0\cos(\phi) \end{aligned}$$

Then, expanding the solution in the same manner as in the proof of theorem 1 yields the desired result.

■

Close to the triple junction, the result in 2 is similar to the convolution in $\mathbb{R}^2$. Again, setting $r_0 = 0$ in equation 4.21 to 4.23 results in the position of the triple junction in the phase portrait at $\left(\frac{\alpha_0}{2\pi}, \frac{\pi-\alpha_0}{2\pi}, \frac{1}{2}\right)$. However, since the implementation of the projection algorithm, see section 3.2 on page 50, concerned two liquids in $\mathbb{R}^2$ the justification that was conducted in $\mathbb{R}^2$ has not been performed in $\mathbb{R}^3$. However, using the exact same method, it is possible to implement a convolution threshold scheme in $\mathbb{R}^3$. With the justification of the projection triangle algorithm at hand, the simulations of capillary flows in arbitrary domains are represented in the next chapter.

5

Results

The section contains the generated results from the implemented algorithm. On the one hand, the simulations concern the affect of the contact angle on capillary flow and on the other hand, the capillary flow through complex domains is simulated. The results illustrate the evolution of a three phase system that is composed of a solid, gas and liquid phase, and each plot illustrates a domain in either $\mathbb{R}^2$ or $\mathbb{R}^3$ after the solution algorithm on page 34 has been applied for a given number of time steps, denoted $\Delta\tau$.

5.1 Effect of changing contact angle

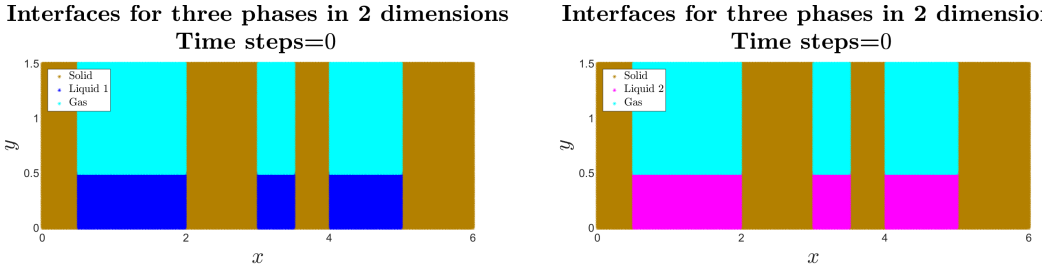
In the following section, two simulations of the capillary flow of two liquids with different contact angle are presented. The first simulation concerns the capillary flow through narrow channels through the solid phase of varying width. The second simulation concerns the capillary flow through a narrow channel with circular obstacles placed in the channel. In both these simulations, the nine point stencil is employed during the diffusion step.

5.1.1 Narrow channels of varying width

In figure 5.1 to 5.3, the capillary flow trough channels of varying width is simulated. The left hand figure in all of these figures illustrates the capillary flow for liquid 1 illustrated in figure 3.11a on page 51, and the right hand side in these figures illustrates the capillary flow for liquid 2 illustrated in figure 3.11b on page 51. The simulations in figure 5.1 to 5.3 are obtained using the solution algorithm for a three phase system on page 34 in combination with the projection triangles illustrated in figure 3.12a and 3.12b on page 53.

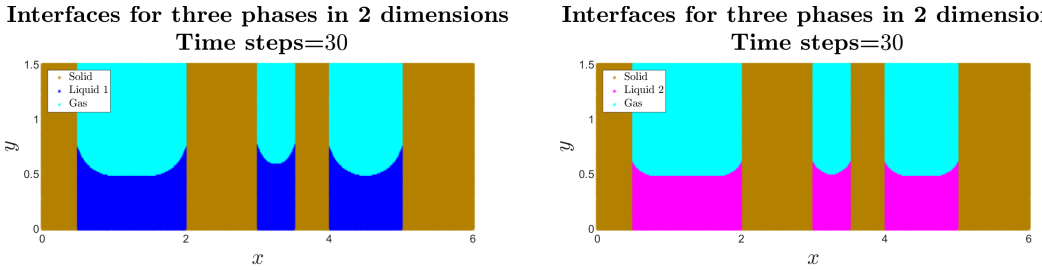
In figure 5.1 to 5.3, the following settings are used. The grid size is set to 600×600 and the step size is $h = \dfrac{6}{600} = 0.01$, the time step in the Euler forward discretization is

$\Delta t = \frac{h^2}{10} = 1 \cdot 10^{-5}$ and the time interval for which the diffusion step is carried out is set to $\Delta\tau = 80 \cdot \Delta t = 8 \cdot 10^{-4}$.



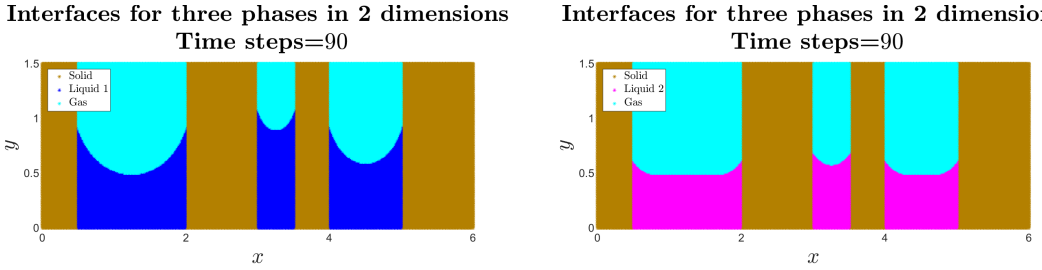
(a) Liquid 1. **(b)** Liquid 2.

Figure 5.1: Flow in channels of varying width, 0 times steps.



(a) Liquid 1. **(b)** Liquid 2.

Figure 5.2: Flow in channels of varying width, 30 times steps.



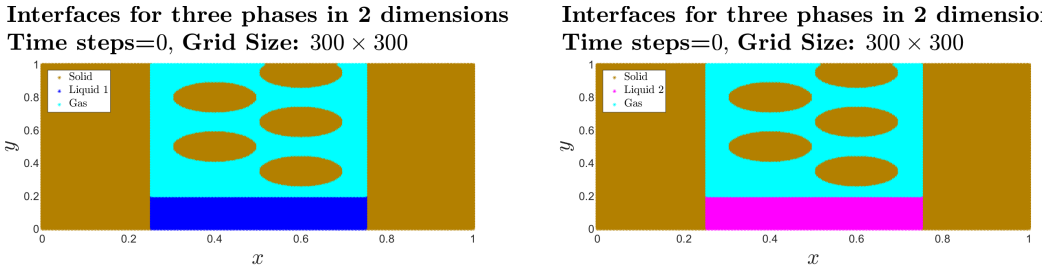
(a) Liquid 1. **(b)** Liquid 2.

Figure 5.3: Flow in channels of varying width, 90 times steps.

In figure 5.1 to 5.3, the rise of liquid 1 is higher than that of liquid 2. Next the capillary flow through a narrow fiber channel with various obstacles placed in the channel is presented.

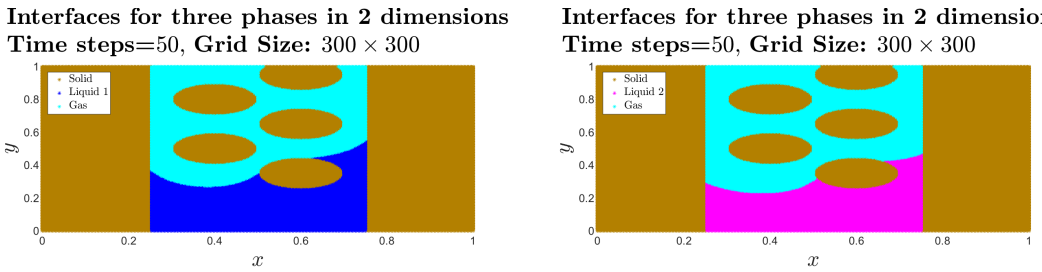
5.1.2 Narrow channel filled with obstacles

In figure 5.4 to 5.8 the capillary flow trough a narrow channel filled with circular obstacles is simulated. Again the same two liquids illustrated in figure 3.11a and 3.11b on page 51 were simulated. However, the algorithm used in order to generate these simulations is slightly modified from the solution algorithm on page 34. In the sharpening step, the fiber phase is always "reset" to its initial state, so that the fiber phase remain stationary troughout the simulation. Otherwise, the liquid and gas phase are sharpened using the line $\hat{\Gamma}_{12}$ in figure 3.12a on page 53. In figure 5.4 to 5.8 the grid size is 300×300, the step size is $h = \frac{1}{300} \approx 0.0033$, the time step is $\Delta t = \frac{h^2}{10} \approx 1.1 \cdot 10^{-6}$ and the diffusion step is conducted for a time interval of size $\Delta\tau = 3.3 \cdot 10^{-4}$.



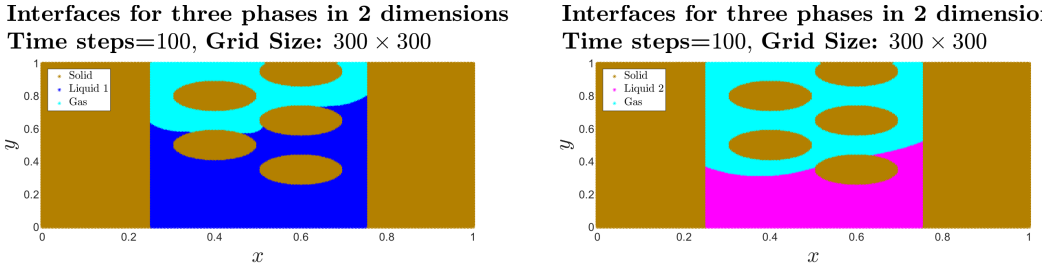
(a) Liquid 1. **(b)** Liquid 2.

Figure 5.4: Flow through channel with obstacles, 0 time steps.



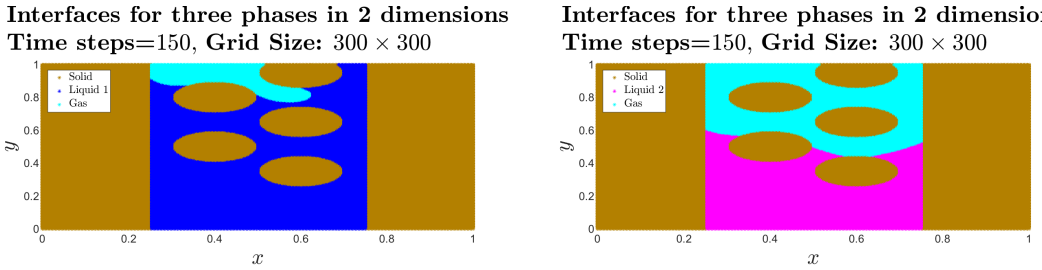
(a) Liquid 1. **(b)** Liquid 2.

Figure 5.5: Flow through channel with obstacles, 50 time steps.



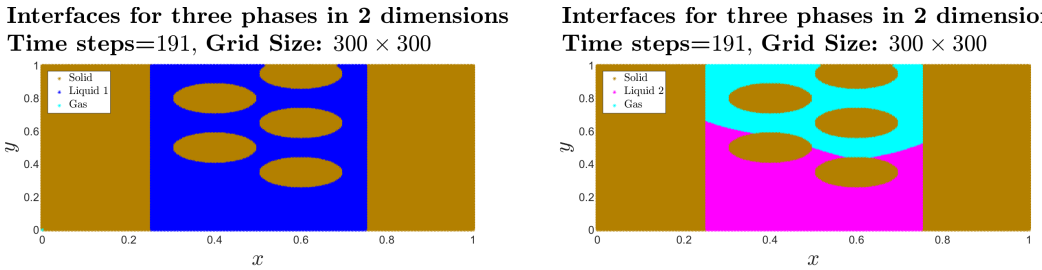
(a) Liquid 1. **(b)** Liquid 2.

Figure 5.6: Flow through channel with obstacles, 100 time steps.



(a) Liquid 1. **(b)** Liquid 2.

Figure 5.7: Flow through channel with obstacles, 150 time steps.



(a) Liquid 1. **(b)** Liquid 2.

Figure 5.8: Flow through maize, 191 time steps.

In figure 5.4 to 5.8, liquid 1 flows through the narrow channel while liquid 2 reaches a steady state. Next the simulations of capillary flow through complex domains are presented.

5.2 Capillary flow through complex domains

Two simulations of capillary flow through complex domains are presented. The first simulation contains a simulation through a complex domain in $\mathbb{R}^2$ and the second simulation illustrate capillary flow through a complex domain in $\mathbb{R}^3$. Both these simulations illustrates the capillary flow of liquid 1 illustrated in figure 3.11a on page 51, in other words, in the sharpening step, the projection triangle in figure 3.12a on page 53 is applied.

5.2.1 Simulation in $\mathbb{R}^2$

In figure 5.9 to **??** the simulation through a complicated domain in $\mathbb{R}^2$ is illustrated. To conduct the diffusion step, the nine point stencil is employed with a grid size of 400×400. Further, the step size is set to $h = \frac{2}{400} = 0.005$, the time step is set to $\Delta t = 2.5 \cdot 10^{-6}$ and the diffusion step is conducted for a time interval $\Delta\tau = 5 \cdot 10^{-4}$.

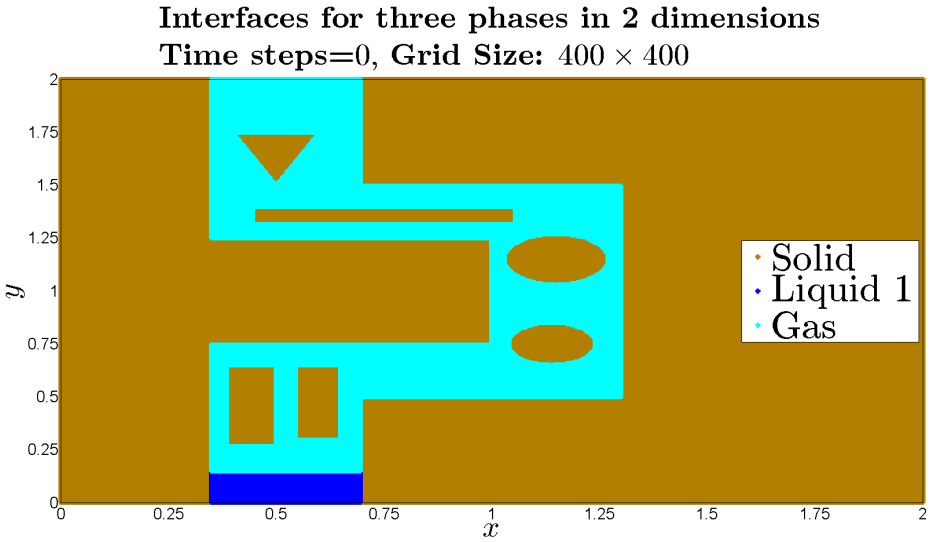


Figure 5.9: 2D Simulation of capillary flow, 0 time steps

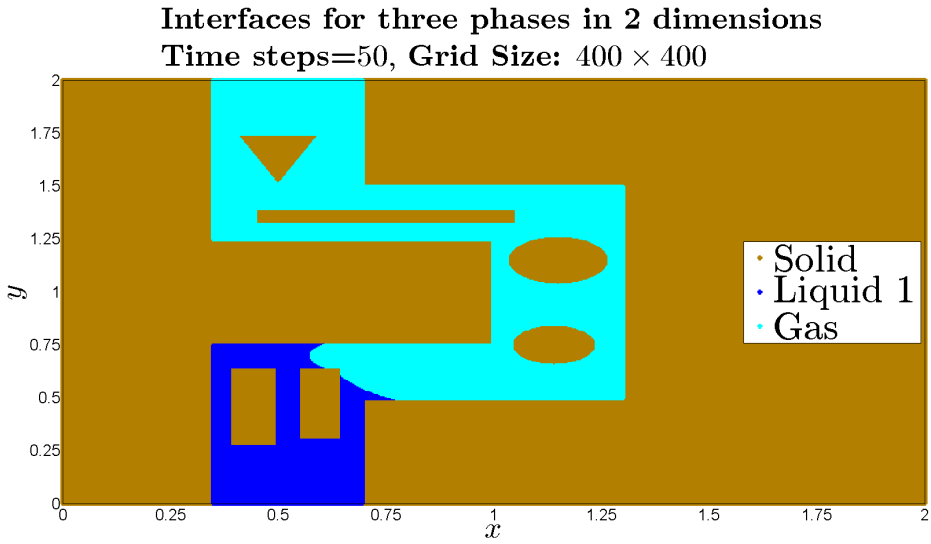


Figure 5.10: 2D Simulation of capillary flow, 50 time steps

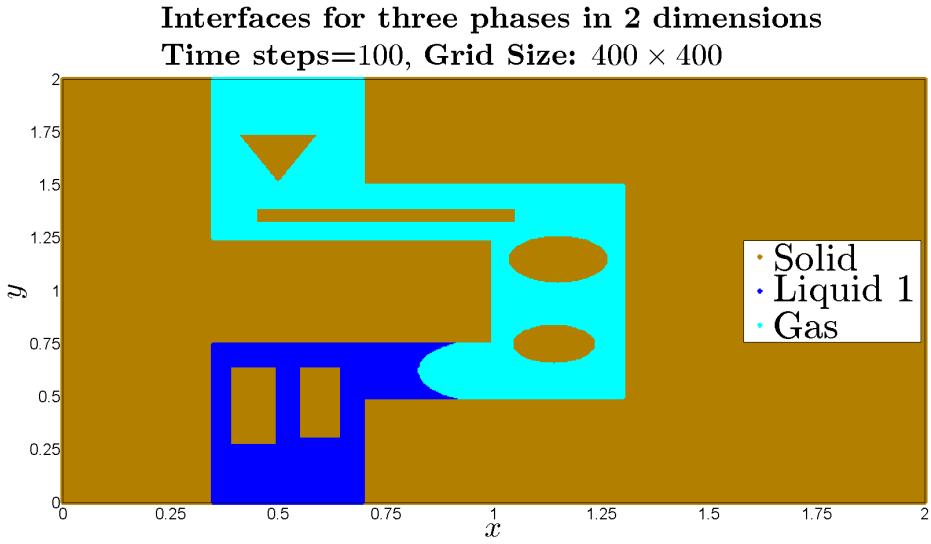


Figure 5.11: 2D Simulation of capillary flow, 100 time steps

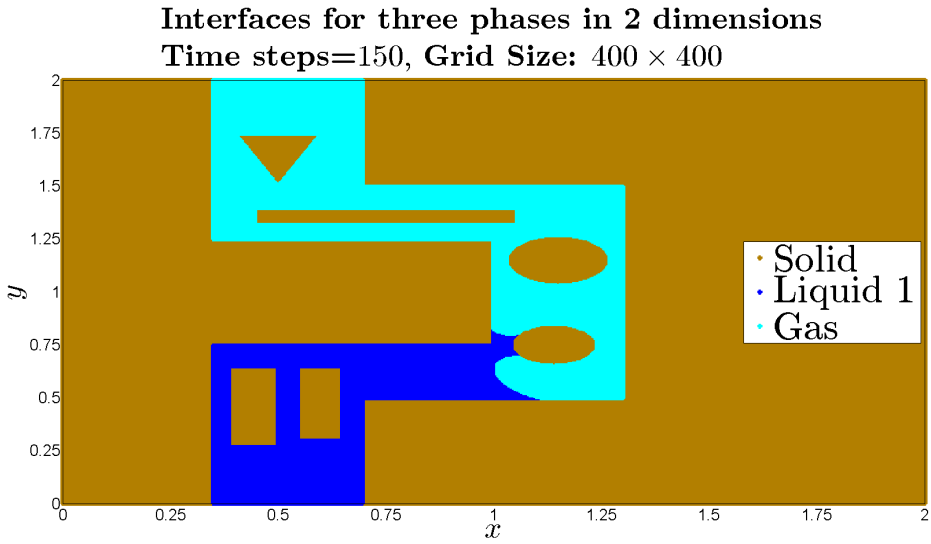


Figure 5.12: 2D Simulation of capillary flow, 150 time steps

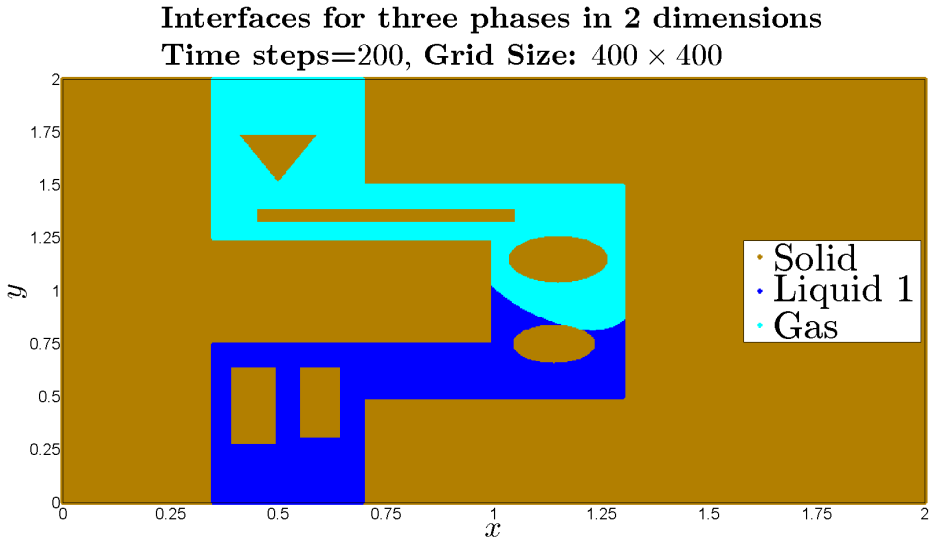


Figure 5.13: 2D Simulation of capillary flow, 200 time steps

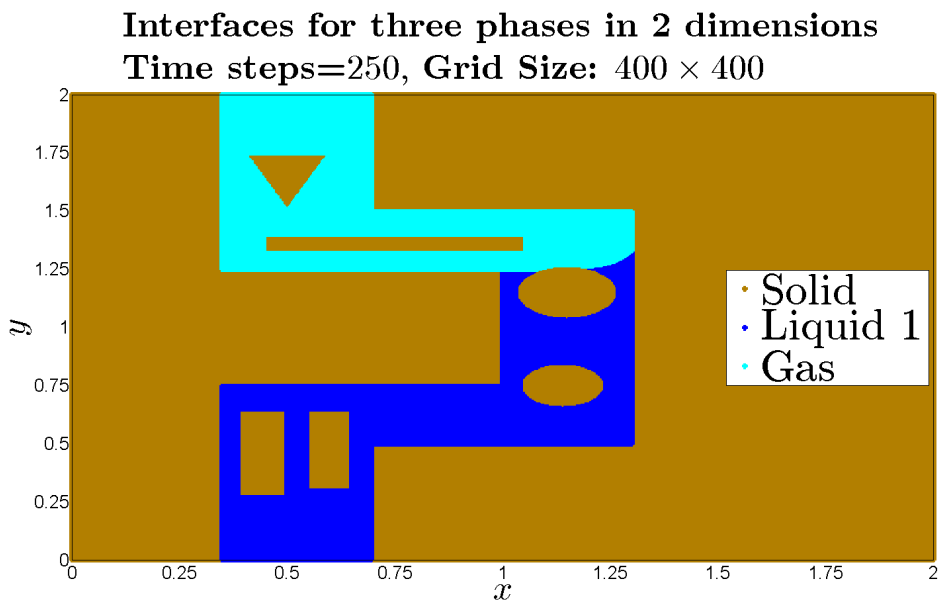


Figure 5.14: 2D Simulation of capillary flow, 250 time steps

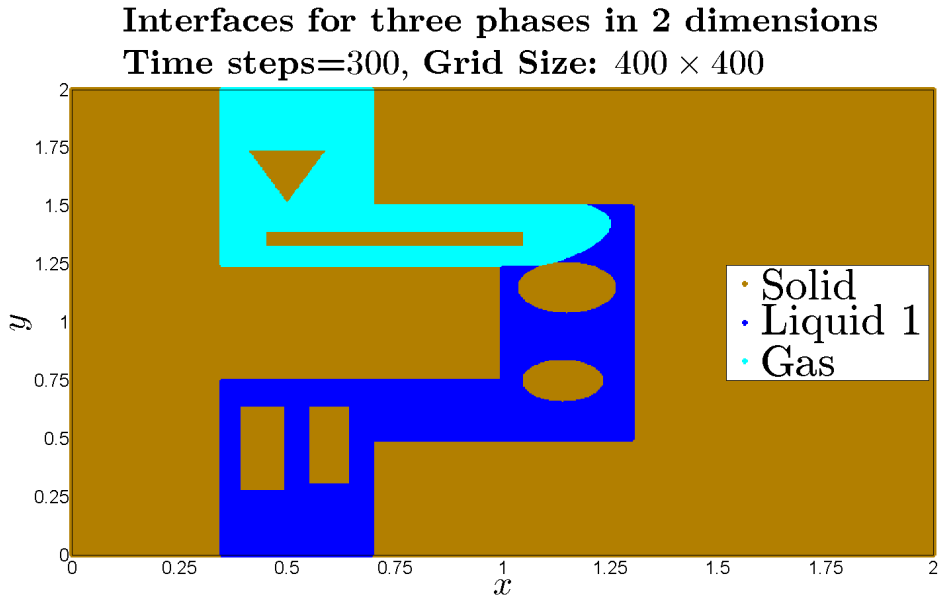


Figure 5.15: 2D Simulation of capillary flow, 300 time steps

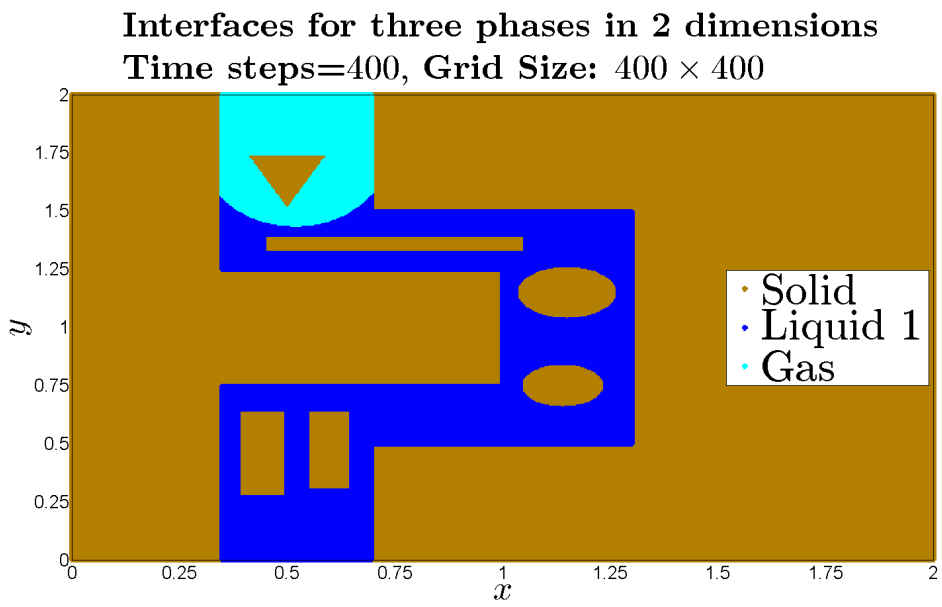


Figure 5.16: 2D Simulation of capillary flow, 400 time steps

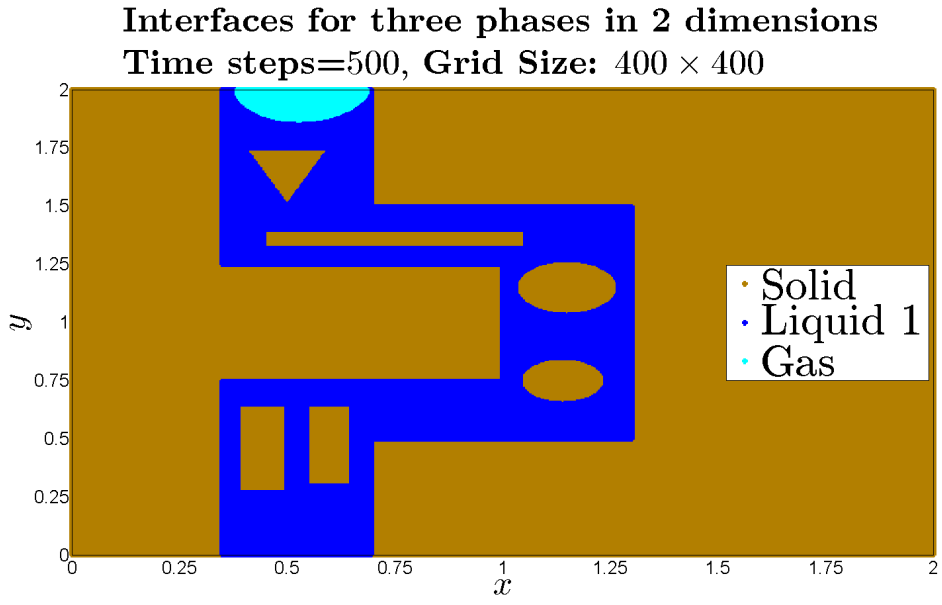


Figure 5.17: 2D Simulation of capillary flow, 500 time steps

The results in figure 5.9 to 5.11 illustrate that the capillary flow through the narrow parts of the solid labyrinth is faster compared to the wider part of the labyrinth.

5.2.2 Simulation in $\mathbb{R}^3$

In figure 5.17 to 5.21, the numerical simulation of capillary flow in $\mathbb{R}^3$ is illustrated. In the simulation, the capillary flow through two differently shaped pores that are connected by a channel is illustrated. Otherwise, the projection triangle in figure 3.12a on page 53 is implemented during the sharpening step with the modification that the solid phase is "reset" during the sharpening step. Furthermore, the heat equation is solved numerically using the 27 point stencil of the Laplace operator. The grid size of these simulations is $150 \times 150 \times 150$, the step size is $h = \frac{1}{150} \approx 0.0067$, the time step is $\Delta t = \frac{h^2}{10} \approx 4.4 \cdot 10^{-6}$ and the diffusion step is carried out for a time interval of size $\Delta\tau = 100 \cdot \Delta t \approx 4.4 \cdot 10^{-4}$.

