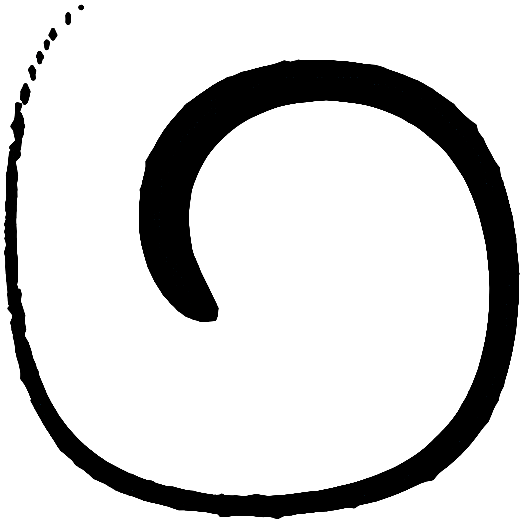
Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Engineering
Department of Energy and Process Engineering

Marcus Sommersel

Interface Tracking for 3D Immersed Boundary Method in Biofluid Dynamics

Master's thesis in Mechanical Engineering
Supervisor: Bernhard Müller
June 2022



NTNU
Norwegian University of
Science and Technology

Marcus Sommersel

Interface Tracking for 3D Immersed Boundary Method in Biofluid Dynamics

Master's thesis in Mechanical Engineering
Supervisor: Bernhard Müller
June 2022

Norwegian University of Science and Technology
Faculty of Engineering
Department of Energy and Process Engineering

NTNU
Norwegian University of
Science and Technology

NTNU

Norwegian University
of Science and Technology

Department of Energy
and Process Engineering

EPT-M-2022

MASTER THESIS

for

Student Marcus Sommersel

Spring 2022

Interface Tracking for 3D Immersed Boundary Method in Biofluid Dynamics

Grensesnittsporing for 3D immersed boundary-metode i biofluiddynamikk

Background and objective

The prediction of fluid-structure interaction (FSI) is not only important for flutter of wings and for flow-induced vibrations in power plants, but also for the flapping motion of the soft palate in the human pharynx. During sleep, the soft palate can make contact with the pharynx wall and lead to obstructive sleep apnea (OSA). Also the sound generated by FSI, e.g. FSI of the inhaled air and the soft palate causing snoring, can be of interest. Because of its great importance for public health, OSA is investigated in a larger interdisciplinary research project entitled “Virtual Surgery in the Upper Airways - New Solutions to Obstructive Sleep Apnea Treatment (VirtuOSA)”, which is funded by the Research Council of Norway.

The objective of the master thesis is to develop, implement and test a method to track the fluid-solid interface for a 3D immersed boundary method (IBM) in biofluid dynamics. The IBM will be used in VirtuOSA to simulate FSI in the upper airways of OSA patients. For testing the interface tracking method, the velocity will be prescribed. Efficient ways of identifying the Cartesian grid points adjacent to marker points on the interface will be investigated. Those Cartesian grid points adjacent to interface marker points identified as solid points will serve as ghost points in our existing IBM for FSI. Fluid velocity, pressure and temperature will be assigned at the ghost points such that the boundary conditions at the fluid-solid interface are approximated. Thus, the compressible Navier-Stokes equations can be easily solved at the fluid points even for complex moving fluid-solid interfaces without the need for any grid generation. The 3D interface tracking method is to be verified for benchmark problems. The master thesis will be a part of VirtuOSA.

The following tasks are to be considered:

1. to check the literature for efficient 3D interface tracking methods,
2. to develop, implement and test an efficient interface tracking method in 3D,
3. to verify the 3D interface tracking method for benchmark problems,
4. to investigate the identification of fluid and solid Cartesian grid points adjacent to interface marker points and the assignment of proper fluid values at ghost points.

-- ” --

Within 14 days of receiving the written text on the master thesis, the candidate shall submit a research plan for his project to his supervisor.

When the thesis is evaluated, emphasis is put on processing of the results, and that they are presented in tabular and/or graphic form in a clear manner, and that they are analyzed carefully.

The thesis should be formulated as a research report with summary in English, conclusion, literature references, table of contents etc. During the preparation of the text, the candidate should make an effort to produce a well-structured and easily readable report. In order to ease the evaluation of the thesis, it is important that the cross-references are correct. In the making of the report, strong emphasis should be placed on both a thorough discussion of the results and an orderly presentation.

The candidate is requested to initiate and keep close contact with his academic supervisor throughout the working period. The candidate must follow the rules and regulations of NTNU as well as possible directions given by the Department of Energy and Process Engineering.

Risk assessment of the candidate's work shall be carried out, in cooperation with the supervisor, according to the department's procedures. The risk assessment must be documented and included as part of the final report. Events related to the candidate's work adversely affecting the health, safety or security, must be documented and included as part of the final report. If the documentation on risk assessment represents a large number of pages, the full version is to be submitted electronically to the supervisor and an excerpt is included in the report. Those who have a theoretical exercise only need to check this and fill out page 1 of the form provided by the Department of Energy and Process Engineering.

Pursuant to “Regulations concerning the supplementary provisions to the technology study program/Master of Science” at NTNU §20, the Department reserves the permission to utilize all the results and data for teaching and research purposes as well as in future publications.

The master's thesis is to be submitted in NTNU’s examination system Inspera Assessment by 15:00 h on June 11, 2022.

☐ Work to be done in lab
☐ Field work

Department of Energy and Process Engineering, January 10, 2022

_____
Bernhard Müller
Academic Supervisor

Abstract

Obstructive sleep apnea is a medical issue caused by collapse of the upper airways restricting oxygen supply during sleep, leading to increased mortality and decreased quality of life. Currently, surgical treatment exists, but there is no way to know the exact outcome of surgery. The VirtuOSA project is a collaboration between St. Olav's University Hospital, NTNU, and SINTEF to develop a CFD tool to predict the outcome of surgical treatment. For the CFD analysis to be accurate, it has to account for the deformation of the upper airways. The fluid-solid interface is essential when modeling the deformations, and an accurate interface tracking method is needed to get good results.

The present work aims at giving a basic understanding of interface tracking methods and investigates the possibility of using the level set method in combination with the ghost point immersed boundary method to track the deformations of the upper airways. The level set method is chosen specifically for the properties that combine well with the ghost point immersed boundary method, allowing it to assign values at the ghost points seamlessly. The end goal is to find a suitable interface tracking method that may be used in the further development of the VirtuOSA project.

The three-dimensional particle level set method has been implemented and is tested for benchmark problems. Extending the standard level set method to the particle level set method showed a significant improvement in mass conservation with little extra CPU time needed. At the same time, it keeps the properties that made the level set method suitable for combination with the ghost point immersed boundary method. The order of convergence is low compared to the order of the numerical schemes used to solve the governing equations. A possible future outlook is to improve the order of convergence, either by tuning the method further or using different numerical schemes.

The present results show that the particle level set method may be tested in the full Navier-Stokes solver to see if it can be used for tracking the deformations of the upper airways in the CFD tool developed in the VirtuOSA project.

Sammendrag

Obstruktiv søvnapné er et medisinsk problem forårsaket av kollaps i de øvre luftveiene som begrenser oksygentilførselen under søvn, som fører til økt dødelighet og senket livskvalitet. For øyeblikket finnes det kirurgiske behandlinger, men ingen måte å vite nøyaktig utfall av operasjonen. VirtuOSA prosjektet er et samarbeid mellom St. Olavs Universitetssykehus, NTNU, og SINTEF med formål å utvikle et CFD-verktøy som kan predikere utfallet av en operasjon. For at CFD-analysen skal være nøyaktig må den ta hensyn til deformasjoner i de øvre luftveiene. Fluid-solid-grensen er essensiell når man modellerer deformasjoner, og en nøyaktig grensesporingsmetode er nødvendig for å få gode resultater.

Denne oppgaven har som mål å gi grunnleggende forståelse av grensesporingsmetoder og undersøke muligheten for å bruke level set-metoden kombinert med immersed boundary-metoden for å følge deformasjonene i de øvre luftveiene. Level set-metoden er valgt fordi den kombinerer godt med ghost point immersed boundary-metoden, og tillater tilegning av verdier i ghost-punktene direkte. Det endelige målet er å finne en passende grensesporingsmetode som kan brukes i den videre utviklingen av VirtuOSA prosjektet.

Den tredimensjonale particle level set-metoden har blitt implementert og testet for to test-caser. Utvidelsen av standard level set-metoden til particle level set-metoden har vist stor forbedring i massebevarelse med lite ekstra CPU-tid. Samtidig beholdes egenskapene som gjorde level set-metoden passende for kombinasjon med ghost point immersed boundary-metoden. Konvergensorden for metoden er lav sammenlignet med orden på de numeriske skjemaene som er brukt for å løse de gjeldende ligningene. En mulig framtidig forbedring vil være å øke konvergensorden, enten ved å finjustere metoden, eller ved å bruke andre numeriske skjemaer.

De nåværende resultatene viser at particle level set-metoden er klar for å testes i den fulle Navier-Stokes-løseren for å se om den kan brukes til å følge deformasjonene i de øvre luftveiene i CFD-verktøyet som utvikles i VirtuOSA prosjektet.

Acknowledgements

I want to give a special thanks to my supervisor Bernhard Müller and Ph.D. candidate Frederik Kristoffersen for being great help and discussion partners during the work with my master's thesis and my project work last semester. I would also like to thank my friends and family for their great support during my five years as a student in Trondheim.

Contents

1 Introduction **1**

1.1 Obstructive Sleep Apnea 1

1.2 VirtuOSA 1

1.3 Outline of Master's Thesis 2

2 Literature review **5**

2.1 CFD in medicine 5

2.2 Interface Tracking 5

2.2.1 Front Tracking Methods 5

2.2.2 Volume of Fluid Methods 6

2.2.3 Level Set Methods 6

2.3 Immersed Boundary Method 7

2.3.1 Ghost Point Immersed Boundary Method 7

3 Governing Equations for the Level Set Method **9**

3.1 Standard Level Set Method 9

3.2 Particle Level Set Method 10

3.3 Combining the Level Set Method with the Immersed Boundary Method 14

3.4 Marching Cubes Algorithm 14

4 Discretization of the Level Set Method **15**

4.1 WENO Method 15

4.2 TVD Runge-Kutta Method 16

4.3 Godunov's Scheme 17

4.4 CFL Number 17

4.5 Boundary Conditions 17
4.6 Measures of Error 18

5 Results **21**

5.1 Sphere in two-dimensional vortex velocity field 21
5.2 Sphere in three-dimensional vortex velocity field 25
5.3 CPU time 29

6 Conclusions and Outlook **31**

Appendices **37**

A LSM3D **39**

A.1 main.cpp 41
A.2 initialization.h and initialization.cpp 49
A.3 particleLSM.h and particleLSM.cpp 53
A.4 schemes.h and schemes.cpp 59
A.5 testCases.h and testCases.cpp 71
A.6 vectorUtilities.h and vectorUtilities.cpp 77
A.7 plotter.py 83

List of Tables

5.1 Error for two-dimensional vortex test. Procedure is explained in section 4.6. 25
5.2 Error for three-dimensional vortex test. Procedure is explained in section 4.6. 29
5.3 CPU time and number of time steps for all test cases at different grid sizes. 30

List of Figures

1.1 The different parts of the upper airways in the sagittal plane [7]. 2

2.1 Representation of the GPIBM showing the interface, a ghost point, the respective boundary intercept and image point, and the fluid points next to the image point. Consider the inside of the circle as the solid domain, and the outside of the circle as the fluid domain. 8

3.1 One-dimensional level set method with a signed distance function. In the one-dimensional case the different domains are defined as lines, and the interface is points between the domains. . . 10

3.2 Figure of the correction step in the particle level set method in two dimensions. The red positive particle has escaped the interface, and is inside a grid cell. ϕ_p is the distance from a cell corner to the particle, while ϕ is the distance from a cell corner to the interface found by the standard level set method. The correct interface location is marked by the dashed line. 13

5.1 Initial sphere in two-dimensional velocity field in the $x - y$ plane for a $400 \times 400 \times 400$ grid. The same shape should be found at $t = T$. 22

5.2 Domains where $\phi(\boldsymbol{x}, t) \leq 0$ at $t = 1/2\,T$ for the sphere in the two-dimensional velocity field with $n = 400$. 22

5.3 Domains where $\phi(\boldsymbol{x}, T) \leq 0$ for the sphere in the two-dimensional velocity field at four different grid resolutions in the $x - y$ plane. 23

5.4 Domains where $\phi(\boldsymbol{x}, t) \leq 0$ at $t = 1/2\,T$ for the sphere in the two-dimensional velocity field with $n = 50$, $n = 100$, and $n = 200$. 24

5.5 Initial sphere in three-dimensional velocity field for a $400 \times 400 \times 400$ grid. The same shape should be found at $t = T$. 26

5.6 Domains where $\phi(\boldsymbol{x}, t) \leq 0$ at $t = 1/2\,T$ for the sphere in the three-dimensional velocity field with $n = 400$. 26

5.7 Domains where $\phi(\boldsymbol{x}, T) \leq 0$ for the sphere in the three-dimensional velocity field at four different grid resolutions. 27

5.8 Domains where $\phi(\boldsymbol{x}, t) \leq 0$ at $t = 1/2\,T$ for the sphere in the three-dimensional velocity field with $n = 50$, $n = 100$, and $n = 200$. 28

Nomenclature

$\boldsymbol{\nabla}$	Nabla operator
$\boldsymbol{N}$	Normal vector
$\boldsymbol{u}$	Velocity vector
$\boldsymbol{x}$	Position vector
$\boldsymbol{x}_p$	Particle position vector
$\Delta\tau$	Pseudo time step used during reinitialization
Δt	Time step
Δx	Step size in x-direction
Δy	Step size in y-direction
Δz	Step size in z-direction
ϕ	Level set method variable
τ	Pseudo time used during reinitialization
BI	Boundary intercept
GP	Ghost point
IP	Image point
n	Grid points in x-, y- and z-direction, creating an $n \times n \times n$ grid
p	Order of convergence
r_p	Radius of the particle
$S(\phi)$	Sign function of ϕ
s_p	Sign of the particle
T	End time, used in the test cases
t	Time

u	Velocity in x-direction
v	Velocity in y-direction
w	Velocity in z-direction
x	Position in x-direction
y	Position in y-direction
z	Position in z-direction
$\delta\Omega$	Boundary of domain
Ω^+	Outside domain
Ω^-	Inside domain
CFD	Computational fluid dynamics
CFL	Courant-Friedrichs-Lewy, CFL number
FSI	Fluid-structure interactions
GPIBM	Ghost point immersed boundary method
IBM	Immersed boundary method
LSM	Level set method
OSA	Obstructive sleep apnea
RK	Runge-Kutta
TVD	Total variation diminishing
VOF	Volume of fluid method
WENO	Weighted essentially non-oscillating

Chapter 1

Introduction

1.1 Obstructive Sleep Apnea

Obstructive sleep apnea (OSA) is a problem that affects 2-4 % of the population and is caused by a collapse in the upper airways while sleeping [1]. Deformations in the upper airways restrict the airflow and may cause heavy snoring. If the deformation is severe, the upper airways will collapse and cause a lack of oxygen supply. The severity of the disease is categorized based on how many apneas the patient develops on average during one hour of sleep, where an apnea is defined as the complete stop of airflow for at least 10 seconds. If a patient has an average of five or more apneas per hour of sleep, the patient is diagnosed with OSA [1]. OSA has shown to be more common among men and people over the age of 50 years [2]. The largest risk factors for developing OSA are obesity and high Body Mass Index (BMI). More details of the worldwide prevalence of OSA can be found in the article by Benjafield et al. [3].

The reduced sleep results in tiredness, while the lack of oxygen can lead to an increased chance of heart failure and stroke, increased mortality, and decreased quality of life [4]. Temporary treatment by wearing a nasal or oronasal mask while sleeping has shown positive results by applying pressure to the upper airways [5]. Surgical treatments have been tried in the past, but there is vast uncertainty related to effectiveness and no way to predict the outcome of the surgery. In some cases, surgery has shown significant improvement for the patients, but many cases show no improvement, and there are even cases where surgery has worsened the condition [6].