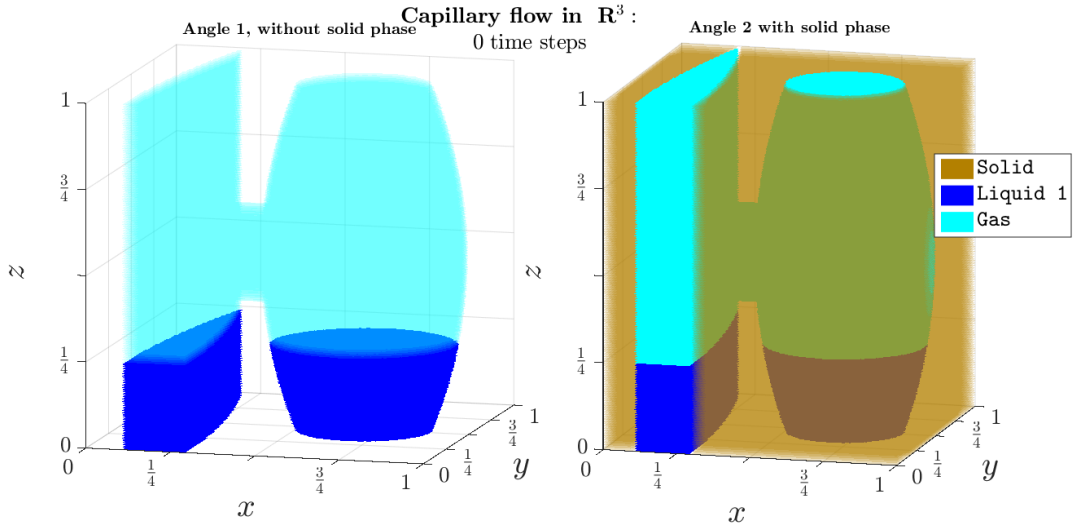


Figure 5.18: 3D Simulation of capillary flow, 0 time steps

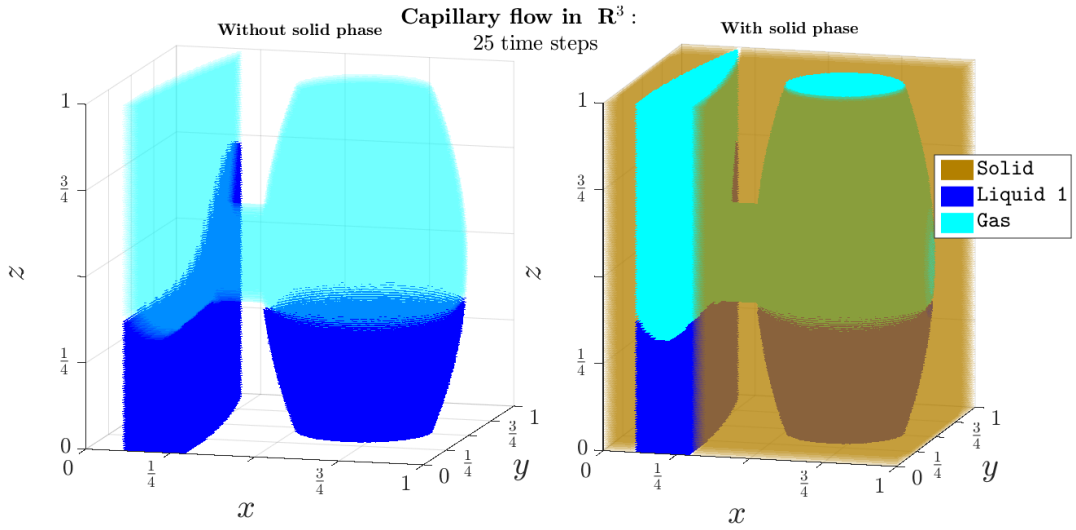


Figure 5.19: 3D Simulation of capillary flow, 25 time steps

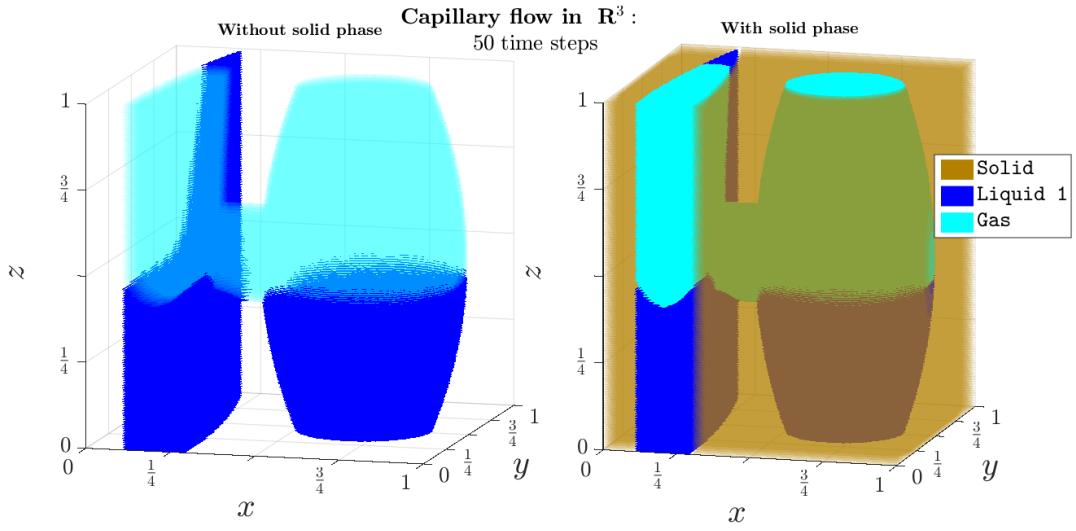


Figure 5.20: 3D Simulation of capillary flow, 50 time steps

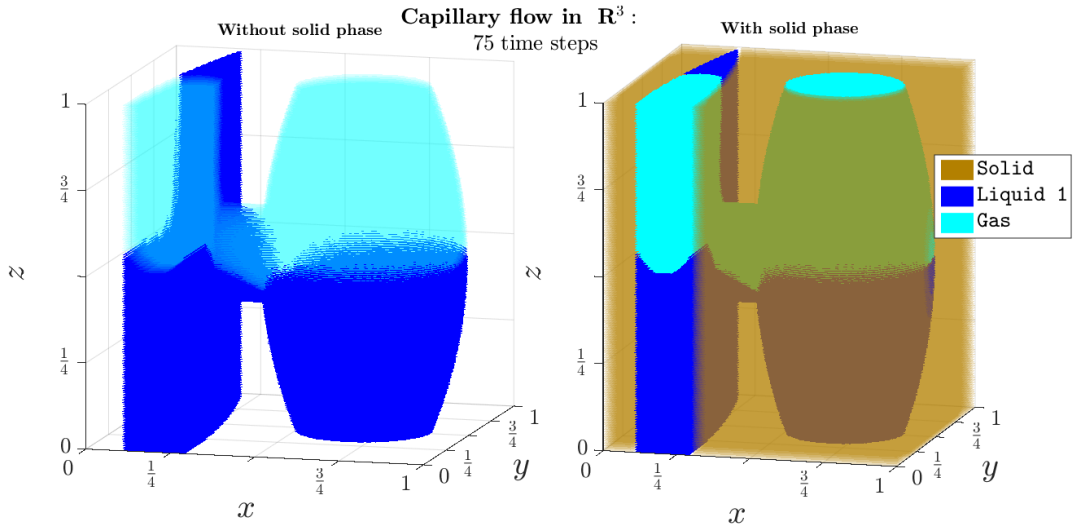


Figure 5.21: 3D Simulation of capillary flow, 75 time steps

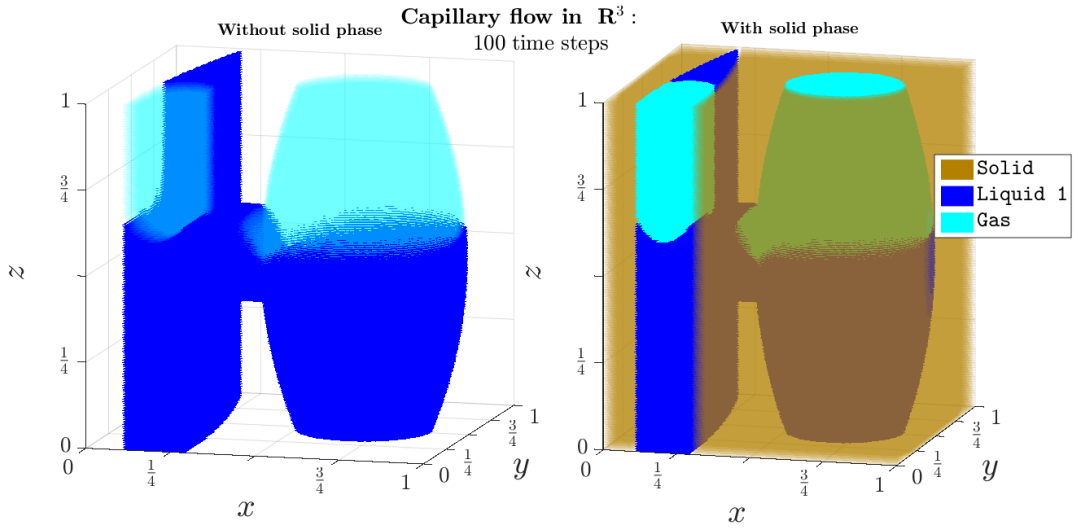


Figure 5.22: 3D Simulation of capillary flow, 100 time steps

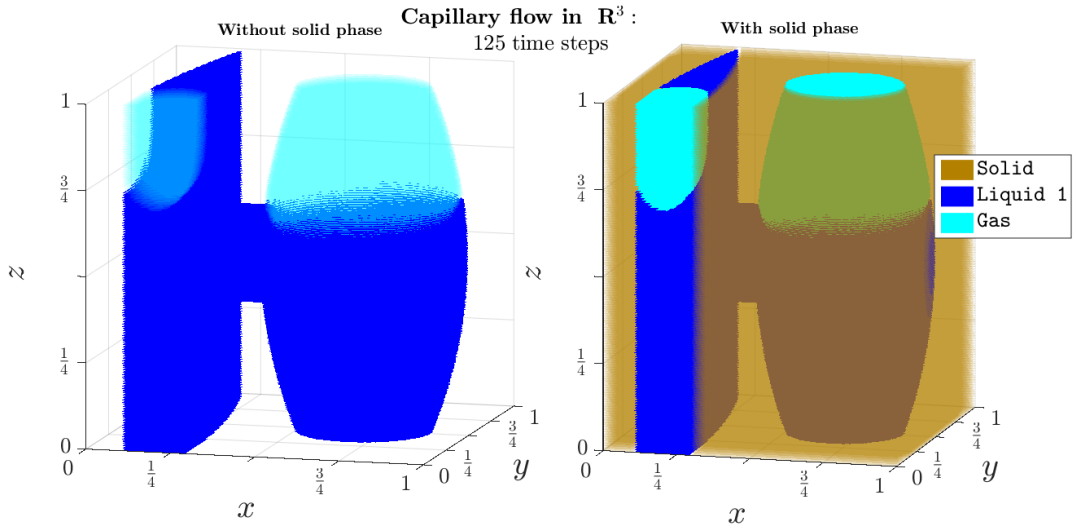


Figure 5.23: 3D Simulation of capillary flow, 125 time steps

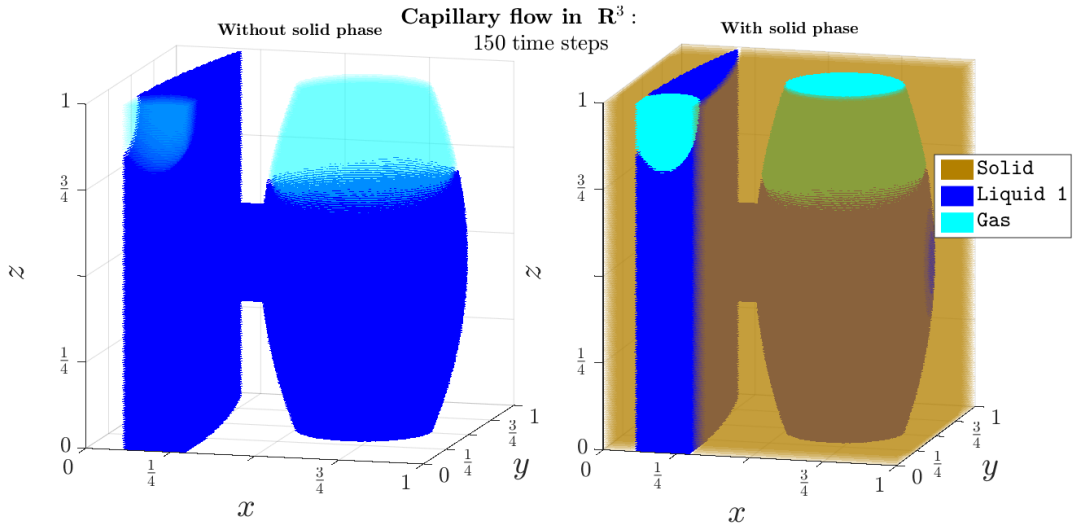


Figure 5.24: 3D Simulation of capillary flow, 150 time steps

The results in figure 5.17 to 5.21 illustrate two properties. The capillary flow through the right hand pore is symmetric, that is the water surface remains symmetric throughout the simulation. Concerning the capillary flow in the left hand pore, the water surface at the more narrow part of the pore evolves faster compared to the more wider part of the pore. Next, a discussion of the findings is presented in the next chapter.

6

Discussion

In summary, the capillary flow for a three phase system using the solution algorithm on page 34 was succesfully simulated. The algorithm could also simulate arbitrarily complex domains, which is an appealing property. Furthermore, the motion of the triple junction after a time interval $\Delta\tau$ was derived using a convolution thresholding calculation in chapter 4. This, calculation confirms the fact that the contact angle α_0 is preserved at a certain point in proximity to the solid wall in the projection triangle algorithm. The overall objective was hence satisfied, but here follows a more detail discussion of the results. The discussion is divided into three parts, namely the implementation of the algorithm, the numerical results and future work.

6.1 Implementation of the algorithm

There are two topics to discuss concerning the implementation of the algorithm, namely the implementation of the diffusion step, that is the modelling of the mean curvature flow, and the implementation of projection triangle algorithm.

6.1.1 Diffusion step

The implementation of the diffusion step is highly accurate. This conclusion is supported by the simulations concerning the "shrinking sphere", that generate an error in the range of 10^{-3}. Furthermore, the error was decreased when a high grid size, that is $175 \times 175 \times 175$ compared to $100 \times 100 \times 100$ was used. However, the grid size increased the computation time substantially which is a drawback of the implemetation of the algorithm. The error in the "shrinking sphere" experiment, was further reduced when the 27 point stencil was used compared to the 7 point stencil which motivated the usage of the first of the two mentioned stencils in subsequent simulations. The computation time of the implementation was very large, especially when it comes to the 3D simulation, and this is explained by the high grid size that is employed in all simulations.

The computation time is dependent on the grid size since a high grid size results in a large amount of points for which the Laplace operator must be estimated. The large number of points included in the diffusion step increase the length of the diffusion step. Therefore, it would be preferable to use a smaller grid in order to increase the speed of the implementation but still maintain a high accuracy of the diffusion step.

A way of implementing a fast and yet accurate solution to the two phase algorithm can be obtain by using an adaptive grid. In the diffusion step, only the points in the grid close to the interface between the various phases are affected in the sharpening step. Therefore, a smaller grid that *adapts* to the position of the interfaces would substantially decrease computation time and still generate an accurate implementation of the algorithm. An adaptive grid has previously been implemented by Ruuth[17] with succesful results.

6.1.2 Projection Triangle

The implementation of the projection triangle step agrees with the simulations conducted by Ruuth[16]. For a straight solid wall, the interfaces between the solid phase and the other phases, i.e. $\hat{\Gamma}_{13}$ and $\hat{\Gamma}_{23}$ in the projection triangle, should be straight lines from the triple junction to the points $\left(\frac{1}{2},0,\frac{1}{2}\right)$ and $\left(0,\frac{1}{2},\frac{1}{2}\right)$ respectively. Exactly these results are obtained in figure 3.12a and 3.12b on page 53, which implies that the implementation of the projection triangle algorithm was succesful. Also, the two conditions on the projection triangle algorithm in section 2.2.2 are met as the results in section 3.2 show.

In addition to a succesful implementation, the convolution thresholding analysis supports the validity of the results from the projection triangle algorithm. By comparing the numerical implementation of the analytical formulas with the interface generated in the projection triangle algorithm as in figure 4.5 on page 74 the similarity is tangible. Thus, the condition on conservation of the contact angle at a point close to the solid phase after a time $\Delta\tau$, see equation 4.18 and 4.19, validates the usage of the projection triangle algorithm in simulating capillary flows. Provided that both the diffusion and the sharpening step were succesfully implemented, the numerical results can be discussed.

6.2 Numerical Results

The results in figure 5.1 to 5.1 indicates that a small contact angle corresponds to a fast capillary rise. Since the contact angle between the solid phase and liquid 1 is higher than the contact angle between the solid phase and liquid 2, the capillary rise is expected to be higher for liquid 1 than for liquid 2[13]. Consequently, the results in figure 5.1 to 5.3 confirm the theory, since the capillary rise of liquid 1 is higher than liquid 2, and thus the results are reasonable.

Secondly, the simulations of capillary flow through a narrow channel filled with obstacles, see figure 5.4 to 5.8, show that a threshold value for the contact angle between the liquid and the solid phase exists. A contact angle above this threshold value, as in the case of liquid 1, will cause the liquid to pass through the channel unhindered.

However, a contact angle below this threshold value will cause the liquid to get stuck in the maize, as in the case of liquid 2. Note that already after 150 times steps, see figure 5.7b and 5.8b, the right hand side of liquid 2 remains stationary.

Furthermore, the results in section 5.2 indicate that the algorithm can be used in order to simulate capillary flow in complex domains in both two and three dimensions. When it comes to the 3D simulation, see figure 5.17 to 5.21, a conclusion concerning the motion of the water surface in symetric versus unsymetric pores can be drawn. In the symmetric pore to the right in these figures, the water surface rises with a constant speed. However, in the unsymmetric pore to the left, the water surface rises quicker at the narrow part at the position $y = 1$ than in the wider part where $y = 0$. This simulation confirms the theory for capillary flow that states that a water surface will rise quicker in a narrow channel compared to a wider channel. The capillary flow through the labyrinth in section 5.2.1, shows the same behaviour as the capillary flow is faster at the more narrow parts of the labyrinth compared to the wider parts which agrees with the theory concerning capillary flow.

It should be noted that in the simulations in section 5.2 and 5.1.2, the sharpening step was conducted by using the projection triangle in combination with ”resetting” the solid phase so that it remains stationary. The reason why this approach was chosen was due to the fact that the circular obstacles in the narrow channel simulated in section 5.1.2 were consumed, in fact they decreased with a speed proportional to their mean curvature as in the case of the shrinking sphere. However, in the future it would be of interest to construct a projection triangle in the sharpening step, such that the solid phase remains stationary, and subsequently the future work for this project is discussed.

6.3 Future work

In the future, three prospects are especially interesting. First, the solution algorithm can be implemented more efficiently, secondly a different expansion in the convolution thresholding calculation can be performed in order to construct projection triangles for more general domains and thirdly the algorithm can be applied when simulating capillary flow in industrial applications. The efficiency of the algorithm can be increased by using an adaptive grid in solving the heat equation. An adaptive grid that covers a sufficiently small area that contains the interfaces between the phases could increase the accuracy of the implementation as well as decreasing the computation time.

In addition to the implementation of the diffusion step, the convolution threshold calculation could potentially include curvature terms in both two and three dimensions in order to determine how the curvature of the interfaces affects the appearence of the projection triangle. In addition, it is of interest to derive a convolution threshold scheme that connects the tripple junction to the two phase situation, that is mean curvature flow. Furthermore, in the case of capillary flow that rises oppose to a gravitational force, it would be of interest to include the gravitational contribution in the diffusion step so that the liquid air interface evolves with a speed proportional to its mean curvature plus a contribution from gravity.

Finally, future work could include modelling capillary flows in industrial application in order to determine the validity of the implemented model. On the one hand, it would be interested to compare the numerical results with empirical data for certain industrial applications of capillary flows. If this implementation would generate accurate results, the algorithm can be applied to situations where it is perhaps difficult to perform empirical test in order to predict the outcome of unknown experiments.

Bibliography

[1] Robert A. Adams and Christopher Essex. *Calculus : a complete course*. Pearson Addison Wesley, Toronto, 7. ed. edition, 2009.

[2] P. W. Atkins and Loretta Jones. *Chemical principles : the quest for insight*. W.H. Freeman, New York, 4. ed. edition, 2008.

[3] Bronsard. On three-phase boundary motion and the singular limit of a vector-valued ginzburg-landau equation. *Archive for Rational Mechanics and Analysis*, 124(4):355–379, 1993.

[4] Lawrence C. Evans. Convergence of an algorithm for mean curvature motion. *Indiana Univ. Math. J.*, 42(2):533–557, 1993.

[5] Alexey Heintz and Richards Grzhibovskis. A convolution thresholding scheme for the willmore flow. *Interfaces and Free Boundaries*, 10:139–153, 2008.

[6] Hitoshi Ishii and Moto-Hiko Sato. Nonlinear oblique derivative problems for singular degenerate parabolic equations on a general domain. *Nonlinear Analysis: Theory, Methods & Applications*, 57(7–8):1077 – 1098, 2004.

[7] Stig Larsson and Vidar Thomée. *Partial differential equations with numerical methods*. Springer, New York, NY, 1st ed. 2003 edition, 2005.

[8] Rafael. López. *Constant Mean Curvature Surfaces with Boundary [Elektronisk resurs]*. 2013.

[9] Robert E. Lynch. Fundamental solutions of nine-point discrete laplacians. *Applied Numerical Mathematics*, 10(3–4):325 – 334, 1992.

[10] Barry Merriman, James K. Bence, and Stanley J. Osher. Motion of multiple junctions: A level set approach. *Journal of Computational Physics*, 112(2):334 – 363, 1994.

[11] Randall O'Reilly. A family of large-stencil discrete laplacian approximations in three dimensions. University of Colorado Boulder, 2006.

[12] N. C. Owen, J. Rubinstein, and P. Sternberg. Minimizers and gradient flows for singularly perturbed bi-stable potentials with a dirichlet condition. *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences*, 429(1877):pp. 505–532, 1990.

[13] Richard M. Pashley and Marilyn E. Karaman. *Applied colloid and surface chemistry.* Wiley, Chichester, 2004.

[14] Jacob Rubinstein, Peter Sternberg, and Joseph B. Keller. Fast reaction, slow diffusion, and curve shortening. *SIAM Journal on Applied Mathematics*, 49(1):116–133, 1989.

[15] Jacob Rubinstein, Peter Sternberg, and Joseph B. Keller. Front interaction and nonhomogeneous equilibria for tristable reaction-diffusion equations. *SIAM J. Appl. Math.*, 53(6):1669 – 1685, 1993.

[16] Steven J. Ruuth. A diffusion-generated approach to multiphase motion. *Journal of Computational Physics*, 145(1):166 – 192, 1998.

[17] Steven J. Ruuth. Efficient algorithms for diffusion-generated motion by mean curvature. *Journal of Computational Physics*, 144(2):603 – 625, 1998.

[18] I.M. Sigal. Lectures on mean curvature flow and stability (mat 1063 hs), November 23 2014.

[19] Dirk Jan Struik. *Lectures on classical differential geometry.* Dover Publications, New York, 2. ed. edition, 1988.

Appendices

The appendix consists of two parts. Appendix A contains the code written in C++ and appendix B contains the computer scripts written in Matlab. The scripts in C++ generates the numerical solution to the evolution of various multiphase systems, while the Matlab scripts are used in order to plot the matrices resulting from the C++ programs. The simulations have been conducted on a PC laptop.

A

C++ code

Appendix A consists of five scripts called ”Shrinking Sphere”,”Narrow Channels”, ”Narrow channels filled with obstacles”, ”Labyrinth simulation” and ”3D simulation”. The script in section ”Shrinking Sphere” generated the results in section 3.1 on page 41. The script in section ”Narrow Channels” generates the results in section 5.1.1 on page 77. The script in section ”Narrow Channels filled with obstacles” generates the results in section 5.1.2 on page 80. The script in section ”Labyritnh simulation” generates the results in section 5.2.1 on page 82. Finally, the script in section ”3D simulation” generates the results in section 5.2.2 on page 87.

The general idea behind the scripts, is that the scripts generates various output matrices, one for each phase, where each position in these matrices contains either the value 0, corresponding to a position *outside* the phase, or 1, corresponding to a position *within* the phase. Furthermore, each position in these matrices are furthermore connected to a position in space which is generated by the same position in two other grid matrices, called ”XPlane” and ”YPlane”, and these matrices give the x- and the y-coordinate for the position in question. In $\mathbb{R}^3$ a vector containing the z-coordinate is also generated.