1.2 VirtuOSA

The VirtuOSA project is a collaboration between St. Olav's University Hospital, NTNU, and SINTEF, trying to use Computational Fluid Dynamics (CFD) to conduct virtual surgery of the upper airways [8]. The project consists of specialists in ENT surgery, computational fluid dynamics, and structural engineering, with the primary goal to gain a better understanding of OSA and to create a diagnostic tool to help predict the outcome of the surgical treatment [6]. The project has created four work packages to incorporate the different scientific fields. Of these work packages, work package three is of particular interest to the present work, as it covers mathematical modeling of the fluid-structure interactions (FSI). Work package three aims at developing an FSI model to be used for calibration of the three-dimensional CFD model to be developed in work package four. The Ph.D. thesis by Moxness [6] gives further information on VirtuOSA and the goals of the different work packages.

For the CFD model to be accurate, it has to account for the deformations of the upper airways. There are many challenges when modeling deforming interfaces. Many of these challenges occur at the interface between two fluids or the fluid-solid interface because of the interplay between the different substances. Interactions between the substances may be caused by forces from one substance affecting the other, chemical reactions, or diffusion between the substances at the interface. All of these interactions affect the position of the interface, which

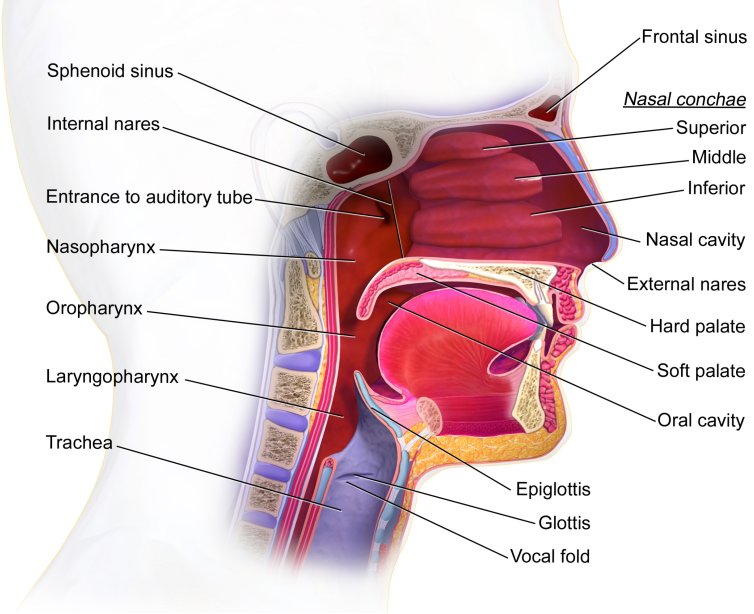


Figure 1.1: The different parts of the upper airways in the sagittal plane [7].

is why accurate interface tracking is an essential and challenging topic for problems with deformations. It is impossible to conduct a good CFD analysis without knowing the interface's position, as the wrong properties and equations may be used and solved for the wrong fluid or solid.

1.3 Outline of Master's Thesis

The present work gives an overview of the immersed boundary method for fluid-structure interactions and various interface tracking methods. However, the main goals have been developing and testing the level set method for interface tracking when the velocity field is prescribed and verifying the implementation against benchmark problems. Further, compatibility between the level set method and the ghost point immersed boundary method is investigated. The investigation aims to provide a tool for tracking the deformation in the upper airways as a part of the VirtuOSA project.

The structure of the remaining chapters is as follows. Chapter 2 aims to give a short review of relevant topics for the present work, starting with CFD in medicine in section 2.1, before reviewing interface tracking in section 2.2, and explaining the immersed boundary method in section 2.3.

The governing equations for the level set method are introduced in chapter 3. The chapter starts with the equations for the standard level set method in section 3.1, before explaining the particle level set method in section 3.2, and how it combines with the ghost point immersed boundary method in section 3.3.

Chapter 4 explains how the equations given in chapter 3 are discretized before introducing three measures of errors in section 4.6.

The test cases applied to the level set method are showcased and discussed in chapter 5, where section 5.1 is devoted to a sphere deformed in the two-dimensional velocity field, and section 5.2 is devoted to a sphere deformed in the three-dimensional velocity field.

Finally, the conclusions and a future outlook are presented in chapter 6.

The three-dimensional level set method implemented in C++ can be found in the appendix and on GitHub at `https://github.com/marcussommersel/LSM3D`.

The present work is a continuation of the project work by Sommersel in the autumn of 2021 [9]. Hence certain

parts of the article are similar. The main difference is the extension into three dimensions and using a different interface tracking method to improve the results. However, the goal and the reason behind the present work have stayed the same.

Chapter 2

Literature review

2.1 CFD in medicine

Computational Fluid Dynamics (CFD) has had wide use in engineering for many decades, but in recent years the use of CFD has gained interest in other fields as well [10]. The use of CFD in medicine and biomechanics has proved useful and, in some cases, enables the treatment of diseases that were previously not treatable. CFD is primarily applied in cardiovascular medicine, although other areas of medicine, i.e., respiratory medicine, are extending its use. The use of CFD enables detailed characterization and computations of metrics that cannot be measured directly [11]. The development of CFD models into clinical tools helps treat the patients with less invasive methods, which may ease the load on the patients. The ability to see the outcome of surgery before the procedure itself may reduce the cost of unnecessary treatment. Running different disease scenarios through a CFD model may also help predict the optimal treatment plan for individual patients. When modeling a complex coupled system such as the respiratory or cardiovascular systems, the fluid flow and solid tissue interact. An efficient and effective approach to fluid-structure interactions is essential [12].

2.2 Interface Tracking

Fluid-structure interactions (FSI) are essential in many engineering applications [13]. These applications come in a wide range, and among them are many mechanical problems like the flutter of wings, wind-turbine applications, and airplane response mechanisms. However, they are also important in biomechanics, like blood flow through the heart and airflow through the upper airways. To account for FSI in a numerical model, it is important to know the exact location of the fluid-solid interface. There are many ways to do interface tracking numerically, and three different classes of methods are often used: front tracking methods, volume of fluid methods, and level set methods.

2.2.1 Front Tracking Methods

The term "Front Tracking" was first introduced in 1967 by Richtmyer and Morton [14], although the method was not implemented before 1981 in the work by Glimm et al. [15]. In front tracking methods, marker points are defined on the interface, and the front is advected with the flow [16]. The front is not only the marker points, but also information about the connectivity of the points, and sometimes the description of the physics at the interface. The interface can then be found by connecting these marker points. The method's accuracy depends on the number of marker points used to represent the interface and the interpolation method used to reconstruct the interface. The placement of these marker points is crucial, as more points are needed in regions of more significant interface curvature than in regions with a flatter interface. The placement of the marker points changes as the interface is stretched or compressed, as the stretched region can be lacking points, and the compressed region may be overflowing with points. Adding and removing marker points is therefore necessary. When extending front tracking methods to three dimensions, the complexity increases significantly, and having

a proper data structure to describe the front is essential. The mass conservation for front tracking methods varies and depends on the method used to advect the flow and the method used to reconstruct the interface. For further reading on front tracking methods, the reader is directed towards chapter 6 in the book by Tryggvason et al. [16].

2.2.2 Volume of Fluid Methods

The Volume of Fluid (VOF) method was introduced in the 1981 paper by Hirt and Nichols [17]. For a flow with two different fluids, the volume of fluid method defines a color function [16]. This color function uses one of the fluids as a reference, and its value in a particular cell represents how much of the total cell volume is filled with the reference fluid. If it occupies the entire cell, the color function has a value of 1, while if the cell is filled with the other fluid, it has a value of 0. This exact procedure can also be used for fluid-solid problems, where either the fluid or the solid would be used as a reference. An algorithm to reconstruct the interface is necessary for the volume of fluid method, as the interface is not directly available from the values of the color function in the cells. The volume of fluid method has good mass conservation, and the extension to three dimensions is straightforward, although not as easy as the level set method introduced in section 2.2.3. Chapter 5 in the book by Tryggvason et al. [16] presents the volume of fluid method further, including some interface reconstruction algorithms.

2.2.3 Level Set Methods

The Level Set Method (LSM) was introduced by Osher and Sethian in 1988 [18]. Level set methods initialize a scalar field in the entire computational domain, where the interface between two fluids or the fluid-solid interface is set to a specific scalar value [19]. The contour at this value, or level, gives the interface. Usually, the scalar field is constructed from a signed distance function with reference to the interface, and the interface can be found at the zero-contour. The level set function can be visualized in two dimensions as a deforming three-dimensional body. An intersection at a certain height would give a two-dimensional domain where the boundary of this domain is equal to the interface. The level set function is advected with the velocity field, and the three-dimensional body is deformed, but the interface can still be found from the intersection at the same height. One of the positive sides of the level set method is its ability to handle splitting bodies. If the interface deforms and splits apart, the interface can still be found from the zero-level set. The signed distance properties of the level set method will deteriorate as the method is iterated forward in time. In 1994, Sussmann et al. introduced periodic reinitialization of the level set method to restore the signed distance field [20]. For information on the standard level set method, the book by Osher and Fedkiw [19] and the book by Sethian [21] are advised, as well as a recent review of level set methods by Gibou et al. [22] and the references therein.

The standard level set method suffers from poor mass conservation. Previous work has shown methods to improve mass conservation [23], [24], [25], [26]. In [23], Russo and Smereka noticed that the reinitialization equation, eq. (3.3), would move the interface location, and implementing the exact interface location would improve the mass conservation. The method was second-order accurate, and in [24] Chéné et al. extended the method to fourth-order accuracy. Hartmann et al. [25] obtained better accuracy by replacing the reinitialization equation with a higher-order constrained reinitialization (HCR). Methods combining the volume of fluid and level set method, which conserve the total volume by construction, have also been implemented [27] [28]. However, these methods become more complicated than the level set method in three spatial dimensions.

The present work has tested both the higher-order reinitialization methods of Russo and Smereka [23], and the particle level set method by Enright et al. [26]. Early testing showed favorable results with the particle level set method. The particle level set method places particles in a band around the interface. These particles are then advected with the local velocity independently of each other and the level set function. If one of the particles at some point crosses the interface, it indicates the level set method may have found a wrong solution, leading to an incorrect interface. The particles that have incorrectly crossed the interface are then used to rebuild the scalar field to correct the wrong solution.

The present work implements the particle level set method introduced in section 3.2. A level set method is favored because it gives easy extension to three dimensions and because the interface can always be found from the zero-contour. Since the level set method has shown poor mass conservation, the particle level set method is implemented to combat this flaw. In addition, the level set method works well with the Ghost Point Immersed Boundary Method (GPIBM), as shown in section 3.3, which is one of the main requirements for the possibility of incorporation with the VirtuOSA project.

2.3 Immersed Boundary Method

The standard in CFD analysis when simulating FSI is to use body-fitted grids. These grids are dependent on the problem and change as the simulations go on. The need to do re-meshing for simulations undergoing large deformations can be computationally expensive, and further problems may occur as grids of different regions overlap each other. Other methods with constant grids have been developed to alleviate the problems occurring with body-fitted grids and re-meshing. One of these methods is the immersed boundary method (IBM) introduced by Peskin [29]. The IBM utilizes a fixed background grid to solve the governing equations by introducing fictive body forces or locally allocating flow values to approximate the boundary conditions at the fluid-solid interface. Although re-meshing is no longer a problem, the results are less accurate. Different IBMs can be classified as either sharp or diffuse interface methods [13]. The diffuse IBMs smear the immersed boundaries to the surrounding grid nodes, which can be done by applying smeared delta functions to fictitious body forces. The sharp IBMs do not smear the interface to the surrounding grid nodes and may cut the nodes at the interface creating a local unstructured grid, or they may apply jump conditions at the interface. Many different IBMs are available, and the keen reader is directed towards the article by Sotiropoulos and Yang [13] and the references therein.

2.3.1 Ghost Point Immersed Boundary Method

The computational domain is divided into fluid and solid domains in fluid-structure interactions. The Navier-Stokes equations are used to determine how the fluid flow behaves, while a corresponding set of equations are used to govern the behavior of the solid. All points in the fluid domain are denoted as fluid points, and all points in the solid domain are denoted as solid points. To solve the Navier-Stokes equations for one fluid point, it needs to consider the fluid properties of the points next to it. However, some of the points next to the fluid points at the interface between the fluid and solid domain are solid points, which means they do not have fluid properties. The Ghost Point Immersed Boundary Method (GPIBM) introduced by Tseng and Ferziger [30] solves this problem by defining a set of ghost points to be the solid points next to the interface. These ghost points have no physical meaning, but they are artificially extending the fluid domain into the solid domain, which means there are enough fluid points to solve the governing equations for all real fluid points. The layout is easier to see in Figure 2.1. To use the ghost points, they need to be assigned fluid properties, which is done by applying different boundary conditions at the interface. By drawing a line from the ghost point to the interface parallel to the normal vector at the interface, the correct boundary condition can be found for each ghost point. The interception of this line with the interface is called the boundary intercept. A point in the fluid domain is also needed to enforce the boundary conditions. The image point is defined by extending the line from the ghost point to the boundary intercept with the same length. Two examples of boundary conditions are given below to show how the GPIBM is used.

Neumann boundary conditions for a flow variable U at the fluid-solid interface, i.e.,

$$\frac{\partial U(\boldsymbol{x}_{BI})}{\partial n} = U_{n\,BI}, \tag{2.1}$$

where $\boldsymbol{x}_{BI}$, the body intercept, is the intersection of the fluid-solid interface and the line between the ghost point $\boldsymbol{x}_{GP}$ and image point $\boldsymbol{x}_{IP}$, are approximated by

$$\frac{U_{IP} - U_{GP}}{|\boldsymbol{x}_{IP} - \boldsymbol{x}_{GP}|} = U_{n\,BI}, \tag{2.2}$$

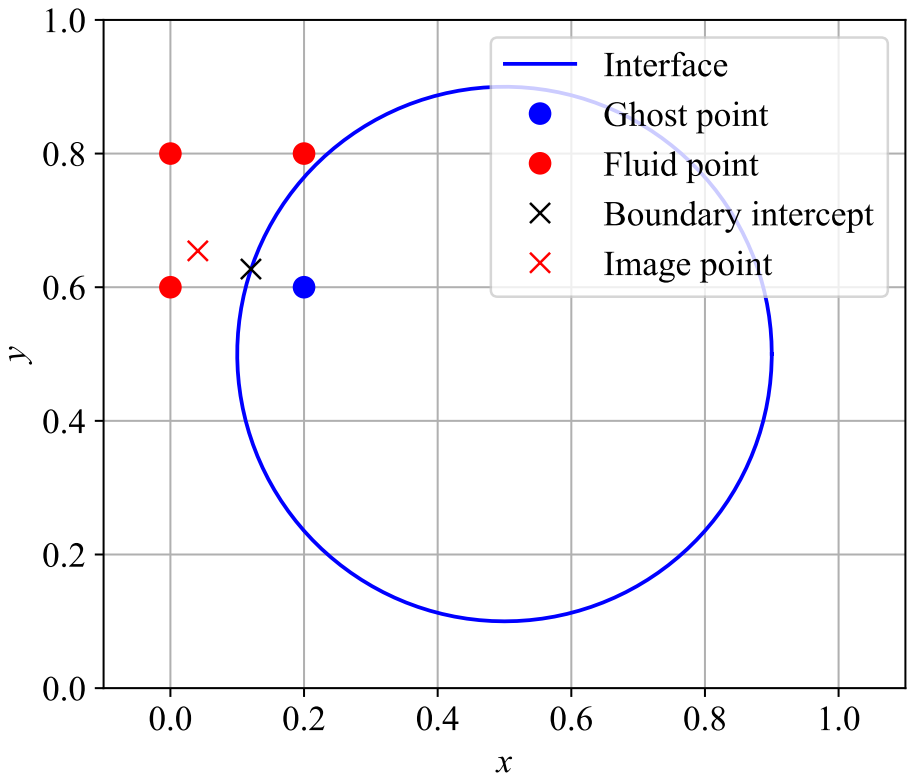


Figure 2.1: Representation of the GPIBM showing the interface, a ghost point, the respective boundary intercept and image point, and the fluid points next to the image point. Consider the inside of the circle as the solid domain, and the outside of the circle as the fluid domain.

yielding an equation to determine U_{GP} [31].