A.1 ”Shrinking Sphere”

```
// User defines
// Mathematics stuff
//#define _USE_MATH_DEFINES
//#define WIN32_LEAN_AND_MEAN
// Include headers
#include <SDKDDKVer.h>
#include <windows.h>
#include <stdlib.h>
#include <malloc.h>
#include <memory.h>
#include <tchar.h>
#include <iostream>
#include <string>
#include <cmath>
//#include "engine.h"
#include <fstream>
#include<direct.h>
#include <sstream>
```

```
#include <stdio.h>
#include "math.h"
#include<vector>
#include<time.h>

// Constants
// number Of pixels
static const int  nuOfEle = 50;
static const int  timeTotal = 200;
// radius of sphere
static const int radius = 1;
// step size
//double step, tTot1step1, tTot1step2, tTot2step1, tTot2step2, tStep1, tStep2, tTot1step3, tTot2step3, tStep3;
double step, tTot, tStep;
// x and y indices
double x, y;
// row and column indices
int rI, cI, zI;


// Standard stuff
using std::string;
using namespace std;
using std::vector;

// Declaration of functions
// Save file in folder
void saveVecDirectory(vector<double>, string);
void saveFileDirectory(vector<vector<vector<vector<double>>>>, int, string);
void cube(vector<vector<vector<double>>>&, vector<vector<vector<double>>>&, int, int, int);
void laplace(vector<vector<vector<double>>>&, vector<vector<vector<double>>>&, double);
void eulerForward(vector<vector<vector<vector<double>>>>&, int, double, double, double);
void sharpening(vector<vector<vector<vector<double>>>>&, int);
double radiusSphere(vector<vector<vector<vector<double>>>>&, int, vector<vector<double>>&, vector<vector<double>>&,
vector<double>&);

// MAIN PROGRAM
int main()
{
string run;
std::ostringstream eleStr;
eleStr << nuOfEle;
std::string eleString = eleStr.str();
std::ostringstream timeStr;
timeStr << timeTotal;
std::string timeString = timeStr.str();
run = ".//ShrinkingSphere20150519" + eleString + "Time" + timeString + "//";
CreateDirectory(run.c_str(), NULL);

//Save number of element
std::ofstream outputnuOfEle(run + "nuOfEle.txt");
outputnuOfEle << nuOfEle;
outputnuOfEle.close();
//Save radius
std::ofstream outputRadius(run + "radius.txt");
outputRadius << radius;
outputRadius.close();
// Define mesh
vector<vector<double>>Xplane;
vector<vector<double>>Yplane;
Xplane.resize(nuOfEle + 1);
Yplane.resize(nuOfEle + 1);
for (int rI = 0; rI < (nuOfEle + 1); rI++)
{
Xplane[rI].resize(nuOfEle + 1);
Yplane[rI].resize(nuOfEle + 1);
}
// Zvec
vector<double>Zvec;
Zvec.resize(nuOfEle + 1);
//Step length
step = ((2 * radius) / ((double)nuOfEle));
// Assignvalues to mesh
//Save matrices in files
std::ofstream outputX(run + "Xplane.txt");
std::ofstream outputY(run + "Yplane.txt");
for (rI = 0, y = radius; rI < (nuOfEle + 1); rI++, y += (-step))
{
for (cI = 0, x = -radius; cI < (nuOfEle + 1); cI++, x += (step))
{
Xplane[rI][cI] = x;
Yplane[rI][cI] = y;
outputX << Xplane[rI][cI] << " ";
```

```
outputY << Yplane[rI][cI] << " ";
}
outputX << "\n";
outputY << "\n";
}
// Close files
outputX.close();
outputY.close();
for (int zI = 0; zI < (nuOfEle + 1); zI++)
{
Zvec[zI] = Xplane[1][zI];
}
std::string fileName = run + "z.txt";
saveVecDirectory(Zvec, fileName);
// Define timeSteps
tStep = ((step*step) / (8));
tTot = 50 * tStep;
/* Save all the variables above*/
// save tStep
std::ofstream outputtStep1(run + "tStep.txt");
outputtStep1 << tStep;
outputtStep1.close();
// Save tTot
std::ofstream outputtTot2step3(run + "tTot.txt");
outputtTot2step3 << tTot;
outputtTot2step3.close();
// REAL PROGRAM STARTS
vector<double>r;
r.resize(timeTotal + 1);
//3D cube
// Zero time steps
vector<vector<vector<vector<double>>>>sphere((nuOfEle + 1), vector<vector<vector<double>>>((nuOfEle + 1),
vector<vector<double>>((nuOfEle + 1), vector<double>((2)))));


// Assign values sphere
for (int zI = 0; zI < (nuOfEle + 1);zI++)
{
for (int rI = 0; rI < (nuOfEle + 1); rI++)
{
for (int cI = 0; cI < (nuOfEle + 1); cI++)
{
if (((Xplane[rI][cI] * Xplane[rI][cI]) + (Yplane[rI][cI] * Yplane[rI][cI]) + (Zvec[zI] * Zvec[zI]))<=(radius*radius))
{
sphere[rI][cI][zI][0] = 1;
}
else
{
sphere[rI][cI][zI][0] = 0;
}
}
}
}
r[0] = 1;


string folder = run + ".//timefolder";
CreateDirectory(folder.c_str(), NULL);

std::ostringstream ini;
ini << 0;
std::string theNumberString = ini.str();
string tempStr = folder + "//t" + theNumberString;
saveFileDirectory(sphere, 0, tempStr);



for (int tI = 1; tI < (timeTotal + 1); tI++)
{
// EULER FORWARD!!!
// Fill sphere using eulerForward function
eulerForward(sphere, 1, tTot, step, tStep);
// Sharpen newly filled vector
r[tI] = radiusSphere(sphere, 1, Xplane, Yplane, Zvec);
//r[tI] = radiusCylinder(sphere, 1, Xplane, Yplane, Zvec);
sharpening(sphere, 1);
cout << "timeStep\t" << tI << "\t of " << timeTotal << "\n";
// Save stuff
std::ostringstream ostr;
ostr << tI;
std::string theNumberString = ostr.str();
string tempStr = folder + "//t" + theNumberString;
```

```
saveFileDirectory(sphere, 1, tempStr);
// Change values of first sphere
for (int zI = 0; zI < (nuOfEle + 1); zI++)
{
for (int rI = 0; rI < (nuOfEle + 1); rI++)
{
for (int cI = 0; cI < (nuOfEle + 1); cI++)
{
sphere[rI][cI][zI][0] = sphere[rI][zI][cI][1];
}
}
}
}
string hajjPaDajj = run + "r.txt";
saveVecDirectory(r, hajjPaDajj);

// Program worked!
cout << "Program worked!\n\n";
// end of main
return 0;
}// END MAIN
```

```
// NEW FUNCTIONS: YIIIIIIIIIIIHAAAAAAAAAAAAAAA!!!
// FUNCTION SAVE VECTOR DIRECTORY
void saveVecDirectory(vector<double> VEC, string directory)
{
std::string filefolder = directory;
std::ofstream output(filefolder);
int length = VEC.size();
int zI;
//std::ofstream output(directory.c_str());
for (zI = 0; zI < length; zI++)
{
/*std::ostringstream ostr;
ostr << zI;
std::string theNumberString = ostr.str();
std::string end = ".txt";
std::string fileFolder = directory + "//" + theNumberString + end;*/
output << VEC[zI] << "\n";
}
output.close();
}
```

```
// FUNCTION SAVE FILE DIRECTORY
void saveFileDirectory(vector<vector<vector<vector<double>>>> ARRAY, int tI, string directory)
{
//std::string folder = ".//fib0";
CreateDirectory(directory.c_str(), NULL);
for (int zI = 0; zI < ((double)nuOfEle + 1); zI++)
{
std::ostringstream ostr;
ostr << zI;
std::string theNumberString = ostr.str();
std::string end = ".txt";
std::string fileFolder = directory + "//" + theNumberString + end;
std::ofstream output(fileFolder);

for (int rI = 0; rI < (nuOfEle + 1); rI++)
{
for (int cI = 0; cI < (nuOfEle + 1); cI++)
{
output << ARRAY[rI][cI][zI][tI] << " ";
}
output << "\n";
}
output.close();
}
}
```

```
// Void cube
void cube(vector<vector<vector<double>>>& cube, vector<vector<vector<double>>>& lpMat, int zI, int rI, int cI)
{
```

```
/*
// cube is a 3X3X3-tensor
//Fill it with an arbitrary value such as five

for (int z = 0; z < 3; z++)
{
for (int x = 0; x < 3; x++)
{
for (int y = 0; y < 3; y++)
{
cube[y][x][z] = 5;
}
}
}

// Check all the extreme cases
//that is if zI = 0, zI = nuOfEle,
//rI = 0, rI = nuOfEle,cI = 0 and cI = nuOfEle,


// Roof
if (zI == nuOfEle)
{
for (int x = 0; x < 3; x++)
{
for (int y = 0; y < 3; y++)
{
cube[y][x][2] = lpMat[((rI - 1) + y)][((cI - 1) + x)][0];
}
}
}
// Floor
if (zI == 0)
{
for (int x = 0; x < 3; x++)
{
for (int y = 0; y < 3; y++)
{
cube[y][x][0] = lpMat[((rI - 1) + y)][((cI - 1) + x)][nuOfEle];
}
}
}
// Right Wall
if (cI == nuOfEle)
{
for (int z = 0; z < 3; z++)
{
for (int y = 0; y < 3; y++)
{
cube[y][2][z] = 0;
}
}
}
// Left Wall
if (cI == 0)
{
for (int z = 0; z < 3; z++)
{
for (int y = 0; y < 3; y++)
{
cube[y][0][z] = 0;
}
}
}
// Front Wall
if (rI == nuOfEle)
{
for (int z = 0; z < 3; z++)
{
for (int x = 0; x < 3; x++)
{
cube[2][x][z] = 0;
}
}
}

// Back Wall
if (rI == 0)
{
for (int z = 0; z < 3; z++)
{
for (int x = 0; x < 3; x++)
{
```

```
cube[0][x][z] = 0;
}
}
}

// Fill with values
for (int z = 0; z < 3; z++)
{
for (int x = 0; x < 3; x++)
{
for (int y = 0; y < 3; y++)
{
if (cube[y][x][z] == 5)
{
cube[y][x][z] = lpMat[((rI - 1) + y)][((cI - 1) + x)][((zI - 1) + z)];
}
}
}
}*/
int yTemp, xTemp, zTemp;
for (int x = 0; x < 3; x++)
{
for (int y = 0; y < 3; y++)
{
for (int z = 0; z < 3; z++)
{
xTemp = cI - 1 + x;
yTemp = rI - 1 + y;
zTemp = zI - 1 + z;
if (yTemp == (nuOfEle + 1))
{

yTemp = nuOfEle;
}
if (yTemp == -1)
{

yTemp = 0;
}
if (xTemp == (nuOfEle + 1))
{

xTemp = nuOfEle;
}
if (xTemp == -1)
{

xTemp = 0;
}
if (zTemp == (nuOfEle + 1))
{

zTemp = 0;
}
if (zTemp == -1)
{

zTemp = nuOfEle;
}
cube[y][x][z] = lpMat[yTemp][xTemp][zTemp];
}
}
}

}// end cube function

// Void function Laplace
void laplace(vector<vector<vector<double>>>& lpMat, vector<vector<vector<double>>>& inputMat, double step)
{
// Allocate memory for the sums
double cornerSum, faceSum, edgeSum, nom;
// Create a cube 3x3x3
//vector<vector<vector<double>>>cubeTemp((3), vector<vector<double>>((3), vector<double>(3)));

// Create a cube 3x3x3
vector<vector<vector<double>>>cubeTemp;
cubeTemp.resize(3);
for (int index = 0; index < 3; index++)
{
cubeTemp[index].resize(3);
for (int hej = 0; hej < 3; hej++)
{
cubeTemp[index][hej].resize(3);
```

```
}
}

//LOOPS
for (int zI = 0; zI < (nuOfEle + 1); zI++)
{
for (int rI = 0; rI < (nuOfEle + 1); rI++)
{
for (int cI = 0; cI < (nuOfEle + 1); cI++)
{
cube(cubeTemp, inputMat, zI, rI, cI);
// Calculate stuff
// FACES
faceSum = cubeTemp[2][1][1] + cubeTemp[0][1][1] + //Forward +Backward
cubeTemp[1][1][2] + cubeTemp[1][1][0] + //Up + Down
cubeTemp[1][0][1] + cubeTemp[1][2][1]; //Left + Right
// EDGES
edgeSum = cubeTemp[2][1][2] + cubeTemp[2][1][0] + //ForwardUp +ForwardDown
cubeTemp[2][0][1] + cubeTemp[2][2][1] + //ForwardLeft +ForwardRight
cubeTemp[0][1][2] + cubeTemp[0][1][0] + //BackwardUp + BackwardDown
cubeTemp[0][0][1] + cubeTemp[0][2][1] + //BackwardLeft + BackwardRight
cubeTemp[1][0][2] + cubeTemp[1][0][0] + //LeftUp + LeftDown
cubeTemp[1][2][2] + cubeTemp[1][2][0];//RightUp + RightDown
cornerSum = cubeTemp[2][2][2] + cubeTemp[2][2][0] + //ForwardRightUp +ForwardRightDown
cubeTemp[2][0][2] + cubeTemp[2][0][0] + //ForwardLeftUp +ForwardLeftDown
cubeTemp[0][2][2] + cubeTemp[0][2][0] + //BackwardRightUp +BackwardRightDown
cubeTemp[0][0][2] + cubeTemp[0][0][0]; //BackwardLeftUp +BackwardLeftDown
//nom = ((3 * faceSum) / (13)) + ((3 * edgeSum) / (26)) + ((cornerSum) / (13)) - ((44 * inputMat[rI][cI][zI]) / (13));// Stencil 27
nom = ((faceSum) / (3)) + ((edgeSum) / (6)) - (4 * inputMat[rI][cI][zI]);//Stencil 19
//nom = faceSum - 6.*inputMat[rI][cI][zI];// Stencil 7
//cout << "\t\Face:\t" << faceSum << "\n\tEdge:\t" << edgeSum << "\n\tCorner:\t" << cornerSum;
//cout << "\n\tNom:" << nom << "\n-----------------\n\n";

lpMat[rI][cI][zI] = ((nom) / (step*step));
}
}
}

/*double left, right, up, down, forward, backward;
for (int zI = 0; zI < (nuOfEle + 1); zI++)
{
for (int rI = 0; rI < (nuOfEle + 1); rI++)
{
for (int cI = 0; cI < (nuOfEle + 1); cI++)
{

//LEFT RIGHT DIMENSION
if (cI == 0)
{
left = 0;
right = inputMat[rI][cI + 1][zI];
}
if (cI == nuOfEle)
{
left = inputMat[rI][cI - 1][zI];
right = 0;
}
if (cI>0 && cI<nuOfEle)
{
left = inputMat[rI][cI - 1][zI];
right = inputMat[rI][cI + 1][zI];
}
//FORWARD BACKWARD DIMENSION
if (rI == 0)
{
forward = inputMat[rI + 1][cI][zI];
backward = 0;
}
if (rI == nuOfEle)
{
forward = 0;
backward = inputMat[rI - 1][cI][zI];
}
if (rI>0 && rI<nuOfEle)
{
forward = inputMat[rI + 1][cI][zI];
backward = inputMat[rI - 1][cI][zI];
}
if (zI == 0)
{
up = inputMat[rI][cI][zI + 1];
down = 0; // Sphere option
//down = inputMat[rI][cI][nuOfEle];//Cylinder, periodic option
```

```
}
//UP DOWN DIMENSION
if (zI == nuOfEle)
{
down = inputMat[rI][cI][zI-1];
up = 0;//Sphere option
//up = inputMat[rI][cI][0];//Cylinder, periodic option
}
if (zI>0 && zI<nuOfEle)
{
up = inputMat[rI][cI][zI + 1];
down = inputMat[rI][cI][zI - 1];
}
double nom = (up + down + left + right + forward + backward - (6 * inputMat[rI][cI][zI]));
lpMat[rI][cI][zI] = ((nom) / (step*step));
}
}
}*/

}


// Euler Forward!
void eulerForward(vector<vector<vector<vector<double>>>>& afterMat, int timeIndex, double tTot, double step, double tStep)
{

//Create temporary vector beforeMat
vector<vector<vector<double>>> beforeMat;
beforeMat.resize(nuOfEle + 1);
for (int rI = 0; rI < (nuOfEle + 1); rI++) {
beforeMat[rI].resize(nuOfEle + 1);
for (int cI = 0; cI < (nuOfEle + 1); cI++)
{
beforeMat[rI][cI].resize(nuOfEle + 1);
}

}
// Initialiaze beforeMat to initialMat
for (int zI = 0; zI < (nuOfEle + 1); zI++)
{
for (int rI = 0; rI < (nuOfEle + 1); rI++)
{
for (int cI = 0; cI < (nuOfEle + 1); cI++)
{
beforeMat[rI][cI][zI] = afterMat[rI][cI][zI][timeIndex - 1];
}
}
}

// Euler loops: one for time, one for each
//spatial dimension
for (double tI = 0; tI < (tTot + tStep); tI += tStep)
{
vector<vector<vector<double>>> lpTemp;
lpTemp.resize(nuOfEle + 1);
for (int rI = 0; rI < (nuOfEle + 1); rI++) {
lpTemp[rI].resize(nuOfEle + 1);
for (int cI = 0; cI < (nuOfEle + 1); cI++)
{
lpTemp[rI][cI].resize(nuOfEle + 1);
}

}
laplace(lpTemp, beforeMat, step);
for (int zI = 0; zI < (nuOfEle + 1); zI++)
{
for (int rI = 0; rI < (nuOfEle + 1); rI++)
{
for (int cI = 0; cI < (nuOfEle + 1); cI++)
{
afterMat[rI][cI][zI][timeIndex] = beforeMat[rI][cI][zI] + tStep*lpTemp[rI][cI][zI];
beforeMat[rI][cI][zI] = afterMat[rI][cI][zI][timeIndex];
}
}
}
lpTemp.erase(lpTemp.begin(), lpTemp.end());
//lpTemp.shrink_to_fit();
lpTemp.clear();
}

beforeMat.erase(beforeMat.begin(), beforeMat.end());
//beforeMat.shrink_to_fit();
beforeMat.clear();
```

```
}// end Euler


// SHARPENING FUNCTION
void sharpening(vector<vector<vector<vector<double>>>>& input, int tI)
{
for (int zI = 0; zI < (nuOfEle + 1); zI++)
{
for (int rI = 0; rI < (nuOfEle + 1); rI++)
{
for (int cI = 0; cI < (nuOfEle + 1); cI++)
{
if (input[rI][cI][zI][tI] < 0.5)
{
input[rI][cI][zI][tI] = 0;
}
else
{
input[rI][cI][zI][tI] = 1;
}
}
}
}
}// end sharpening


double radiusSphere(vector<vector<vector<vector<double>>>>& sphere, int tI, vector<vector<double>>& Xplane, vector<vector<double>>& Yplane,
vector<double>& zVec)
{
double radiusMean, radSum, temp, k, Xtemp, Ytemp, Ztemp;
int den;
radSum = 0;
den = 0;
int zI, rI, cI;
for (zI = 0; zI < (nuOfEle + 1); zI++)
{
for (rI = 0; rI < (nuOfEle + 1); rI++)
{
for (cI = 0; cI < (nuOfEle + 1); cI++)
{

if (sphere[rI][cI][zI][tI] == 0.5)
{
temp = sqrt((zVec[zI] * zVec[zI]) + (Xplane[rI][cI] * Xplane[rI][cI]) + (Yplane[rI][cI] * Yplane[rI][cI]));
radSum += temp;
den++;
continue;
}
if (zI > 0 && cI > 0 && rI > 0 &&
zI < nuOfEle && cI < nuOfEle && rI < nuOfEle)
{
// RIGHT SIDE
if (sphere[rI][cI + 1][zI][tI] > 0.5 && sphere[rI][cI][zI][tI] < 0.5)
{
k = ((sphere[rI][cI + 1][zI][tI] - sphere[rI][cI][zI][tI]) / (Xplane[rI][cI + 1] - Xplane[rI][cI]));
Xtemp = ((0.5 - sphere[rI][cI][zI][tI]) / (k)) + Xplane[rI][cI];
temp = sqrt((zVec[zI] * zVec[zI]) + (Xtemp * Xtemp) + (Yplane[rI][cI] * Yplane[rI][cI]));
radSum += temp;
den++;
}
// LEFT SIDE
if (sphere[rI][cI - 1][zI][tI] > 0.5 && sphere[rI][cI][zI][tI] < 0.5)
{
k = ((sphere[rI][cI - 1][zI][tI] - sphere[rI][cI][zI][tI]) / (Xplane[rI][cI - 1] - Xplane[rI][cI]));
Xtemp = ((0.5 - sphere[rI][cI][zI][tI]) / (k)) + Xplane[rI][cI];
temp = sqrt((zVec[zI] * zVec[zI]) + (Xtemp * Xtemp) + (Yplane[rI][cI] * Yplane[rI][cI]));
radSum += temp;
den++;
}
// FORWARD
if (sphere[rI + 1][cI][zI][tI] > 0.5 && sphere[rI][cI][zI][tI] < 0.5)
{
k = ((sphere[rI + 1][cI][zI][tI] - sphere[rI][cI][zI][tI]) / (Yplane[rI + 1][cI] - Yplane[rI][cI]));
Ytemp = ((0.5 - sphere[rI][cI][zI][tI]) / (k)) + Yplane[rI][cI];
temp = sqrt((zVec[zI] * zVec[zI]) + (Xplane[rI][cI] * Xplane[rI][cI]) + (Ytemp * Ytemp));
radSum += temp;
den++;
}
```

```
// BACKWARD
if (sphere[rI - 1][cI][zI][tI] > 0.5 && sphere[rI][cI][zI][tI] < 0.5)
{
k = ((sphere[rI - 1][cI][zI][tI] - sphere[rI][cI][zI][tI]) / (Yplane[rI - 1][cI] - Yplane[rI][cI]));
Ytemp = ((0.5 - sphere[rI][cI][zI][tI]) / (k)) + Yplane[rI][cI];
temp = sqrt((zVec[zI] * zVec[zI]) + (Xplane[rI][cI] * Xplane[rI][cI]) + (Ytemp * Ytemp));
radSum += temp;
den++;

}

}

}
}
}
radiusMean = ((radSum) / ((double)den));
return radiusMean;
}// End radius sphere
```

A.2 Narrow Channels

```
#define _USE_MATH_DEFINES

// User defines
// Mathematics stuff
//#define _USE_MATH_DEFINES
//#define WIN32_LEAN_AND_MEAN
// Include headers
#include <SDKDDKVer.h>
#include <windows.h>
#include <stdlib.h>
#include <malloc.h>
#include <memory.h>
#include <tchar.h>
#include <iostream>
#include <string>
#include <cmath>
//#include "engine.h"
#include <fstream>
#include<direct.h>
#include <sstream>
#include <stdio.h>
#include "math.h"
#include<vector>
#include<time.h>

// Constants
// number Of pixels
static const int  xDim = 600;
static const int  yDim = 600;
static const int  xDimZoom = 500;
static const int  yDimZoom = 500;
static const int  timeTotal = 160;
// radius of sphere
static const int heightSquare = 6;
static const int lengthSquare = 6;
static const int heightZoom = 1;
static const int lengthZoom = 1;
// step size
double step, tTot, tStep, stepZoom, tStepZoom, tTotZoom;
// x and y indices
double x, y;
// row and column indices
int rI, cI, zI;


// Standard stuff
using std::string;
using namespace std;
using std::vector;


// Declaration of functions
// Save file in folder
void saveFileDirectory(vector<vector<vector<double>>>, int, string, int, int, int);
void cube(vector<vector<double>>&, vector<vector<double>>&, int, int, int, int, int, int);
void laplace(vector<vector<double>>&, vector<vector<double>>&, double, int, int, int, int);
void eulerForward(vector<vector<vector<double>>>&, int, double, double, double, int, int, int, int);
void sharpening(vector<vector<vector<double>>>&, vector<vector<vector<double>>>&, vector<vector<vector<double>>>&, int,
```

```
vector<vector<double>>&, int);

// MAIN PROGRAM
int main()
{
// Working directory
string run;
std::ostringstream eleStrFib;
eleStrFib << xDim;
std::string eleStringFib = eleStrFib.str();
std::ostringstream eleStrFib2;
eleStrFib2 << yDim;
std::string eleStringFib2 = eleStrFib2.str();
std::ostringstream timeStrFib;
timeStrFib << timeTotal;
std::string timeStringFib = timeStrFib.str();
run = ".//20150518FirstDomain" + eleStringFib + "Ydim" + eleStringFib2 + "Time" + timeStringFib + "//";
//run = ".//Repetitive//";
//run = ".//newFolder//";
CreateDirectory(run.c_str(), NULL);
string fiberStr = run + "fiber";
CreateDirectory(fiberStr.c_str(), NULL);
string waterStr = run + "water";
CreateDirectory(waterStr.c_str(), NULL);
string airStr = run + "air";
CreateDirectory(airStr.c_str(), NULL);
// Create timefolder
string timeFolderFib = fiberStr + "//timeFolder//";
CreateDirectory(timeFolderFib.c_str(), NULL);
string timeFolderWat = waterStr + "//timeFolder//";
CreateDirectory(timeFolderWat.c_str(), NULL);
string timeFolderAir = airStr + "//timeFolder//";
CreateDirectory(timeFolderAir.c_str(), NULL);


cout << "First print\n";

//Projection triangle
string diffFiber = run + "//DiffusedFiber//";
CreateDirectory(diffFiber.c_str(), NULL);
string diffWater = run + "//DiffusedWater//";
CreateDirectory(diffWater.c_str(), NULL);
string diffAir = run + "//DiffusedAir//";
CreateDirectory(diffAir.c_str(), NULL);


//Save number of element
std::ofstream outputnuOfEle(run + "xDim.txt");
outputnuOfEle << xDim;
outputnuOfEle.close();
//Save number of element
std::ofstream outputnuOfEle2(run + "yDim.txt");
outputnuOfEle2 << yDim;
outputnuOfEle2.close();
//Save radius
std::ofstream outputRadius(run + "length.txt");
outputRadius << lengthSquare;
outputRadius.close();
//Save radius
std::ofstream outputRadius2(run + "height.txt");
outputRadius2 << heightSquare;
outputRadius2.close();
// Define mesh
vector<vector<double>>Xplane;
vector<vector<double>>Yplane;
vector<vector<double>>XplaneZoom;
vector<vector<double>>YplaneZoom;
Xplane.resize(yDim + 1);
Yplane.resize(yDim + 1);
for (int rI = 0; rI < (yDim + 1); rI++)
{
Xplane[rI].resize(xDim + 1);
Yplane[rI].resize(xDim + 1);

}

XplaneZoom.resize(yDimZoom + 1);
YplaneZoom.resize(yDimZoom + 1);
for (int rI = 0; rI < (yDimZoom + 1); rI++)
{
XplaneZoom[rI].resize(xDimZoom + 1);
YplaneZoom[rI].resize(xDimZoom + 1);
}
```

```
//Step length
step = ((lengthSquare) / ((double)xDim));
std::ofstream outputnuOfEle3(run + "step.txt");
outputnuOfEle3 << step;
outputnuOfEle3.close();
// Assignvalues to mesh
//Save matrices in files
std::ofstream outputX(run + "Xplane.txt");
std::ofstream outputY(run + "Yplane.txt");
for (rI = 0, y = lengthSquare; rI < (yDim + 1); rI++, y -= (step))
{
for (cI = 0, x = 0; cI < (xDim + 1); cI++, x += (step))
{
Xplane[rI][cI] = x;
Yplane[rI][cI] = y;
outputX << Xplane[rI][cI] << " ";
outputY << Yplane[rI][cI] << " ";
}
outputX << "\n";
outputY << "\n";
}
// Close files
outputX.close();
outputY.close();


//Zoomed in stuff

//Step length
stepZoom = ((lengthZoom) / ((double)xDimZoom));
std::ofstream outputstepZoom(run + "stepZoom.txt");
outputstepZoom << stepZoom;
outputstepZoom.close();
// Assignvalues to mesh
//Save matrices in files
std::ofstream outputXZoom(run + "XplaneZoom.txt");
std::ofstream outputYZoom(run + "YplaneZoom.txt");
for (rI = 0, y = 1; rI < (yDimZoom + 1); rI++, y -= (stepZoom))
{
for (cI = 0, x = 0; cI < (xDimZoom + 1); cI++, x += (stepZoom))
{
XplaneZoom[rI][cI] = x;
YplaneZoom[rI][cI] = y;
outputXZoom << XplaneZoom[rI][cI] << " ";
outputYZoom << YplaneZoom[rI][cI] << " ";
}
outputXZoom << "\n";
outputYZoom << "\n";
}
// Close files
outputXZoom.close();
outputYZoom.close();


cout << "Second print\n";

vector<vector<vector<double>>> fiberZoom;
vector<vector<vector<double>>> waterZoom;
vector<vector<vector<double>>> airZoom;
fiberZoom.resize(yDimZoom + 1);
waterZoom.resize(yDimZoom + 1);
airZoom.resize(yDimZoom + 1);
for (int rI = 0; rI < (yDimZoom + 1); rI++)
{
fiberZoom[rI].resize(xDimZoom + 1);
waterZoom[rI].resize(xDimZoom + 1);
airZoom[rI].resize(xDimZoom + 1);
for (int cI = 0; cI < (xDimZoom + 1); cI++)
{
fiberZoom[rI][cI].resize(2);
waterZoom[rI][cI].resize(2);
airZoom[rI][cI].resize(2);
}
}

cout << "Third print\n";

for (int rI = 0; rI < (yDimZoom + 1); rI++)
{
for (int cI = 0; cI < (xDimZoom + 1); cI++)
{
if (XplaneZoom[rI][cI] <= 0.5)
```

```
{
fiberZoom[rI][cI][0] = 1;
waterZoom[rI][cI][0] = 0;
airZoom[rI][cI][0] = 0;
}
if (XplaneZoom[rI][cI] > 0.5)
{
if (YplaneZoom[rI][cI] <= (1 - XplaneZoom[rI][cI]))
{
fiberZoom[rI][cI][0] = 0;
waterZoom[rI][cI][0] = 1;
airZoom[rI][cI][0] = 0;
}
if (YplaneZoom[rI][cI] > (1 - XplaneZoom[rI][cI]))
{
fiberZoom[rI][cI][0] = 0;
waterZoom[rI][cI][0] = 0;
airZoom[rI][cI][0] = 1;
}
}
}
}

cout << "Fourth print\n";


saveFileDirectory(fiberZoom, 0, diffFiber, 0, xDimZoom, yDimZoom);
saveFileDirectory(waterZoom, 0, diffWater, 0, xDimZoom, yDimZoom);
saveFileDirectory(airZoom, 0, diffAir, 0, xDimZoom, yDimZoom);



// Allocate Memory fibermatrix.
//vector<vector<vector<double>>>fiber((yDim + 1), vector<vector<double>>((xDim + 1), vector<double>(2)));
vector<vector<vector<double>>> fiber;
vector<vector<vector<double>>> water;
vector<vector<vector<double>>> air;
fiber.resize(yDim + 1);
water.resize(yDim + 1);
air.resize(yDim + 1);
for (int rI = 0; rI < (yDim + 1); rI++)
{
fiber[rI].resize(xDim + 1);
water[rI].resize(xDim + 1);
air[rI].resize(xDim + 1);
for (int cI = 0; cI < (xDim + 1); cI++)
{
fiber[rI][cI].resize(2);
water[rI][cI].resize(2);
air[rI][cI].resize(2);
}
}


// Define timeSteps
tStepZoom = ((stepZoom*stepZoom) / (10));
tTotZoom = 3500 * tStepZoom;
// Save all the variables above
// save tStep
std::ofstream outputtStep1(run + "tStepZoom.txt");
outputtStep1 << tStepZoom;
outputtStep1.close();
// Save tTot
std::ofstream outputtTot2step3(run + "tTotZoom.txt");
outputtTot2step3 << tTotZoom;
outputtTot2step3.close();


double minEleX, minEleXSigned, minEleY, minEleYSigned, diffX, diffY, minDiffX, minDiffY;
minDiffX = 25;
minDiffY = 25;
for (int index = 0; index < (xDimZoom + 1); index++)
{
diffX = XplaneZoom[1][index] - 0.5;
//cout << "Difference X:\t" << diffX << "\n";
if (abs(diffX) < minDiffX)
{
minEleX = abs(XplaneZoom[1][index]);
minEleXSigned = XplaneZoom[1][index];
minDiffX = abs(diffX);
//cout << "Minimal element:\t" << minEleXSigned << "\n";
}
}
```

```
for (int index = 0; index < (yDimZoom + 1); index++)
{
diffY = YplaneZoom[index][1] - 0.5;
//cout << "Difference Y:\t" << diffY << "\n";
if (abs(diffY) < minDiffY)
{
minEleY = abs(YplaneZoom[index][1]);
minEleYSigned = YplaneZoom[index][1];
minDiffY = abs(diffY);
//cout << "Minimal element:\t" << minEleYSigned << "\n";
}
}

cout << "Minimal value X:\t" << minEleXSigned << "\n";
cout << "Minimal value Y:\t" << minEleYSigned << "\n";

// Number of edge elements
int fibWat = 0;
int fibAir = 0;
int watAir = 0;


for (int rI = 1; rI < (yDimZoom + 1); rI++)
{
for (int cI = 0; cI < (xDimZoom + 1); cI++)
{
// Interface fiber air
if (XplaneZoom[rI][cI] == minEleXSigned && YplaneZoom[rI][cI] >= minEleYSigned && YplaneZoom[rI][cI] <= 0.7)
{
fibAir++;
}
// Interface fiber water
if (XplaneZoom[rI][cI] == minEleXSigned && YplaneZoom[rI][cI] <= minEleYSigned && YplaneZoom[rI][cI] > 0.2)
{
fibWat++;
}
// Interface water air
if (YplaneZoom[rI][cI] <= (1 - XplaneZoom[rI][cI])
&& YplaneZoom[rI - 1][cI] > (1 - XplaneZoom[rI][cI])
&& XplaneZoom[rI][cI] >= minEleXSigned && XplaneZoom[rI][cI] <= 0.8)
{
watAir++;
}
}
}


std::ofstream outputtStepfibWat(run + "fibWatInt.txt");
outputtStepfibWat << fibWat;
outputtStepfibWat.close();
// Save tTot
std::ofstream outputtTot2stepfibAir(run + "fibAirInt.txt");
outputtTot2stepfibAir << fibAir;
outputtTot2stepfibAir.close();
std::ofstream outputtTot2stepwatAir(run + "watAirInt.txt");
outputtTot2stepwatAir << watAir;
outputtTot2stepwatAir.close();


vector<vector<double>>fibWatVec((fibWat), vector<double>(2));
vector<vector<double>>fibAirVec((fibAir), vector<double>(2));
vector<vector<double>>watAirVec((watAir), vector<double>(2));
int indexFibWat = 0;
int indexFibAir = 0;
int indexWatAir = 0;


for (int rI = 1; rI < (yDimZoom + 1); rI++)
{
for (int cI = 0; cI < (xDimZoom + 1); cI++)
{
// Interface fiber air
if (XplaneZoom[rI][cI] == minEleXSigned && YplaneZoom[rI][cI] >= minEleYSigned && YplaneZoom[rI][cI] <= 0.7)
{
fibAirVec[indexFibAir][0] = rI;
fibAirVec[indexFibAir][1] = cI;
indexFibAir++;
}
// Interface fiber water
if (XplaneZoom[rI][cI] == minEleXSigned && YplaneZoom[rI][cI] <= minEleYSigned &&  YplaneZoom[rI][cI] > 0.2)
{
fibWatVec[indexFibWat][0] = rI;
fibWatVec[indexFibWat][1] = cI;
```

```
indexFibWat++;
}
// Interface water air
if (YplaneZoom[rI][cI] <= (1 - XplaneZoom[rI][cI])
&& YplaneZoom[rI - 1][cI] > (1 - XplaneZoom[rI][cI])
&& XplaneZoom[rI][cI] >= minEleXSigned && XplaneZoom[rI][cI] <= 0.8)
{
watAirVec[indexWatAir][0] = rI;
watAirVec[indexWatAir][1] = cI;
indexWatAir++;
}
}
}


vector<vector<double>>fibWatIntVec((fibWat), vector<double>(3));
vector<vector<double>>fibAirIntVec((fibAir), vector<double>(3));
vector<vector<double>>watAirIntVec((watAir+1), vector<double>(3));

cout << "Now starts short Euler\n";

eulerForward(fiberZoom, 1, tTotZoom, stepZoom, tStepZoom, 1, 1, xDimZoom, yDimZoom);
eulerForward(waterZoom, 1, tTotZoom, stepZoom, tStepZoom, 1, 1, xDimZoom, yDimZoom);
eulerForward(airZoom, 1, tTotZoom, stepZoom, tStepZoom, 1, 1, xDimZoom, yDimZoom);

for (int index = 0; index < (fibWat - 1); index++)
{
fibWatIntVec[index][0] = fiberZoom[fibWatVec[index][0]][fibWatVec[index][1]][1];
fibWatIntVec[index][1] = waterZoom[fibWatVec[index][0]][fibWatVec[index][1]][1];
fibWatIntVec[index][2] = airZoom[fibWatVec[index][0]][fibWatVec[index][1]][1];
}

for (int index = 0; index < (fibAir - 1); index++)
{
fibAirIntVec[index][0] = fiberZoom[fibAirVec[index][0]][fibAirVec[index][1]][1];
fibAirIntVec[index][1] = waterZoom[fibAirVec[index][0]][fibAirVec[index][1]][1];
fibAirIntVec[index][2] = airZoom[fibAirVec[index][0]][fibAirVec[index][1]][1];
}

for (int index = 0; index < (watAir - 1); index++)
{
watAirIntVec[index][0] = ((fiberZoom[watAirVec[index][0]][watAirVec[index][1]][1] + fiberZoom[watAirVec[index][0]+1][watAirVec[index][1]][1]) / (2));
watAirIntVec[index][1] = ((waterZoom[watAirVec[index][0]][watAirVec[index][1]][1] + waterZoom[watAirVec[index][0]+1][watAirVec[index][1]][1]) / (2));
watAirIntVec[index][2] = ((airZoom[watAirVec[index][0]][watAirVec[index][1]][1] + airZoom[watAirVec[index][0]+1][watAirVec[index][1]][1]) / (2));
}

fibWatIntVec[0][0] = fibWatIntVec[1][0];
fibWatIntVec[0][1] = fibWatIntVec[1][1];
fibWatIntVec[0][2] = fibWatIntVec[1][2];
fibAirIntVec[0][0] = fibAirIntVec[1][0];
fibAirIntVec[0][1] = fibAirIntVec[1][1];
fibAirIntVec[0][2] = fibAirIntVec[1][2];

cout << "Finished! Let us organize the vectors!\n";

// Organize them based on fiber column
// Fiber Water
for (int i = 0; i < fibWat; ++i)
for (int j = 0; j < i; ++j)
if (fibWatIntVec[j][0] > fibWatIntVec[i][0])
std::swap(fibWatIntVec[i], fibWatIntVec[j]);
// Fiber Air
for (int i = 0; i < fibAir; ++i)
for (int j = 0; j < i; ++j)
if (fibAirIntVec[j][0] > fibAirIntVec[i][0])
std::swap(fibAirIntVec[i], fibAirIntVec[j]);
// Water Air
for (int i = 0; i < watAir; ++i)
for (int j = 0; j < i; ++j)
if (watAirIntVec[j][0] > watAirIntVec[i][0])
std::swap(watAirIntVec[i], watAirIntVec[j]);


cout << "Save 'em!\n";
std::ofstream outputFibWat(run + "fibWat.txt");
for (int i = 0; i < (fibWat - 1); i++)
{
outputFibWat << fibWatIntVec[i][0] << "\t" << fibWatIntVec[i][1] << "\t" << fibWatIntVec[i][2] << "\n";
}
outputFibWat.close();
std::ofstream outputFibAir(run + "fibAir.txt");
for (int i = 0; i < (fibAir - 1); i++)
{
```

```
outputFibAir << fibAirIntVec[i][0] << "\t" << fibAirIntVec[i][1] << "\t" << fibAirIntVec[i][2] << "\n";
}
outputFibAir.close();


double aFib = watAirIntVec[watAir - 1][0];
double aAir = watAirIntVec[watAir - 1][2];
double bFib = watAirIntVec[watAir - 2][0];
double bAir = watAirIntVec[watAir - 2][2];
double airHej = aAir + ((0.5 - aFib) / (bFib - aFib))*(bAir - aAir);
double fibHej = 0.5;
double watHej = 1 - fibHej - airHej;

watAirIntVec[watAir][0] = fibHej;
watAirIntVec[watAir][1] = watHej;
watAirIntVec[watAir][2] = airHej;

// Water Air
for (int i = 0; i < watAir; ++i)
for (int j = 0; j < i; ++j)
if (watAirIntVec[j][0] > watAirIntVec[i][0])
std::swap(watAirIntVec[i], watAirIntVec[j]);

std::ofstream outputwatAir(run + "watAir.txt");
for (int i = 0; i < watAir; i++)
{
outputwatAir << watAirIntVec[i][0] << "\t" << watAirIntVec[i][1] << "\t" << watAirIntVec[i][2] << "\n";
}
outputwatAir.close();

cout << "Okay! Now a projection triangle is formed, let us move on!\n";

// Initialize fiberMatrix.

//Loop through matrix
//std::ofstream outputFib(timeFolder + "0.txt");

for (int rI = 0; rI < (yDim + 1); rI++)
{
for (int cI = 0; cI < (xDim + 1); cI++)
{
if (Xplane[rI][cI]<0.5)
{
fiber[rI][cI][0] = 1;
water[rI][cI][0] = 0;
air[rI][cI][0] = 0;
}
if (Xplane[rI][cI]>2 && Xplane[rI][cI]<3)
{
fiber[rI][cI][0] = 1;
water[rI][cI][0] = 0;
air[rI][cI][0] = 0;
}
if (Xplane[rI][cI]>3.5 && Xplane[rI][cI]<4)
{
fiber[rI][cI][0] = 1;
water[rI][cI][0] = 0;
air[rI][cI][0] = 0;
}
if (Xplane[rI][cI]>5)
{
fiber[rI][cI][0] = 1;
water[rI][cI][0] = 0;
air[rI][cI][0] = 0;
}
if (fiber[rI][cI][0] == 0)
{
if (Yplane[rI][cI] < 0.5)
{
fiber[rI][cI][0] = 0;
water[rI][cI][0] = 1;
air[rI][cI][0] = 0;
}
else
{
fiber[rI][cI][0] = 0;
water[rI][cI][0] = 0;
air[rI][cI][0] = 1;
}
}
}
}
// Initialize fiberMatrix.
```

```
//outputFib.close();

saveFileDirectory(fiber, 0, timeFolderFib, 0, xDim, yDim);
saveFileDirectory(water, 0, timeFolderWat, 0, xDim, yDim);
saveFileDirectory(air, 0, timeFolderAir, 0, xDim, yDim);



//Step length
step = ((lengthSquare) / ((double)xDim));
std::ofstream outputnuOfEle39(run + "step.txt");
outputnuOfEle39 << step;
outputnuOfEle39.close();

// Define timeSteps
tStep = ((step*step) / (10));
tTot = 80 * tStep;
// Save all the variables above
// save tStep
std::ofstream outputtStep22(run + "tStep.txt");
outputtStep22 << tStep;
outputtStep22.close();
// Save tTot
std::ofstream outputtTot2step32(run + "tTot.txt");
outputtTot2step32 << tTot;
outputtTot2step32.close();



// REAL PROGRAM STARTS
for (int tI = 1; tI < (timeTotal + 1); tI++)
{
// EULER FORWARD!!!
// FIBER
eulerForward(fiber, 1, tTot, step, tStep, 0, 1, xDim, yDim);
cout << "\t\t\tFiber Ready!\n";
eulerForward(water, 1, tTot, step, tStep, 0, 1, xDim, yDim);
cout << "\t\t\tWater Ready!\n";
eulerForward(air, 1, tTot, step, tStep, 0, 1, xDim, yDim);
cout << "\t\t\tAir Ready!\n";


saveFileDirectory(fiber, 1, diffWater, tI, xDim, yDim);
saveFileDirectory(water, 1, diffFiber, tI, xDim, yDim);
saveFileDirectory(air, 1, diffAir, tI, xDim, yDim);

sharpening(fiber, water, air, 1, watAirIntVec, watAir);


saveFileDirectory(fiber, 1, timeFolderFib, tI, xDim, yDim);
saveFileDirectory(water, 1, timeFolderWat, tI, xDim, yDim);
saveFileDirectory(air, 1, timeFolderAir, tI, xDim, yDim);


// Change values of first vector

for (int rI = 0; rI < (yDim + 1); rI++)
{
for (int cI = 0; cI < (xDim + 1); cI++)
{
fiber[rI][cI][0] = fiber[rI][cI][1];
water[rI][cI][0] = water[rI][cI][1];
air[rI][cI][0] = air[rI][cI][1];
}
}

cout << "\nTime step\t" << tI << "\tof\t" << timeTotal << "\n\n";
}







// End of program
cout << "Program worked hey?! \n\n\n\n";
return 0;
```

```
}


// FUNCTION SAVE FILE DIRECTORY
void saveFileDirectory(vector<vector<vector<double>>> ARRAY, int tI, string directory, int time, int xDim, int yDim)
{
//std::string folder = ".//fib0";
//CreateDirectory(directory.c_str(), NULL);
std::ostringstream ostr;
ostr << time;
std::string theNumberString = ostr.str();
std::string end = ".txt";
std::string fileFolder = directory + "//t" + theNumberString + end;
std::ofstream output(fileFolder);

for (int rI = 0; rI < (yDim + 1); rI++)
{
for (int cI = 0; cI < (xDim + 1); cI++)
{
output << ARRAY[rI][cI][tI] << " ";
}
output << "\n";
}
output.close();
}



// Void cube
void cube(vector<vector<double>>& cube, vector<vector<double>>& lpMat, int rI, int cI, int neumannX, int neumannY,
int xDim, int yDim)
{
int yTemp, xTemp;
for (int x = 0; x < 3; x++)
{
for (int y = 0; y < 3; y++)
{
xTemp = cI - 1 + x;
yTemp = rI - 1 + y;
if (yTemp == (yDim + 1))
{
if (neumannY == 1)
{
yTemp = yDim;
}
if (neumannY == 0)
{
yTemp = 0;
}
}
if (yTemp == -1)
{
if (neumannY == 1)
{
yTemp = 0;
}
if (neumannY == 0)
{
yTemp = yDim;
}
}
if (xTemp == (xDim + 1))
{
if (neumannX == 1)
{
xTemp = xDim;
}
if (neumannX == 0)
{
xTemp = 0;
}
}
if (xTemp == -1)
{
if (neumannX == 1)
{
xTemp = 0;
}
if (neumannX == 0)
{
xTemp = xDim;
}
```

```
}
cube[y][x] = lpMat[yTemp][xTemp];
}
}


}// end cube function



// Void function Laplace
void laplace(vector<vector<double>>& lpMat, vector<vector<double>>& inputMat, double step, int neumannX, int neumannY,
int xDim, int yDim)
{
// Allocate memory for the sums
double cornerSum, faceSum, nom;
// Create a cube 3x3x3
//vector<vector<vector<double>>>cubeTemp((3), vector<vector<double>>((3), vector<double>(3)));

// Create a cube 3x3x3
vector<vector<double>>cubeTemp;
cubeTemp.resize(3);
for (int index = 0; index < 3; index++)
{
cubeTemp[index].resize(3);
}

//LOOPS

for (int rI = 0; rI < (yDim + 1); rI++)
{
for (int cI = 0; cI < (xDim + 1); cI++)
{
cube(cubeTemp, inputMat, rI, cI, neumannX, neumannY, xDim, yDim);
// Calculate stuff
// FACES
faceSum = cubeTemp[1][2] + cubeTemp[1][0] + //Right + Left
cubeTemp[2][1] + cubeTemp[0][1]; // Forward + Backward
// EDGES
cornerSum = cubeTemp[2][2] + cubeTemp[2][0] + //ForwardRight + ForwardLeft
cubeTemp[0][2] + cubeTemp[0][0]; //BackwardRight + BackwardLeft
nom = ((2 * faceSum) / (3)) + ((cornerSum) / (6)) - ((10 * inputMat[rI][cI]) / (3));
// Laplace matrix
lpMat[rI][cI] = ((nom) / (step*step));
//cout <<"Laplace matrix:\t" <<lpMat[rI][cI]<<"\n";
}
}

}


// Euler Forward!
void eulerForward(vector<vector<vector<double>>>& afterMat, int timeIndex, double tTot, double step, double tStep, int neumannX, int neumannY,
int xDim, int yDim)
{

//Create temporary vector beforeMat
vector<vector<double>> beforeMat;
beforeMat.resize(yDim + 1);
for (int rI = 0; rI < (yDim + 1); rI++) {
beforeMat[rI].resize(xDim + 1);
}
// Initialiaze beforeMat to initialMat

for (int rI = 0; rI < (yDim + 1); rI++)
{
for (int cI = 0; cI < (xDim + 1); cI++)
{
beforeMat[rI][cI] = afterMat[rI][cI][timeIndex - 1];
}
}

// Euler loops: one for time, one for each
//spatial dimension
for (double tI = 0; tI < (tTot + tStep); tI += tStep)
{
vector<vector<double>> lpTemp;
lpTemp.resize(yDim + 1);
for (int rI = 0; rI < (yDim + 1); rI++) {
lpTemp[rI].resize(xDim + 1);
}
```

```
laplace(lpTemp, beforeMat, step, neumannX, neumannY, xDim, yDim);

for (int rI = 0; rI < (yDim + 1); rI++)
{
for (int cI = 0; cI < (xDim + 1); cI++)
{
afterMat[rI][cI][timeIndex] = beforeMat[rI][cI] + tStep*lpTemp[rI][cI];
beforeMat[rI][cI] = afterMat[rI][cI][timeIndex];
}
}
lpTemp.erase(lpTemp.begin(), lpTemp.end());
//lpTemp.shrink_to_fit();
lpTemp.clear();
}

beforeMat.erase(beforeMat.begin(), beforeMat.end());
//beforeMat.shrink_to_fit();
beforeMat.clear();
}// end Euler


// SHARPENING FUNCTION
// SHARPENING FUNCTION
//void sharpening(vector<vector<vector<vector<double>>>>& input, int tI, double threshold)
void sharpening(vector<vector<vector<double>>>& fiber, vector<vector<vector<double>>>& water, vector<vector<vector<double>>>& air, int tI,
vector<vector<double>>& waterAir, int lengthInterface)
{
double z0, z1, x0, x1, xd;
for (int rI = 0; rI < (yDim + 1); rI++)
{
for (int cI = 0; cI < (xDim + 1); cI++)
{
vector<double>tempVec(3);
/*tempVec[0] = fiber[rI][cI][zI][tI];
tempVec[1] = water[rI][cI][zI][tI];
tempVec[2] = air[rI][cI][zI][tI];*/
/*tempVec[0] = ((fiber[rI][cI][tI]) / (fiber[rI][cI][tI] + water[rI][cI][tI] + air[rI][cI][tI]));
tempVec[1] = ((water[rI][cI][tI]) / (fiber[rI][cI][tI] + water[rI][cI][tI] + air[rI][cI][tI]));
tempVec[2] = ((air[rI][cI][tI]) / (fiber[rI][cI][tI] + water[rI][cI][tI] + air[rI][cI][tI]));*/
tempVec[0] = fiber[rI][cI][tI];
tempVec[1] = water[rI][cI][tI];
tempVec[2] = air[rI][cI][tI];
if (tempVec[0] > 0.5)
{
fiber[rI][cI][tI] = 1;
water[rI][cI][tI] = 0;
air[rI][cI][tI] = 0;
}
else
{
z0 = 0;
z1 = 0;
x0 = 0;
x1 = 0;
xd = 0;
for (int index = 0; index < (lengthInterface - 1); index++)
{
if (tempVec[2] >= waterAir[index + 1][2] && tempVec[2] <= waterAir[index][2] && waterAir[index][2] != 0)
{
z0 = waterAir[index][2];
z1 = waterAir[index + 1][2];
x0 = waterAir[index][1];
x1 = waterAir[index + 1][1];
xd = x0 + (((tempVec[2] - z0) / (z1 - z0))*(x1 - x0));
//cout << "Great success dude! xd=\t"<<xd<<"\n";
break;
}
}
if (xd == 0)
{
double minimum = 25;
double minIndex = 1;
for (int index = 0; index < (lengthInterface - 1); index++)
{
if (abs(tempVec[2] - waterAir[index + 1][2])<minimum)
{
minimum = abs(tempVec[2] - waterAir[index + 1][2]);
minIndex = index + 1;
}
}
xd = waterAir[minIndex][1];
//cout << "Big fail dude! xd=\t" << xd <<"\n";
}
```

```
if (tempVec[1] > xd)
{
fiber[rI][cI][tI] = 0;
water[rI][cI][tI] = 1;
air[rI][cI][tI] = 0;
}
else
{
fiber[rI][cI][tI] = 0;
water[rI][cI][tI] = 0;
air[rI][cI][tI] = 1;
}
}

}
}
}// end sharpening
```