Dirichlet boundary conditions for a flow variable U at the fluid-solid interface, i.e.,

$$U(\boldsymbol{x}_{BI}) = U_{BI}, \tag{2.3}$$

are approximated by

$$\frac{1}{2}(U_{IP} + U_{GP}) = U_{BI}, \tag{2.4}$$

yielding an equation to determine U_{GP} [31].

These two boundary conditions allow the assignment of values at the ghost points to uphold the boundary conditions. The goal is to find the values at the ghost points. The values at the boundary intercept are given, and the values at image points can be found by interpolating the values at the fluid points next to it. What remains is finding the position of the image point and the interface.

Chapter 3

Governing Equations for the Level Set Method

3.1 Standard Level Set Method

The level set equation for an externally created velocity field $\boldsymbol{u}$ is

$$\phi_t + \boldsymbol{u} \cdot \boldsymbol{\nabla}\phi = 0 \tag{3.1}$$

where ϕ is the level set variable [19]. Often, including the present work, a signed distance function with reference to the interface location is used for ϕ. A region Ω is defined where Ω^- is inside the region, Ω^+ is outside the region, and $\partial\Omega$ is the boundary of the region [19]. The signed distance function is defined for the whole computational domain. The value of the signed distance function is the shortest distance to $\partial\Omega$, where values within Ω^- are negative, values within Ω^+ are positive, and values on $\partial\Omega$ are zero. This can be defined as

$$\phi(\boldsymbol{x}) = \begin{cases} \min(|\boldsymbol{x} - \boldsymbol{x}_I|) & \text{if } \boldsymbol{x} \in \Omega^+, \\ -\min(|\boldsymbol{x} - \boldsymbol{x}_I|) & \text{if } \boldsymbol{x} \in \Omega^-, \text{ and} \\ 0 & \text{if } \boldsymbol{x} \in \delta\Omega \end{cases} \tag{3.2}$$

where $\boldsymbol{x}_I$ are all points on $\partial\Omega$. An example of a signed distance function in one dimension can be seen in Figure 3.1.

The signed distance function does not retain its signed distance properties through evolution in time. This may be caused by distorted solutions leading to very large or small gradients around the interface [23]. In addition, the level set solution is prone to jumps at the interface when interfaces merge [20]. To fix the signed distance function, regular reinitialization is applied to the signed distance field. Reinitialization is done by keeping the interface location fixed. At the same time, the rest of the field is iterated a number of pseudo time steps forward in pseudo time to fulfill $|\boldsymbol{\nabla}\phi| = 1$, which is what characterizes a signed distance field. The reinitialization equation is defined as [19]

$$\phi_\tau + S(\phi_0)(|\boldsymbol{\nabla}\phi| - 1) = 0 \tag{3.3}$$

where τ is an artificial time. The $S(\boldsymbol{x})$ term is a sign function.

The present work uses a sign function defined as

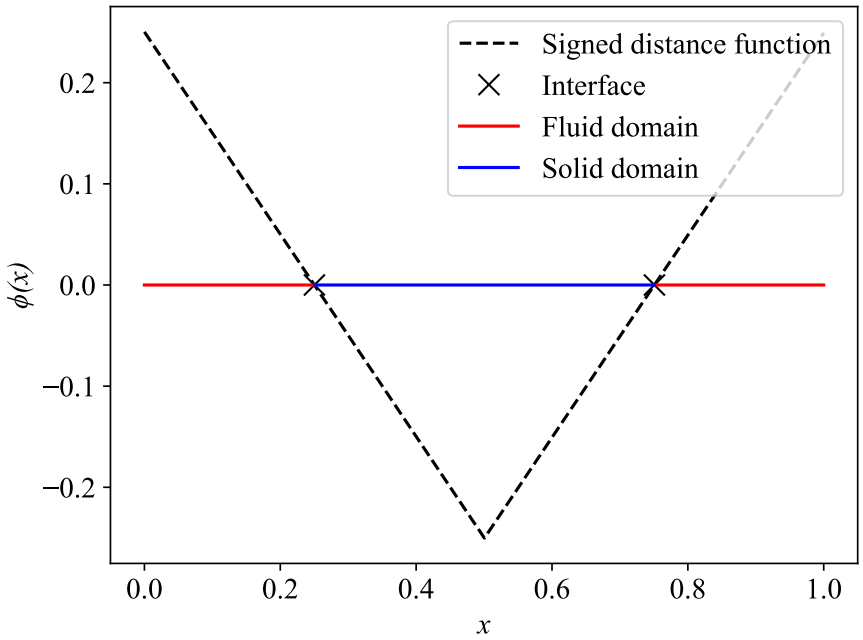


Figure 3.1: One-dimensional level set method with a signed distance function. In the one-dimensional case the different domains are defined as lines, and the interface is points between the domains.

$$S(\phi_0) = \begin{cases} 1 & \text{if } \phi_0(\boldsymbol{x}) > 0, \\ -1 & \text{if } \phi_0(\boldsymbol{x}) < 0, \text{ and} \\ 0 & \text{if } \phi_0(\boldsymbol{x}) = 0, \end{cases} \tag{3.4}$$

where $\phi_0(\boldsymbol{x})$ is the level set function before the first pseudo time step. Other sign functions can also be used, and a common choice is [19]

$$S(\phi) = \frac{\phi}{\sqrt{\phi^2 + |\boldsymbol{\nabla}\phi|^2 \Delta x^2}}, \tag{3.5}$$

where the sign function is updated for each pseudo time step.

3.2 Particle Level Set Method

The particle level set method utilizes a set of independently advected marker particles to correct the interface when the level set method computes a wrong solution [26]. The initial scalar field is used to initialize a band of particles on both sides of the interface. There are two sets of particles, defined to be positive and negative particles. The particles are placed in every cell with at least one corner within three cell widths from the interface. This can be found from the signed distance function as wherever $|\phi| < 3 \max(\Delta x, \Delta y, \Delta z)$. In three dimensions, 128 particles are seeded by placing them randomly distributed in each cell, where half of them are positive, and the other half are negative. The number of particles of each type is set to 4^1 in one-dimensional flows, 4^2 in two-dimensional flows, and 4^3 in three-dimensional flows [26].

After each particle is seeded, it is attracted to the the correct side of the interface. Positive particles are attracted to the $\phi \geq 0$ side of the interface, and the negative particles to the $\phi < 0$ side of the interface. The attraction is done by randomly picking a distance from the interface, ϕ_{goal}, within a band of $b_{\text{min}} \leq \phi_{\text{goal}} \leq b_{\text{max}}$ for positive

particles, and within $-b_{\text{max}} \leq \phi_{goal} \leq -b_{\text{min}}$ for negative particles, where $b_{\text{min}} = 0.1 \min(\Delta x, \Delta y, \Delta z)$ and $b_{\text{max}} = 3.0 \min(\Delta x, \Delta y, \Delta z)$. The attraction step is done with

$$\boldsymbol{x}_{new} = \boldsymbol{x}_p + \lambda \left(\phi_{\text{goal}} - \phi(\boldsymbol{x}_p)\right) \boldsymbol{N}(\boldsymbol{x}_p), \tag{3.6}$$

where $\boldsymbol{x}_p$ is the coordinates of the particle, $\boldsymbol{N}$ is the normal vector to the interface, $\lambda = 1$, and the local values of ϕ and $\boldsymbol{N}$ are found by trilinear interpolation. The unit normal vector can be found from the scalar field and is defined as

$$\boldsymbol{N} = \frac{\boldsymbol{\nabla}\phi}{|\boldsymbol{\nabla}\phi|}. \tag{3.7}$$

The attraction step is used to gain a smoother distribution in the direction normal to the interface [26].

The trilinear interpolation is an extension from line interpolation in one dimension and bilinear interpolation in two dimensions. The interpolation starts by finding the correct point on each axis of the cell

$$x_d = \frac{x - x_0}{x_1 - x_0}, \tag{3.8}$$

$$y_d = \frac{y - y_0}{y_1 - y_0}, \tag{3.9}$$

and

$$z_d = \frac{z - z_0}{z_1 - z_0}, \tag{3.10}$$

where x, y and z is the coordinate of the interpolated value, and x_0, x_1, y_0, y_1, z_0 and z_1 are the cell boundaries with $x_0 < x_1$, $y_0 < y_1$ and $z_0 < z_1$. By first interpolating along the x-axis, the corresponding values are found from

$$c_{00} = c_{000}(1 - x_d) + c_{100}x_d, \tag{3.11}$$

$$c_{01} = c_{001}(1 - x_d) + c_{101}x_d, \tag{3.12}$$

$$c_{10} = c_{010}(1 - x_d) + c_{110}x_d, \tag{3.13}$$

and

$$c_{11} = c_{011}(1 - x_d) + c_{111}x_d, \tag{3.14}$$

where c_{ijk} denote the values at (x_i, y_j, z_k). Interpolation across the y-axis gives the values

$$c_0 = c_{00}(1 - y_d) + c_{10}y_d, \tag{3.15}$$

and

$$c_1 = c_{01}(1 - y_d) + c_{11}y_d, \tag{3.16}$$

before interpolation along the z-axis gives the value at (x, y, z) by

$$c = c_0(1 - z_d) + c_1 z_d. \tag{3.17}$$

If the particle is not within the correct band, λ is halved, and $\boldsymbol{x}_p = \boldsymbol{x}_{new}$ is used along with the new local values $\phi(\boldsymbol{x}_p)$ and $\boldsymbol{N}(\boldsymbol{x}_p)$ to find the new particle location with eq. (3.6). This process is repeated a maximum of 15 times, and if the particle is still not within the correct band, it is deleted. If the particle is in the correct band, the new coordinates of the particle are saved, and $\boldsymbol{x}_p = \boldsymbol{x}_{new}$. The particle is then assigned a radius set by

$$r_p = \begin{cases} r_{\text{max}} & \text{if } s_p\phi(\boldsymbol{x}_p) > r_{\text{max}} \\ s_p\phi(\boldsymbol{x}_p) & \text{if } r_{\text{min}} \leq s_p\phi(\boldsymbol{x}_p) \leq r_{\text{max}} \\ r_{\text{min}} & \text{if } s_p\phi(\boldsymbol{x}_p) < r_{\text{min}}, \end{cases} \tag{3.18}$$

where $r_{\text{min}} = 0.1 \min(\Delta x, \Delta y, \Delta z)$, $r_{\text{max}} = 0.5 \min(\Delta x, \Delta y, \Delta z)$, and $s_p = 1$ for positive particles and $s_p = -1$ for negative particles.

At each time step, the particles are advected with the external velocity field independently of each other and the level set method. The advection is governed by

$$\frac{d\boldsymbol{x}_p}{dt} = \boldsymbol{u}(\boldsymbol{x}_p), \tag{3.19}$$

where $\boldsymbol{u}(\boldsymbol{x}_p)$ is the local velocity.

After both the level set equation (3.1) and the particle advection equation (3.19) are moved forward in time, the particles are used to correct the interface. All particles on the wrong side of the interface with more than their radius, i.e. if $\phi(\boldsymbol{x}_p) < -r_p$ for positive particles and $\phi(\boldsymbol{x}_p) > r_p$ for negative particles, are denoted as escaped particles. The escaped positive particles are tasked with rebuilding the $\phi > 0$ region, while the escaped negative particles are tasked with rebuilding the $\phi < 0$ region. An escaped particle indicates an error in at least one of the eight corner values for the cell containing the escaped particle. The particles can generate their own local level set functions, where the zero level set gives the surface of the particles. This local level set function is defined as

$$\phi_p(\boldsymbol{x}) = s_p \cdot (r_p - |\boldsymbol{x} - \boldsymbol{x}_p|). \tag{3.20}$$

As the particles escape, equation (3.20) may be used to predict the value of the global level set variable and may be used to rebuild the signed distance field. The escaped particles predict the values of ϕ at each cell corner by applying equation (3.20) to all eight corners of the cell. The ϕ_p value at a cell corner represents the distance from the corner to the particle surface. An example in two dimensions is seen in Figure 3.2, showing an escaped positive particle in a grid cell, the current interface found by the level set method, the correct interface, and the distances ϕ_p and ϕ. The particle would never physically cross the correct interface, which means the actual length from the corner to the correct interface location is maximum ϕ_p for the escaped particle in Figure 3.2. The predicted value at the corner, ϕ_p, is checked with the global value of the corner, ϕ, and two new parameters are defined; ϕ^+ to rebuild the positive region and ϕ^- to rebuild the negative region, where

$$\phi^+ = \max(\phi_p, \phi), \tag{3.21}$$

and

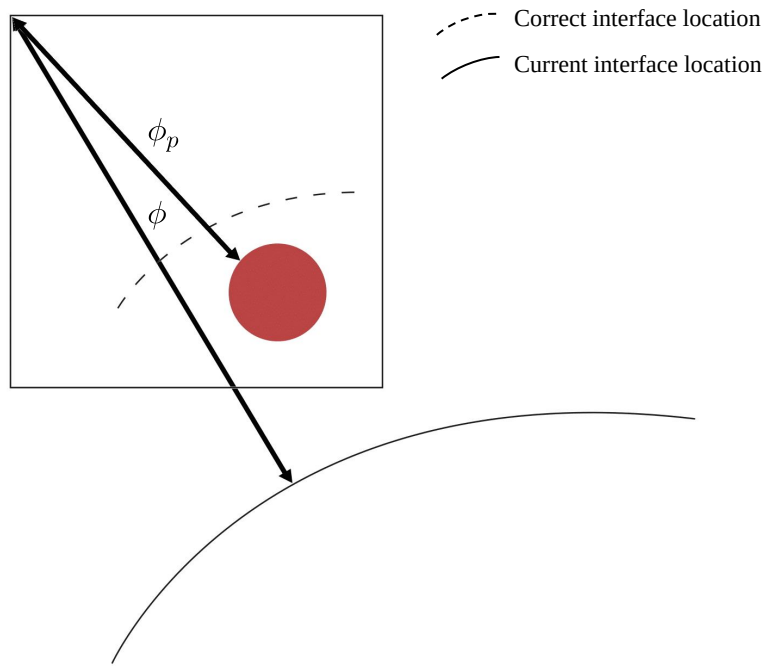


Figure 3.2: Figure of the correction step in the particle level set method in two dimensions. The red positive particle has escaped the interface, and is inside a grid cell. ϕ_p is the distance from a cell corner to the particle, while ϕ is the distance from a cell corner to the interface found by the standard level set method. The correct interface location is marked by the dashed line.

$$\phi^- = \min(\phi_p, \phi). \tag{3.22}$$

The correct value is set at the corner by applying

$$\phi = \begin{cases} \phi^+ & \text{if } |\phi^+| \leq |\phi^-| \\ \phi^- & \text{if } |\phi^+| > |\phi^-|, \end{cases} \tag{3.23}$$

as it gives priority to the values closer to the interface. Once the values of ϕ are set on all corners of the cell, the radius is updated with Eq. (3.18). The particle radius is small close to the interface to determine if the particle escapes more accurately. The same procedure is repeated for all the particles.

After the particles are advected multiple times, or there are large deformations at the interface, the particles may need to be reseeded. Reseeding includes deleting particles that no longer provide valuable information and inserting new particles in cells with few remaining particles. The reseeding algorithms used in [26] are more complex than in the present work. The more straightforward methods used here are due to time restrictions, and more complex methods were not prioritized because the implemented methods showed promising results. However, readers are encouraged to read the original article [26] to get further insight into the reseeding algorithms available. In the present work, the deletion of unnecessary particles is coupled with the correction step, and all particles with $|\phi(\boldsymbol{x}_p)| - r_p > b_{\max}$, meaning they are entirely outside the band around the interface, are deleted. The insertion of new particles is done after the correction step, eq. (3.23), after a number of iterations depending on the test case. It is done by finding how many particles are left in the cells and simply inserting the missing amount from each cell's previously defined number of particles. In addition to periodic reseeding, Enright et al. [26] also suggest reseeding when the interface has undergone a certain amount of

compression or stretching.