A.3 Narrow channels filled with obstacles

```
#define _USE_MATH_DEFINES

// User defines
// Mathematics stuff
//#define _USE_MATH_DEFINES
//#define WIN32_LEAN_AND_MEAN
// Include headers
#include <SDKDDKVer.h>
#include <windows.h>
#include <stdlib.h>
#include <malloc.h>
#include <memory.h>
#include <tchar.h>
#include <iostream>
#include <string>
#include <cmath>
//#include "engine.h"
#include <fstream>
#include<direct.h>
#include <sstream>
#include <stdio.h>
#include "math.h"
#include<vector>
#include<time.h>
#include <ctime>
#include <chrono>
// Constants
// number Of pixels
static const int  xDim = 300;
static const int  yDim = 300;
static const int  xDimZoom = 500;
static const int  yDimZoom = 500;
static const int  timeTotal = 300;
// radius of sphere
static const int heightSquare = 1;
static const int lengthSquare = 1;
static const int heightZoom = 1;
static const int lengthZoom = 1;
// step size
double step, tTot, tStep, stepZoom, tStepZoom, tTotZoom;
// x and y indices
double x, y;
// row and column indices
int rI, cI, zI;


// Standard stuff
using std::string;
using namespace std;
using std::vector;


// Declaration of functions
// Save file in folder
void saveFileDirectory(vector<vector<vector<double>>>, int, string, int, int, int);
void cube(vector<vector<double>>&, vector<vector<double>>&, int, int, int, int, int, int);
void laplace(vector<vector<double>>&, vector<vector<double>>&, double, int, int, int, int);
```

```
void eulerForward(vector<vector<vector<double>>>&, int, double, double, double, int, int, int, int);
void sharpening(vector<vector<vector<double>>>&, vector<vector<vector<double>>>&, vector<vector<vector<double>>>&, int,
vector<vector<double>>&, int);



struct tm newtime;
__time32_t aclock;



// MAIN PROGRAM
int main()
{
// Working directory
string run;
std::ostringstream eleStrFib;
eleStrFib << xDim;
std::string eleStringFib = eleStrFib.str();
std::ostringstream eleStrFib2;
eleStrFib2 << yDim;
std::string eleStringFib2 = eleStrFib2.str();
std::ostringstream timeStrFib;
timeStrFib << timeTotal;
std::string timeStringFib = timeStrFib.str();
run = ".//20150516FifthTry" + eleStringFib + "Ydim" + eleStringFib2 + "Time" + timeStringFib + "//";
//run = ".//Repetitive//";
//run = ".//newFolder//";
CreateDirectory(run.c_str(), NULL);
string fiberStr = run + "fiber";
CreateDirectory(fiberStr.c_str(), NULL);
string waterStr = run + "water";
CreateDirectory(waterStr.c_str(), NULL);
string airStr = run + "air";
CreateDirectory(airStr.c_str(), NULL);
// Create timefolder
string timeFolderFib = fiberStr + "//timeFolder//";
CreateDirectory(timeFolderFib.c_str(), NULL);
string timeFolderWat = waterStr + "//timeFolder//";
CreateDirectory(timeFolderWat.c_str(), NULL);
string timeFolderAir = airStr + "//timeFolder//";
CreateDirectory(timeFolderAir.c_str(), NULL);


cout << "\tHello there!\n\n\tNow starts the program!\n\n----------------------------------\n\n";
// current date/time based on current system


//Time stuff
char buffer[32];
errno_t errNum;
_time32(&aclock);   // Get time in seconds.
_localtime32_s(&newtime, &aclock);   // Convert time to struct tm form.

// Print local time as a string.

errNum = asctime_s(buffer, 32, &newtime);
if (errNum)
{
printf("Error code: %d", (int)errNum);
return 1;
}
printf("Current date and time: %s\n\n", buffer);



//Projection triangle
string diffFiber = run + "//DiffusedFiber//";
CreateDirectory(diffFiber.c_str(), NULL);
string diffWater = run + "//DiffusedWater//";
CreateDirectory(diffWater.c_str(), NULL);
string diffAir = run + "//DiffusedAir//";
CreateDirectory(diffAir.c_str(), NULL);


//Save number of element
std::ofstream outputnuOfEle(run + "xDim.txt");
outputnuOfEle << xDim;
outputnuOfEle.close();
//Save number of element
std::ofstream outputnuOfEle2(run + "yDim.txt");
```

```
outputnuOfEle2 << yDim;
outputnuOfEle2.close();
//Save radius
std::ofstream outputRadius(run + "length.txt");
outputRadius << lengthSquare;
outputRadius.close();
//Save radius
std::ofstream outputRadius2(run + "height.txt");
outputRadius2 << heightSquare;
outputRadius2.close();
// Define mesh
vector<vector<double>>Xplane;
vector<vector<double>>Yplane;
Xplane.resize(yDim + 1);
Yplane.resize(yDim + 1);
for (int rI = 0; rI < (yDim + 1); rI++)
{
Xplane[rI].resize(xDim + 1);
Yplane[rI].resize(xDim + 1);

}
vector<vector<double>>XplaneZoom;
vector<vector<double>>YplaneZoom;
XplaneZoom.resize(yDimZoom + 1);
YplaneZoom.resize(yDimZoom + 1);
for (int rI = 0; rI < (yDimZoom + 1); rI++)
{
XplaneZoom[rI].resize(xDimZoom + 1);
YplaneZoom[rI].resize(xDimZoom + 1);
}

//Step length
step = ((lengthSquare) / ((double)xDim));
std::ofstream outputnuOfEle3(run + "step.txt");
outputnuOfEle3 << step;
outputnuOfEle3.close();
// Assignvalues to mesh
//Save matrices in files
std::ofstream outputX(run + "Xplane.txt");
std::ofstream outputY(run + "Yplane.txt");
for (rI = 0, y = lengthSquare; rI < (yDim + 1); rI++, y -= (step))
{
for (cI = 0, x = 0; cI < (xDim + 1); cI++, x += (step))
{
Xplane[rI][cI] = x;
Yplane[rI][cI] = y;
outputX << Xplane[rI][cI] << " ";
outputY << Yplane[rI][cI] << " ";
}
outputX << "\n";
outputY << "\n";
}
// Close files
outputX.close();
outputY.close();




//Zoomed in stuff

//Step length
stepZoom = ((lengthZoom) / ((double)xDimZoom));
std::ofstream outputstepZoom(run + "stepZoom.txt");
outputstepZoom << stepZoom;
outputstepZoom.close();
// Assignvalues to mesh
//Save matrices in files
std::ofstream outputXZoom(run + "XplaneZoom.txt");
std::ofstream outputYZoom(run + "YplaneZoom.txt");
for (rI = 0, y = 1; rI < (yDimZoom + 1); rI++, y -= (stepZoom))
{
for (cI = 0, x = 0; cI < (xDimZoom + 1); cI++, x += (stepZoom))
{
XplaneZoom[rI][cI] = x;
YplaneZoom[rI][cI] = y;
outputXZoom << XplaneZoom[rI][cI] << " ";
outputYZoom << YplaneZoom[rI][cI] << " ";
}
outputXZoom << "\n";
outputYZoom << "\n";
}
```

```
// Close files
outputXZoom.close();
outputYZoom.close();


cout << "Second print\n";

vector<vector<vector<double>>> fiberZoom;
vector<vector<vector<double>>> waterZoom;
vector<vector<vector<double>>> airZoom;
fiberZoom.resize(yDimZoom + 1);
waterZoom.resize(yDimZoom + 1);
airZoom.resize(yDimZoom + 1);
for (int rI = 0; rI < (yDimZoom + 1); rI++)
{
fiberZoom[rI].resize(xDimZoom + 1);
waterZoom[rI].resize(xDimZoom + 1);
airZoom[rI].resize(xDimZoom + 1);
for (int cI = 0; cI < (xDimZoom + 1); cI++)
{
fiberZoom[rI][cI].resize(2);
waterZoom[rI][cI].resize(2);
airZoom[rI][cI].resize(2);
}
}

cout << "Third print\n";

for (int rI = 0; rI < (yDimZoom + 1); rI++)
{
for (int cI = 0; cI < (xDimZoom + 1); cI++)
{
if (XplaneZoom[rI][cI] <= 0.5)
{
fiberZoom[rI][cI][0] = 1;
waterZoom[rI][cI][0] = 0;
airZoom[rI][cI][0] = 0;
}
if (XplaneZoom[rI][cI] > 0.5)
{
if (YplaneZoom[rI][cI] <= (1 - XplaneZoom[rI][cI]))
{
fiberZoom[rI][cI][0] = 0;
waterZoom[rI][cI][0] = 1;
airZoom[rI][cI][0] = 0;
}
if (YplaneZoom[rI][cI] > (1 - XplaneZoom[rI][cI]))
{
fiberZoom[rI][cI][0] = 0;
waterZoom[rI][cI][0] = 0;
airZoom[rI][cI][0] = 1;
}
}
}
}

cout << "Fourth print\n";


saveFileDirectory(fiberZoom, 0, diffFiber, 0, xDimZoom, yDimZoom);
saveFileDirectory(waterZoom, 0, diffWater, 0, xDimZoom, yDimZoom);
saveFileDirectory(airZoom, 0, diffAir, 0, xDimZoom, yDimZoom);



// Allocate Memory fibermatrix.
//vector<vector<vector<double>>>fiber((yDim + 1), vector<vector<double>>((xDim + 1), vector<double>(2)));
vector<vector<vector<double>>> fiber;
vector<vector<vector<double>>> water;
vector<vector<vector<double>>> air;
fiber.resize(yDim + 1);
water.resize(yDim + 1);
air.resize(yDim + 1);
for (int rI = 0; rI < (yDim + 1); rI++)
{
fiber[rI].resize(xDim + 1);
water[rI].resize(xDim + 1);
air[rI].resize(xDim + 1);
for (int cI = 0; cI < (xDim + 1); cI++)
{
fiber[rI][cI].resize(2);
water[rI][cI].resize(2);
air[rI][cI].resize(2);
```

```
}
}


// Define timeSteps
tStepZoom = ((stepZoom*stepZoom) / (10));
tTotZoom = 3500 * tStepZoom;
// Save all the variables above
// save tStep
std::ofstream outputtStep1(run + "tStepZoom.txt");
outputtStep1 << tStepZoom;
outputtStep1.close();
// Save tTot
std::ofstream outputtTot2step3(run + "tTotZoom.txt");
outputtTot2step3 << tTotZoom;
outputtTot2step3.close();


double minEleX, minEleXSigned, minEleY, minEleYSigned, diffX, diffY, minDiffX, minDiffY;
minDiffX = 25;
minDiffY = 25;
for (int index = 0; index < (xDimZoom + 1); index++)
{
diffX = XplaneZoom[1][index] - 0.5;
//cout << "Difference X:\t" << diffX << "\n";
if (abs(diffX) < minDiffX)
{
minEleX = abs(XplaneZoom[1][index]);
minEleXSigned = XplaneZoom[1][index];
minDiffX = abs(diffX);
//cout << "Minimal element:\t" << minEleXSigned << "\n";
}
}
for (int index = 0; index < (yDimZoom + 1); index++)
{
diffY = YplaneZoom[index][1] - 0.5;
//cout << "Difference Y:\t" << diffY << "\n";
if (abs(diffY) < minDiffY)
{
minEleY = abs(YplaneZoom[index][1]);
minEleYSigned = YplaneZoom[index][1];
minDiffY = abs(diffY);
//cout << "Minimal element:\t" << minEleYSigned << "\n";
}
}

cout << "Minimal value X:\t" << minEleXSigned << "\n";
cout << "Minimal value Y:\t" << minEleYSigned << "\n";


// Number of edge elements
int fibWat = 0;
int fibAir = 0;
int watAir = 0;



for (int rI = 1; rI < (yDimZoom + 1); rI++)
{
for (int cI = 0; cI < (xDimZoom + 1); cI++)
{
// Interface fiber air
if (XplaneZoom[rI][cI] == minEleXSigned && YplaneZoom[rI][cI] >= minEleYSigned && YplaneZoom[rI][cI] <= 0.7)
{
fibAir++;
}
// Interface fiber water
if (XplaneZoom[rI][cI] == minEleXSigned && YplaneZoom[rI][cI] <= minEleYSigned && YplaneZoom[rI][cI] > 0.2)
{
fibWat++;
}
// Interface water air
if (YplaneZoom[rI][cI] <= (1 - XplaneZoom[rI][cI])
&& YplaneZoom[rI - 1][cI] > (1 - XplaneZoom[rI][cI])
&& XplaneZoom[rI][cI] >= minEleXSigned && XplaneZoom[rI][cI] <= 0.8)
{
watAir++;
}
}
}
```

```
std::ofstream outputtStepfibWat(run + "fibWatInt.txt");
outputtStepfibWat << fibWat;
outputtStepfibWat.close();
// Save tTot
std::ofstream outputtTot2stepfibAir(run + "fibAirInt.txt");
outputtTot2stepfibAir << fibAir;
outputtTot2stepfibAir.close();
std::ofstream outputtTot2stepwatAir(run + "watAirInt.txt");
outputtTot2stepwatAir << watAir;
outputtTot2stepwatAir.close();


vector<vector<double>>fibWatVec((fibWat), vector<double>(2));
vector<vector<double>>fibAirVec((fibAir), vector<double>(2));
vector<vector<double>>watAirVec((watAir), vector<double>(2));
int indexFibWat = 0;
int indexFibAir = 0;
int indexWatAir = 0;


for (int rI = 1; rI < (yDimZoom + 1); rI++)
{
for (int cI = 0; cI < (xDimZoom + 1); cI++)
{
// Interface fiber air
if (XplaneZoom[rI][cI] == minEleXSigned && YplaneZoom[rI][cI] >= minEleYSigned && YplaneZoom[rI][cI] <= 0.7)
{
fibAirVec[indexFibAir][0] = rI;
fibAirVec[indexFibAir][1] = cI;
indexFibAir++;
}
// Interface fiber water
if (XplaneZoom[rI][cI] == minEleXSigned && YplaneZoom[rI][cI] <= minEleYSigned &&  YplaneZoom[rI][cI] > 0.2)
{
fibWatVec[indexFibWat][0] = rI;
fibWatVec[indexFibWat][1] = cI;
indexFibWat++;
}
// Interface water air
if (YplaneZoom[rI][cI] <= (1 - XplaneZoom[rI][cI])
&& YplaneZoom[rI - 1][cI] > (1 - XplaneZoom[rI][cI])
&& XplaneZoom[rI][cI] >= minEleXSigned && XplaneZoom[rI][cI] <= 0.8)
{
watAirVec[indexWatAir][0] = rI;
watAirVec[indexWatAir][1] = cI;
indexWatAir++;
}
}
}



vector<vector<double>>fibWatIntVec((fibWat), vector<double>(3));
vector<vector<double>>fibAirIntVec((fibAir), vector<double>(3));
vector<vector<double>>watAirIntVec((watAir + 1), vector<double>(3));

cout << "Now starts short Euler\n";
_time32(&aclock);   // Get time in seconds.
_localtime32_s(&newtime, &aclock);   // Convert time to struct tm form.
errNum = asctime_s(buffer, 32, &newtime);
if (errNum)
{
printf("Error code: %d", (int)errNum);
return 1;
}
printf("Current date and time: %s\n\n", buffer);

eulerForward(fiberZoom, 1, tTotZoom, stepZoom, tStepZoom, 1, 1, xDimZoom, yDimZoom);
eulerForward(waterZoom, 1, tTotZoom, stepZoom, tStepZoom, 1, 1, xDimZoom, yDimZoom);
eulerForward(airZoom, 1, tTotZoom, stepZoom, tStepZoom, 1, 1, xDimZoom, yDimZoom);

for (int index = 0; index < (fibWat - 1); index++)
{
fibWatIntVec[index][0] = fiberZoom[fibWatVec[index][0]][fibWatVec[index][1]][1];
fibWatIntVec[index][1] = waterZoom[fibWatVec[index][0]][fibWatVec[index][1]][1];
fibWatIntVec[index][2] = airZoom[fibWatVec[index][0]][fibWatVec[index][1]][1];
}

for (int index = 0; index < (fibAir - 1); index++)
{
fibAirIntVec[index][0] = fiberZoom[fibAirVec[index][0]][fibAirVec[index][1]][1];
fibAirIntVec[index][1] = waterZoom[fibAirVec[index][0]][fibAirVec[index][1]][1];
```

```
fibAirIntVec[index][2] = airZoom[fibAirVec[index][0]][fibAirVec[index][1]][1];
}

for (int index = 0; index < (watAir - 1); index++)
{
watAirIntVec[index][0] = ((fiberZoom[watAirVec[index][0]][watAirVec[index][1]][1] + fiberZoom[watAirVec[index][0] + 1][watAirVec[index][1]][1]) / (2));
watAirIntVec[index][1] = ((waterZoom[watAirVec[index][0]][watAirVec[index][1]][1] + waterZoom[watAirVec[index][0] + 1][watAirVec[index][1]][1]) / (2));
watAirIntVec[index][2] = ((airZoom[watAirVec[index][0]][watAirVec[index][1]][1] + airZoom[watAirVec[index][0] + 1][watAirVec[index][1]][1]) / (2));
}

fibWatIntVec[0][0] = fibWatIntVec[1][0];
fibWatIntVec[0][1] = fibWatIntVec[1][1];
fibWatIntVec[0][2] = fibWatIntVec[1][2];
fibAirIntVec[0][0] = fibAirIntVec[1][0];
fibAirIntVec[0][1] = fibAirIntVec[1][1];
fibAirIntVec[0][2] = fibAirIntVec[1][2];




cout << "Finished! Let us organize the suckers!\n";

// Organize them based on fiber column
// Fiber Water
for (int i = 0; i < fibWat; ++i)
for (int j = 0; j < i; ++j)
if (fibWatIntVec[j][0] > fibWatIntVec[i][0])
std::swap(fibWatIntVec[i], fibWatIntVec[j]);
// Fiber Air
for (int i = 0; i < fibAir; ++i)
for (int j = 0; j < i; ++j)
if (fibAirIntVec[j][0] > fibAirIntVec[i][0])
std::swap(fibAirIntVec[i], fibAirIntVec[j]);
// Water Air
for (int i = 0; i < watAir; ++i)
for (int j = 0; j < i; ++j)
if (watAirIntVec[j][0] > watAirIntVec[i][0])
std::swap(watAirIntVec[i], watAirIntVec[j]);


cout << "Save 'em!\n";
std::ofstream outputFibWat(run + "fibWat.txt");
for (int i = 0; i < (fibWat - 1); i++)
{
outputFibWat << fibWatIntVec[i][0] << "\t" << fibWatIntVec[i][1] << "\t" << fibWatIntVec[i][2] << "\n";
}
outputFibWat.close();
std::ofstream outputFibAir(run + "fibAir.txt");
for (int i = 0; i < (fibAir - 1); i++)
{
outputFibAir << fibAirIntVec[i][0] << "\t" << fibAirIntVec[i][1] << "\t" << fibAirIntVec[i][2] << "\n";
}
outputFibAir.close();


double aFib = watAirIntVec[watAir - 1][0];
double aAir = watAirIntVec[watAir - 1][2];
double bFib = watAirIntVec[watAir - 2][0];
double bAir = watAirIntVec[watAir - 2][2];
double airHej = aAir + ((0.5 - aFib) / (bFib - aFib))*(bAir - aAir);
double fibHej = 0.5;
double watHej = 1 - fibHej - airHej;

watAirIntVec[watAir][0] = fibHej;
watAirIntVec[watAir][1] = watHej;
watAirIntVec[watAir][2] = airHej;

// Water Air
for (int i = 0; i < watAir; ++i)
for (int j = 0; j < i; ++j)
if (watAirIntVec[j][0] > watAirIntVec[i][0])
std::swap(watAirIntVec[i], watAirIntVec[j]);

std::ofstream outputwatAir(run + "watAir.txt");
for (int i = 0; i < watAir; i++)
{
outputwatAir << watAirIntVec[i][0] << "\t" << watAirIntVec[i][1] << "\t" << watAirIntVec[i][2] << "\n";
}
outputwatAir.close();
```

```
cout << "Okay! Now a projection triangle is formed, let us move on!\n";
_time32(&aclock);   // Get time in seconds.
_localtime32_s(&newtime, &aclock);   // Convert time to struct tm form.
errNum = asctime_s(buffer, 32, &newtime);
if (errNum)
{
printf("Error code: %d", (int)errNum);
return 1;
}
printf("Current date and time: %s\n\n", buffer);
```

```
// Initialize fiberMatrix.

//Loop through matrix
//std::ofstream outputFib(timeFolder + "0.txt");
for (int rI = 0; rI < (yDim + 1); rI++)
{
for (int cI = 0; cI < (xDim + 1); cI++)
{
if (Xplane[rI][cI]<0.25)
{
fiber[rI][cI][0] = 1;
water[rI][cI][0] = 0;
air[rI][cI][0] = 0;
}
if (Xplane[rI][cI]>0.75)
{
fiber[rI][cI][0] = 1;
water[rI][cI][0] = 0;
air[rI][cI][0] = 0;
}
if (((Xplane[rI][cI] - 0.6)*(Xplane[rI][cI] - 0.6)) + ((Yplane[rI][cI] - 0.35)*(Yplane[rI][cI] - 0.35))<(0.1*0.1))
{
fiber[rI][cI][0] = 1;
water[rI][cI][0] = 0;
air[rI][cI][0] = 0;
}
if (((Xplane[rI][cI] - 0.4)*(Xplane[rI][cI] - 0.4)) + ((Yplane[rI][cI] - 0.5)*(Yplane[rI][cI] - 0.5))<(0.1*0.1))
{
fiber[rI][cI][0] = 1;
water[rI][cI][0] = 0;
air[rI][cI][0] = 0;
}
if (((Xplane[rI][cI] - 0.6)*(Xplane[rI][cI] - 0.6)) + ((Yplane[rI][cI] - 0.65)*(Yplane[rI][cI] - 0.65))<(0.1*0.1))
{
fiber[rI][cI][0] = 1;
water[rI][cI][0] = 0;
air[rI][cI][0] = 0;
}
if (((Xplane[rI][cI] - 0.4)*(Xplane[rI][cI] - 0.4)) + ((Yplane[rI][cI] - 0.8)*(Yplane[rI][cI] - 0.8))<(0.1*0.1))
{
fiber[rI][cI][0] = 1;
water[rI][cI][0] = 0;
air[rI][cI][0] = 0;
}
if (((Xplane[rI][cI] - 0.6)*(Xplane[rI][cI] - 0.6)) + ((Yplane[rI][cI] - 0.95)*(Yplane[rI][cI] - 0.95))<(0.1*0.1))
{
fiber[rI][cI][0] = 1;
water[rI][cI][0] = 0;
air[rI][cI][0] = 0;
}
//Rest of crap änna!
```

```
if (fiber[rI][cI][0] == 0)
{
if (Yplane[rI][cI] < 0.2)
{
fiber[rI][cI][0] = 0;
water[rI][cI][0] = 1;
air[rI][cI][0] = 0;
}
else
{
fiber[rI][cI][0] = 0;
water[rI][cI][0] = 0;
air[rI][cI][0] = 1;
}
}
}
}
```

```
//outputFib.close();

saveFileDirectory(fiber, 0, timeFolderFib, 0, xDim, yDim);
saveFileDirectory(water, 0, timeFolderWat, 0, xDim, yDim);
saveFileDirectory(air, 0, timeFolderAir, 0, xDim, yDim);
```

```
//Step length
step = ((lengthSquare) / ((double)xDim));
std::ofstream outputnuOfEle39(run + "step.txt");
outputnuOfEle39 << step;
outputnuOfEle39.close();

// Define timeSteps
tStep = ((step*step) / (10));
tTot = 300 * tStep;
// Save all the variables above
// save tStep
std::ofstream outputtStep22(run + "tStep.txt");
outputtStep22 << tStep;
outputtStep22.close();
// Save tTot
std::ofstream outputtTot2step32(run + "tTot.txt");
outputtTot2step32 << tTot;
outputtTot2step32.close();
```

```
// REAL PROGRAM STARTS
for (int tI = 1; tI < (timeTotal + 1); tI++)
{
// EULER FORWARD!!!
// FIBER
eulerForward(fiber, 1, tTot, step, tStep, 0, 1, xDim, yDim);
cout << "\t\t\tFiber Ready!\n";
eulerForward(water, 1, tTot, step, tStep, 0, 1, xDim, yDim);
cout << "\t\t\tWater Ready!\n";
eulerForward(air, 1, tTot, step, tStep, 0, 1, xDim, yDim);
cout << "\t\t\tAir Ready!\n";


saveFileDirectory(fiber, 1, diffWater, tI, xDim, yDim);
saveFileDirectory(water, 1, diffFiber, tI, xDim, yDim);
saveFileDirectory(air, 1, diffAir, tI, xDim, yDim);

sharpening(fiber, water, air, 1, watAirIntVec, watAir);


saveFileDirectory(fiber, 1, timeFolderFib, tI, xDim, yDim);
saveFileDirectory(water, 1, timeFolderWat, tI, xDim, yDim);
saveFileDirectory(air, 1, timeFolderAir, tI, xDim, yDim);


// Change values of first vector

for (int rI = 0; rI < (yDim + 1); rI++)
{
for (int cI = 0; cI < (xDim + 1); cI++)
{
```

```
fiber[rI][cI][0] = fiber[rI][cI][1];
water[rI][cI][0] = water[rI][cI][1];
air[rI][cI][0] = air[rI][cI][1];
}
}
_time32(&aclock);   // Get time in seconds.
_localtime32_s(&newtime, &aclock);   // Convert time to struct tm form.
cout << "\nTime step\t" << tI << "\tof\t" << timeTotal << "\n";
errNum = asctime_s(buffer, 32, &newtime);
if (errNum)
{
printf("Error code: %d", (int)errNum);
return 1;
}
printf("Current date and time: %s\n\n", buffer);
}
```

```
// End of program
cout << "Program worked hey?! \n\n\n\n";
return 0;
}
```

```
// FUNCTION SAVE FILE DIRECTORY
void saveFileDirectory(vector<vector<vector<double>>> ARRAY, int tI, string directory, int time, int xDim, int yDim)
{
//std::string folder = ".//fib0";
//CreateDirectory(directory.c_str(), NULL);
std::ostringstream ostr;
ostr << time;
std::string theNumberString = ostr.str();
std::string end = ".txt";
std::string fileFolder = directory + "//t" + theNumberString + end;
std::ofstream output(fileFolder);

for (int rI = 0; rI < (yDim + 1); rI++)
{
for (int cI = 0; cI < (xDim + 1); cI++)
{
output << ARRAY[rI][cI][tI] << " ";
}
output << "\n";
}
output.close();
}
```

```
// Void cube
void cube(vector<vector<double>>& cube, vector<vector<double>>& lpMat, int rI, int cI, int neumannX, int neumannY,
int xDim, int yDim)
{
int yTemp, xTemp;
for (int x = 0; x < 3; x++)
{
for (int y = 0; y < 3; y++)
{
xTemp = cI - 1 + x;
yTemp = rI - 1 + y;
if (yTemp == (yDim + 1))
{
if (neumannY == 1)
{
yTemp = yDim;
}
if (neumannY == 0)
{
yTemp = 0;
}
}
if (yTemp == -1)
{
if (neumannY == 1)
{
yTemp = 0;
```

```
}
if (neumannY == 0)
{
yTemp = yDim;
}
}
if (xTemp == (xDim + 1))
{
if (neumannX == 1)
{
xTemp = xDim;
}
if (neumannX == 0)
{
xTemp = 0;
}
}
if (xTemp == -1)
{
if (neumannX == 1)
{
xTemp = 0;
}
if (neumannX == 0)
{
xTemp = xDim;
}
}
cube[y][x] = lpMat[yTemp][xTemp];
}
}


}// end cube function



// Void function Laplace
void laplace(vector<vector<double>>& lpMat, vector<vector<double>>& inputMat, double step, int neumannX, int neumannY,
int xDim, int yDim)
{
// Allocate memory for the sums
double cornerSum, faceSum, nom;
// Create a cube 3x3x3
//vector<vector<vector<double>>>cubeTemp((3), vector<vector<double>>((3), vector<double>(3)));

// Create a cube 3x3x3
vector<vector<double>>cubeTemp;
cubeTemp.resize(3);
for (int index = 0; index < 3; index++)
{
cubeTemp[index].resize(3);
}

//LOOPS

for (int rI = 0; rI < (yDim + 1); rI++)
{
for (int cI = 0; cI < (xDim + 1); cI++)
{
cube(cubeTemp, inputMat, rI, cI, neumannX, neumannY, xDim, yDim);
// Calculate stuff
// FACES
faceSum = cubeTemp[1][2] + cubeTemp[1][0] + //Right + Left
cubeTemp[2][1] + cubeTemp[0][1]; // Forward + Backward
// EDGES
cornerSum = cubeTemp[2][2] + cubeTemp[2][0] + //ForwardRight + ForwardLeft
cubeTemp[0][2] + cubeTemp[0][0]; //BackwardRight + BackwardLeft
nom = ((2 * faceSum) / (3)) + ((cornerSum) / (6)) - ((10 * inputMat[rI][cI]) / (3));
// Laplace matrix
lpMat[rI][cI] = ((nom) / (step*step));
//cout <<"Laplace matrix:\t" <<lpMat[rI][cI]<<"\n";
}
}

}


// Euler Forward!
void eulerForward(vector<vector<vector<double>>>& afterMat, int timeIndex, double tTot, double step, double tStep, int neumannX, int neumannY,
int xDim, int yDim)
```

```
{

//Create temporary vector beforeMat
vector<vector<double>> beforeMat;
beforeMat.resize(yDim + 1);
for (int rI = 0; rI < (yDim + 1); rI++) {
beforeMat[rI].resize(xDim + 1);
}
// Initialiaze beforeMat to initialMat

for (int rI = 0; rI < (yDim + 1); rI++)
{
for (int cI = 0; cI < (xDim + 1); cI++)
{
beforeMat[rI][cI] = afterMat[rI][cI][timeIndex - 1];
}
}

// Euler loops: one for time, one for each
//spatial dimension
for (double tI = 0; tI < (tTot + tStep); tI += tStep)
{
vector<vector<double>> lpTemp;
lpTemp.resize(yDim + 1);
for (int rI = 0; rI < (yDim + 1); rI++) {
lpTemp[rI].resize(xDim + 1);
}
laplace(lpTemp, beforeMat, step, neumannX, neumannY, xDim, yDim);

for (int rI = 0; rI < (yDim + 1); rI++)
{
for (int cI = 0; cI < (xDim + 1); cI++)
{
afterMat[rI][cI][timeIndex] = beforeMat[rI][cI] + tStep*lpTemp[rI][cI];
beforeMat[rI][cI] = afterMat[rI][cI][timeIndex];
}
}
lpTemp.erase(lpTemp.begin(), lpTemp.end());
//lpTemp.shrink_to_fit();
lpTemp.clear();
}

beforeMat.erase(beforeMat.begin(), beforeMat.end());
//beforeMat.shrink_to_fit();
beforeMat.clear();
}// end Euler


// SHARPENING FUNCTION
// SHARPENING FUNCTION
//void sharpening(vector<vector<vector<vector<double>>>>& input, int tI, double threshold)
void sharpening(vector<vector<vector<double>>>& fiber, vector<vector<vector<double>>>& water, vector<vector<vector<double>>>& air, int tI,
vector<vector<double>>& waterAir, int lengthInterface)
{
double z0, z1, x0, x1, xd;
for (int rI = 0; rI < (yDim + 1); rI++)
{
for (int cI = 0; cI < (xDim + 1); cI++)
{
vector<double>tempVec(3);
/*tempVec[0] = fiber[rI][cI][zI][tI];
tempVec[1] = water[rI][cI][zI][tI];
tempVec[2] = air[rI][cI][zI][tI];*/
/*tempVec[0] = ((fiber[rI][cI][tI]) / (fiber[rI][cI][tI] + water[rI][cI][tI] + air[rI][cI][tI]));
tempVec[1] = ((water[rI][cI][tI]) / (fiber[rI][cI][tI] + water[rI][cI][tI] + air[rI][cI][tI]));
tempVec[2] = ((air[rI][cI][tI]) / (fiber[rI][cI][tI] + water[rI][cI][tI] + air[rI][cI][tI]));*/
tempVec[0] = fiber[rI][cI][tI];
tempVec[1] = water[rI][cI][tI];
tempVec[2] = air[rI][cI][tI];
//if (tempVec[0] > 0.5)
if (fiber[rI][cI][0] == 1)
{
fiber[rI][cI][tI] = 1;
water[rI][cI][tI] = 0;
air[rI][cI][tI] = 0;
}
else
{
z0 = 0;
z1 = 0;
x0 = 0;
x1 = 0;
xd = 0;
```

```
for (int index = 0; index < (lengthInterface - 1); index++)
{
if (tempVec[2] >= waterAir[index + 1][2] && tempVec[2] <= waterAir[index][2] && waterAir[index][2] != 0)
{
z0 = waterAir[index][2];
z1 = waterAir[index + 1][2];
x0 = waterAir[index][1];
x1 = waterAir[index + 1][1];
xd = x0 + (((tempVec[2] - z0) / (z1 - z0))*(x1 - x0));
//cout << "Great success dude! xd=\t" << xd << "\n";
break;
}
}
if (xd == 0)
{
double minimum = 25;
double minIndex = 1;
for (int index = 0; index < (lengthInterface - 1); index++)
{
if (abs(tempVec[2] - waterAir[index + 1][2])<minimum)
{
minimum = abs(tempVec[2] - waterAir[index + 1][2]);
minIndex = index + 1;
}
}
xd = waterAir[minIndex][1];
//cout << "Big fail dude! xd=\t" << xd << "\n";
}
if (tempVec[1] > xd)
{
fiber[rI][cI][tI] = 0;
water[rI][cI][tI] = 1;
air[rI][cI][tI] = 0;
}
else
{
fiber[rI][cI][tI] = 0;
water[rI][cI][tI] = 0;
air[rI][cI][tI] = 1;
}
}

}
}
}// end sharpening
```

A.4 Labyrinth simulation

```
#define _USE_MATH_DEFINES

// User defines
// Mathematics stuff
//#define _USE_MATH_DEFINES
//#define WIN32_LEAN_AND_MEAN
// Include headers
#include <SDKDDKVer.h>
#include <windows.h>
#include <stdlib.h>
#include <malloc.h>
#include <memory.h>
#include <tchar.h>
#include <iostream>
#include <string>
#include <cmath>
//#include "engine.h"
#include <fstream>
#include<direct.h>
#include <sstream>
#include <stdio.h>
#include "math.h"
#include<vector>
#include<time.h>
#include <ctime>
#include <chrono>
// Constants
// number Of pixels
static const int  xDim = 600;
static const int  yDim = 600;
static const int  xDimZoom = 500;
static const int  yDimZoom = 500;
```

```
static const int  timeTotal = 300;
// radius of sphere
static const int heightSquare = 2;
static const int lengthSquare = 2;
static const int heightZoom = 1;
static const int lengthZoom = 1;
// step size
double step, tTot, tStep, stepZoom, tStepZoom, tTotZoom;
// x and y indices
double x, y;
// row and column indices
int rI, cI, zI;


// Standard stuff
using std::string;
using namespace std;
using std::vector;


// Declaration of functions
// Save file in folder
void saveFileDirectory(vector<vector<vector<double>>>, int, string, int, int, int);
void cube(vector<vector<double>>&, vector<vector<double>>&, int, int, int, int, int, int);
void laplace(vector<vector<double>>&, vector<vector<double>>&, double, int, int, int, int);
void eulerForward(vector<vector<vector<double>>>&, int, double, double, double, int, int, int, int);
void sharpening(vector<vector<vector<double>>>&, vector<vector<vector<double>>>&, vector<vector<vector<double>>>&, int,
vector<vector<double>>&, int);



struct tm newtime;
__time32_t aclock;



// MAIN PROGRAM
int main()
{
// Working directory
string run;
std::ostringstream eleStrFib;
eleStrFib << xDim;
std::string eleStringFib = eleStrFib.str();
std::ostringstream eleStrFib2;
eleStrFib2 << yDim;
std::string eleStringFib2 = eleStrFib2.str();
std::ostringstream timeStrFib;
timeStrFib << timeTotal;
std::string timeStringFib = timeStrFib.str();
run = ".//20150529CreepyLabyrinthThirdTry" + eleStringFib + "Ydim" + eleStringFib2 + "Time" + timeStringFib + "//";
//run = ".//Repetitive//";
//run = ".//newFolder//";
CreateDirectory(run.c_str(), NULL);
string fiberStr = run + "fiber";
CreateDirectory(fiberStr.c_str(), NULL);
string waterStr = run + "water";
CreateDirectory(waterStr.c_str(), NULL);
string airStr = run + "air";
CreateDirectory(airStr.c_str(), NULL);
// Create timefolder
string timeFolderFib = fiberStr + "//timeFolder//";
CreateDirectory(timeFolderFib.c_str(), NULL);
string timeFolderWat = waterStr + "//timeFolder//";
CreateDirectory(timeFolderWat.c_str(), NULL);
string timeFolderAir = airStr + "//timeFolder//";
CreateDirectory(timeFolderAir.c_str(), NULL);


cout << "\tHello there!\n\n\tNow starts the program!\n\n-----------------------------------\n\n";
// current date/time based on current system


//Time stuff
char buffer[32];
errno_t errNum;
_time32(&aclock);   // Get time in seconds.
_localtime32_s(&newtime, &aclock);   // Convert time to struct tm form.

// Print local time as a string.

errNum = asctime_s(buffer, 32, &newtime);
if (errNum)
```

```
{
printf("Error code: %d", (int)errNum);
return 1;
}
printf("Current date and time: %s\n\n", buffer);


//Projection triangle
string diffFiber = run + "//DiffusedFiber//";
CreateDirectory(diffFiber.c_str(), NULL);
string diffWater = run + "//DiffusedWater//";
CreateDirectory(diffWater.c_str(), NULL);
string diffAir = run + "//DiffusedAir//";
CreateDirectory(diffAir.c_str(), NULL);


//Save number of element
std::ofstream outputnuOfEle(run + "xDim.txt");
outputnuOfEle << xDim;
outputnuOfEle.close();
//Save number of element
std::ofstream outputnuOfEle2(run + "yDim.txt");
outputnuOfEle2 << yDim;
outputnuOfEle2.close();
//Save radius
std::ofstream outputRadius(run + "length.txt");
outputRadius << lengthSquare;
outputRadius.close();
//Save radius
std::ofstream outputRadius2(run + "height.txt");
outputRadius2 << heightSquare;
outputRadius2.close();
// Define mesh
vector<vector<double>>Xplane;
vector<vector<double>>Yplane;
Xplane.resize(yDim + 1);
Yplane.resize(yDim + 1);
for (int rI = 0; rI < (yDim + 1); rI++)
{
Xplane[rI].resize(xDim + 1);
Yplane[rI].resize(xDim + 1);

}
vector<vector<double>>XplaneZoom;
vector<vector<double>>YplaneZoom;
XplaneZoom.resize(yDimZoom + 1);
YplaneZoom.resize(yDimZoom + 1);
for (int rI = 0; rI < (yDimZoom + 1); rI++)
{
XplaneZoom[rI].resize(xDimZoom + 1);
YplaneZoom[rI].resize(xDimZoom + 1);
}

//Step length
step = ((lengthSquare) / ((double)xDim));
std::ofstream outputnuOfEle3(run + "step.txt");
outputnuOfEle3 << step;
outputnuOfEle3.close();
// Assignvalues to mesh
//Save matrices in files
std::ofstream outputX(run + "Xplane.txt");
std::ofstream outputY(run + "Yplane.txt");
for (rI = 0, y = lengthSquare; rI < (yDim + 1); rI++, y -= (step))
{
for (cI = 0, x = 0; cI < (xDim + 1); cI++, x += (step))
{
Xplane[rI][cI] = x;
Yplane[rI][cI] = y;
outputX << Xplane[rI][cI] << " ";
outputY << Yplane[rI][cI] << " ";
}
outputX << "\n";
outputY << "\n";
}
// Close files
outputX.close();
outputY.close();

//Zoomed in stuff

//Step length
stepZoom = ((lengthZoom) / ((double)xDimZoom));
std::ofstream outputstepZoom(run + "stepZoom.txt");
```

```
outputstepZoom << stepZoom;
outputstepZoom.close();
// Assignvalues to mesh
//Save matrices in files
std::ofstream outputXZoom(run + "XplaneZoom.txt");
std::ofstream outputYZoom(run + "YplaneZoom.txt");
for (rI = 0, y = 1; rI < (yDimZoom + 1); rI++, y -= (stepZoom))
{
for (cI = 0, x = 0; cI < (xDimZoom + 1); cI++, x += (stepZoom))
{
XplaneZoom[rI][cI] = x;
YplaneZoom[rI][cI] = y;
outputXZoom << XplaneZoom[rI][cI] << " ";
outputYZoom << YplaneZoom[rI][cI] << " ";
}
outputXZoom << "\n";
outputYZoom << "\n";
}
// Close files
outputXZoom.close();
outputYZoom.close();


cout << "Second print\n";

vector<vector<vector<double>>> fiberZoom;
vector<vector<vector<double>>> waterZoom;
vector<vector<vector<double>>> airZoom;
fiberZoom.resize(yDimZoom + 1);
waterZoom.resize(yDimZoom + 1);
airZoom.resize(yDimZoom + 1);
for (int rI = 0; rI < (yDimZoom + 1); rI++)
{
fiberZoom[rI].resize(xDimZoom + 1);
waterZoom[rI].resize(xDimZoom + 1);
airZoom[rI].resize(xDimZoom + 1);
for (int cI = 0; cI < (xDimZoom + 1); cI++)
{
fiberZoom[rI][cI].resize(2);
waterZoom[rI][cI].resize(2);
airZoom[rI][cI].resize(2);
}
}

cout << "Third print\n";

for (int rI = 0; rI < (yDimZoom + 1); rI++)
{
for (int cI = 0; cI < (xDimZoom + 1); cI++)
{
if (XplaneZoom[rI][cI] <= 0.5)
{
fiberZoom[rI][cI][0] = 1;
waterZoom[rI][cI][0] = 0;
airZoom[rI][cI][0] = 0;
}
if (XplaneZoom[rI][cI] > 0.5)
{
if (YplaneZoom[rI][cI] <= ((3 - (4 * XplaneZoom[rI][cI])) / (2)))
{
fiberZoom[rI][cI][0] = 0;
waterZoom[rI][cI][0] = 1;
airZoom[rI][cI][0] = 0;
}
if (YplaneZoom[rI][cI] > ((3 - (4 * XplaneZoom[rI][cI])) / (2)))
{
fiberZoom[rI][cI][0] = 0;
waterZoom[rI][cI][0] = 0;
airZoom[rI][cI][0] = 1;
}
}
}
}

cout << "Fourth print\n";


saveFileDirectory(fiberZoom, 0, diffFiber, 0, xDimZoom, yDimZoom);
saveFileDirectory(waterZoom, 0, diffWater, 0, xDimZoom, yDimZoom);
saveFileDirectory(airZoom, 0, diffAir, 0, xDimZoom, yDimZoom);
```

```
// Allocate Memory fibermatrix.
//vector<vector<vector<double>>>fiber((yDim + 1), vector<vector<double>>((xDim + 1), vector<double>(2)));
vector<vector<vector<double>>> fiber;
vector<vector<vector<double>>> water;
vector<vector<vector<double>>> air;
fiber.resize(yDim + 1);
water.resize(yDim + 1);
air.resize(yDim + 1);
for (int rI = 0; rI < (yDim + 1); rI++)
{
fiber[rI].resize(xDim + 1);
water[rI].resize(xDim + 1);
air[rI].resize(xDim + 1);
for (int cI = 0; cI < (xDim + 1); cI++)
{
fiber[rI][cI].resize(2);
water[rI][cI].resize(2);
air[rI][cI].resize(2);
}
}



// Define timeSteps
tStepZoom = ((stepZoom*stepZoom) / (10));
tTotZoom = 3500 * tStepZoom;
// Save all the variables above
// save tStep
std::ofstream outputtStep1(run + "tStepZoom.txt");
outputtStep1 << tStepZoom;
outputtStep1.close();
// Save tTot
std::ofstream outputtTot2step3(run + "tTotZoom.txt");
outputtTot2step3 << tTotZoom;
outputtTot2step3.close();


double minEleX, minEleXSigned, minEleY, minEleYSigned, diffX, diffY, minDiffX, minDiffY;
minDiffX = 25;
minDiffY = 25;
for (int index = 0; index < (xDimZoom + 1); index++)
{
diffX = XplaneZoom[1][index] - 0.5;
//cout << "Difference X:\t" << diffX << "\n";
if (abs(diffX) < minDiffX)
{
minEleX = abs(XplaneZoom[1][index]);
minEleXSigned = XplaneZoom[1][index];
minDiffX = abs(diffX);
//cout << "Minimal element:\t" << minEleXSigned << "\n";
}
}
for (int index = 0; index < (yDimZoom + 1); index++)
{
diffY = YplaneZoom[index][1] - 0.5;
//cout << "Difference Y:\t" << diffY << "\n";
if (abs(diffY) < minDiffY)
{
minEleY = abs(YplaneZoom[index][1]);
minEleYSigned = YplaneZoom[index][1];
minDiffY = abs(diffY);
//cout << "Minimal element:\t" << minEleYSigned << "\n";
}
}

cout << "Minimal value X:\t" << minEleXSigned << "\n";
cout << "Minimal value Y:\t" << minEleYSigned << "\n";


// Number of edge elements
int fibWat = 0;
int fibAir = 0;
int watAir = 0;



for (int rI = 1; rI < (yDimZoom + 1); rI++)
{
for (int cI = 0; cI < (xDimZoom + 1); cI++)
{
// Interface fiber air
if (XplaneZoom[rI][cI] == minEleXSigned && YplaneZoom[rI][cI] >= minEleYSigned && YplaneZoom[rI][cI] <= 0.7)
{
```

```
fibAir++;
}
// Interface fiber water
if (XplaneZoom[rI][cI] == minEleXSigned && YplaneZoom[rI][cI] <= minEleYSigned && YplaneZoom[rI][cI] > 0.2)
{
fibWat++;
}
// Interface water air
if (YplaneZoom[rI][cI] <= ((3 - (4 * XplaneZoom[rI][cI])) / (2))
&& YplaneZoom[rI - 1][cI] > ((3 - (4 * XplaneZoom[rI][cI])) / (2))
&& XplaneZoom[rI][cI] >= minEleXSigned && XplaneZoom[rI][cI] <= 0.7)
{
watAir++;
}
}
}



std::ofstream outputtStepfibWat(run + "fibWatInt.txt");
outputtStepfibWat << fibWat;
outputtStepfibWat.close();
// Save tTot
std::ofstream outputtTot2stepfibAir(run + "fibAirInt.txt");
outputtTot2stepfibAir << fibAir;
outputtTot2stepfibAir.close();
std::ofstream outputtTot2stepwatAir(run + "watAirInt.txt");
outputtTot2stepwatAir << watAir;
outputtTot2stepwatAir.close();


vector<vector<double>>fibWatVec((fibWat), vector<double>(2));
vector<vector<double>>fibAirVec((fibAir), vector<double>(2));
vector<vector<double>>watAirVec((watAir), vector<double>(2));
int indexFibWat = 0;
int indexFibAir = 0;
int indexWatAir = 0;


for (int rI = 1; rI < (yDimZoom + 1); rI++)
{
for (int cI = 0; cI < (xDimZoom + 1); cI++)
{
// Interface fiber air
if (XplaneZoom[rI][cI] == minEleXSigned && YplaneZoom[rI][cI] >= minEleYSigned && YplaneZoom[rI][cI] <= 0.7)
{
fibAirVec[indexFibAir][0] = rI;
fibAirVec[indexFibAir][1] = cI;
indexFibAir++;
}
// Interface fiber water
if (XplaneZoom[rI][cI] == minEleXSigned && YplaneZoom[rI][cI] <= minEleYSigned &&  YplaneZoom[rI][cI] > 0.2)
{
fibWatVec[indexFibWat][0] = rI;
fibWatVec[indexFibWat][1] = cI;
indexFibWat++;
}
// Interface water air
if (YplaneZoom[rI][cI] <= ((3 - (4 * XplaneZoom[rI][cI])) / (2))
&& YplaneZoom[rI - 1][cI] > ((3 - (4 * XplaneZoom[rI][cI])) / (2))
&& XplaneZoom[rI][cI] >= minEleXSigned && XplaneZoom[rI][cI] <= 0.7)
{
watAirVec[indexWatAir][0] = rI;
watAirVec[indexWatAir][1] = cI;
indexWatAir++;
}
}
}



vector<vector<double>>fibWatIntVec((fibWat), vector<double>(3));
vector<vector<double>>fibAirIntVec((fibAir), vector<double>(3));
vector<vector<double>>watAirIntVec((watAir + 1), vector<double>(3));

cout << "Now starts short Euler\n";
_time32(&aclock);   // Get time in seconds.
_localtime32_s(&newtime, &aclock);   // Convert time to struct tm form.
errNum = asctime_s(buffer, 32, &newtime);
if (errNum)
{
printf("Error code: %d", (int)errNum);
return 1;
```

```
}
printf("Current date and time: %s\n\n", buffer);

eulerForward(fiberZoom, 1, tTotZoom, stepZoom, tStepZoom, 1, 1, xDimZoom, yDimZoom);
eulerForward(waterZoom, 1, tTotZoom, stepZoom, tStepZoom, 1, 1, xDimZoom, yDimZoom);
eulerForward(airZoom, 1, tTotZoom, stepZoom, tStepZoom, 1, 1, xDimZoom, yDimZoom);

for (int index = 0; index < (fibWat - 1); index++)
{
fibWatIntVec[index][0] = fiberZoom[fibWatVec[index][0]][fibWatVec[index][1]][1];
fibWatIntVec[index][1] = waterZoom[fibWatVec[index][0]][fibWatVec[index][1]][1];
fibWatIntVec[index][2] = airZoom[fibWatVec[index][0]][fibWatVec[index][1]][1];
}

for (int index = 0; index < (fibAir - 1); index++)
{
fibAirIntVec[index][0] = fiberZoom[fibAirVec[index][0]][fibAirVec[index][1]][1];
fibAirIntVec[index][1] = waterZoom[fibAirVec[index][0]][fibAirVec[index][1]][1];
fibAirIntVec[index][2] = airZoom[fibAirVec[index][0]][fibAirVec[index][1]][1];
}

for (int index = 0; index < (watAir - 1); index++)
{
watAirIntVec[index][0] = ((fiberZoom[watAirVec[index][0]][watAirVec[index][1]][1] + fiberZoom[watAirVec[index][0] + 1][watAirVec[index][1]][1]) / (2));
watAirIntVec[index][1] = ((waterZoom[watAirVec[index][0]][watAirVec[index][1]][1] + waterZoom[watAirVec[index][0] + 1][watAirVec[index][1]][1]) / (2));
watAirIntVec[index][2] = ((airZoom[watAirVec[index][0]][watAirVec[index][1]][1] + airZoom[watAirVec[index][0] + 1][watAirVec[index][1]][1]) / (2));
}

fibWatIntVec[0][0] = fibWatIntVec[1][0];
fibWatIntVec[0][1] = fibWatIntVec[1][1];
fibWatIntVec[0][2] = fibWatIntVec[1][2];
fibAirIntVec[0][0] = fibAirIntVec[1][0];
fibAirIntVec[0][1] = fibAirIntVec[1][1];
fibAirIntVec[0][2] = fibAirIntVec[1][2];

cout << "Finished! Let us organize the vectors!\n";

// Organize them based on fiber column
// Fiber Water
for (int i = 0; i < fibWat; ++i)
for (int j = 0; j < i; ++j)
if (fibWatIntVec[j][0] > fibWatIntVec[i][0])
std::swap(fibWatIntVec[i], fibWatIntVec[j]);
// Fiber Air
for (int i = 0; i < fibAir; ++i)
for (int j = 0; j < i; ++j)
if (fibAirIntVec[j][0] > fibAirIntVec[i][0])
std::swap(fibAirIntVec[i], fibAirIntVec[j]);
// Water Air
for (int i = 0; i < watAir; ++i)
for (int j = 0; j < i; ++j)
if (watAirIntVec[j][0] > watAirIntVec[i][0])
std::swap(watAirIntVec[i], watAirIntVec[j]);


cout << "Save 'em!\n";
std::ofstream outputFibWat(run + "fibWat.txt");
for (int i = 0; i < (fibWat - 1); i++)
{
outputFibWat << fibWatIntVec[i][0] << "\t" << fibWatIntVec[i][1] << "\t" << fibWatIntVec[i][2] << "\n";
}
outputFibWat.close();
std::ofstream outputFibAir(run + "fibAir.txt");
for (int i = 0; i < (fibAir - 1); i++)
{
outputFibAir << fibAirIntVec[i][0] << "\t" << fibAirIntVec[i][1] << "\t" << fibAirIntVec[i][2] << "\n";
}
outputFibAir.close();


double aFib = watAirIntVec[watAir - 1][0];
double aAir = watAirIntVec[watAir - 1][2];
double bFib = watAirIntVec[watAir - 2][0];
double bAir = watAirIntVec[watAir - 2][2];
double airHej = aAir + ((0.5 - aFib) / (bFib - aFib))*(bAir - aAir);
double fibHej = 0.5;
double watHej = 1 - fibHej - airHej;

watAirIntVec[watAir][0] = fibHej;
watAirIntVec[watAir][1] = watHej;
watAirIntVec[watAir][2] = airHej;

// Water Air
```

```
for (int i = 0; i < watAir; ++i)
for (int j = 0; j < i; ++j)
if (watAirIntVec[j][0] > watAirIntVec[i][0])
std::swap(watAirIntVec[i], watAirIntVec[j]);

std::ofstream outputwatAir(run + "watAir.txt");
for (int i = 0; i < watAir; i++)
{
outputwatAir << watAirIntVec[i][0] << "\t" << watAirIntVec[i][1] << "\t" << watAirIntVec[i][2] << "\n";
}
outputwatAir.close();



cout << "Okay! Now a projection triangle is formed, let us move on!\n";
_time32(&aclock);   // Get time in seconds.
_localtime32_s(&newtime, &aclock);   // Convert time to struct tm form.
errNum = asctime_s(buffer, 32, &newtime);
if (errNum)
{
printf("Error code: %d", (int)errNum);
return 1;
}
printf("Current date and time: %s\n\n", buffer);

// Initialize fiberMatrix.

//Loop through matrix
//std::ofstream outputFib(timeFolder + "0.txt");
for (int rI = 0; rI < (yDim + 1); rI++)
{
for (int cI = 0; cI < (xDim + 1); cI++)
{
// Fiber walls
if (Xplane[rI][cI]<=0.35)
{
fiber[rI][cI][0] = 1;
water[rI][cI][0] = 0;
air[rI][cI][0] = 0;
}
if (Xplane[rI][cI]>=1.3)
{
fiber[rI][cI][0] = 1;
water[rI][cI][0] = 0;
air[rI][cI][0] = 0;
}
if (Xplane[rI][cI] >= 0.35 && Xplane[rI][cI] <= 1
&& Yplane[rI][cI] >= 0.75 && Yplane[rI][cI] <= 1.25)
{
fiber[rI][cI][0] = 1;
water[rI][cI][0] = 0;
air[rI][cI][0] = 0;
}
if (Xplane[rI][cI] >= 0.7 && Xplane[rI][cI] <= 1.75
&& Yplane[rI][cI] <= 0.5)
{
fiber[rI][cI][0] = 1;
water[rI][cI][0] = 0;
air[rI][cI][0] = 0;
}
if (Xplane[rI][cI] >= 0.7 && Xplane[rI][cI] <= 1.75
&& Yplane[rI][cI] >= 1.5)
{
fiber[rI][cI][0] = 1;
water[rI][cI][0] = 0;
air[rI][cI][0] = 0;
}
// Fiber Obstacles: Squares
if (Xplane[rI][cI] >= 0.39 && Xplane[rI][cI] <= 0.5
&& Yplane[rI][cI] >= 0.27 && Yplane[rI][cI]<=0.65)
{
fiber[rI][cI][0] = 1;
water[rI][cI][0] = 0;
air[rI][cI][0] = 0;
}
if (Xplane[rI][cI] >= 0.55 && Xplane[rI][cI] <= 0.65
&& Yplane[rI][cI] >= 0.30 && Yplane[rI][cI] <= 0.65)
{
fiber[rI][cI][0] = 1;
water[rI][cI][0] = 0;
air[rI][cI][0] = 0;
}
if (Xplane[rI][cI] >= 0.45 && Xplane[rI][cI] <= 1.05
```

```
&& Yplane[rI][cI] >= 1.32 && Yplane[rI][cI] <= 1.4)
{
fiber[rI][cI][0] = 1;
water[rI][cI][0] = 0;
air[rI][cI][0] = 0;
}
// Fiber Obstacles: Circles
if (((Xplane[rI][cI] - 1.14)*(Xplane[rI][cI] - 1.14)) + ((Yplane[rI][cI] - 0.75)*(Yplane[rI][cI] - 0.75)) < (0.1*0.1))
{
fiber[rI][cI][0] = 1;
water[rI][cI][0] = 0;
air[rI][cI][0] = 0;
}
if (((Xplane[rI][cI] - 1.15)*(Xplane[rI][cI] - 1.15)) + ((Yplane[rI][cI] - 1.15)*(Yplane[rI][cI] - 1.15)) < (0.12*0.12))
{
fiber[rI][cI][0] = 1;
water[rI][cI][0] = 0;
air[rI][cI][0] = 0;
}
// Triangle
if (Yplane[rI][cI] >= 1.5 && Yplane[rI][cI]<=1.75 &&
Xplane[rI][cI] >= ((2.75 - Yplane[rI][cI]) / (2.5)) && Xplane[rI][cI] <= ((Yplane[rI][cI]-0.25) / (2.5)))
{
fiber[rI][cI][0] = 1;
water[rI][cI][0] = 0;
air[rI][cI][0] = 0;
}

//Rest of crap ännа!
if (fiber[rI][cI][0] == 0)
{
if (Yplane[rI][cI] < 0.15)
{
fiber[rI][cI][0] = 0;
water[rI][cI][0] = 1;
air[rI][cI][0] = 0;
}
else
{
fiber[rI][cI][0] = 0;
water[rI][cI][0] = 0;
air[rI][cI][0] = 1;
}
}
}
}


//outputFib.close();

saveFileDirectory(fiber, 0, timeFolderFib, 0, xDim, yDim);
saveFileDirectory(water, 0, timeFolderWat, 0, xDim, yDim);
saveFileDirectory(air, 0, timeFolderAir, 0, xDim, yDim);



//Step length
step = ((lengthSquare) / ((double)xDim));
std::ofstream outputnuOfEle39(run + "step.txt");
outputnuOfEle39 << step;
outputnuOfEle39.close();

// Define timeSteps
tStep = ((step*step) / (10));
tTot = 600 * tStep;
// Save all the variables above
// save tStep
std::ofstream outputtStep22(run + "tStep.txt");
outputtStep22 << tStep;
outputtStep22.close();
// Save tTot
std::ofstream outputtTot2step32(run + "tTot.txt");
outputtTot2step32 << tTot;
outputtTot2step32.close();



// REAL PROGRAM STARTS
for (int tI = 1; tI < (timeTotal + 1); tI++)
{
// EULER FORWARD!!!
// FIBER
```

```
eulerForward(fiber, 1, tTot, step, tStep, 0, 1, xDim, yDim);
cout << "\t\t\tFiber Ready!\n";
eulerForward(water, 1, tTot, step, tStep, 0, 1, xDim, yDim);
cout << "\t\t\tWater Ready!\n";
eulerForward(air, 1, tTot, step, tStep, 0, 1, xDim, yDim);
cout << "\t\t\tAir Ready!\n";


saveFileDirectory(fiber, 1, diffWater, tI, xDim, yDim);
saveFileDirectory(water, 1, diffFiber, tI, xDim, yDim);
saveFileDirectory(air, 1, diffAir, tI, xDim, yDim);

sharpening(fiber, water, air, 1, watAirIntVec, watAir);


saveFileDirectory(fiber, 1, timeFolderFib, tI, xDim, yDim);
saveFileDirectory(water, 1, timeFolderWat, tI, xDim, yDim);
saveFileDirectory(air, 1, timeFolderAir, tI, xDim, yDim);


// Change values of first vector

for (int rI = 0; rI < (yDim + 1); rI++)
{
for (int cI = 0; cI < (xDim + 1); cI++)
{
fiber[rI][cI][0] = fiber[rI][cI][1];
water[rI][cI][0] = water[rI][cI][1];
air[rI][cI][0] = air[rI][cI][1];
}
}
_time32(&aclock);   // Get time in seconds.
_localtime32_s(&newtime, &aclock);   // Convert time to struct tm form.
cout << "\nTime step\t" << tI << "\tof\t" << timeTotal << "\n";
errNum = asctime_s(buffer, 32, &newtime);
if (errNum)
{
printf("Error code: %d", (int)errNum);
return 1;
}
printf("Current date and time: %s\n\n", buffer);
}

// End of program
cout << "Program worked hey?! \n\n\n\n";
return 0;
}



// FUNCTION SAVE FILE DIRECTORY
void saveFileDirectory(vector<vector<vector<double>>> ARRAY, int tI, string directory, int time, int xDim, int yDim)
{
//std::string folder = ".//fib0";
//CreateDirectory(directory.c_str(), NULL);
std::ostringstream ostr;
ostr << time;
std::string theNumberString = ostr.str();
std::string end = ".txt";
std::string fileFolder = directory + "//t" + theNumberString + end;
std::ofstream output(fileFolder);

for (int rI = 0; rI < (yDim + 1); rI++)
{
for (int cI = 0; cI < (xDim + 1); cI++)
{
output << ARRAY[rI][cI][tI] << " ";
}
output << "\n";
}
output.close();
}



// Void cube
void cube(vector<vector<double>>& cube, vector<vector<double>>& lpMat, int rI, int cI, int neumannX, int neumannY,
int xDim, int yDim)
{
int yTemp, xTemp;
for (int x = 0; x < 3; x++)
{
for (int y = 0; y < 3; y++)
```

```
{
xTemp = cI - 1 + x;
yTemp = rI - 1 + y;
if (yTemp == (yDim + 1))
{
if (neumannY == 1)
{
yTemp = yDim;
}
if (neumannY == 0)
{
yTemp = 0;
}
}
if (yTemp == -1)
{
if (neumannY == 1)
{
yTemp = 0;
}
if (neumannY == 0)
{
yTemp = yDim;
}
}
if (xTemp == (xDim + 1))
{
if (neumannX == 1)
{
xTemp = xDim;
}
if (neumannX == 0)
{
xTemp = 0;
}
}
if (xTemp == -1)
{
if (neumannX == 1)
{
xTemp = 0;
}
if (neumannX == 0)
{
xTemp = xDim;
}
}
cube[y][x] = lpMat[yTemp][xTemp];
}
}



}// end cube function



// Void function Laplace
void laplace(vector<vector<double>>& lpMat, vector<vector<double>>& inputMat, double step, int neumannX, int neumannY,
int xDim, int yDim)
{
// Allocate memory for the sums
double cornerSum, faceSum, nom;
// Create a cube 3x3x3
//vector<vector<vector<double>>>cubeTemp((3), vector<vector<double>>((3), vector<double>(3)));

// Create a cube 3x3x3
vector<vector<double>>cubeTemp;
cubeTemp.resize(3);
for (int index = 0; index < 3; index++)
{
cubeTemp[index].resize(3);
}

//LOOPS

for (int rI = 0; rI < (yDim + 1); rI++)
{
for (int cI = 0; cI < (xDim + 1); cI++)
{
cube(cubeTemp, inputMat, rI, cI, neumannX, neumannY, xDim, yDim);
// Calculate stuff
```

```
// FACES
faceSum = cubeTemp[1][2] + cubeTemp[1][0] + //Right + Left
cubeTemp[2][1] + cubeTemp[0][1]; // Forward + Backward
// EDGES
cornerSum = cubeTemp[2][2] + cubeTemp[2][0] + //ForwardRight + ForwardLeft
cubeTemp[0][2] + cubeTemp[0][0]; //BackwardRight + BackwardLeft
nom = ((2 * faceSum) / (3)) + ((cornerSum) / (6)) - ((10 * inputMat[rI][cI]) / (3));
// Laplace matrix
lpMat[rI][cI] = ((nom) / (step*step));
//cout <<"Laplace matrix:\t" <<lpMat[rI][cI]<<"\n";
}
}

}


// Euler Forward!
void eulerForward(vector<vector<vector<double>>>& afterMat, int timeIndex, double tTot, double step, double tStep, int neumannX, int neumannY,
int xDim, int yDim)
{

//Create temporary vector beforeMat
vector<vector<double>> beforeMat;
beforeMat.resize(yDim + 1);
for (int rI = 0; rI < (yDim + 1); rI++) {
beforeMat[rI].resize(xDim + 1);
}
// Initialiaze beforeMat to initialMat

for (int rI = 0; rI < (yDim + 1); rI++)
{
for (int cI = 0; cI < (xDim + 1); cI++)
{
beforeMat[rI][cI] = afterMat[rI][cI][timeIndex - 1];
}
}

// Euler loops: one for time, one for each
//spatial dimension
for (double tI = 0; tI < (tTot + tStep); tI += tStep)
{
vector<vector<double>> lpTemp;
lpTemp.resize(yDim + 1);
for (int rI = 0; rI < (yDim + 1); rI++) {
lpTemp[rI].resize(xDim + 1);
}
laplace(lpTemp, beforeMat, step, neumannX, neumannY, xDim, yDim);

for (int rI = 0; rI < (yDim + 1); rI++)
{
for (int cI = 0; cI < (xDim + 1); cI++)
{
afterMat[rI][cI][timeIndex] = beforeMat[rI][cI] + tStep*lpTemp[rI][cI];
beforeMat[rI][cI] = afterMat[rI][cI][timeIndex];
}
}
lpTemp.erase(lpTemp.begin(), lpTemp.end());
//lpTemp.shrink_to_fit();
lpTemp.clear();
}

beforeMat.erase(beforeMat.begin(), beforeMat.end());
//beforeMat.shrink_to_fit();
beforeMat.clear();
}// end Euler


// SHARPENING FUNCTION
// SHARPENING FUNCTION
//void sharpening(vector<vector<vector<vector<double>>>>& input, int tI, double threshold)
void sharpening(vector<vector<vector<vector<double>>>>& fiber, vector<vector<vector<double>>>& water, vector<vector<vector<double>>>& air, int tI,
vector<vector<double>>& waterAir, int lengthInterface)
{
double z0, z1, x0, x1, xd;
for (int rI = 0; rI < (yDim + 1); rI++)
{
for (int cI = 0; cI < (xDim + 1); cI++)
{
vector<double>tempVec(3);
/*tempVec[0] = fiber[rI][cI][zI][tI];
tempVec[1] = water[rI][cI][zI][tI];
tempVec[2] = air[rI][cI][zI][tI];*/
/*tempVec[0] = ((fiber[rI][cI][tI]) / (fiber[rI][cI][tI] + water[rI][cI][tI] + air[rI][cI][tI]));
```

```
tempVec[1] = ((water[rI][cI][tI]) / (fiber[rI][cI][tI] + water[rI][cI][tI] + air[rI][cI][tI]));
tempVec[2] = ((air[rI][cI][tI]) / (fiber[rI][cI][tI] + water[rI][cI][tI] + air[rI][cI][tI]));*/
tempVec[0] = fiber[rI][cI][tI];
tempVec[1] = water[rI][cI][tI];
tempVec[2] = air[rI][cI][tI];
//if (tempVec[0] > 0.5)
if (fiber[rI][cI][0] == 1)
{
fiber[rI][cI][tI] = 1;
water[rI][cI][tI] = 0;
air[rI][cI][tI] = 0;
}
else
{
z0 = 0;
z1 = 0;
x0 = 0;
x1 = 0;
xd = 0;
for (int index = 0; index < (lengthInterface - 1); index++)
{
if (tempVec[2] >= waterAir[index + 1][2] && tempVec[2] <= waterAir[index][2] && waterAir[index][2] != 0)
{
z0 = waterAir[index][2];
z1 = waterAir[index + 1][2];
x0 = waterAir[index][1];
x1 = waterAir[index + 1][1];
xd = x0 + (((tempVec[2] - z0) / (z1 - z0))*(x1 - x0));
//cout << "Great success dude! xd=\t" << xd << "\n";
break;
}
}
if (xd == 0)
{
double minimum = 25;
double minIndex = 1;
for (int index = 0; index < (lengthInterface - 1); index++)
{
if (abs(tempVec[2] - waterAir[index + 1][2])<minimum)
{
minimum = abs(tempVec[2] - waterAir[index + 1][2]);
minIndex = index + 1;
}
}
xd = waterAir[minIndex][1];
//cout << "Big fail dude! xd=\t" << xd << "\n";
}
if (tempVec[1] > xd)
{
fiber[rI][cI][tI] = 0;
water[rI][cI][tI] = 1;
air[rI][cI][tI] = 0;
}
else
{
fiber[rI][cI][tI] = 0;
water[rI][cI][tI] = 0;
air[rI][cI][tI] = 1;
}
}

}
}
}// end sharpening
```