3.3 Combining the Level Set Method with the Immersed Boundary Method

The end goal of the present work is to find an interface tracking algorithm that can be implemented in combination with the GPIBM. For the interface tracking method to be effective, it needs to identify whether a point is in the fluid or solid region, which is found from the sign of the signed distance function in the level set method, and find the image points for the GPIBM. This can be done with the level set method, as the normal vector from the ghost points to the interface points in the same direction as the gradient of the level set method and can easily be identified.

The normal vector defined in equation 3.7 can be used together with the value of the signed distance function at the ghost points to find the closest points on the interface [19]. By doubling this distance, the location of the image points seen in Figure 2.1 can be found from

$$x_{IP} = x - 2\,\phi(x,y,z)\,\frac{\partial\phi(x,y,z)}{\partial x}, \tag{3.24}$$

$$y_{IP} = y - 2\,\phi(x,y,z)\,\frac{\partial\phi(x,y,z)}{\partial y}, \tag{3.25}$$

and

$$z_{IP} = z - 2\,\phi(x,y,z)\,\frac{\partial\phi(x,y,z)}{\partial z}, \tag{3.26}$$

where (x, y, z) is the position of the ghost point, and $|\boldsymbol{\nabla}\phi| = 1$ was used. This enables the use of the boundary conditions directly after the flow variables at $\boldsymbol{x}_{IP}$ have been interpolated, as explained in section 2.3.1.

3.4 Marching Cubes Algorithm

The interface can be found from the zero-contour. Hence an algorithm to find contours in the scalar field is needed. The present work utilizes the marching cubes algorithm [32]. The algorithm divides the domain into cubes with the grid points at the corners. The algorithm finds cubes where the values at the corner are on different sides of the reference value, which indicates the interface is crossing between these corners. There are 256 different configurations the interface may cross each cube, but accounting for symmetries, the number of unique configurations reduces to 14. After the algorithm has found which sides are crossed by the interface, the crossing point on the side is found from linear interpolation of the values at the corners. Higher-order interpolation schemes have been tested, but they showed no significant improvements [32]. This procedure is repeated for the rest of the domain. The implementation of the marching cubes algorithm is done with the help of the Python library scikit-image [33].

Chapter 4

Discretization of the Level Set Method

Finite difference schemes are used to solve the level set equation (3.1), the reinitialization equation (3.3), and the advection of the particles (3.19). Different schemes are used to discretize the equations according to the accuracy needed. High accuracy is needed for both the level set equation and the reinitialization equation. The temporal discretization in these equations is approximated with a total variation diminishing (TVD) Runge-Kutta (RK) method, while the spatial derivatives are approximated with a weighted essentially non-oscillating (WENO) scheme and the Godunov scheme for the level set equation and the reinitialization equation respectively, as suggested by Osher and Fedkiw [19]. For the temporal derivative in the advection of the particles in equation (3.19), Euler's scheme is used because it gave good initial results. The velocity in equation (3.19) is found through trilinear interpolation of the given externally created velocity field. The trilinear interpolation is introduced in section 3.2. The descriptions of the WENO scheme, TVD RK scheme, and Godunov's scheme below follows the presentation by Osher and Fedkiw [19].

4.1 WENO Method

The weighted essentially non-oscillating (WENO) scheme is used because it handles discontinuities in the derivatives automatically [19]. These discontinuities can occur at places in the flow where the distance to the interface is equal in more than one direction, causing a kink in the signed distance function. The WENO scheme is fifth-order accurate in smooth regions of the flow and third-order accurate in other parts of the flow.

To approximate ϕ_x in $a_x\phi_x$, where the advection velocity is defined as $a_x = u$ in (3.1) and $\phi_x = \frac{\partial\phi}{\partial x}$, the WENO scheme uses a backward difference for a positive velocity, $a_x > 0$, and a forward difference for a negative velocity, $a_x < 0$. The scheme uses three approximations of ϕ_x defined as

$$\phi_x^1 = \frac{v_1}{3} - \frac{7v_2}{6} + \frac{11v_3}{6}, \tag{4.1}$$

$$\phi_x^2 = -\frac{v_2}{6} + \frac{5v_3}{6} + \frac{v_4}{3}, \tag{4.2}$$

and

$$\phi_x^3 = \frac{v_3}{3} + \frac{5v_4}{6} - \frac{v_5}{6} \tag{4.3}$$

where $v_1 = D^-\phi_{i-2}$, $v_2 = D^-\phi_{i-1}$, $v_3 = D^-\phi_i$, $v_4 = D^-\phi_{i+1}$ and $v_5 = D^-\phi_{i+2}$ is defined for a backward difference, and $v_1 = D^+\phi_{i+2}$, $v_2 = D^+\phi_{i+1}$, $v_3 = D^+\phi_i$, $v_4 = D^+\phi_{i-1}$ and $v_5 = D^+\phi_{i-2}$ for a forward difference, where

$$D^-\phi_i = \frac{\phi_i - \phi_{i-1}}{\Delta x}, \tag{4.4}$$

and

$$D^+\phi_i = \frac{\phi_{i+1} - \phi_i}{\Delta x}. \tag{4.5}$$

The three ϕ_x^i terms are weighted and summed to yield the WENO approximation

$$\phi_x = \omega_1\phi_x^1 + \omega_2\phi_x^2 + \omega_3\phi_x^3, \tag{4.6}$$

using

$$\omega_1 = \frac{\alpha_1}{\alpha_1 + \alpha_2 + \alpha_3}, \tag{4.7}$$

$$\omega_2 = \frac{\alpha_2}{\alpha_1 + \alpha_2 + \alpha_3}, \tag{4.8}$$

and

$$\omega_3 = \frac{\alpha_3}{\alpha_1 + \alpha_2 + \alpha_3} \tag{4.9}$$

as weights. In smooth regions of the flow the optimal weights are observed as $\omega_1 = 0.1$, $\omega_2 = 0.6$, and $\omega_3 = 0.3$. For non-smooth regions of the flow the scheme defines

$$\alpha_1 = \frac{0.1}{(S_1 + \epsilon)^2}, \tag{4.10}$$

$$\alpha_2 = \frac{0.6}{(S_2 + \epsilon)^2}, \tag{4.11}$$

and

$$\alpha_3 = \frac{0.3}{(S_3 + \epsilon)^2} \tag{4.12}$$

utilizing the smoothness indicators

$$S_1 = \frac{13}{12}(v_1 - 2v_2 + v_3)^2 + \frac{1}{4}(v_1 - 4v_2 + 3v_3)^2, \tag{4.13}$$

$$S_2 = \frac{13}{12}(v_2 - 2v_3 + v_4)^2 + \frac{1}{4}(v_2 - v_4)^2, \tag{4.14}$$

and

$$S_3 = \frac{13}{12}(v_3 - 2v_4 + v_5)^2 + \frac{1}{4}(3v_3 - 4v_4 + v_5)^2 \tag{4.15}$$

to estimate the smoothness of ϕ_i. $\epsilon = 10^{-6}\ \max(v_1^2,\ v_2^2,\ v_3^2,\ v_4^2,\ v_5^2) + 10^{-99}$, where the first term is a scaling term and the second term is included to avoid division by zero. The same procedure is followed when approximating ϕ_y in $a_y\phi_y$ and ϕ_z in $a_z\phi_z$, where $a_y = v$ and $a_z = w$ in (3.1), and $\phi_y = \frac{\partial\phi}{\partial y}$ and $\phi_z = \frac{\partial\phi}{\partial z}$.

4.2 TVD Runge-Kutta Method

For the temporal discretization, the third-order accurate total variation diminishing (TVD) Runge-Kutta (RK) method is chosen to achieve third-order accuracy [34].

The first step of the third-order accurate TVD RK is doing a forward Euler step

$$\frac{\phi^{n+1} - \phi^n}{\Delta t} + \boldsymbol{u}^n \cdot \boldsymbol{\nabla}\phi^n = 0, \tag{4.16}$$

followed by a second Euler step

$$\frac{\phi^{n+2} - \phi^{n+1}}{\Delta t} + \boldsymbol{u}^{n+1} \cdot \boldsymbol{\nabla}\phi^{n+1} = 0. \tag{4.17}$$

A weighted average is used

$$\phi^{n+\frac{1}{2}} = \frac{3}{4}\phi^n + \frac{1}{4}\phi^{n+2} \tag{4.18}$$

to obtain an approximation for $t^n + 0.5\Delta t$. A third Euler step is used

$$\frac{\phi^{n+\frac{3}{2}} - \phi^{n+\frac{1}{2}}}{\Delta t} + \boldsymbol{u}^{n+\frac{1}{2}} \cdot \boldsymbol{\nabla}\phi^{n+\frac{1}{2}} = 0 \tag{4.19}$$

followed by a final averaging step

$$\phi^{n+1} = \frac{1}{3}\phi^n + \frac{2}{3}\phi^{n+\frac{3}{2}} \tag{4.20}$$

to obtain an approximation for ϕ^{n+1}.

4.3 Godunov's Scheme

The reinitialization equation (3.3) is discretized with the TVD RK method presented in section 4.2 for the time derivative and the Godunov scheme for the spatial derivative [19].

In the reinitialization equation, the sign function (3.4) is used as the advection velocity. ϕ_x, ϕ_y, and ϕ_z are found by taking the square root of the compact form of Godunov's scheme by Rouy and Tourin [35]

$$\phi_x^2 \approx \begin{cases} \max(\max(\phi_x^-,\, 0)^2, \min(\phi_x^+,\, 0)^2) & \text{when } a_x > 0, \text{ and} \\ \max(\min(\phi_x^-,\, 0)^2, \max(\phi_x^+,\, 0)^2) & \text{when } a_x < 0, \end{cases} \tag{4.21}$$

where $\phi_x^- = D^-\phi$ and $\phi_x^+ = D^+\phi$ as in equations (4.4) and (4.5). The same procedure is followed for ϕ_y^2 and ϕ_z^2.

4.4 CFL Number

Courant-Friedrichs-Lewy (CFL) condition is used in numerical simulations to ensure the stability of the scheme. The CFL condition is enforced by defining the CFL number

$$\text{CFL} = \Delta t \max\left(\frac{|u|}{\Delta x} + \frac{|v|}{\Delta y} + \frac{|w|}{\Delta z}\right), \tag{4.22}$$

where $0 \leq \text{CFL} \leq 1$ usually ensures stability [19]. The maximum in eq. (4.22) is taken over all grid points and all time steps.

4.5 Boundary Conditions

The regular reinitialization of the signed distance field and the fact that the interface is kept far away from the boundary of the domain make it possible to assign a wide range of boundary conditions at the domain's boundary. In the present work, the boundary conditions for the level set equation (3.1) are set by extrapolation. As the WENO scheme uses the three neighboring points on either side of the current node, the extrapolation is done for the three outer nodes at each boundary. The extrapolation is given as

$$\phi_{3,j,k} = \phi_{4,j,k} - (\phi_{5,j,k} - \phi_{4,j,k}), \tag{4.23}$$

$$\phi_{2,j,k} = \phi_{3,j,k} - (\phi_{4,j,k} - \phi_{3,j,k}), \tag{4.24}$$

and

$$\phi_{1,j,k} = \phi_{2,j,k} - (\phi_{3,j,k} - \phi_{2,j,k}) \tag{4.25}$$

in the x-direction,

$$\phi_{i,3,k} = \phi_{i,4,k} - (\phi_{i,5,k} - \phi_{i,4,k}), \tag{4.26}$$

$$\phi_{i,2,k} = \phi_{i,3,k} - (\phi_{i,4,k} - \phi_{i,3,k}), \tag{4.27}$$

and

$$\phi_{i,1,k} = \phi_{i,2,k} - (\phi_{i,3,k} - \phi_{i,2,k}) \tag{4.28}$$

in the y-direction, and

$$\phi_{i,j,3} = \phi_{i,j,4} - (\phi_{i,j,5} - \phi_{i,j,4}), \tag{4.29}$$

$$\phi_{i,j,2} = \phi_{i,j,3} - (\phi_{i,j,4} - \phi_{i,j,3}), \tag{4.30}$$

and

$$\phi_{i,j,1} = \phi_{i,j,2} - (\phi_{i,j,3} - \phi_{i,j,2}) \tag{4.31}$$

in the z-direction, where the 1 index is the outermost grid point for all sides of the domain.

4.6 Measures of Error

Three different error measures are implemented to find the error of the particle level set method. The first method is to find the volume change from the initial scalar field to the scalar field at final time. The volume of the Ω^- region can be found from [19]

$$V = \int_\Omega (1 - \tilde{H}(\phi(\boldsymbol{x})))dx\,dy\,dz, \tag{4.32}$$

where $\tilde{H}(\phi)$ is a smeared-out Heaviside function defined as [19]

$$\tilde{H}(\phi) = \begin{cases} 0 & \text{if } \phi < -\epsilon \\ \frac{1}{2} + \frac{\phi}{2\epsilon} + \frac{1}{2\pi}\sin\left(\frac{\pi\phi}{\epsilon}\right) & \text{if } -\epsilon \leq \phi \leq \epsilon \\ 1 & \text{if } \epsilon < \phi, \end{cases} \tag{4.33}$$

where ϵ is the bandwidth of the numerical smearing, and is set to $\epsilon = 1.5\Delta x$.

The discrete version of equation (4.32) is

$$V = \sum_{i=1,\, j=1,\, k=1}^{m} (1 - \tilde{H}(\phi_{i,j,k}))\,\Delta x\,\Delta y\,\Delta z, \tag{4.34}$$

where $_{i=1,\, j=1,\, k=1}$ denote the first indices in each direction, and m grid points in each dimension are assumed.

The interface error and average volume error introduced in [36] are also used to measure the error. Both of these error estimates use another variant of the Heaviside function where $H(x) \equiv 1$ for $x < 0$, and $H(x) \equiv 0$ otherwise. The interface error is slightly adapted for 3D and is defined as

$$I_{error} = \frac{1}{A}\int_\Omega |H(\phi_{expect}) - H(\phi_{compute})|\,dx\,dy\,dz, \tag{4.35}$$

where ϕ_{expect} is the initial signed distance field for the test case, $\phi_{compute}$ is the signed distance field at the time the error is measured, A is the surface area at the initial time, and Ω is the entire domain. The surface area can be found from [19]

$$A = \int_{\Omega} \delta(\phi(\boldsymbol{x}))|\nabla\phi(\boldsymbol{x})|\, dx\, dy\, dz, \tag{4.36}$$

where $\delta(\phi)$ is defined as

$$\delta(\phi) = \begin{cases} 0 & \text{if } \phi < -\epsilon \\ \frac{1}{2\epsilon} + \frac{1}{2\epsilon}\cos\left(\frac{\pi\phi}{\epsilon}\right) & \text{if } -\epsilon \leq \phi \leq \epsilon \\ 0 & \text{if } \epsilon < \phi, \end{cases} \tag{4.37}$$

The discretized form of (4.35) can be written as

$$\text{Interface error} = \sum_{i=1,j=1,k=1}^{m} \frac{1}{A}|H(\phi_{expect,\,ijk}) - H(\phi_{compute,\,ijk})|\Delta x\, \Delta y\, \Delta z, \tag{4.38}$$

where $H(\phi_{ijk})$ is the Heaviside function applied to ϕ_{ijk}, and (4.36) is discretized as

$$A = \sum_{i=1,\,j=1,\,k=1}^{m} \delta(\phi_{i,j,k}) \left| \sqrt{\left(\frac{\partial\phi}{\partial x}\right)^2 + \left(\frac{\partial\phi}{\partial y}\right)^2 + \left(\frac{\partial\phi}{\partial z}\right)^2} \right| \Delta x\, \Delta y\, \Delta z, \tag{4.39}$$

and $\frac{\partial\phi}{\partial \boldsymbol{x}}$, $\frac{\partial\phi}{\partial \boldsymbol{y}}$ and $\frac{\partial\phi}{\partial \boldsymbol{z}}$ are discretized with second-order central differencing.