A.5 3D simulation

```
#define _USE_MATH_DEFINES

// User defines
// Mathematics stuff
//#define _USE_MATH_DEFINES
//#define WIN32_LEAN_AND_MEAN
// Include headers
#include <SDKDDKVer.h>
#include <windows.h>
#include <stdlib.h>
#include <malloc.h>
#include <memory.h>
#include <tchar.h>
```

```
#include <iostream>
#include <string>
#include <cmath>
//#include "engine.h"
#include <fstream>
#include<direct.h>
#include <sstream>
#include <stdio.h>
#include "math.h"
#include<vector>
#include<time.h>



// Constants
// number Of pixels
static const int  nuOfEle = 150;
static const int  timeTotal = 150;
static const int  xDimZoom = 500;
static const int  yDimZoom = 500;
static const int heightZoom = 1;
static const int lengthZoom = 1;
// radius of sphere
static const int cubeDim = 1;
// step size
//double step, tTot1step1, tTot1step2, tTot2step1, tTot2step2, tStep1, tStep2, tTot1step3, tTot2step3, tStep3;
double step, tTot, tStep, stepZoom, tStepZoom, tTotZoom;
// x and y indices
double x, y;
// row and column indices
int rI, cI, zI;


// Standard stuff
using std::string;
using namespace std;
using std::vector;


// Declaration of functions
// Save file in folder
void saveVecDirectory(vector<double>, string);
void saveFileDirectory(vector<vector<vector<vector<double>>>>, int, string);
void cube(vector<vector<vector<double>>>&, vector<vector<vector<double>>>&, int, int, int,
vector<vector<double>>&, vector<vector<double>>&, vector<double>&, int, int, int);
void laplace(vector<vector<vector<double>>>&, vector<vector<vector<double>>>&, double,
vector<vector<double>>&, vector<vector<double>>&, vector<double>&, int, int, int);
void eulerForward(vector<vector<vector<vector<double>>>>&, int, double, double, double,
vector<vector<double>>&, vector<vector<double>>&, vector<double>&, int, int, int);
void sharpening(vector<vector<vector<vector<double>>>>&, vector<vector<vector<vector<double>>>>&, vector<vector<vector<vector<double>>>>&,
int, vector<vector<double>>&, int);
void saveFileDirectory2d(vector<vector<vector<double>>>, int, string, int, int, int);
void cube2d(vector<vector<double>>&, vector<vector<double>>&, int, int, int, int, int, int, int);
void laplace2d(vector<vector<double>>&, vector<vector<double>>&, double, int, int, int, int);
void eulerForward2d(vector<vector<vector<double>>>&, int, double, double, double, int, int, int, int);



struct tm newtime;
__time32_t aclock;



// MAIN PROGRAM
int main()
{
// Working directory
string run;
std::ostringstream eleStrFib;
eleStrFib << nuOfEle;
std::string eleStringFib = eleStrFib.str();
std::ostringstream timeStrFib;
timeStrFib << timeTotal;
std::string timeStringFib = timeStrFib.str();
run = ".//20150514ThreeDimension" + eleStringFib + "Time" + timeStringFib + "tTot150//";
CreateDirectory(run.c_str(), NULL);
// Fiber directory
string fiberStr = run + "fiber//";
CreateDirectory(fiberStr.c_str(), NULL);
string fiberDiffStr = fiberStr + "Diffused//";
CreateDirectory(fiberDiffStr.c_str(), NULL);
```

```
// Water directory
string waterStr = run + "water//";
CreateDirectory(waterStr.c_str(), NULL);
string waterDiffStr = waterStr + "Diffused//";
CreateDirectory(waterDiffStr.c_str(), NULL);
// Air directory
string airStr = run + "air//";
CreateDirectory(airStr.c_str(), NULL);
string airDiffStr = airStr + "Diffused//";
CreateDirectory(airDiffStr.c_str(), NULL);



cout << "\tHello there!\n\n\tNow starts the program!\n\n----------------------------------\n\n";
// current date/time based on current system


//Time stuff
char buffer[32];
errno_t errNum;
_time32(&aclock);   // Get time in seconds.
_localtime32_s(&newtime, &aclock);   // Convert time to struct tm form.

// Print local time as a string.

errNum = asctime_s(buffer, 32, &newtime);
if (errNum)
{
printf("Error code: %d", (int)errNum);
return 1;
}
printf("Current date and time: %s\n\n", buffer);


// Projection triangle stuff in 2D
vector<vector<double>>XplaneZoom;
vector<vector<double>>YplaneZoom;
XplaneZoom.resize(yDimZoom + 1);
YplaneZoom.resize(yDimZoom + 1);
for (int rI = 0; rI < (yDimZoom + 1); rI++)
{
XplaneZoom[rI].resize(xDimZoom + 1);
YplaneZoom[rI].resize(xDimZoom + 1);
}
//Zoomed in stuff

//Step length
stepZoom = ((lengthZoom) / ((double)xDimZoom));
std::ofstream outputstepZoom(run + "stepZoom.txt");
outputstepZoom << stepZoom;
outputstepZoom.close();
// Assignvalues to mesh
//Save matrices in files
std::ofstream outputXZoom(run + "XplaneZoom.txt");
std::ofstream outputYZoom(run + "YplaneZoom.txt");
for (rI = 0, y = 1; rI < (yDimZoom + 1); rI++, y -= (stepZoom))
{
for (cI = 0, x = 0; cI < (xDimZoom + 1); cI++, x += (stepZoom))
{
XplaneZoom[rI][cI] = x;
YplaneZoom[rI][cI] = y;
outputXZoom << XplaneZoom[rI][cI] << " ";
outputYZoom << YplaneZoom[rI][cI] << " ";
}
outputXZoom << "\n";
outputYZoom << "\n";
}
// Close files
outputXZoom.close();
outputYZoom.close();


cout << "Second print\n";

vector<vector<vector<double>>> fiberZoom;
vector<vector<vector<double>>> waterZoom;
vector<vector<vector<double>>> airZoom;
fiberZoom.resize(yDimZoom + 1);
waterZoom.resize(yDimZoom + 1);
airZoom.resize(yDimZoom + 1);
for (int rI = 0; rI < (yDimZoom + 1); rI++)
{
```

```
fiberZoom[rI].resize(xDimZoom + 1);
waterZoom[rI].resize(xDimZoom + 1);
airZoom[rI].resize(xDimZoom + 1);
for (int cI = 0; cI < (xDimZoom + 1); cI++)
{
fiberZoom[rI][cI].resize(2);
waterZoom[rI][cI].resize(2);
airZoom[rI][cI].resize(2);
}
}

cout << "Third print\n";

for (int rI = 0; rI < (yDimZoom + 1); rI++)
{
for (int cI = 0; cI < (xDimZoom + 1); cI++)
{
if (XplaneZoom[rI][cI] <= 0.5)
{
fiberZoom[rI][cI][0] = 1;
waterZoom[rI][cI][0] = 0;
airZoom[rI][cI][0] = 0;
}
if (XplaneZoom[rI][cI] > 0.5)
{
if (YplaneZoom[rI][cI] <= ((3 - (4 * XplaneZoom[rI][cI])) / (2)))
{
fiberZoom[rI][cI][0] = 0;
waterZoom[rI][cI][0] = 1;
airZoom[rI][cI][0] = 0;
}
if (YplaneZoom[rI][cI] > ((3 - (4 * XplaneZoom[rI][cI])) / (2)))
{
fiberZoom[rI][cI][0] = 0;
waterZoom[rI][cI][0] = 0;
airZoom[rI][cI][0] = 1;
}
}
}
}

cout << "Fourth print\n";


saveFileDirectory2d(fiberZoom, 0, fiberDiffStr, 0, xDimZoom, yDimZoom);
saveFileDirectory2d(waterZoom, 0, waterDiffStr, 0, xDimZoom, yDimZoom);
saveFileDirectory2d(airZoom, 0, airDiffStr, 0, xDimZoom, yDimZoom);



// Define timeSteps
tStepZoom = ((stepZoom*stepZoom) / (10));
tTotZoom = 3500 * tStepZoom;
// Save all the variables above
// save tStep
std::ofstream outputtStep1(run + "tStepZoom.txt");
outputtStep1 << tStepZoom;
outputtStep1.close();
// Save tTot
std::ofstream outputtTot2step3(run + "tTotZoom.txt");
outputtTot2step3 << tTotZoom;
outputtTot2step3.close();


double minEleX, minEleXSigned, minEleY, minEleYSigned, diffX, diffY, minDiffX, minDiffY;
minDiffX = 25;
minDiffY = 25;
for (int index = 0; index < (xDimZoom + 1); index++)
{
diffX = XplaneZoom[1][index] - 0.5;
//cout << "Difference X:\t" << diffX << "\n";
if (abs(diffX) < minDiffX)
{
minEleX = abs(XplaneZoom[1][index]);
minEleXSigned = XplaneZoom[1][index];
minDiffX = abs(diffX);
//cout << "Minimal element:\t" << minEleXSigned << "\n";
}
}
for (int index = 0; index < (yDimZoom + 1); index++)
{
diffY = YplaneZoom[index][1] - 0.5;
//cout << "Difference Y:\t" << diffY << "\n";
```

```
if (abs(diffY) < minDiffY)
{
minEleY = abs(YplaneZoom[index][1]);
minEleYSigned = YplaneZoom[index][1];
minDiffY = abs(diffY);
//cout << "Minimal element:\t" << minEleYSigned << "\n";
}
}

cout << "Minimal value X:\t" << minEleXSigned << "\n";
cout << "Minimal value Y:\t" << minEleYSigned << "\n";


// Number of edge elements
int fibWat = 0;
int fibAir = 0;
int watAir = 0;



for (int rI = 1; rI < (yDimZoom + 1); rI++)
{
for (int cI = 0; cI < (xDimZoom + 1); cI++)
{
// Interface fiber air
if (XplaneZoom[rI][cI] == minEleXSigned && YplaneZoom[rI][cI] >= minEleYSigned && YplaneZoom[rI][cI] <= 0.7)
{
fibAir++;
}
// Interface fiber water
if (XplaneZoom[rI][cI] == minEleXSigned && YplaneZoom[rI][cI] <= minEleYSigned && YplaneZoom[rI][cI] > 0.2)
{
fibWat++;
}
// Interface water air
if (YplaneZoom[rI][cI] <= ((3 - (4 * XplaneZoom[rI][cI])) / (2))
&& YplaneZoom[rI - 1][cI] > ((3 - (4 * XplaneZoom[rI][cI])) / (2))
&& XplaneZoom[rI][cI] >= minEleXSigned && XplaneZoom[rI][cI] <= 0.7)
{
watAir++;
}
}
}


std::ofstream outputtStepfibWat(run + "fibWatInt.txt");
outputtStepfibWat << fibWat;
outputtStepfibWat.close();
// Save tTot
std::ofstream outputtTot2stepfibAir(run + "fibAirInt.txt");
outputtTot2stepfibAir << fibAir;
outputtTot2stepfibAir.close();
std::ofstream outputtTot2stepwatAir(run + "watAirInt.txt");
outputtTot2stepwatAir << watAir;
outputtTot2stepwatAir.close();


vector<vector<double>>fibWatVec((fibWat), vector<double>(2));
vector<vector<double>>fibAirVec((fibAir), vector<double>(2));
vector<vector<double>>watAirVec((watAir), vector<double>(2));
int indexFibWat = 0;
int indexFibAir = 0;
int indexWatAir = 0;


for (int rI = 1; rI < (yDimZoom + 1); rI++)
{
for (int cI = 0; cI < (xDimZoom + 1); cI++)
{
// Interface fiber air
if (XplaneZoom[rI][cI] == minEleXSigned && YplaneZoom[rI][cI] >= minEleYSigned && YplaneZoom[rI][cI] <= 0.7)
{
fibAirVec[indexFibAir][0] = rI;
fibAirVec[indexFibAir][1] = cI;
indexFibAir++;
}
// Interface fiber water
if (XplaneZoom[rI][cI] == minEleXSigned && YplaneZoom[rI][cI] <= minEleYSigned &&  YplaneZoom[rI][cI] > 0.2)
{
fibWatVec[indexFibWat][0] = rI;
fibWatVec[indexFibWat][1] = cI;
indexFibWat++;
}
```

```
// Interface water air
if (YplaneZoom[rI][cI] <= ((3 - (4 * XplaneZoom[rI][cI])) / (2))
&& YplaneZoom[rI - 1][cI] > ((3 - (4 * XplaneZoom[rI][cI])) / (2))
&& XplaneZoom[rI][cI] >= minEleXSigned && XplaneZoom[rI][cI] <= 0.7)
{
watAirVec[indexWatAir][0] = rI;
watAirVec[indexWatAir][1] = cI;
indexWatAir++;
}
}
}

vector<vector<double>>fibWatIntVec((fibWat), vector<double>(3));
vector<vector<double>>fibAirIntVec((fibAir), vector<double>(3));
vector<vector<double>>watAirIntVec((watAir + 1), vector<double>(3));

cout << "Now starts short Euler\n";
_time32(&aclock);   // Get time in seconds.
_localtime32_s(&newtime, &aclock);   // Convert time to struct tm form.

// Print local time as a string.

errNum = asctime_s(buffer, 32, &newtime);
if (errNum)
{
printf("Error code: %d", (int)errNum);
return 1;
}
printf("Current date and time: %s\n\n", buffer);

eulerForward2d(fiberZoom, 1, tTotZoom, stepZoom, tStepZoom, 1, 1, xDimZoom, yDimZoom);
eulerForward2d(waterZoom, 1, tTotZoom, stepZoom, tStepZoom, 1, 1, xDimZoom, yDimZoom);
eulerForward2d(airZoom, 1, tTotZoom, stepZoom, tStepZoom, 1, 1, xDimZoom, yDimZoom);

for (int index = 0; index < (fibWat - 1); index++)
{
fibWatIntVec[index][0] = fiberZoom[fibWatVec[index][0]][fibWatVec[index][1]][1];
fibWatIntVec[index][1] = waterZoom[fibWatVec[index][0]][fibWatVec[index][1]][1];
fibWatIntVec[index][2] = airZoom[fibWatVec[index][0]][fibWatVec[index][1]][1];
}

for (int index = 0; index < (fibAir - 1); index++)
{
fibAirIntVec[index][0] = fiberZoom[fibAirVec[index][0]][fibAirVec[index][1]][1];
fibAirIntVec[index][1] = waterZoom[fibAirVec[index][0]][fibAirVec[index][1]][1];
fibAirIntVec[index][2] = airZoom[fibAirVec[index][0]][fibAirVec[index][1]][1];
}

for (int index = 0; index < (watAir - 1); index++)
{
watAirIntVec[index][0] = ((fiberZoom[watAirVec[index][0]][watAirVec[index][1]][1] + fiberZoom[watAirVec[index][0] + 1][watAirVec[index][1]][1]) / (2));
watAirIntVec[index][1] = ((waterZoom[watAirVec[index][0]][watAirVec[index][1]][1] + waterZoom[watAirVec[index][0] + 1][watAirVec[index][1]][1]) / (2));
watAirIntVec[index][2] = ((airZoom[watAirVec[index][0]][watAirVec[index][1]][1] + airZoom[watAirVec[index][0] + 1][watAirVec[index][1]][1]) / (2));
}

fibWatIntVec[0][0] = fibWatIntVec[1][0];
fibWatIntVec[0][1] = fibWatIntVec[1][1];
fibWatIntVec[0][2] = fibWatIntVec[1][2];
fibAirIntVec[0][0] = fibAirIntVec[1][0];
fibAirIntVec[0][1] = fibAirIntVec[1][1];
fibAirIntVec[0][2] = fibAirIntVec[1][2];

cout << "Finished! Let us organize the vectors!\n";

// Organize them based on fiber column
// Fiber Water
for (int i = 0; i < fibWat; ++i)
for (int j = 0; j < i; ++j)
if (fibWatIntVec[j][0] > fibWatIntVec[i][0])
std::swap(fibWatIntVec[i], fibWatIntVec[j]);
// Fiber Air
for (int i = 0; i < fibAir; ++i)
for (int j = 0; j < i; ++j)
if (fibAirIntVec[j][0] > fibAirIntVec[i][0])
std::swap(fibAirIntVec[i], fibAirIntVec[j]);
// Water Air
for (int i = 0; i < watAir; ++i)
for (int j = 0; j < i; ++j)
if (watAirIntVec[j][0] > watAirIntVec[i][0])
std::swap(watAirIntVec[i], watAirIntVec[j]);


cout << "Save 'em!\n";
```

```
std::ofstream outputFibWat(run + "fibWat.txt");
for (int i = 0; i < (fibWat - 1); i++)
{
outputFibWat << fibWatIntVec[i][0] << "\t" << fibWatIntVec[i][1] << "\t" << fibWatIntVec[i][2] << "\n";
}
outputFibWat.close();
std::ofstream outputFibAir(run + "fibAir.txt");
for (int i = 0; i < (fibAir - 1); i++)
{
outputFibAir << fibAirIntVec[i][0] << "\t" << fibAirIntVec[i][1] << "\t" << fibAirIntVec[i][2] << "\n";
}
outputFibAir.close();


double aFib = watAirIntVec[watAir - 1][0];
double aAir = watAirIntVec[watAir - 1][2];
double bFib = watAirIntVec[watAir - 2][0];
double bAir = watAirIntVec[watAir - 2][2];
double airHej = aAir + ((0.5 - aFib) / (bFib - aFib))*(bAir - aAir);
double fibHej = 0.5;
double watHej = 1 - fibHej - airHej;

watAirIntVec[watAir][0] = fibHej;
watAirIntVec[watAir][1] = watHej;
watAirIntVec[watAir][2] = airHej;

// Water Air
for (int i = 0; i < watAir; ++i)
for (int j = 0; j < i; ++j)
if (watAirIntVec[j][0] > watAirIntVec[i][0])
std::swap(watAirIntVec[i], watAirIntVec[j]);

std::ofstream outputwatAir(run + "watAir.txt");
for (int i = 0; i < watAir; i++)
{
outputwatAir << watAirIntVec[i][0] << "\t" << watAirIntVec[i][1] << "\t" << watAirIntVec[i][2] << "\n";
}
outputwatAir.close();



cout << "Okay! Now a projection triangle is formed, let us move on!\n";

_time32(&aclock);   // Get time in seconds.
_localtime32_s(&newtime, &aclock);   // Convert time to struct tm form.

// Print local time as a string.

errNum = asctime_s(buffer, 32, &newtime);
if (errNum)
{
printf("Error code: %d", (int)errNum);
return 1;
}
printf("Current date and time: %s\n\n", buffer);









// Real simulation stuff in 3D


//Save number of element
std::ofstream outputnuOfEle(run + "nuOfEle.txt");
outputnuOfEle << nuOfEle;
outputnuOfEle.close();
//Save radius
std::ofstream outputRadius(run + "cubeDim.txt");
outputRadius << cubeDim;
outputRadius.close();
// Define mesh
vector<vector<double>>Xplane;
vector<vector<double>>Yplane;
```

```
Xplane.resize(nuOfEle + 1);
Yplane.resize(nuOfEle + 1);
for (int rI = 0; rI < (nuOfEle + 1); rI++)
{
Xplane[rI].resize(nuOfEle + 1);
Yplane[rI].resize(nuOfEle + 1);
}
// Zvec
vector<double>Zvec;
Zvec.resize(nuOfEle + 1);
//Step length
step = ((cubeDim) / ((double)nuOfEle));
// Assignvalues to mesh
//Save matrices in files
std::ofstream outputX(run + "Xplane.txt");
std::ofstream outputY(run + "Yplane.txt");
for (rI = 0, y = cubeDim; rI < (nuOfEle + 1); rI++, y += (-step))
{
for (cI = 0, x = 0; cI < (nuOfEle + 1); cI++, x += (step))
{
Xplane[rI][cI] = x;
Yplane[rI][cI] = y;
outputX << Xplane[rI][cI] << " ";
outputY << Yplane[rI][cI] << " ";
}
outputX << "\n";
outputY << "\n";
}
// Close files
outputX.close();
outputY.close();
for (int zI = 0; zI < (nuOfEle + 1); zI++)
{
Zvec[zI] = Xplane[1][zI];
}
std::string fileName = run + "z.txt";
saveVecDirectory(Zvec, fileName);

// Define timeSteps
tStep = ((step*step) / (10));
tTot = 100 * tStep;
// Save all the variables above
// save tStep
std::ofstream outputtStep12(run + "tStep.txt");
outputtStep12 << tStep;
outputtStep12.close();
// Save tTot
std::ofstream outputtTot2step32(run + "tTot.txt");
outputtTot2step32 << tTot;
outputtTot2step32.close();
// REAL PROGRAM STARTS



//3D cube
// Zero time steps
vector<vector<vector<vector<double>>>>fiber((nuOfEle + 1), vector<vector<vector<double>>>((nuOfEle + 1),
  vector<vector<double>>((nuOfEle + 1), vector<double>((2)))));

vector<vector<vector<vector<double>>>>water((nuOfEle + 1), vector<vector<vector<double>>>((nuOfEle + 1),
  vector<vector<double>>((nuOfEle + 1), vector<double>((2)))));

vector<vector<vector<vector<double>>>>air((nuOfEle + 1), vector<vector<vector<double>>>((nuOfEle + 1),
 vector<vector<double>>((nuOfEle + 1), vector<double>((2)))));



double leftWall, rightWall, cylinderRadius, cylinderWall;

// Assign values sphere

for (int zI = 0; zI < (nuOfEle + 1); zI++)
{
for (int rI = 0; rI < (nuOfEle + 1); rI++)
{
for (int cI = 0; cI < (nuOfEle + 1); cI++)
{
double leftWall = (0.1*Yplane[rI][cI] * Yplane[rI][cI]) + 0.1;
double rightWall = (-0.1*Yplane[rI][cI] * Yplane[rI][cI]) + 0.3;
double cylinderRadius = (-0.4)*(Zvec[zI] * (Zvec[zI] - 1)) + 0.2;
double cylinderWall = ((Xplane[rI][cI] - 0.69) * (Xplane[rI][cI] - 0.69)) + ((Yplane[rI][cI] - 0.5)*(Yplane[rI][cI] - 0.5));
//cout << "Radius:\t" << cylinderRadius << "\t,Z-value:\t" << Zvec[zI]<<"\n";
```

```
if (Xplane[rI][cI] < leftWall)
{
fiber[rI][cI][zI][0] = 1;
air[rI][cI][zI][0] = 0;
water[rI][cI][zI][0] = 0;
}
if (Xplane[rI][cI] > rightWall && (cylinderRadius*cylinderRadius) < cylinderWall)
{
if (((Yplane[rI][cI] - 0.5)*(Yplane[rI][cI] - 0.5)) + ((Zvec[zI] - 0.5)*(Zvec[zI] - 0.5))<0.02 &&
Xplane[rI][cI]<0.7)
{
fiber[rI][cI][zI][0] = 0;
air[rI][cI][zI][0] = 1;
water[rI][cI][zI][0] = 0;
}
else
{
fiber[rI][cI][zI][0] = 1;
air[rI][cI][zI][0] = 0;
water[rI][cI][zI][0] = 0;
}
}
if (fiber[rI][cI][zI][0] == 0)
{
if (Zvec[zI]<0.25)
{
fiber[rI][cI][zI][0] = 0;
air[rI][cI][zI][0] = 0;
water[rI][cI][zI][0] = 1;
}
else
{
fiber[rI][cI][zI][0] = 0;
air[rI][cI][zI][0] = 1;
water[rI][cI][zI][0] = 0;
}
}

}//rI
}//cI
}//zI


// Save initial fiber
string folderFib = fiberStr + ".//timefolder";
CreateDirectory(folderFib.c_str(), NULL);

std::ostringstream iniFib;
iniFib << 0;
std::string theNumberStringFib = iniFib.str();
string tempStrFib = folderFib + "//t" + theNumberStringFib;
saveFileDirectory(fiber, 0, tempStrFib);


// Save initial water
string folderWat = waterStr + ".//timefolder";
CreateDirectory(folderWat.c_str(), NULL);

std::ostringstream iniWat;
iniWat << 0;
std::string theNumberStringWat = iniWat.str();
string tempStrWat = folderWat + "//t" + theNumberStringWat;
saveFileDirectory(water, 0, tempStrWat);

// Save initial air
string folderAir = airStr + ".//timefolder";
CreateDirectory(folderAir.c_str(), NULL);

std::ostringstream iniAir;
iniAir << 0;
std::string theNumberStringAir = iniAir.str();
string tempStrAir = folderAir + "//t" + theNumberStringAir;
saveFileDirectory(air, 0, tempStrAir);


// TIME LOOP
cout << "\nTime step\t" << 1 << "\tof\t" << timeTotal << "\n\n";

for (int tI = 1; tI < (timeTotal + 1); tI++)
{
// EULER FORWARD!!!
// FIBER
eulerForward(fiber, 1, tTot, step, tStep, Xplane, Yplane, Zvec, 1, 1, 1);
```

```
// Sharpen newly filled vector
//sharpening(fiber, 1, (1 / 2));
// Water
cout << "\t\t\tFiber Ready!\n";
eulerForward(water, 1, tTot, step, tStep, Xplane, Yplane, Zvec, 1, 1, 1);
// Sharpen newly filled vector
//sharpening(water, 1, (1/4));
// Air
cout << "\t\t\tWater Ready!\n";
eulerForward(air, 1, tTot, step, tStep, Xplane, Yplane, Zvec, 1, 1, 1);
// Sharpen newly filled vector
//sharpening(air, 1, (1 / 4));
cout << "\t\t\tAir Ready!\n";

std::ostringstream ostrFib2;
ostrFib2 << tI;
std::string theNumberStringFib2 = ostrFib2.str();
string tempStrFib2 = fiberDiffStr + "//t" + theNumberStringFib2;
saveFileDirectory(fiber, 1, tempStrFib2);
// Save stuff
std::ostringstream ostrWat2;
ostrWat2 << tI;
std::string theNumberStringWat2 = ostrWat2.str();
string tempStrWat2 = waterDiffStr + "//t" + theNumberStringWat2;
saveFileDirectory(water, 1, tempStrWat2);
// Save stuff
std::ostringstream ostrAir2;
ostrAir2 << tI;
std::string theNumberStringAir2 = ostrAir2.str();
string tempStrAir2 = airDiffStr + "//t" + theNumberStringAir2;
saveFileDirectory(air, 1, tempStrAir2);
sharpening(fiber, water, air, 1, watAirIntVec, watAir);
//sharpening(fiber, water, air);
// Save stuff
std::ostringstream ostrFib;
ostrFib << tI;
std::string theNumberStringFib = ostrFib.str();
string tempStrFib = folderFib + "//t" + theNumberStringFib;
saveFileDirectory(fiber, 1, tempStrFib);
// Save stuff
std::ostringstream ostrWat;
ostrWat << tI;
std::string theNumberStringWat = ostrWat.str();
string tempStrWat = folderWat + "//t" + theNumberStringWat;
saveFileDirectory(water, 1, tempStrWat);
// Save stuff
std::ostringstream ostrAir;
ostrAir << tI;
std::string theNumberStringAir = ostrAir.str();
string tempStrAir = folderAir + "//t" + theNumberStringAir;
saveFileDirectory(air, 1, tempStrAir);

// Change values of first vector
for (int zI = 0; zI < (nuOfEle + 1); zI++)
{
for (int rI = 0; rI < (nuOfEle + 1); rI++)
{
for (int cI = 0; cI < (nuOfEle + 1); cI++)
{
fiber[rI][cI][zI][0] = fiber[rI][cI][zI][1];
water[rI][cI][zI][0] = water[rI][cI][zI][1];
air[rI][cI][zI][0] = air[rI][cI][zI][1];
}
}
}
cout << "\nTime step\t" << tI << "\tof\t" << timeTotal << "\n\n";
_time32(&aclock);   // Get time in seconds.
_localtime32_s(&newtime, &aclock);   // Convert time to struct tm form.

// Print local time as a string.

errNum = asctime_s(buffer, 32, &newtime);
if (errNum)
{
printf("Error code: %d", (int)errNum);
return 1;
}
printf("Current date and time: %s\n\n", buffer);
}

// Program worked!
cout << "Program worked!\n\n";
// end of main
```

```
return 0;
}// END MAIN




// NEW FUNCTIONS: YIIIIIIIIIIIHAAAAAAAAAAAAAAA!!!
// FUNCTION SAVE VECTOR DIRECTORY
void saveVecDirectory(vector<double> VEC, string directory)
{
std::string filefolder = directory;
std::ofstream output(filefolder);
int length = VEC.size();
int zI;
//std::ofstream output(directory.c_str());
for (zI = 0; zI < length; zI++)
{
/*std::ostringstream ostr;
ostr << zI;
std::string theNumberString = ostr.str();
std::string end = ".txt";
std::string fileFolder = directory + "//" + theNumberString + end;*/
output << VEC[zI] << "\n";
}
output.close();
}



// FUNCTION SAVE FILE DIRECTORY
void saveFileDirectory(vector<vector<vector<vector<double>>>> ARRAY, int tI, string directory)
{
//std::string folder = ".//fib0";
CreateDirectory(directory.c_str(), NULL);
for (int zI = 0; zI < (nuOfEle + 1); zI++)
{
std::ostringstream ostr;
ostr << zI;
std::string theNumberString = ostr.str();
std::string end = ".txt";
std::string fileFolder = directory + "//" + theNumberString + end;
std::ofstream output(fileFolder);

for (int rI = 0; rI < (nuOfEle + 1); rI++)
{
for (int cI = 0; cI < (nuOfEle + 1); cI++)
{
output << ARRAY[rI][cI][zI][tI] << " ";
}
output << "\n";
}
output.close();
}
}


// Void cube
void cube(vector<vector<vector<double>>>& cube, vector<vector<vector<double>>>& lpMat, int zI, int rI, int cI,
vector<vector<double>>& Xplane, vector<vector<double>>& Yplane, vector<double>& Zvec, int neumannX, int neumannY, int neumannZ)
{

int xTemp, yTemp, zTemp;
for (int z = 0; z < 3; z++)
{
for (int y = 0; y < 3; y++)
{
for (int x = 0; x < 3; x++)
{
xTemp = cI - 1 + x;
yTemp = rI - 1 + y;
zTemp = zI - 1 + z;
if (zTemp == (nuOfEle + 1))
{
if (neumannZ == 1)
{
zTemp = nuOfEle;
}
if (neumannZ == 0)
```

```
{
zTemp = 0;
}
}
if (zTemp == -1)
{
if (neumannZ == 1)
{
zTemp = 0;
}
if (neumannZ == 0)
{
zTemp = nuOfEle;
}
}
if (yTemp == (nuOfEle + 1))
{
if (neumannY == 1)
{
yTemp = nuOfEle;
}
if (neumannY == 0)
{
yTemp = 0;
}
}
if (yTemp == -1)
{
if (neumannY == 1)
{
yTemp = 0;
}
if (neumannY == 0)
{
yTemp = nuOfEle;
}
}
if (xTemp == (nuOfEle + 1))
{
if (neumannX == 1)
{
xTemp = nuOfEle;
}
if (neumannX == 0)
{
xTemp = 0;
}
}
if (xTemp == -1)
{
if (neumannX == 1)
{
xTemp = 0;
}
if (neumannX == 0)
{
xTemp = nuOfEle;
}
}
cube[y][x][z] = lpMat[yTemp][xTemp][zTemp];
}//end x
}//end y
}//end z

}// end cube function

// Void function Laplace
void laplace(vector<vector<vector<double>>>& lpMat, vector<vector<vector<double>>>& inputMat, double step,
vector<vector<double>>& Xplane, vector<vector<double>>& Yplane, vector<double>& Zvec, int neumannX, int neumannY, int neumannZ)
{
// Allocate memory for the sums
double cornerSum, faceSum, edgeSum, nom;
double radius = 1;
// Create a cube 3x3x3
//vector<vector<vector<double>>>cubeTemp((3), vector<vector<double>>((3), vector<double>(3)));

// Create a cube 3x3x3
vector<vector<vector<double>>>cubeTemp;
cubeTemp.resize(3);
for (int index = 0; index < 3; index++)
{
cubeTemp[index].resize(3);
for (int hej = 0; hej < 3; hej++)
```

```
{
cubeTemp[index][hej].resize(3);
}
}

//LOOPS
for (int zI = 0; zI < (nuOfEle + 1); zI++)
{
for (int rI = 0; rI < (nuOfEle + 1); rI++)
{
for (int cI = 0; cI < (nuOfEle + 1); cI++)
{

cube(cubeTemp, inputMat, zI, rI, cI, Xplane, Yplane, Zvec, neumannX, neumannY, neumannZ);
// Calculate stuff
// FACES
faceSum = cubeTemp[2][1][1] + cubeTemp[0][1][1] + //Forward +Backward
cubeTemp[1][1][2] + cubeTemp[1][1][0] + //Up + Down
cubeTemp[1][0][1] + cubeTemp[1][2][1]; //Left + Right
// EDGES
edgeSum = cubeTemp[2][1][2] + cubeTemp[2][1][0] + //ForwardUp +ForwardDown
cubeTemp[2][0][1] + cubeTemp[2][2][1] + //ForwardLeft +ForwardRight
cubeTemp[0][1][2] + cubeTemp[0][1][0] + //BackwardUp + BackwardDown
cubeTemp[0][0][1] + cubeTemp[0][2][1] + //BackwardLeft + BackwardRight
cubeTemp[1][0][2] + cubeTemp[1][0][0] + //LeftUp + LeftDown
cubeTemp[1][2][2] + cubeTemp[1][2][0];//RightUp + RightDown
cornerSum = cubeTemp[2][2][2] + cubeTemp[2][2][0] + //ForwardRightUp +ForwardRightDown
cubeTemp[2][0][2] + cubeTemp[2][0][0] + //ForwardLeftUp +ForwardLeftDown
cubeTemp[0][2][2] + cubeTemp[0][2][0] + //BackwardRightUp +BackwardRightDown
cubeTemp[0][0][2] + cubeTemp[0][0][0]; //BackwardLeftUp +BackwardLeftDown
nom = ((3 * faceSum) / (13)) + ((3 * edgeSum) / (26)) + ((cornerSum) / (13)) - ((44 * inputMat[rI][cI][zI]) / (13));// Stencil 27
//nom = ((faceSum) / (3)) + ((edgeSum) / (6)) - (4 * inputMat[rI][cI][zI]);
//nom = faceSum - 6*inputMat[rI][cI][zI];// Stencil 7
//cout << "\t\Face:\t" << faceSum << "\n\tEdge:\t" << edgeSum << "\n\tCorner:\t" << cornerSum;
//cout << "\n\tNom:" << nom << "\n------------------\n\n";

lpMat[rI][cI][zI] = ((nom) / (step*step));
}
}
}
}


// Euler Forward!
void eulerForward(vector<vector<vector<vector<double>>>>& afterMat, int timeIndex, double tTot, double step, double tStep,
vector<vector<double>>& Xplane, vector<vector<double>>& Yplane, vector<double>& Zvec, int neumannX, int neumannY, int neumannZ)
{

//Create temporary vector beforeMat
vector<vector<vector<double>>> beforeMat;
beforeMat.resize(nuOfEle + 1);
for (int rI = 0; rI < (nuOfEle + 1); rI++) {
beforeMat[rI].resize(nuOfEle + 1);
for (int cI = 0; cI < (nuOfEle + 1); cI++)
{
beforeMat[rI][cI].resize(nuOfEle + 1);
}

}
// Initialiaze beforeMat to initialMat
for (int zI = 0; zI < (nuOfEle + 1); zI++)
{
for (int rI = 0; rI < (nuOfEle + 1); rI++)
{
for (int cI = 0; cI < (nuOfEle + 1); cI++)
{
beforeMat[rI][cI][zI] = afterMat[rI][cI][zI][timeIndex - 1];
}
}
}

// Euler loops: one for time, one for each
//spatial dimension
for (double tI = 0; tI < (tTot + tStep); tI += tStep)
{
vector<vector<vector<double>>> lpTemp;
lpTemp.resize(nuOfEle + 1);
for (int rI = 0; rI < (nuOfEle + 1); rI++) {
lpTemp[rI].resize(nuOfEle + 1);
for (int cI = 0; cI < (nuOfEle + 1); cI++)
{
lpTemp[rI][cI].resize(nuOfEle + 1);
}
```

```
}
laplace(lpTemp, beforeMat, step, Xplane, Yplane, Zvec, neumannX, neumannY, neumannZ);
for (int zI = 0; zI < (nuOfEle + 1); zI++)
{
for (int rI = 0; rI < (nuOfEle + 1); rI++)
{
for (int cI = 0; cI < (nuOfEle + 1); cI++)
{
afterMat[rI][cI][zI][timeIndex] = beforeMat[rI][cI][zI] + tStep*lpTemp[rI][cI][zI];
beforeMat[rI][cI][zI] = afterMat[rI][cI][zI][timeIndex];
}
}
}
lpTemp.erase(lpTemp.begin(), lpTemp.end());
//lpTemp.shrink_to_fit();
lpTemp.clear();
}

beforeMat.erase(beforeMat.begin(), beforeMat.end());
//beforeMat.shrink_to_fit();
beforeMat.clear();

}// end Euler

// SHARPENING FUNCTION
// SHARPENING FUNCTION
//void sharpening(vector<vector<vector<vector<double>>>>& input, int tI, double threshold)
void sharpening(vector<vector<vector<vector<double>>>>& fiber, vector<vector<vector<vector<double>>>>& water, vector<vector<vector<vector<double>>>>& air,
 int tI,vector<vector<double>>& waterAir, int lengthInterface)
{
//double denom, watHat, airHat, thresHold;
double z0, z1, x0, x1, xd;
for (int rI = 0; rI < (nuOfEle + 1); rI++)
{
for (int cI = 0; cI < (nuOfEle + 1); cI++)
{
for (int zI = 0; zI < (nuOfEle + 1); zI++)
{
vector<double>tempVec(3);
tempVec[0] = fiber[rI][cI][zI][tI];
tempVec[1] = water[rI][cI][zI][tI];
tempVec[2] = air[rI][cI][zI][tI];
if (tempVec[0] > 0.5)
{
fiber[rI][cI][zI][tI] = 1;
water[rI][cI][zI][tI] = 0;
air[rI][cI][zI][tI] = 0;
}
else
{
z0 = 0;
z1 = 0;
x0 = 0;
x1 = 0;
xd = 0;
for (int index = 0; index < (lengthInterface - 1); index++)
{
if (tempVec[2] >= waterAir[index + 1][2] && tempVec[2] <= waterAir[index][2] && waterAir[index][2] != 0)
{
z0 = waterAir[index][2];
z1 = waterAir[index + 1][2];
x0 = waterAir[index][1];
x1 = waterAir[index + 1][1];
xd = x0 + (((tempVec[2] - z0) / (z1 - z0))*(x1 - x0));
//cout << "Great success dude! xd=\t" << xd << "\n";
break;
}
}
if (xd == 0)
{
double minimum = 25;
double minIndex = 1;
for (int index = 0; index < (lengthInterface - 1); index++)
{
if (abs(tempVec[2] - waterAir[index + 1][2])<minimum)
{
minimum = abs(tempVec[2] - waterAir[index + 1][2]);
minIndex = index + 1;
}
}
xd = waterAir[minIndex][1];
//cout << "Big fail dude! xd=\t" << xd << "\n";
```

```
}
if (tempVec[1] > xd)
{
fiber[rI][cI][zI][tI] = 0;
water[rI][cI][zI][tI] = 1;
air[rI][cI][zI][tI] = 0;
}
else
{
fiber[rI][cI][zI][tI] = 0;
water[rI][cI][zI][tI] = 0;
air[rI][cI][zI][tI] = 1;
}
}
}//zI
}//cI
}//rI
}// end sharpening



// FUNCTION SAVE FILE DIRECTORY
void saveFileDirectory2d(vector<vector<vector<double>>> ARRAY, int tI, string directory, int time, int xDim, int yDim)
{
//std::string folder = ".//fib0";
//CreateDirectory(directory.c_str(), NULL);
std::ostringstream ostr;
ostr << time;
std::string theNumberString = ostr.str();
std::string end = ".txt";
std::string fileFolder = directory + "//t" + theNumberString + end;
std::ofstream output(fileFolder);

for (int rI = 0; rI < (yDim + 1); rI++)
{
for (int cI = 0; cI < (xDim + 1); cI++)
{
output << ARRAY[rI][cI][tI] << " ";
}
output << "\n";
}
output.close();
}



// Void cube
void cube2d(vector<vector<double>>& cube, vector<vector<double>>& lpMat, int rI, int cI, int neumannX, int neumannY,
int xDim, int yDim)
{
int yTemp, xTemp;
for (int x = 0; x < 3; x++)
{
for (int y = 0; y < 3; y++)
{
xTemp = cI - 1 + x;
yTemp = rI - 1 + y;
if (yTemp == (yDim + 1))
{
if (neumannY == 1)
{
yTemp = yDim;
}
if (neumannY == 0)
{
yTemp = 0;
}
}
if (yTemp == -1)
{
if (neumannY == 1)
{
yTemp = 0;
}
if (neumannY == 0)
{
yTemp = yDim;
}
}
if (xTemp == (xDim + 1))
{
if (neumannX == 1)
```

```
{
xTemp = xDim;
}
if (neumannX == 0)
{
xTemp = 0;
}
}
if (xTemp == -1)
{
if (neumannX == 1)
{
xTemp = 0;
}
if (neumannX == 0)
{
xTemp = xDim;
}
}
cube[y][x] = lpMat[yTemp][xTemp];
}
}



}// end cube function

// Void function Laplace
void laplace2d(vector<vector<double>>& lpMat, vector<vector<double>>& inputMat, double step, int neumannX, int neumannY,
int xDim, int yDim)
{
// Allocate memory for the sums
double cornerSum, faceSum, nom;
// Create a cube 3x3x3
//vector<vector<vector<double>>>cubeTemp((3), vector<vector<double>>((3), vector<double>(3)));

// Create a cube 3x3x3
vector<vector<double>>cubeTemp;
cubeTemp.resize(3);
for (int index = 0; index < 3; index++)
{
cubeTemp[index].resize(3);
}

//LOOPS

for (int rI = 0; rI < (yDim + 1); rI++)
{
for (int cI = 0; cI < (xDim + 1); cI++)
{
cube2d(cubeTemp, inputMat, rI, cI, neumannX, neumannY, xDim, yDim);
// Calculate stuff
// FACES
faceSum = cubeTemp[1][2] + cubeTemp[1][0] + //Right + Left
cubeTemp[2][1] + cubeTemp[0][1]; // Forward + Backward
// EDGES
cornerSum = cubeTemp[2][2] + cubeTemp[2][0] + //ForwardRight + ForwardLeft
cubeTemp[0][2] + cubeTemp[0][0]; //BackwardRight + BackwardLeft
nom = ((2 * faceSum) / (3)) + ((cornerSum) / (6)) - ((10 * inputMat[rI][cI]) / (3));
// Laplace matrix
lpMat[rI][cI] = ((nom) / (step*step));
//cout <<"Laplace matrix:\t" <<lpMat[rI][cI]<<"\n";
}
}

}


// Euler Forward!
void eulerForward2d(vector<vector<vector<double>>>& afterMat, int timeIndex, double tTot, double step, double tStep, int neumannX, int neumannY,
int xDim, int yDim)
{

//Create temporary vector beforeMat
vector<vector<double>> beforeMat;
beforeMat.resize(yDim + 1);
for (int rI = 0; rI < (yDim + 1); rI++) {
beforeMat[rI].resize(xDim + 1);
}
// Initialiaze beforeMat to initialMat

for (int rI = 0; rI < (yDim + 1); rI++)
{
```

```
for (int cI = 0; cI < (xDim + 1); cI++)
{
beforeMat[rI][cI] = afterMat[rI][cI][timeIndex - 1];
}
}

// Euler loops: one for time, one for each
//spatial dimension
for (double tI = 0; tI < (tTot + tStep); tI += tStep)
{
vector<vector<double>> lpTemp;
lpTemp.resize(yDim + 1);
for (int rI = 0; rI < (yDim + 1); rI++) {
lpTemp[rI].resize(xDim + 1);
}
laplace2d(lpTemp, beforeMat, step, neumannX, neumannY, xDim, yDim);

for (int rI = 0; rI < (yDim + 1); rI++)
{
for (int cI = 0; cI < (xDim + 1); cI++)
{
afterMat[rI][cI][timeIndex] = beforeMat[rI][cI] + tStep*lpTemp[rI][cI];
beforeMat[rI][cI] = afterMat[rI][cI][timeIndex];
}
}
lpTemp.erase(lpTemp.begin(), lpTemp.end());
//lpTemp.shrink_to_fit();
lpTemp.clear();
}

beforeMat.erase(beforeMat.begin(), beforeMat.end());
//beforeMat.shrink_to_fit();
beforeMat.clear();
}// end Euler
```