The average volume error is defined as

$$M_{error} = \int_{t=0}^{t_f} \frac{|M(t) - M(0)|}{t_f} dt \tag{4.40}$$

where t_f is the time the error is measured, and

$$M(t) = \int_{\Omega} |H(\phi(x, y, z, t)))|dx\, dy\, dz. \tag{4.41}$$

The discretized form of equation (4.41) can be written as

$$M^n = \sum_{i=1,j=1,k=1}^{m} |H(\phi)^n_{ijk}|\Delta x\, \Delta y\, \Delta z, \tag{4.42}$$

where n is the current time step. The discretized version (4.40) follows as

$$\text{Average volume error} = \sum_{n=1}^{t_f} \frac{|M^n - M^0|}{t_f} \Delta t. \tag{4.43}$$

The order of convergence is shown in Table 5.2, and is defined as

$$p = \frac{\ln(\text{err}(\Delta x)/\text{err}(\Delta x/2))}{\ln\,(2)} \tag{4.44}$$

where $\text{err}(\Delta x)$ is the error at grid spacing Δx.

Chapter 5

Results

The simulations in the present work are done at a CFL number of 0.9, which ensures the stability of the numerical methods. The time step is then found from eq. (4.22). The tests are conducted on an $n \times n \times n$ grid, where n is the number of grid cells in each spatial direction. To see the order of convergence of the method, the grid size is halved for each simulation, where the coarsest grid uses $n = 50$, and the finest uses $n = 400$. The reinitialization equation (3.3) is solved every time step with 5 pseudo time steps. The pseudo time step size is set to $\Delta\tau = 0.5\Delta x$ as suggested in [19]. Particle reseeding is done every 100 time steps. These parameters are kept constant and were chosen because they seemed to give the best results for the test case explained in section 5.2. However, other parameters may show better results depending on the test case. During simple testing with a sphere advected with $u = 1$, $v = 1$, and $w = 1$, less frequent reinitialization also showed good results, indicating more reinitialization may be needed in velocity fields with large deformations.

5.1 Sphere in two-dimensional vortex velocity field

This test case consists of a sphere of radius 0.15 in a unit cube of $[0, 1]\times[0, 1]\times[0, 1]$, where the center is initially placed at (0.5, 0.75, 0.5), with the signed distance with reference to the sphere surface as the initial condition for the scalar field as explained in section 3.1. The initial sphere is shown in Figure 5.1. The externally created velocity field was introduced by Morgan and Waltz [37] and is given by

$$u = \sin(\pi x)\cos(\pi y)\cos(\pi t/T), \tag{5.1}$$

$$v = -\cos(\pi x)\sin(\pi y)\cos(\pi t/T), \tag{5.2}$$

and

$$w = 0, \tag{5.3}$$

where $T = 10$ and is the total time of the deformation. The velocity field ensures the initial sphere is deformed in the $x - y$ plane. The cosine term ensures the velocity field is reversed after $t = 1/2\,T$, and the interface is deformed back to the initial sphere at $t = T$. The maximum deformations at $t = 1/2\,T$ in the $x - y$ and $y - z$ planes for $n = 400$ are shown in Figure 5.2. As the initial and final interface are supposed to be identical, the error can be found from the measures introduced in section 4.6.

Figure 5.3 shows the first and final time step of the deformation at four different grid sizes. The approximation of the initial volume is best for the finest grid. The error measures in Table 5.1 also show decreasing errors for increasing number of grid nodes. The maximum deformations at $t = 1/2\,T$ for $n = 50$, $n = 100$, and $n = 200$ are shown in Figure 5.4. All grids manage to keep the interface without creating droplets that separate from the main structure.

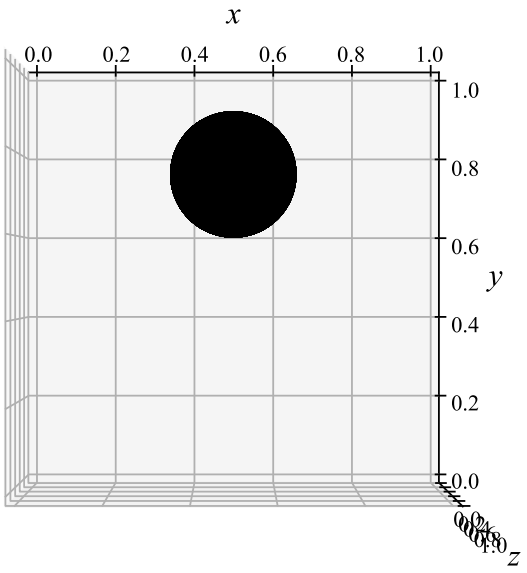
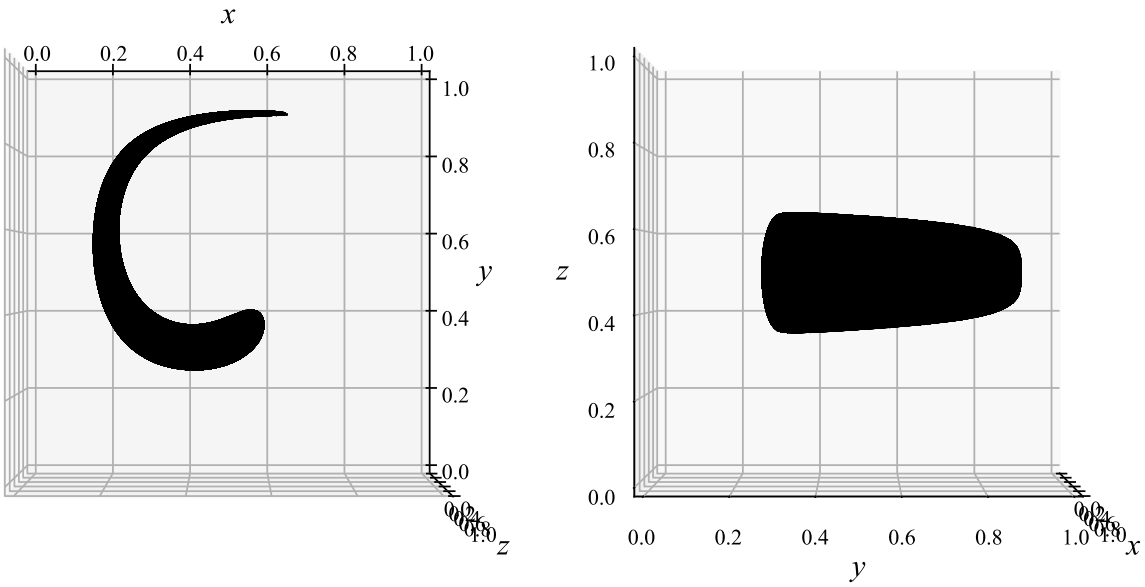


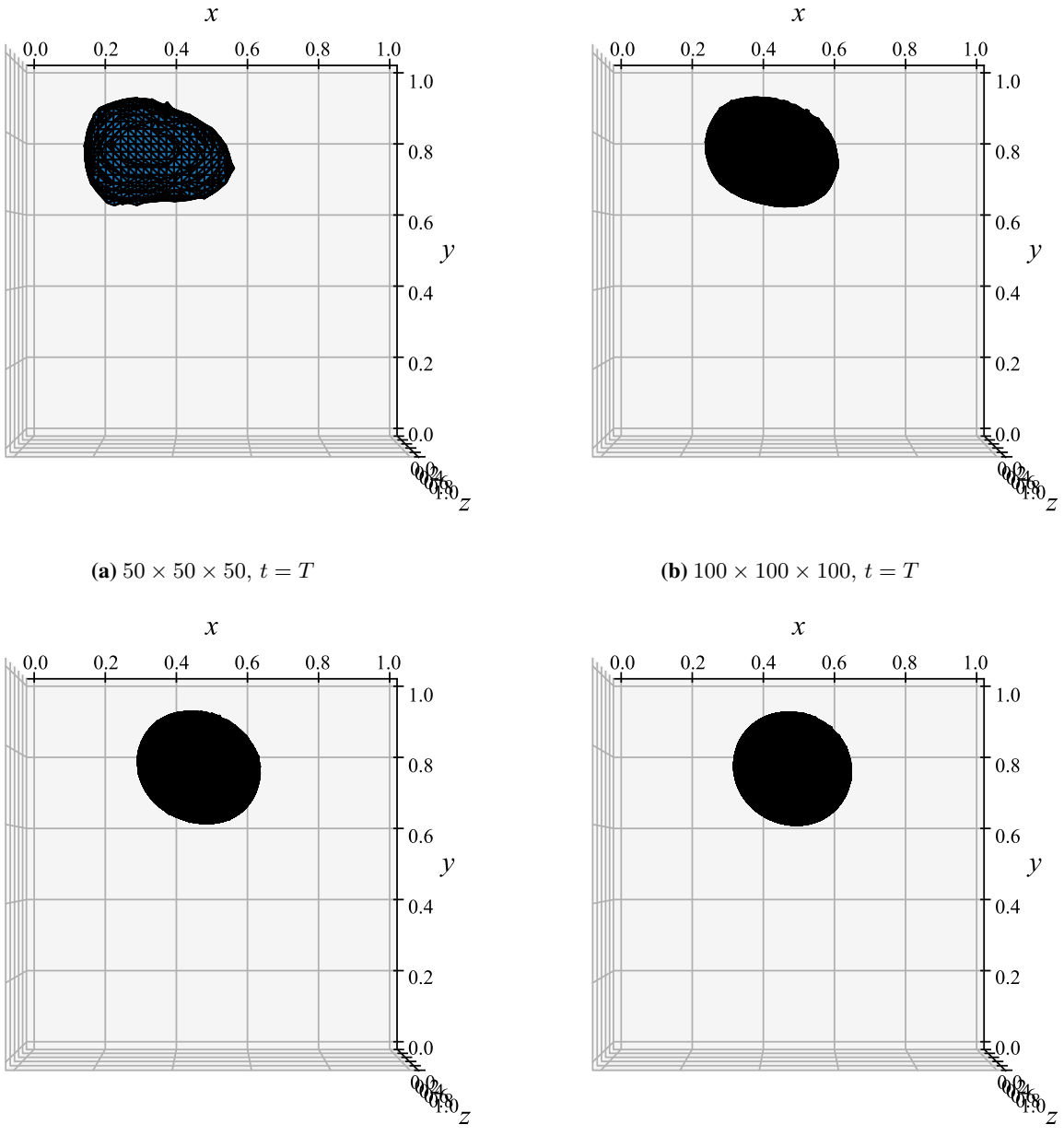
Figure 5.1: Initial sphere in two-dimensional velocity field in the $x-y$ plane for a $400 \times 400 \times 400$ grid. The same shape should be found at $t = T$.



(a) $x - y$ plane

(b) $y - z$ plane

Figure 5.2: Domains where $\phi(\boldsymbol{x}, t) \leq 0$ at $t = 1/2\,T$ for the sphere in the two-dimensional velocity field with $n = 400$.



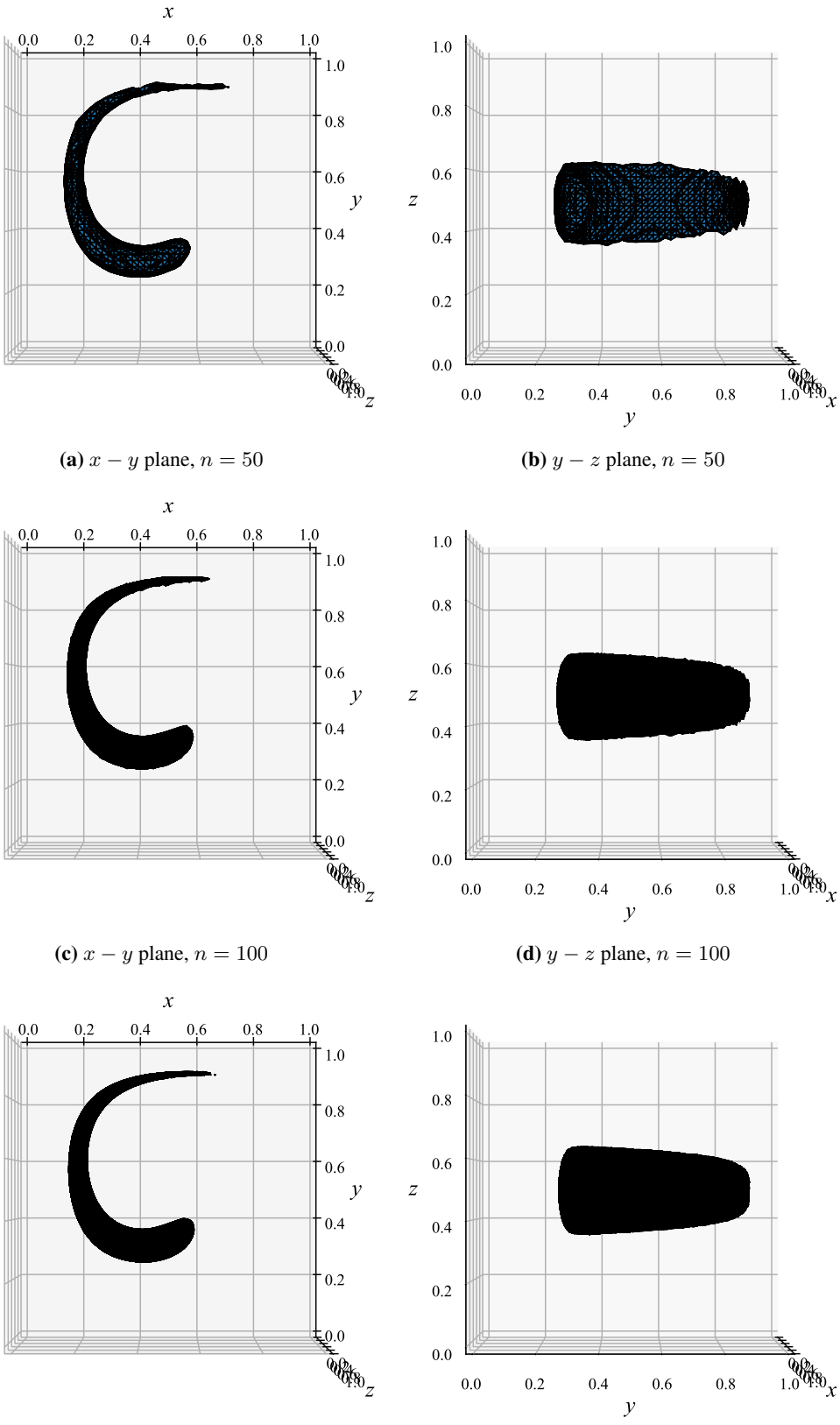
(a) $50 \times 50 \times 50$, $t = T$

(b) $100 \times 100 \times 100$, $t = T$

(c) $200 \times 200 \times 200$, $t = T$

(d) $400 \times 400 \times 400$, $t = T$

Figure 5.3: Domains where $\phi(\boldsymbol{x}, T) \leq 0$ for the sphere in the two-dimensional velocity field at four different grid resolutions in the $x - y$ plane.



(a) $x - y$ plane, $n = 50$

(b) $y - z$ plane, $n = 50$

(c) $x - y$ plane, $n = 100$

(d) $y - z$ plane, $n = 100$

(e) $x - y$ plane, $n = 200$

(f) $y - z$ plane, $n = 200$

Figure 5.4: Domains where $\phi(\boldsymbol{x}, t) \leq 0$ at $t = 1/2\,T$ for the sphere in the two-dimensional velocity field with $n = 50$, $n = 100$, and $n = 200$.

Table 5.1: Error for two-dimensional vortex test. Procedure is explained in section 4.6.

n	Interface error	p	Average volume error	p	Volume change [%]
50	6.84E-2		1.12E-3		23.77
		0.95		0.59	
100	3.54E-2		7.40E-4		13.95
		1.01		0.86	
200	1.76E-2		4.06E-4		7.44
		1.02		0.93	
400	8.67E-3		2.13E-4		3.82

The order of convergence for the sphere in the two-dimensional vortex test is shown in Table 5.1. The order of convergence for the interface error, eq. (4.35), is stable at around 1.0, while the order of convergence for the average volume error, eq (4.40), varies. The order of convergence for the average volume error starts low and increases towards 1.0 as the grids are refined. The volume is overshot for all the grid sizes but decreases towards the initial volume as more grid nodes are used. Morgan et al. [37] saw first-order convergence for the same test case.

5.2 Sphere in three-dimensional vortex velocity field

The test case was introduced by LeVeque [38] and consists of a sphere of radius 0.15 in a unit cube of $[0, 1] \times [0, 1] \times [0, 1]$, where the center is initially placed at (0.35, 0.35, 0.35). The initial condition for the scalar field is the signed distance with reference to the sphere surface as explained in section 3.1. The velocity field is given by

$$u = 2\,\sin^2(\pi x)\sin(2\pi y)\sin(2\pi z)\cos(\pi t/T), \tag{5.4}$$

$$v = -\sin(2\pi x)\sin^2(\pi y)\sin(2\pi z)\cos(\pi t/T), \tag{5.5}$$

and

$$w = -\sin(2\pi x)\sin(2\pi y)\sin^2(\pi z)\cos(\pi t/T), \tag{5.6}$$

where $T = 3$. The velocity field deforms the initial sphere seen in Figure 5.5 in both the $x - y$ plane and the $x - z$ plane. The cosine term ensures the velocity field is reversed after $t = 1/2\,T$, and the interface is deformed back to the initial sphere at $t = T$. The maximum deformations in the $x - y$ and $x - z$ planes for $n = 400$ are shown in Figure 5.6. As the initial and final interface are supposed to be identical, the error can be found from the measures introduced in section 4.6.

Figure 5.7 shows the deformation's first and final time step at four different grid sizes. The finer grids clearly show a better match with the initial interface. Compared with the error measures in Table 5.2, this is also true for both the interface error and the average volume error, as they both decrease at finer grids. However, that is not the case for the volume change, as the coarsest grid shows the second best volume conservation. Although, by looking at Figure 5.7(a), the shape of the interface is far off compared to the finer grids.

Figure 5.8 shows the maximum deformation in both the $x - y$ and $x - z$ planes for $n = 50$, $n = 100$, and $n = 200$. It is evident that there are less separated droplets from the main structures for the finer grids at $t = 1/2\,T$ when the interface is stretched. None of these droplets are shown in Figure 5.6, showing the deformations at $t = 1/2\,T$ for $n = 400$.

The order of convergence is given in Table 5.2. For the interface error, the order of convergence is approximately stable at around 1.0. The order of convergence for the average volume error is slightly more changing but seems to stabilize around 1.5. The relatively large difference in order of convergence between $n = 50$ and $n = 100$ to $n = 100$ and $n = 200$ may be explained by $n = 50$ being too coarse to give a good approximation of the interface.

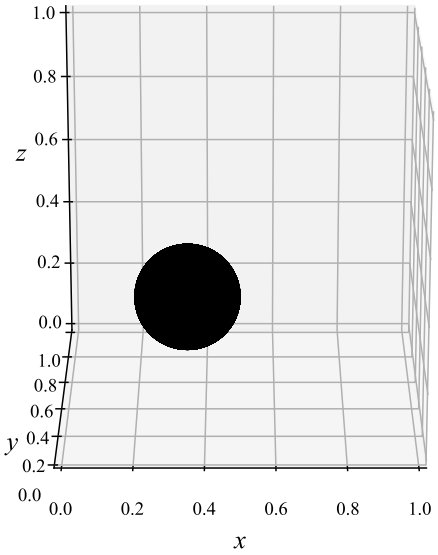
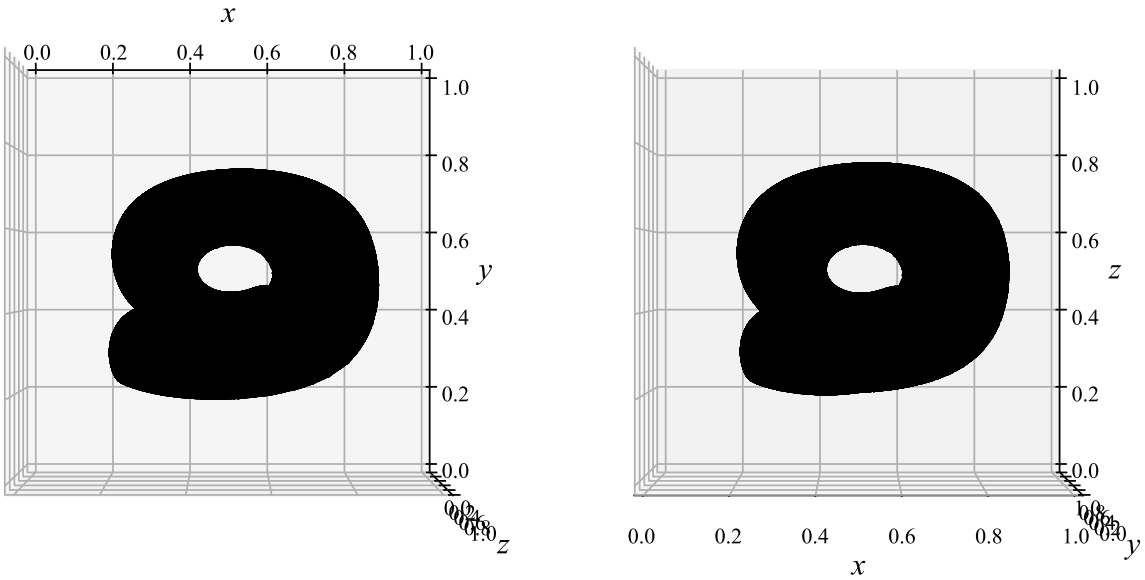


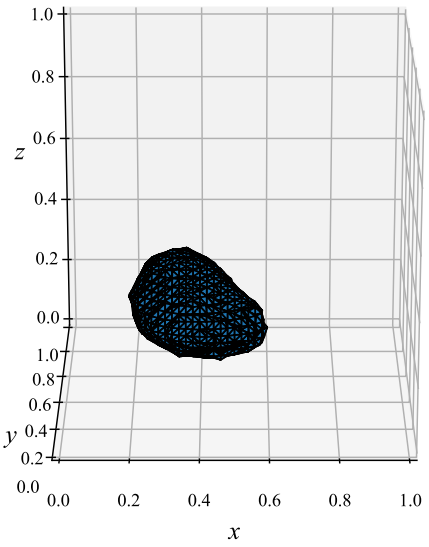
Figure 5.5: Initial sphere in three-dimensional velocity field for a $400 \times 400 \times 400$ grid. The same shape should be found at $t = T$.



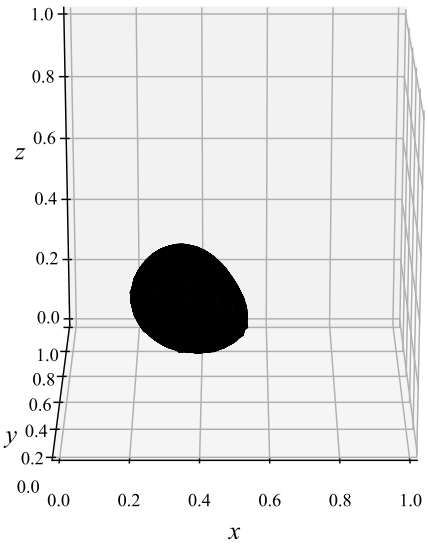
(a) $x - y$ plane

(b) $x - z$ plane

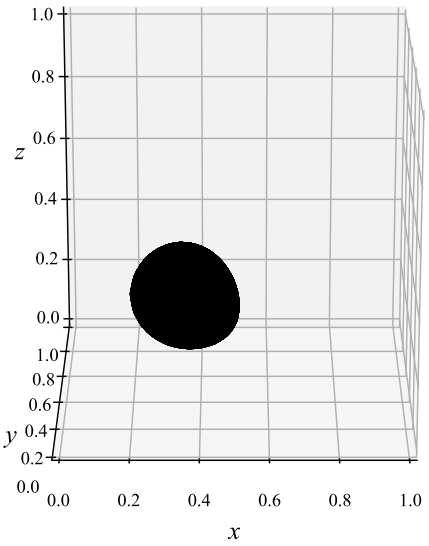
Figure 5.6: Domains where $\phi(\boldsymbol{x}, t) \leq 0$ at $t = 1/2\,T$ for the sphere in the three-dimensional velocity field with $n = 400$.



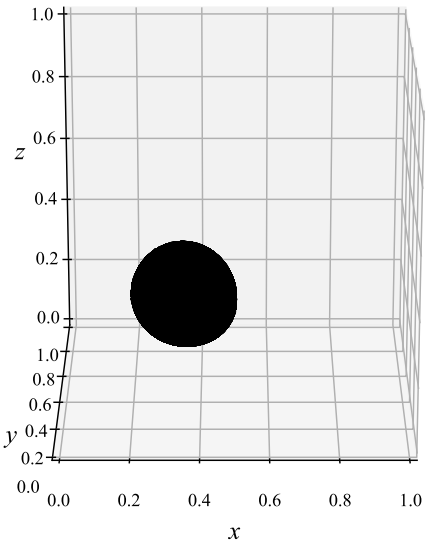
(a) $50 \times 50 \times 50$, $t = T$



(b) $100 \times 100 \times 100$, $t = T$

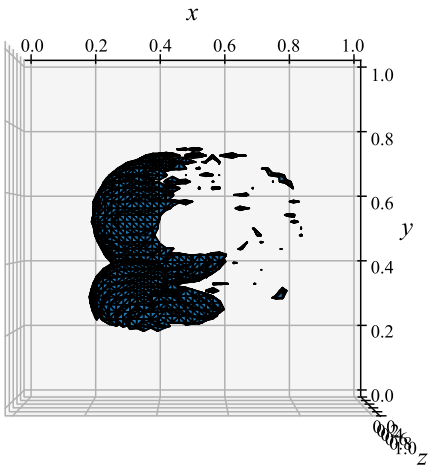


(c) $200 \times 200 \times 200$, $t = T$

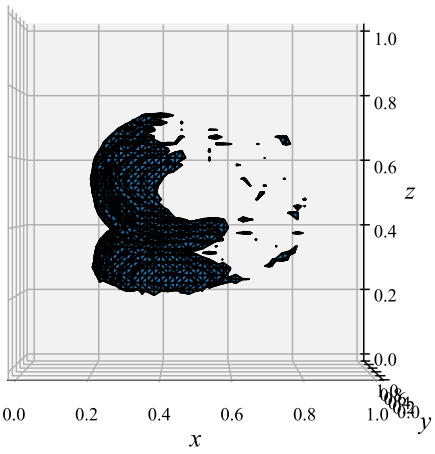


(d) $400 \times 400 \times 400$, $t = T$

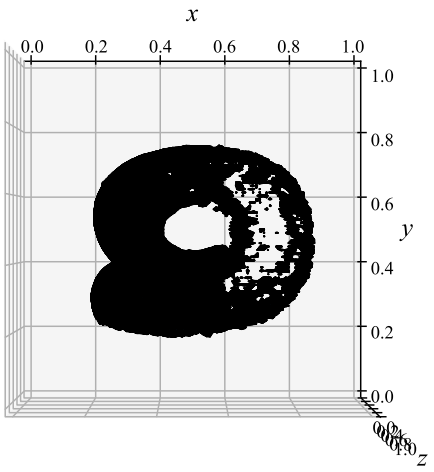
Figure 5.7: Domains where $\phi(\boldsymbol{x}, T) \leq 0$ for the sphere in the three-dimensional velocity field at four different grid resolutions.



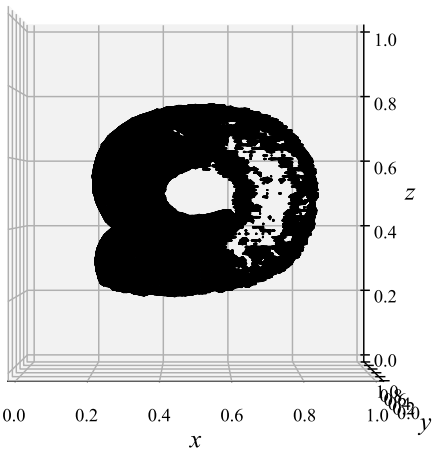
(a) $x - y$ plane, $n = 50$



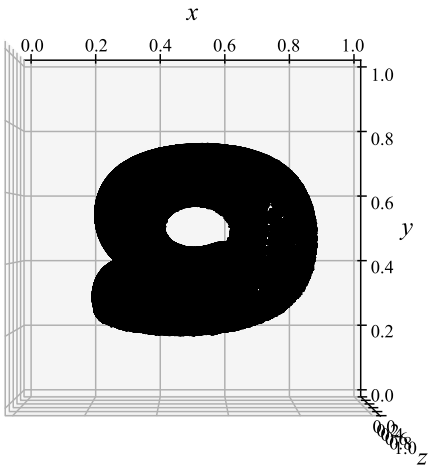
(b) $x - z$ plane, $n = 50$



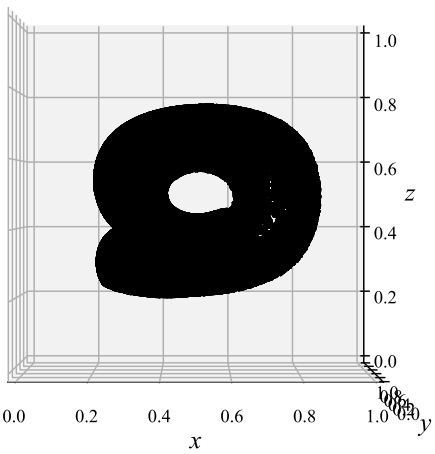
(c) $x - y$ plane, $n = 100$



(d) $x - z$ plane, $n = 100$



(e) $x - y$ plane, $n = 200$



(f) $x - z$ plane, $n = 200$

Figure 5.8: Domains where $\phi(\boldsymbol{x}, t) \leq 0$ at $t = 1/2\,T$ for the sphere in the three-dimensional velocity field with $n = 50$, $n = 100$, and $n = 200$.

Table 5.2: Error for three-dimensional vortex test. Procedure is explained in section 4.6.

n	Interface error	p	Average volume error	p	Volume change [%]
50	2.82E-2		2.02E-3		-0.73
		0.97		1.88	
100	1.43E-2		5.50E-4		1.43
		1.01		1.46	
200	7.11E-3		1.99E-4		1.06
		1.02		1.51	
400	3.50E-3		7.00E-5		0.62

The order of convergence is at the same size as for Enright et al. [26], who observed an order of convergence in the region between 1 and 1.5. Note that the order of convergence calculated for Enright et al. [26] is for the two-dimensional vortex test, and no order of convergence was calculated for the three-dimensional test case. First-order convergence was also seen in the two-dimensional standard level set method implementation by Sommersel [9]. The same test case was also run for the standard level set method at grids up to the size of $n = 400$, where the entire volume was lost before $t = 1/2\,T$. The CPU times for the tests with the standard level set method are shown in Table 5.3(c). The CPU times for the particle level set method in Table 5.3(b) shows an increase of 70.5 %, 42.6 %, 19.0 %, and 24.5 %, compared to the standard level set method in Table 5.3(c) with grid sizes of $n = 50$, $n = 100$, $n = 200$, and $n = 400$ respectively. Considering the massive improvement in mass conservation, extending the standard level set method to the particle level set method is worth the extra work.

5.3 CPU time

All tests are run on a computer with a clock rate of 1.4 GHz, with an 11th Gen Intel Core i7-11700T processor with 16 CPUs and 32 GB RAM. The elapsed time and number of time steps for each run are given in Table 5.3. Further optimization of the code is possible and may lead to shorter tun times. The CPU times presented in Table 5.3 are meant as a pointer for the code as it is, and changing the parameters explained at the start of chapter 5 may lead to better results and shorter run times.