B

Matlab code

Appendix B is divided into four different parts named ”Projection Triangle”, ”Plotting in $\mathbb{R}^2$”, ”Plotting in $\mathbb{R}^3$” and ”Test of convolution formula”. The first three scripts plots the generated output matrices from the c++ scripts in appendix A. The script in section ”Test of convolution formula” generates the graphs illustrated in section 4.1.2 on page 71.

B.1 Plotting of projection triangle

```
folder = './20150526CreepyLabyrinth600Ydim600Time300/';
fibWat = load([folder,'fibWat.txt']);
[r, c] = size(fibWat);
fibWat2 = zeros(r-1,c);
fibAir = load([folder,'fibAir.txt']);
[r1, c1] = size(fibAir);
fibAir2 = zeros(r-1,c);
watAir = load([folder,'watAir.txt']);
[r2, c2] = size(watAir);
watAir2 = zeros(r,c);
tTot = load([folder,'tTotZoom.txt']);
tStep = load([folder,'tStepZoom.txt']);
hej = tTot/tStep;
tTot2 = load([folder,'tTot.txt']);
tStep2 = load([folder,'tStep.txt']);
hej2 = tTot2/tStep;

aFib = watAir(r2,1);
aAir = watAir(r2,3);
bFib = watAir(r2-1,1);
bAir = watAir(r2-1,3);
airHej = aAir + ((0.5 - aFib) / (bFib - aFib))*(bAir - aAir);
fibHej = 0.5;
watHej = 1-fibHej-airHej;

%%
for index = 2:r
        fibWat2(index,1) = ((fibWat(index,1))/(sum(fibWat(index,:))));
        fibWat2(index,2) = ((fibWat(index,2))/(sum(fibWat(index,:))));
        fibWat2(index,3) = ((fibWat(index,3))/(sum(fibWat(index,:))));
end

for index = 2:r1

    fibAir2(index,1) = ((fibAir(index,1))/(sum(fibAir(index,:))));
    fibAir2(index,2) = ((fibAir(index,2))/(sum(fibAir(index,:))));
```

```
    fibAir2(index,3) = ((fibAir(index,3))/(sum(fibAir(index,:))));

end
for index = 2:r2
   % Air
   watAir2(index,1) = ((watAir(index,1))/(sum(watAir(index,:))));
   watAir2(index,2) = ((watAir(index,2))/(sum(watAir(index,:))));
   watAir2(index,3) = ((watAir(index,3))/(sum(watAir(index,:))));
end

[r5,c5]=size(fibAir2);

fibAir2(r5+1,1) = (1/2);
fibAir2(r5+1,2) = (1/8);
fibAir2(r5+1,3) = (3/8);


%% Figure 1

%watAir(r2+1,1) = fibHej;
%watAir(r2+1,2) = watHej;
%watAir(r2+1,3) = airHej;

a = [0.5; 0.5];
b = [0; (1/8)];
c = [0.5; (3/8)];

% %%
figure(1)
clf
plot3([0;0],[1;0],[0;1],'black','Linewidth',2)
hold on
plot3([0;1],[1;0],[0;0],'black','Linewidth',2)
hold on
plot3([1;0],[0;0],[0;1],'black','Linewidth',2)
hold on
plot3(0.5,0,0.5,'*black','Linewidth',55)
hold on
plot3(0,0.5,0.5,'*black','Linewidth',55)
hold on
h1 = plot3(fibWat(2:r,3),fibWat(2:r,2),fibWat(2:r,1),'--blue','Linewidth',2);
hold on
h2 = plot3(watAir(:,3),watAir(:,2),watAir(:,1),'red','Linewidth',3);
hold on
h3 = plot3(fibAir(2:r1,3),fibAir(2:r1,2),fibAir(2:r1,1),'--green','Linewidth',2);
hold on
plot3(0.5,0.5,0,'*red','Linewidth',2.5)
%h3 = plot3(c,b,a,'green','Linewidth',1.5);
hold on
plot3(0.5,0.5,0,'*black','Linewidth',55)
hold on
hej = plot3(0.4262,0.0738,0.5,'*cyan','Linewidth',60);
hold off
grid on
x = [0.54,0.51];
y = [0.4, 0.463];
x2 = [0.38,0.4];
y2 = [0.15, 0.11];
j2 = annotation('textarrow', x2,y2,...
          'String', '$\left(\frac{1}{2},\frac{1}{2},0\right)$');
set(j2,'interpreter','latex','Fontsize',20)
x3 = [0.23,0.245];
y3 = [0.52, 0.515];
j3 = annotation('textarrow', x3,y3,...
          'String', '$\left(\frac{1}{2},0,\frac{1}{2}\right)$');
set(j3,'interpreter','latex','Fontsize',20)
x4 = [0.575,0.55];
y4 = [0.53, 0.51];
j4 = annotation('textarrow', x4,y4,...
          'String', '$\left(0,\frac{1}{2},\frac{1}{2}\right)$');
set(j4,'interpreter','latex','Fontsize',20)
j5 = annotation('textbox', [0.15, 0.65, 0.1, 0.1],...
          'String', {'$u_{1}\Leftrightarrow\textrm{Gas}$','$u_{2}\Leftrightarrow\textrm{Liquid}$','$u_{3}\Leftrightarrow\textrm{Solid}$'});
set(j5,'interpreter','latex','Fontsize',20)
x6 = [0.135; 0.103];
y6 = [0.135; 0.115];
j6 = annotation('textarrow', x6,y6,...
          'String', '$u_{1}=(1,0,0)$');
set(j6,'interpreter','latex','Fontsize',20)
x7 = [0.4; 0.4];
y7 = [0.8; 0.91];
j7 = annotation('textarrow', x7,y7,...
          'String', '$u_{3}=(0,0,1)$');
```

```
set(j7,'interpreter','latex','Fontsize',20)
x8 = [0.64; 0.695];
y8 = [0.135; 0.115];
j8 = annotation('textarrow', x8,y8,...
          'String', '$u_{2}=(0,1,0)$');
set(j8,'interpreter','latex','Fontsize',20)

h = legend([h1,h2,h3,hej],{'$\Gamma_{\textrm{Solid},\textrm{Gas}}$','$\Gamma_{\textrm{Liquid},\textrm{Gas}}$',...
    '$\Gamma_{\textrm{Solid},\textrm{Liquid}}$','$\left(0.4262,0.0738,0.5\right)$'});
set(h,'interpreter','latex','Fontsize',30)
title(['\textbf{Projection triangle:}','Grid size: $500\times 500$'],'interpreter','latex','Fontsize',40)
set(gca,'xtick',[],'ytick',[],'ztick',[])
%view(3)
view([-45, 360])
%view([90,90])
```

B.2 Plotting in $\mathbb{R}^2$

```
folder = './20150529CreepyLabyrinthThirdTry600Ydim600Time300/';
%X = load([folder,'XplaneZoom.txt']);
%Y = load([folder,'YplaneZoom.txt']);
X = load([folder,'Xplane.txt']);
Y = load([folder,'Yplane.txt']);



%fibAir = load([folder,'fibAirInt.txt']);
%fibWat = load([folder,'fibWatInt.txt']);
%watAir = load([folder,'watAirInt.txt']);

time = 38;
fiber = load([folder,'fiber/timeFolder/t',num2str(time),'.txt']);
%fiber = load([folder,'DiffusedFiber/t',num2str(time),'.txt']);
water = load([folder,'water/timeFolder/t',num2str(time),'.txt']);
%water = load([folder,'DiffusedWater/t',num2str(time),'.txt']);
air = load([folder,'air/timeFolder/t',num2str(time),'.txt']);
%air = load([folder,'DiffusedAir/t',num2str(time),'.txt']);
[r, c] = size(fiber);

indFib = 1;
indWat = 1;
indAir = 1;
fibRep = zeros(1,2);
watRep = zeros(1,2);
airRep = zeros(1,2);
for rI = 1:r
   for cI = 1:c
        if fiber(rI,cI)==1
            fibRep(indFib,1) = X(rI,cI);
            fibRep(indFib,2) = Y(rI,cI);
            indFib = indFib + 1;
        end
        if water(rI,cI)==1
            watRep(indWat,1) = X(rI,cI);
            watRep(indWat,2) = Y(rI,cI);
            indWat = indWat + 1;
        end
        if air(rI,cI)==1
            airRep(indAir,1) = X(rI,cI);
            airRep(indAir,2) = Y(rI,cI);
            indAir = indAir + 1;
        end
   end
end

%%


figure(1)
clf
h1=plot(fibRep(:,1),fibRep(:,2),'*','Color',[.7 .5 0]);
hold on
h2 = plot(watRep(:,1),watRep(:,2),'*blue');
hold on
h3 = plot(airRep(:,1),airRep(:,2),'*cyan');
leg = legend([h1,h2,h3],'Solid','Liquid 1','Gas');
set(leg,'interpreter','latex','Fontsize',50,'Location','East')
xlabel('$x$','interpreter','latex','Fontsize',40)
ylabel('$y$','interpreter','latex','Fontsize',40)
title({'\textbf{Interfaces for three phases in 2 dimensions}',...
```

```
   ['\textbf{Time steps=}',num2str(time),', \textbf{Grid Size:} $600\times 600$']},...
   'interpreter','latex','Fontsize',40)
%title({'\textbf{Interfaces for three phases in 2 dimensions}','$\;\;\;\;\;\;\;\;\;\;\;\;\;\;\;\;\;\;\;\;\;\;\;\;\;\;$\textbf{Zoomed in domain}'},...
%       'interpreter','latex','Fontsize',43)
grid on
set(gca,'Fontsize',20,...
    'Xtick',[0:0.25:2],...
    'Ytick',[0:0.25:2])
%set(gca,...
%    'Yticklabel',{'0','$\frac{1}{4}$','$\frac{1}{2}$','$\frac{3}{4}$','$1$','$\frac{5}{4}$','$\frac{3}{2}$','$\frac{7}{4}$','2'},...
%    'Xticklabel',{'0','$\frac{1}{4}$','$\frac{1}{2}$','$\frac{3}{4}$','$1$','$\frac{5}{4}$','$\frac{3}{2}$','$\frac{7}{4}$','2'},...
%    'TickLabelInterpreter','latex','Fontsize',30);
axis([0,2,0,2])
%axis([0,6,0,6])
%axis([0,1,0,1])
```

B.3 Plotting in $\mathbb{R}^3$

The plotting in $\mathbb{R}^3$ is conducted in two steps. First the 3D tensors are loaded into Matlab, and secondly they are plotted.

B.3.1 Loading of 3D tensors

```
function [x,y,z] = representation3D(sphere,Xplane,Yplane,zVec, nuOfEle)
matrix = zeros((nuOfEle + 1),(nuOfEle + 1),(nuOfEle + 1));

for zI = 1:(nuOfEle+1)
    for rI = 1:(nuOfEle+1)
        for cI = 1:(nuOfEle+1)
            if sphere(rI,cI,zI) == 1
                matrix(rI,cI,zI) = zVec(zI);
            else
                matrix(rI,cI,zI) = NaN;
            end
        end
    end
end
x = Xplane(:,:);
y = Yplane(:,:);
z = matrix(:,:,1);

for index = 2:(nuOfEle + 1)
    x = [x;Xplane];
    y = [y;Yplane];
    z = [z;matrix(:,:,index)];
end


end
```

B.3.2 Plotting of tensors

```
folder = './20150514ThreeDimension150Time300tTot150/';
tTot = load([folder,'tTot.txt']);
%radius = load([folder,'radius.txt']);
Xplane = load([folder,'Xplane.txt']);
Yplane = load([folder,'Yplane.txt']);
nuOfEle = load([folder,'nuOfEle.txt']);
zVec = load([folder,'z.txt']);
timeStep = 108;
fiber = zeros(nuOfEle+1,nuOfEle+1,nuOfEle+1);
water = zeros(nuOfEle+1,nuOfEle+1,nuOfEle+1);
air = zeros(nuOfEle+1,nuOfEle+1,nuOfEle+1);
for zI = 1:(nuOfEle+1)
    fiber(:,:,zI) = load([folder,'fiber/timefolder/t',num2str(timeStep),'/',num2str(zI-1),'.txt']);
    air(:,:,zI) = load([folder,'air/timefolder/t',num2str(timeStep),'/',num2str(zI-1),'.txt']);
    water(:,:,zI) = load([folder,'water/timefolder/t',num2str(timeStep),'/',num2str(zI-1),'.txt']);
end


[xFib,yFib,zFib] = representation3D(fiber,Xplane,Yplane,zVec,nuOfEle);
[xWat,yWat,zWat] = representation3D(water,Xplane,Yplane,zVec,nuOfEle);
[xAir,yAir,zAir] = representation3D(air,Xplane,Yplane,zVec,nuOfEle);
```

```
%% Plottning
figure(1)
clf
%subplot(2,2,3)
subplot(1,2,1)
%plot3(xFib,yFib,zFib,'red')
%surf(xFib,yFib,zFib,'FaceColor',[0.7,0.5,0],'Facealpha',0.15,'edgecolor','none')
%hold on
%plot3(xWat,yWat,zWat,'blue')
surf(xWat,yWat,zWat,'FaceColor','blue','Facealpha',1,'edgecolor','none')
%mesh(xWat,yWat,zWat,[1,0,0]);
hold on
surf(xAir,yAir,zAir,'FaceColor','cyan','Facealpha',0.15,'edgecolor','none')
hold off
%h = legend('\texttt{Fiber}','\texttt{Water}','\texttt{Air}');
%set(h,'interpreter','latex','Fontsize',20,'Location','NorthEast');
xlabel('$x$','interpreter','latex','Fontsize',30)
ylabel('$y$','interpreter','latex','Fontsize',30)
zlabel('$z$','interpreter','latex','Fontsize',30)
title('\textbf{Without solid phase}',...
    'interpreter','latex','Fontsize',15)
grid on
% a = annotation('textbox', [0.18,0.7,0.1,0.1],...
%            'String',...
%            {'\textbf{\underbar{Fact box}}',...
%            ['Grid:$\;\;\;$',num2str(nuOfEle),'$\times$',num2str(nuOfEle),'$\times$',num2str(nuOfEle)],...
%            ['Time step, $\Delta\tau=$',num2str(2*tTot)]});
% set(a,'interpreter','latex','Fontsize',20)
ax = gca;
ax.XTick = [0,0.25,0.5,0.75,1];
ax.YTick = [0,0.25,0.5,0.75,1];
ax.ZTick = [0,0.25,0.5,0.75,1];
ax.TickLabelInterpreter = 'latex';
ax.XTickLabel = {'$0$','$\frac{1}{4}$','$\dfrac{1}{2}$','$\frac{3}{4}$','$1$'};
ax.YTickLabel = {'$0$','$\frac{1}{4}$','$\dfrac{1}{2}$','$\frac{3}{4}$','$1$'};
ax.ZTickLabel = {'$0$','$\frac{1}{4}$','$\dfrac{1}{2}$','$\frac{3}{4}$','$1$'};
set(gca,'Fontsize',20)
axis([0,1,0,1,0,1])
%view(3)
view([15 10])
zoom(1.15)
%view([5,5])
%camzoom(1.1)
% light('Position',[0 0 -1],'Style','local')
% %light('Position',[0 -1 0],'Style','local')
% light('Position',[0 0 1],'Style','local')
% light('Position',[0.1 0.1 0.9],'Style','local')
% light('Position',[0 -0.1 1],'Style','local')
% light('Position',[1 0 0],'Style','local')
% %camlight('headlight')
%subplot(2,2,4)
subplot(1,2,2)
%plot3(xFib,yFib,zFib,'red')
a1 = surf(xFib,yFib,zFib,'FaceColor',[0.7,0.5,0],'Facealpha',0.25,'edgecolor','none');%,'edgealpha',0.2);
hold on
%plot3(xWat,yWat,zWat,'blue')
a2 = surf(xWat,yWat,zWat,'FaceColor','blue','Facealpha',1,'edgecolor','none');
%mesh(xWat,yWat,zWat,[1,0,0]);
hold on
a3 = surf(xAir,yAir,zAir,'FaceColor','cyan','Facealpha',1,'edgecolor','none');
hold off
%set(a1,'handlevisibility','off')
%a1.Color(4) = 0.4;
%alpha(0.5)
h = legend('\texttt{Solid}','\texttt{Liquid 1}','\texttt{Gas}');
pos = get(h,'position');
set(h,'interpreter','latex','Fontsize',20,'position',[0.88 0.6 pos(3:4)]);%,'Location','NorthWest');
xlabel('$x$','interpreter','latex','Fontsize',30)
ylabel('$y$','interpreter','latex','Fontsize',30)
zlabel('$z$','interpreter','latex','Fontsize',30)
title('\textbf{With solid phase}',...
    'interpreter','latex','Fontsize',15)
grid on
% a = annotation('textbox', [0.18,0.7,0.1,0.1],...
%            'String',...
%            {'\textbf{\underbar{Fact box}}',...
%            ['Grid:$\;\;\;$',num2str(nuOfEle),'$\times$',num2str(nuOfEle),'$\times$',num2str(nuOfEle)],...
%            ['Time step, $\Delta\tau=$',num2str(2*tTot)]});
% set(a,'interpreter','latex','Fontsize',20)
ax = gca;
```

```
ax.XTick = [0,0.25,0.5,0.75,1];
ax.YTick = [0,0.25,0.5,0.75,1];
ax.ZTick = [0,0.25,0.5,0.75,1];
ax.TickLabelInterpreter = 'latex';
ax.XTickLabel = {'$0$','$\frac{1}{4}$','$\dfrac{1}{2}$','$\frac{3}{4}$','$1$'};
ax.YTickLabel = {'$0$','$\frac{1}{4}$','$\dfrac{1}{2}$','$\frac{3}{4}$','$1$'};
ax.ZTickLabel = {'$0$','$\frac{1}{4}$','$\dfrac{1}{2}$','$\frac{3}{4}$','$1$'};
set(gca,'Fontsize',20)
axis([0,1,0,1,0,1])
%view(3)
view([15 10])
%view([60 20])
hej = annotation('textbox', [0 0.9 1 0.1], ...
    'String', {'\textbf{Capillary flow in } $\mathbf{R}^{3}:\;\;$',['$\;\;\;\;\;\;\;\;\;\;\;\;$',num2str(timeStep),'$\;$time steps']}, ...
    'EdgeColor', 'none', ...
    'HorizontalAlignment', 'center');
set(hej,'interpreter','latex','Fontsize',20)
```

B.4 Test of convolution formulas

The test of the convolution formulas is done by plotting various points using the two function files u_1 and u_2 which correspond to the convolutions in equation 4.18 and 4.19 on page 71. Thus the first script represents the plotting of various points, and this program calls the functions u_1 and u_2 in order to calculate the convolutions.

B.4.1 Plotting of points

```
%% Test of analytical formulae
%y1 = 0.001;
%y1 = 0.001;
%y1 = sqrt()
 tau = 1e-6;
%y1 = 0.001;
%u3Ini = 0.5+((y1^2)/(8*tau));
u3Ini = 0.502;
%u3Ini = 0.51;
y1 = sqrt(8*tau*(u3Ini-0.5));
 alpha = pi/4;
 xInt = y1*tan(alpha);
k = ((y1-0)/(0-xInt));
% Initial u1 and u2
u1Ini = ((alpha)/(2*pi))-((cos(alpha))/(4*sqrt(pi)*sqrt(tau)))*y1+((alpha+(sin(alpha)*cos(alpha)))/(8*pi*tau))*y1^2;
u2Ini = 1-u1Ini-u3Ini;


%Point 4
x1p = xInt/4;
y1p = k*x1p+y1;
alpha1 = atan(((y1-y1p)/(x1p)));
u3x1py1p = u3Conv(y1p,x1p,alpha1,tau,u3Ini);
u1x1py1p = u1Conv(y1p,x1p,alpha,tau,u1Ini,alpha1);
u2x1py1p = 1-u1x1py1p-u3x1py1p;


% Point 2
x2p = x1p-0.000005;
y2p = y1p;
alpha2 = atan(((y1-y2p)/(x2p)));
u3x2py2p = u3Conv(y2p,x2p,alpha2,tau,u3Ini);
u1x2py2p = u1Conv(y2p,x2p,alpha,tau,u1Ini,alpha2);
u2x2py2p = 1-u1x2py2p-u3x2py2p;


% Point 3
x3p = x1p+0.000005;
y3p = y1p;
alpha3 = atan(((y1-y3p)/(x2p)));
u3x3py3p = u3Conv(y3p,x3p,alpha2,tau,u3Ini);
u1x3py3p = u1Conv(y3p,x3p,alpha,tau,u1Ini,alpha3);
u2x3py3p = 1-u1x3py3p-u3x3py3p;

 %%
figure(3)
clf
```

```
h0 = plot(0,y1,'*black','Linewidth',5);
hold on
h6 = plot(x1p,y1p,'*green','Linewidth',5);
%hold on
%h7 = plot(x2p,y2p,'*blue','Linewidth',5);
%hold on
%h8 = plot(x3p,y3p,'*red','Linewidth',5);
hold on
h0 = plot(xInt,0,'*black','Linewidth',5);
hold on
h8 = plot(linspace(0,1.4e-04,100),k*linspace(0,1.4e-04,100)+y1,'--black');
hold off
grid on
apa = legend(['Triple Junction:(',num2str(0),',',num2str(y1),')'],['Point 1:(',num2str(x1p),',',num2str(y1p),')'],...%,...
             ['Intersection:(',num2str(xInt),',',num2str(0),')'],'$\Gamma_{12}(t=\Delta\tau)$');
            %['Point 2:(',num2str(x2p),',',num2str(y2p),')'],...
            %['Point 3:(',num2str(x3p),',',num2str(y3p),')'],...
set(apa,'interpreter','latex','Fontsize',20)
%set(get(gca,'Xticklabel'),'Interpreter','latex');
set(gca,'Ytick',[0:0.00003:0.00012],...
    'Yticklabel',{'0','$0.3\times 10^{-3}$','$0.6\times 10^{-3}$','$0.9\times 10^{-3}$','$1.2\times 10^{-3}$'},...
    'Xtick',[0:0.00003:0.00012],...
    'Xticklabel',{'0','$0.3\times 10^{-3}$','$0.6\times 10^{-3}$','$0.9\times 10^{-3}$','$1.2\times 10^{-3}$'},...
    'TickLabelInterpreter','latex','Fontsize',15);
xlabel('$x$','interpreter','latex','Fontsize',25);
ylabel('$y$','interpreter','latex','Fontsize',25);
title({'\textbf{Numerical test of analytical formulae: Representation of points}','\textbf{Conservation of contact angle,...
 $\alpha_{0}=\frac{\pi}{4}$}'},'interpreter','latex','Fontsize',30)
axis([0,1.3e-04,0,1.3e-04])



%% figure 4

figure(4)
clf
plot3([0;0],[1;0],[0;1],'black','Linewidth',2)
hold on
plot3([0;1],[1;0],[0;0],'black','Linewidth',2)
hold on
plot3([1;0],[0;0],[0;1],'black','Linewidth',2);
hold on
hej = plot3(3/8,1/8,0.5,'*black','Linewidth',120);
hold on
h3 = plot3(u2x1py1p,u1x1py1p,u3x1py1p,'*green','Linewidth',100);
%hold on
%h4 = plot3(u2x2py2p,u1x2py2p,u3x2py2p,'*blue','Linewidth',100);
%hold on
%h5 = plot3(u2x3py3p,u1x3py3p,u3x3py3p,'*red','Linewidth',100);
hold on
plot3(0.5,0.5,0,'*black','Linewidth',4)
 hold on
 plot3([0.5;0],[0;0.5],[0.5;0.5],'yellow','Linewidth',3)
hold on
plot3([3/8; u2x1py1p; 0.5],[1/8; u1x1py1p; 0.5],[0.5; u3x1py1p; 0],'--green','Linewidth',1.5)
hold on
%plot3([3/8; u2x2py2p; 0.5],[1/8; u1x2py2p; 0.5],[0.5; u3x2py2p; 0],'--blue','Linewidth',1.5)
%hold on
%plot3([3/8; u2x3py3p; 0.5],[1/8; u1x3py3p; 0.5],[0.5; u3x3py3p; 0],'--red','Linewidth',1.5)
%hold on
plot3(0.5,0.5,0,'*red','Linewidth',2.5)
%h3 = plot3(c,b,a,'green','Linewidth',1.5);
hold on
plot3(0.5,0.5,0,'*black','Linewidth',55)
hold off
grid on
x = [0.54,0.51];
y = [0.4, 0.463];
x2 = [0.3,0.32];
y2 = [0.15, 0.11];
j2 = annotation('textarrow', x2,y2,...
          'String', '$\left(\frac{1}{2},\frac{1}{2},0\right)$');
set(j2,'interpreter','latex','Fontsize',20)
x3 = [0.18,0.2];
y3 = [0.52, 0.5];
j3 = annotation('textarrow', x3,y3,...
          'String', '$\left(\frac{1}{2},0,\frac{1}{2}\right)$');
set(j3,'interpreter','latex','Fontsize',20)
x4 = [0.47,0.44];
y4 = [0.53, 0.50];
j4 = annotation('textarrow', x4,y4,...
          'String', '$\left(0,\frac{1}{2},\frac{1}{2}\right)$');
set(j4,'interpreter','latex','Fontsize',20)
```

```
j5 = annotation('textbox', [0.15, 0.65, 0.1, 0.1],...
          'String', {'$u_{1}\Leftrightarrow\textrm{Gas}$','$u_{2}\Leftrightarrow\textrm{Liquid}$','$u_{3}\Leftrightarrow\textrm{Solid}$'});
set(j5,'interpreter','latex','Fontsize',20)
x6 = [0.11; 0.085];
y6 = [0.135; 0.115];
j6 = annotation('textarrow', x6,y6,...
          'String', '$u_{1}=(1,0,0)$');
set(j6,'interpreter','latex','Fontsize',20)
x7 = [0.32; 0.32];
y7 = [0.75; 0.885];
j7 = annotation('textarrow', x7,y7,...
          'String', '$u_{3}=(0,0,1)$');
set(j7,'interpreter','latex','Fontsize',20)
x8 = [0.52; 0.555];
y8 = [0.135; 0.115];
j8 = annotation('textarrow', x8,y8,...
          'String', '$u_{2}=(0,1,0)$');
set(j8,'interpreter','latex','Fontsize',20)


h = legend([h3],['Point 1: $\left(',num2str(u2x1py1p),',',num2str(u1x1py1p),',',num2str(u3x1py1p),'\right)$']);%,...
                    %['Point 2: $\left(',num2str(u2x2py2p),',',num2str(u1x2py2p),',',num2str(u3x2py2p),'\right)$'],...%,...
                    %['Point 3: $\left(',num2str(u2x3py3p),',',num2str(u1x3py3p),',',num2str(u3x3py3p),'\right)$']});
%             ['Point 2: $\left(',num2str(u2x6y6),',',num2str(u1x6y6),',',num2str(u3x6y6),'\right)$'],...
%             ['Point 3: $\left(',num2str(u2x5y5),',',num2str(u1x5y5),',',num2str(u3x5y5),'\right)$'],...
%             ['Point 4: $\left(',num2str(u2x4y4),',',num2str(u1x4y4),',',num2str(u3x4y4),'\right)$'],...
%             ['Point 5: $\left(',num2str(u2x3y3),',',num2str(u1x3y3),',',num2str(u3x3y3),'\right)$'],...
%             ['Point 6: $\left(',num2str(u2x2y2),',',num2str(u1x2y2),',',num2str(u3x2y2),'\right)$'],...
%             '$\left(u_{2}^{\star\star},u_{1}^{\star\star},u_{3}^{\star\star}\right)=\left(\frac{3}{8},\frac{1}{8},\frac{1}{2}\right)$'});
set(h,'interpreter','latex','Fontsize',30)
title({'\textbf{Numerical test of analytical formulae: Projection Triangle}','\textbf{Conservation of contact angle, $\alpha_{0}=\frac{\pi}{4}$}'},...
'interpreter','latex','Fontsize',30)
set(gca,'xtick',[],'ytick',[],'ztick',[])
 view([-45, 360])
```

B.4.2 u_1

```
function u1 = u1Conv(y0,x0,alpha,tau,u1Ini,alpha1)
    Q = y0/x0;
    theta = atan(Q);
    %termIni = ((alpha)/(2*pi));
    N = (1+Q^2)*(alpha+cos(theta-alpha)*sin(theta-alpha))-Q;
    g = ((sqrt(pi)*sqrt(tau)*cos(alpha))/(alpha+(sin(alpha)*cos(alpha))))-(Q*x0);
    f = ((pi*tau*cos(alpha))/(alpha+(sin(alpha)*cos(alpha))));
    h = ((sqrt(pi)*sqrt(tau)*((sqrt(1+Q^2)*sin(theta-alpha))-1))/(N));
    j = ((pi*tau*(1-(sqrt(1+Q^2)*sin(theta-alpha)))^2)/(N^2));
    diff = u1Ini-((alpha)/(2*pi));
    fraction = ((2*tau*diff)/(pi*(alpha+(sin(alpha)*cos(alpha)))));
    factor = ((N)/(8*pi*tau));
    % Första försök: u1 = ((alpha)/(2*pi))+factor*((g/tan(alpha))-h-((sqrt(f+fraction))/(tan(alpha))))^2-factor*j;
    % Andra försök: u1 = ((alpha)/(2*pi))+factor*((g/tan(alpha))-h+((sqrt(f+fraction))/(tan(alpha))))^2-factor*j;
    u1 = ((alpha)/(2*pi))+factor*((g/tan(alpha1))-h-((sqrt(f+fraction))/(tan(alpha1))))^2-factor*j;
end
```

B.4.3 u_3

```
function u3 = u3Conv(y0,x0,alpha1,tau,u3Ini)
    Q = y0/x0;
    %theta = atan(Q);
    %disp('First Thing')
    diff = u3Ini - 0.5;
    nom = sqrt(8*tau*diff)-(Q.*x0);
    frac = ((nom)/(tan(alpha1)));
    f = ((2*sqrt(tau))/(sqrt(pi)*(1+Q^2)));
    g = ((4*tau)/(pi*(1+Q^2)));
    square = (frac-f)^2;
    factor = ((1+Q^2)/(8*tau));
    u3 = 0.5+factor*(square-g);
end
```