Table 5.3: CPU time and number of time steps for all test cases at different grid sizes.

(a) Two-dimensional vortex test with the particle level set method.

n	CPU time [s]	Number of time steps
50	65	546
100	921	1100
200	14078	2212
400	224277	4434

(b) Three-dimensional vortex test with the particle level set method.

n	CPU time [s]	Number of time steps
50	75	404
100	1038	818
200	13865	1644
400	223832	3298

(c) Three-dimensional vortex test with the standard level set method.

n	CPU time [s]	Number of time steps
50	44	404
100	728	818
200	11655	1644
400	179808	3298

Chapter 6

Conclusions and Outlook

The three-dimensional particle level set method has been implemented, and the possibility of using this approach in the VirtuOSA project has been investigated. The conclusion is that the level set method can track the fluid-solid interface in the upper airways for OSA patients with the GPIBM, as is needed in the VirtuOSA project. The properties of the signed distance function explained in section 3.3 are a big bonus when computing the values at the ghost points to satisfy the correct boundary conditions. These properties are the main reasons the level set method has been preferred over methods like the volume of fluid method and front tracking method in the present study. Implementing the particle level set method has significantly improved the poor mass conservation of the standard level set method. By comparing the the CPU times in Table 5.3(b) and Table 5.3(c) an average of 21.8 % longer CPU times are seen for the two finest grids. Considering that the standard level set method lost all volume for the finest grid with $n = 400$, while the particle level set method saw a volume change of 23.77 % for the coarsest grid with $n = 50$, this extra CPU time is worthwhile.

An order of convergence in the region between 0.9 and 1.5 is seen, which is low compared to the WENO method, which is fifth-order accurate in smooth regions and at least third-order accurate elsewhere. The reason behind the low order for the particle level set method may be the low order method used to advect the particle advection equation (3.19), or low order in the implemented trilinear interpolation method introduced in section 3.2. Experimenting with different alternatives to these methods may be the focus of future work and may lead to an increase in the order of convergence. The possibility of combining the particle level set method with other methods, i.e., the different modifications to the reinitialization equation as discussed in section 2.2.3, may also increase the order of the method even further.

The low order of convergence is slightly worrying. Lower-order methods may give the same results and be computationally cheaper. Further code optimization is possible and will be a case to consider before possible implementation in the VirtuOSA project. Changing when particles are reseeded for the particle level set method may be more efficient and give better results. Nevertheless, the results from the particle level set method are a significant improvement compared to the standard level set method. The easy implementation with the GPIBM shows that the particle level set method is a candidate to consider when choosing the final front tracking method.

Bibliography

[1] Terry Young, Mari Palta, Jerome Dempsey, James Skatrud, Steven Weber and Safwan Badr. 'The Occurrence of Sleep-Disordered Breathing among Middle-Aged Adults'. In: *New England Journal of Medicine* 328.17 (1993). PMID: 8464434, pp. 1230–1235. DOI: 10.1056/NEJM199304293281704.

[2] Amy S. Jordan, David G. McSharry and Atul Malhotra. 'Adult obstructive sleep apnoea'. In: *The Lancet* 383.9918 (2014), pp. 736–747. ISSN: 0140-6736. DOI: 10.1016/S0140-6736(13)60734-5.

[3] Adam V. Benjafield, Najib T. Ayas, Peter R. Eastwood, Raphael Heinzer, Mary S. M. Ip, Mary J. Morrell, Carlos M. Nunez, Sanjay R. Patel, Thomas Penzel, Jean-Louis Pépin, Paul E. Peppard, Sanjeev Sinha, Sergio Tufik, Kate Valentine and Atul Malhotra. 'Estimation of the global prevalence and burden of obstructive sleep apnoea: a literature-based analysis'. In: *The Lancet Respiratory Medicine* 7.8 (2019), pp. 687–698. ISSN: 2213-2600. DOI: 10.1016/S2213-2600(19)30198-5.

[4] T. Douglas Bradley and John S. Floras. 'Obstructive sleep apnoea and its cardiovascular consequences'. In: *The Lancet* 373.9657 (2009), pp. 82–93. ISSN: 0140-6736. DOI: 10.1016/S0140-6736(08)61622-0.

[5] Lucia Spicuzza, Daniela Caruso and Giuseppe Di Maria. 'Obstructive sleep apnoea syndrome and its management'. In: *Therapeutic Advances in Chronic Disease* 6.5 (2015). PMID: 26336596, pp. 273–285. DOI: 10.1177/2040622315590318.

[6] Mads Henrik Strand Moxness. *The Influence of the Nasal Airway in Obstructive Sleep Apnea*. PhD thesis, NTNU, June 2018. URL: https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/2560447.

[7] Blausen.com staff. 'Medical gallery of Blausen Medical 2014'. In: *WikiJournal of Medicine 1 (2)* (2014). ISSN: 2002-4436. DOI: 10.15347/wjm/2014.010.

[8] Sverre Gullikstad Johnsen. *VirtuOSA - Virtuell Kirurgi i de Øvre Luftveiene – Nye Løsninger for Behandling av Obstruktiv Søvnapne*. SINTEF, Jan. 2021. URL: https://www.sintef.no/prosjekter/2020/virtuosa/.

[9] Marcus Sommersel. *Interface Tracking for Immersed Boundary Method in Biofluid Dynamics*. Project Work, NTNU, 2021.

[10] Luke Reid. 'An Introduction to Biomedical Computational Fluid Dynamics'. In: *Biomedical Visualisation: Volume 10*. Ed. by Paul M. Rea. Cham: Springer International Publishing, 2021, pp. 205–222. ISBN: 978-3-030-76951-2. DOI: 10.1007/978-3-030-76951-2_10.

[11] Paul D. Morris, Andrew Narracott, Hendrik von Tengg-Kobligk, Daniel Alejandro Silva Soto, Sarah Hsiao, Angela Lungu, Paul Evans, Neil W. Bressloff, Patricia V. Lawford, D. Rodney Hose and Julian P. Gunn. 'Computational fluid dynamics modelling in cardiovascular medicine'. In: *Heart* 102.1 (2016), pp. 18–28. ISSN: 1355-6037. DOI: 10.1136/heartjnl-2015-308044.

[12] S. S. Khalafvand, E. Y. K. Ng and L. Zhong. 'CFD simulation of flow through heart: a perspective review'. In: *Computer Methods in Biomechanics and Biomedical Engineering* 14.1 (2011). PMID: 21271418, pp. 113–132. DOI: 10.1080/10255842.2010.493515.

[13] Fotis Sotiropoulos and Xiaolei Yang. 'Immersed boundary methods for simulating fluid–structure interaction'. In: *Progress in Aerospace Sciences* 65 (2014), pp. 1–21. ISSN: 0376-0421. DOI: 10.1016/j.paerosci.2013.09.003.

[14] Richtmyer and Morton. *Difference Methods for Initial Value Problems*. Interscience Publishers, 1967.

[15] J. Glimm, E. Isaacson, D. Marchesin and O. McBryan. 'Front tracking for hyperbolic systems'. In: *Advances in Applied Mathematics* 2.1 (1981), pp. 91–119. ISSN: 0196-8858. DOI: 10.1016/0196-8858(81)90040-3.

[16] Grétar Tryggvason, Ruben Scardovelli and Stéphane Zaleski. *Direct Numerical Simulations of Gas–Liquid Multiphase Flows*. Cambridge University Press, 2011. DOI: 10.1017/CBO9780511975264.

[17] C. W. Hirt and B. D. Nichols. 'Volume of fluid (VOF) method for the dynamics of free boundaries'. In: *Journal of Computational Physics* 39.1 (1981), pp. 201–225. ISSN: 0021-9991. DOI: 10.1016/0021-9991(81)90145-5.

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

[19] Stanley Osher and Ronald Fedkiw. *The Level Set Methods and Dynamic Implicit Surfaces*. Springer, 2003. ISBN: 978-0-387-95482-0. DOI: 10.1007/b98879.

[20] Mark Sussman, Peter Smereka and Stanley Osher. 'A Level Set Approach for Computing Solutions to Incompressible Two-Phase Flow'. In: *Journal of Computational Physics* 114.1 (1994), pp. 146–159. ISSN: 0021-9991. DOI: 10.1006/jcph.1994.1155.

[21] J. A. Sethian. *Level set methods and fast marching methods*. Cambridge University Press, 1999.

[22] Frederic Gibou, Ronald Fedkiw and Stanley Osher. 'A review of level-set methods and some recent applications'. In: *Journal of Computational Physics* 353 (2018), pp. 82–109. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2017.10.006.

[23] Giovanni Russo and Peter Smereka. 'A Remark on Computing Distance Functions'. In: *Journal of Computational Physics* 163.1 (2000), pp. 51–67. ISSN: 0021-9991. DOI: 10.1006/jcph.2000.6553.

[24] Antoine du Chéné, Chohong Min and Frédéric Gibou. 'Second-Order Accurate Computation of Curvatures in a Level Set Framework Using Novel High-Order Reinitialization Schemes'. In: *Journal of Scientific Computing* 35 (2008), pp. 114–131. DOI: 10.1007/s10915-007-9177-1.

[25] Daniel Hartmann, Matthias Meinke and Wolfgang Schröder. 'The constrained reinitialization equation for level set methods'. In: *Journal of Computational Physics* 229.5 (2010), pp. 1514–1535. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2009.10.042.

[26] Douglas Enright, Ronald Fedkiw, Joel Ferziger and Ian Mitchell. 'A Hybrid Particle Level Set Method for Improved Interface Capturing'. In: *Journal of Computational Physics* 183.1 (2002), pp. 83–116. ISSN: 0021-9991. DOI: 10.1006/jcph.2002.7166.

[27] Mark Sussman and Elbridge Gerry Puckett. 'A Coupled Level Set and Volume-of-Fluid Method for Computing 3D and Axisymmetric Incompressible Two-Phase Flows'. In: *Journal of Computational Physics* 162.2 (2000), pp. 301–337. ISSN: 0021-9991. DOI: 10.1006/jcph.2000.6537.

[28] Xiaofeng Yang, Ashley J. James, John Lowengrub, Xiaoming Zheng and Vittorio Cristini. 'An adaptive coupled level-set/volume-of-fluid interface capturing method for unstructured triangular grids'. In: *Journal of Computational Physics* 217.2 (2006), pp. 364–394. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2006.01.007.

[29] Charles S. Peskin. 'Flow patterns around heart valves: A numerical method'. In: *Journal of Computational Physics* 10.2 (1972), pp. 252–271. ISSN: 0021-9991. DOI: 10.1016/0021-9991(72)90065-4.

[30] Yu-Heng Tseng and Joel H. Ferziger. 'A ghost-cell immersed boundary method for flow in complex geometry'. In: *Journal of Computational Physics* 192.2 (2003), pp. 593–623. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2003.07.024.

[31] R. Mittal, H. Dong, M. Bozkurttas, F. M. Najjar, A. Vargas and A. von Loebbecke. 'A versatile sharp interface immersed boundary method for incompressible flows with complex boundaries'. In: *Journal of Computational Physics* 227.10 (2008), pp. 4825–4852. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2008.01.028.

[32] William E. Lorensen and Harvey E. Cline. 'Marching Cubes: A High Resolution 3D Surface Construction Algorithm'. In: *Computer Graphics* 21 (Aug. 1987), pp. 163–169. DOI: 10.1145/37401.37422.

[33] Stéfan van der Walt, Johannes L. Schönberger, Juan Nunez-Iglesias, François Boulogne, Joshua D. Warner, Neil Yager, Emmanuelle Gouillart, Tony Yu and the scikit-image contributors. 'scikit-image: image processing in Python'. In: *PeerJ* 2 (June 2014), e453. ISSN: 2167-8359. DOI: 10.7717/peerj.453.

[34] Chi-Wang Shu and Stanley Osher. 'Efficient implementation of essentially non-oscillatory shock-capturing schemes'. In: *Journal of Computational Physics* 77.2 (1988), pp. 439–471. ISSN: 0021-9991. DOI: 10.1016/0021-9991(88)90177-5.

[35] Elisabeth Rouy and Agnès Tourin. 'A Viscosity Solutions Approach to Shape-From-Shading'. In: *SIAM Journal on Numerical Analysis* 29.3 (1992), pp. 867–884. ISSN: 00361429. URL: http://www.jstor.org/stable/2158283.

[36] Mark Sussman and Emad Fatemi. 'An Efficient, Interface-Preserving Level Set Redistancing Algorithm and Its Application to Interfacial Incompressible Fluid Flow'. In: *SIAM Journal on Scientific Computing* 20.4 (1999), pp. 1165–1191. DOI: 10.1137/S1064827596298245.

[37] Nathaniel R. Morgan and Jacob I. Waltz. '3D level set methods for evolving fronts on tetrahedral meshes with adaptive mesh refinement'. In: *Journal of Computational Physics* 336 (2017), pp. 492–512. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2017.02.030.

[38] Randall J. LeVeque. 'High-Resolution Conservative Algorithms for Advection in Incompressible Flow'. In: *SIAM Journal on Numerical Analysis* 33.2 (1996), pp. 627–665. DOI: 10.1137/0733033.

Appendices

Appendix A

LSM3D

The following appendix includes code for the implemented particle level set method. The main computations are programmed in C++, while the post-processing is programmed in Python. The complete code is also found on GitHub at `https://github.com/marcussommersel/LSM3D`.

A.1 main.cpp

This file contains the main structure of the particle level set method and reinitialization. The computation of the different measures of error from section 4.6 are also included.

```cpp
#include <vector>
#include <iostream>
#include <ctime>
#include <chrono>
#include <fstream>
#include "initialization.h"
#include "schemes.h"
#include "vectorUtilities.h"
#include "particleLSM.h"
#include "testCases.h"
using namespace std;

int main(){

    // start of setup.
    chrono::steady_clock::time_point startTime = chrono::steady_clock::now();

    cout << "Program start." << endl;

    // grid nodes in each spatial direction
    const int m = 200; // x-direction
    const int n = 200; // y-direction
    const int p = 200; // z-direction

    // size of domain
    const double xStart = 0.0;
    const double xEnd = 1.0;
    const double yStart = 0.0;
    const double yEnd = 1.0;
    const double zStart = 0.0;
    const double zEnd = 1.0;

    // vector of nodes in each direction
    vector<double> x = linspace(xStart, xEnd, m);
    vector<double> y = linspace(yStart, yEnd, n);
    vector<double> z = linspace(zStart, zEnd, p);

    // grid spacing in each direction
    double dx = x[1] - x[0];
    double dy = y[1] - y[0];
    double dz = z[1] - z[0];

    // varius parameters that may be changed
    double dtau = 0.5*dx; // size of pseudo-time step
    bool doReinit = true; // true if reinitialization should be done
    bool doParticle = false; // true if particles should be used for level set method
    bool saveParticles = false; // true if particles should be saved for plotting
    int nParticles = 64; // number of particles of each type (positive and negative)
    in each cell
    int reinitFreq = 1; // how often reinitialization should be conducted
    int reinitSteps = 5; // how many pseudo-time steps should be done
    int plotFreq = 20000; // how often signed distance field is saved, not including
    first and last time step
    int reseedFreq = 100; // how often particles are reseeded
    int itmax = 20000; // maximum number of iterations
    double CFL = 0.9; // CFL-number
    bool halfplot = true; // true if signed distance field should be saved at 0.5 t/T

    // The three implemented test cases
    string testcase = "vortex";
    // string testcase = "sheared";
    // string testcase = "simple";
    string savePath = "figures/";

    Point c;
    double r;
    double T;
    if (testcase == "vortex"){
        c = Point(0.35,0.35,0.35);
        r = 0.15;
        T = 3.0;
    } else if (testcase == "sheared"){
        c = Point(0.5,0.75,0.5);
```

```
72          r = 0.15;
73          T = 10.0;
74      } else if (testcase == "simple"){
75          c = Point(0.35,0.35,0.35);
76          r = 0.15;
77          T = 0.4;
78      }
79
80      vector<double> phi;
81
82      // initial signed distance field
83      signedDistanceField(phi, x, y, z, r, c, m, n, p);
84
85      // measures of error
86      double initialVolume = volume(phi, dx, dy, dz);
87      vector<double> phi0 = phi;
88      double MError0 = massError(phi, dx, dy, dz, m, n, p);
89      double MError = 0;
90
91      vector<string> plotTimes;
92      vector<string> plotTimesParticle;
93
94      // save signed distance field
95      saveScalarField(savePath + to_string(0.000000) + ".txt", phi, x, y, z, m, n, p);
96      plotTimes.push_back(to_string(0.000000));
97
98      // parameters used in particle level set method
99      vector<Particle> particles;
100     double rmin = 0.1*min(dx, min(dy, dz));
101     double rmax = 0.5*max(dx, max(dy, dz));
102     double bmin = rmin;
103     double bmax = 3.0*max(dx, max(dy, dz));
104
105     // initializing particles
106     cout << "Initializing particles." << endl;
107     if (doParticle){
108         Derivative norm = normal(phi, dx, dy, dz, m, n, p);
109         for (int k = 0; k < p; ++k){
110             for (int j = 0; j < n; ++j){
111                 for (int i = 0; i < m; ++i){
112                     if (abs(phi[i + j*n + k*p*p]) < 3*max(dx, max(dy,dz))){
113                         vector<Particle> newParticles = initializeParticles(x[i], y[j
                            ], z[k], dx, dy, dz, x, y, z, phi, norm, m, n, p, nParticles);
114                         particles.insert(particles.end(), newParticles.begin(),
                            newParticles.end());
115                     }
116                 }
117             }
118         }
119
120         cout << "Initialization finished." << endl;
121
122         if (saveParticles){
123             plotParticles(savePath + to_string(0.000000) + "particle.txt" , particles
                );
124             plotTimesParticle.push_back(to_string(0.000000) + "particle");
125         }
126     }
127
128     vector<double> ax;
129     vector<double> ay;
130     vector<double> az;
131
132     double t = 0;
133
134     // initial velocity field
135     if (testcase == "vortex"){
136         Velocity a = vortexVelocity(m, n, p, x, y, z, t, T);
137         ax = a.x;
138         ay = a.y;
139         az = a.z;
140     } else if (testcase == "sheared"){
141         Velocity a = shearedSphereVelocity(m, n, p, x, y, z, t, T);
```

```
142             ax = a.x;
143             ay = a.y;
144             az = a.z;
145         } else if (testcase == "simple"){
146             Velocity a = simpleVelocity(m, n, p);
147             ax = a.x;
148             ay = a.y;
149             az = a.z;
150         }
151 
152         double dtmax = CFL/(vectorMax(vectorAbs(ax)/dx + vectorAbs(ay)/dy + vectorAbs(az)/
            dz)); // max time step
153         double dt;
154         int numIt = 0;
155 
156         cout << "Setup complete." << endl;
157         chrono::steady_clock::time_point currentTime = chrono::steady_clock::now();
158         cout << "Elapsed time: " << (chrono::duration_cast<chrono::seconds>(currentTime -
            startTime).count())  << " s." << endl << endl;
159 
160         // main iteration loop
161         for (int it = 1; it < itmax; ++it){
162 
163             // statement to make sure loop has one iteration at 0.5 t/T
164             if (t < 0.5*T){
165                 dt = min(dtmax, 0.5*T - t);
166             } else if (t >= 0.5*T){
167                 dt = min(dtmax, T - t);
168             }
169 
170             t += dt;
171 
172             // advection of level set method
173             TVDRK3_weno(phi, ax, ay, az, m, n, p, dx, dy, dz, dt);
174 
175             // advection of particles
176             if (doParticle){
177                 for (int a = 0; a < particles.size(); ++a){
178 
179                     int i = (int)(particles[a].x/dx);
180                     int j = (int)(particles[a].y/dy);
181                     int k = (int)(particles[a].z/dz);
182 
183                     double Up = trilinearInterpolation(particles[a].x, particles[a].y,
                        particles[a].z,
184                         x[i], x[i+1], y[j], y[j+1], z[k], z[k+1],
185                         ax[i+j*n+k*p*p],
186                         ax[(i+1)+j*n+k*p*p],
187                         ax[(i+1)+(j+1)*n+k*p*p],
188                         ax[i+(j+1)*n+k*p*p],
189                         ax[i+j*n+(k+1)*p*p],
190                         ax[(i+1)+j*n+(k+1)*p*p],
191                         ax[(i+1)+(j+1)*n+(k+1)*p*p],
192                         ax[i+(j+1)*n+(k+1)*p*p]);
193 
194                     double Vp = trilinearInterpolation(particles[a].x, particles[a].y,
                        particles[a].z,
195                         x[i], x[i+1], y[j], y[j+1], z[k], z[k+1],
196                         ay[i+j*n+k*p*p],
197                         ay[(i+1)+j*n+k*p*p],
198                         ay[(i+1)+(j+1)*n+k*p*p],
199                         ay[i+(j+1)*n+k*p*p],
200                         ay[i+j*n+(k+1)*p*p],
201                         ay[(i+1)+j*n+(k+1)*p*p],
202                         ay[(i+1)+(j+1)*n+(k+1)*p*p],
203                         ay[i+(j+1)*n+(k+1)*p*p]);
204 
205                     double Wp = trilinearInterpolation(particles[a].x, particles[a].y,
                        particles[a].z,
206                         x[i], x[i+1], y[j], y[j+1], z[k], z[k+1],
207                         az[i+j*n+k*p*p],
208                         az[(i+1)+j*n+k*p*p],
209                         az[(i+1)+(j+1)*n+k*p*p],
```

```
                    az[i+(j+1)*n+k*p*p],
                    az[i+j*n+(k+1)*p*p],
                    az[(i+1)+j*n+(k+1)*p*p],
                    az[(i+1)+(j+1)*n+(k+1)*p*p],
                    az[i+(j+1)*n+(k+1)*p*p]);

                particles[a].x = particles[a].x + dt*Up;
                particles[a].y = particles[a].y + dt*Vp;
                particles[a].z = particles[a].z + dt*Wp;

                i = (int)(particles[a].x/dx);
                j = (int)(particles[a].y/dy);
                k = (int)(particles[a].z/dz);

                double phip = trilinearInterpolation(particles[a].x, particles[a].y,
                particles[a].z,
                    x[i], x[i+1], y[j], y[j+1], z[k], z[k+1],
                    phi[i+j*n+k*p*p],
                    phi[(i+1)+j*n+k*p*p],
                    phi[(i+1)+(j+1)*n+k*p*p],
                    phi[i+(j+1)*n+k*p*p],
                    phi[i+j*n+(k+1)*p*p],
                    phi[(i+1)+j*n+(k+1)*p*p],
                    phi[(i+1)+(j+1)*n+(k+1)*p*p],
                    phi[i+(j+1)*n+(k+1)*p*p]);

                // delete particles
                if (abs(phip) - particles[a].r > bmax){
                    particles.erase(particles.begin() + a);
                }
                // interface correction
                else if ((phip < 0 && particles[a].positive) || (phip > 0 && !
                particles[a].positive)  && (abs(phip) > particles[a].r)){

                    vector<double> phiCorrected =
                        correctInterface(particles[a], x[i], x[i+1], y[j], y[j+1], z[k
                        ], z[k+1],
                        phi[(i)+(j)*n+(k)*p*p],
                        phi[(i+1)+(j)*n+(k)*p*p],
                        phi[(i+1)+(j+1)*n+(k)*p*p],
                        phi[(i)+(j+1)*n+(k)*p*p],
                        phi[(i)+(j)*n+(k+1)*p*p],
                        phi[(i+1)+(j)*n+(k+1)*p*p],
                        phi[(i+1)+(j+1)*n+(k+1)*p*p],
                        phi[(i)+(j+1)*n+(k+1)*p*p],
                        phip);

                    phi[(i)+(j)*n+(k)*p*p] = phiCorrected[0];
                    phi[(i+1)+(j)*n+(k)*p*p] = phiCorrected[1];
                    phi[(i+1)+(j+1)*n+(k)*p*p] = phiCorrected[2];
                    phi[(i)+(j+1)*n+(k)*p*p] = phiCorrected[3];
                    phi[(i)+(j)*n+(k+1)*p*p] = phiCorrected[4];
                    phi[(i+1)+(j)*n+(k+1)*p*p] = phiCorrected[5];
                    phi[(i+1)+(j+1)*n+(k+1)*p*p] = phiCorrected[6];
                    phi[(i)+(j+1)*n+(k+1)*p*p] = phiCorrected[7];

                }

                phip = trilinearInterpolation(particles[a].x, particles[a].y,
                particles[a].z,
                    x[i], x[i+1], y[j], y[j+1], z[k], z[k+1],
                    phi[i+j*n+k*p*p],
                    phi[(i+1)+j*n+k*p*p],
                    phi[(i+1)+(j+1)*n+k*p*p],
                    phi[i+(j+1)*n+k*p*p],
                    phi[i+j*n+(k+1)*p*p],
                    phi[(i+1)+j*n+(k+1)*p*p],
                    phi[(i+1)+(j+1)*n+(k+1)*p*p],
                    phi[i+(j+1)*n+(k+1)*p*p]);

                // adjust radius
                if (sign(phip)*phip > rmax){
                    particles[a].r = rmax;
```

```
279                 } else if (sign(phip)*phip < rmin){
280                     particles[a].r = rmin;
281                 } else {
282                     particles[a].r = sign(phip)*phip;
283                 }
284 
285             }
286 
287             // initialize new particles
288             if (it%reseedFreq == 0 && it != 0){
289                 vector<int> cellx;
290                 vector<int> celly;
291                 vector<int> cellz;
292                 vector<int> cellParticles;
293                 cellx.push_back((int)(particles[0].x/dx));
294                 celly.push_back((int)(particles[0].y/dy));
295                 cellz.push_back((int)(particles[0].z/dz));
296                 cellParticles.push_back(1);
297 
298                 bool found = false;
299                 for (int a = 1; a < particles.size(); ++a){
300                     for (int b = 0; b < cellx.size(); ++b){
301                         if ((int)(particles[a].x/dx) == cellx[b] && (int)(particles[a
                            ].y/dy) == celly[b] && (int)(particles[a].z/dz) == cellz[b]){
302                             cellParticles[b] += 1;
303                             if (cellParticles[b] > nParticles){
304                                 particles.erase(particles.begin() + a);
305                                 a -= 1;
306                             }
307                             found = true;
308                             break;
309                         }
310                     }
311                     if (!found){
312                         cellx.push_back((int)(particles[a].x/dx));
313                         celly.push_back((int)(particles[a].y/dy));
314                         cellz.push_back((int)(particles[a].z/dz));
315                         cellParticles.push_back(1);
316                         found = false;
317                     }
318                 }
319 
320                 Derivative norm = normal(phi, dx, dy, dz, m, n, p);
321                 for (int a = 0; a < cellx.size(); ++a){
322                     int num = nParticles - cellParticles[a];
323                     if (num > 0){
324                         vector<Particle> newParticles = initializeParticles(x[cellx[a
                            ]], y[celly[a]], z[cellz[a]], dx, dy, dz, x, y, z, phi, norm,
                            m, n, p, num);
325                         particles.insert(particles.end(), newParticles.begin(),
                            newParticles.end());
326                     }
327                 }
328             }
329         }
330 
331         // reinitialization loop
332         if (doReinit && (it%reinitFreq == 0)){
333             vector<double> phi0 = phi;
334             for (int i = 0; i < reinitSteps - 1; ++i){
335                 // euler_godunov_reinit(phi, m, n, p, dx, dy, dz, dtau, phi0);
336                 TVDRK3_godunov_reinit(phi, m, n, p, dx, dy, dz, dtau, phi0);
337             }
338         }
339 
340         // mass error
341         MError += abs(massError(phi, dx, dy, dz, m, n, p) - MError0)*dt;
342 
343         // printing to console
344         if (it%10 == 0){
345             cout << "Iteration: " << it << endl;
346             chrono::steady_clock::time_point currentTime = chrono::steady_clock::now
                ();
```

```
347             cout << "Elapsed time: " << (chrono::duration_cast<chrono::seconds>(
                currentTime - startTime).count())  << " s." << endl;
348             cout << "t = " << t << endl;
349         }
350         if (it%plotFreq == 0){
351             cout << "Saving scalar field." << endl;
352             saveScalarField(savePath + to_string(t) + ".txt", phi, x, y, z, m, n, p);
353             plotTimes.push_back(to_string(t));
354 
355             if (doParticle && saveParticles){
356                 plotParticles(savePath + to_string(t) + "particle.txt" , particles);
357                 plotTimesParticle.push_back(to_string(t) + "particle");
358             }
359             cout << "Done saving." << endl;
360         }
361 
362         // plotting at 0.5 t/T
363         if (t/T == 0.5 && halfplot){
364             cout << "Saving scalar field." << endl;
365             saveScalarField(savePath + to_string(t) + ".txt", phi, x, y, z, m, n, p);
366             plotTimes.push_back(to_string(t));
367             cout << "Done saving." << endl;
368         }
369 
370         // finish iterations if t = T
371         if (t == T){
372             numIt = it;
373             break;
374         }
375 
376         // find velocity for next time step
377         if (testcase == "vortex"){
378             Velocity a = vortexVelocity(m, n, p, x, y, z, t, T);
379             ax = a.x;
380             ay = a.y;
381             az = a.z;
382         } else if (testcase == "sheared"){
383             Velocity a = shearedSphereVelocity(m, n, p, x, y, z, t, T);
384             ax = a.x;
385             ay = a.y;
386             az = a.z;
387         } else if (testcase == "simple"){
388             Velocity a = simpleVelocity(m, n, p);
389             ax = a.x;
390             ay = a.y;
391             az = a.z;
392         }
393 
394     }
395 
396     // writing last time step to console
397     {
398         cout << "Iteration: " << numIt << endl;
399         chrono::steady_clock::time_point currentTime = chrono::steady_clock::now();
400         cout << "Elapsed time: " << (chrono::duration_cast<chrono::seconds>(
            currentTime - startTime).count())  << " s." << endl;
401         cout << "t = " << t << endl;
402     }
403 
404     // error measures
405     double endVolume = volume(phi, dx, dy, dz);
406     double volumeChange = 100*(endVolume-initialVolume)/initialVolume;
407     double L1Error = interfaceError(phi0, phi, dx, dy, dz, m, n, p);
408     MError = MError/t;
409 
410     // saving of signed distance field for final time step
411     saveScalarField(savePath + to_string(T) + ".txt", phi, x, y, z, m, n, p);
412     plotTimes.push_back(to_string(T));
413 
414     if (doParticle && saveParticles){
415         plotParticles(savePath + to_string(t) + "particle.txt" , particles);
416         plotTimesParticle.push_back(to_string(t) + "particle");
417     }
```

```
418
419         {
420             ofstream file;
421             file.open(savePath + "plotTimes.txt");
422             if (!file.is_open()){cerr << "could not open file." << endl;}
423             for (int i = 0; i < plotTimes.size(); ++i){
424                 file << plotTimes[i] << endl;
425             }
426             file.close();
427         }
428 
429         if (doParticle && saveParticles){
430             ofstream file;
431             file.open(savePath + "plotTimesParticle.txt");
432             if (!file.is_open()){cerr << "could not open file." << endl;}
433             for (int i = 0; i < plotTimesParticle.size(); ++i){
434                 file << plotTimesParticle[i] << endl;
435             }
436             file.close();
437         }
438 
439         // printing log to file
440         {
441             ofstream file;
442             file.open(savePath + "log.txt");
443             if (!file.is_open()){cerr << "could not open file." << endl;}
444             file << "Iterations: " << numIt << endl;
445             chrono::steady_clock::time_point currentTime = chrono::steady_clock::now();
446             file << "Elapsed time: " << (chrono::duration_cast<chrono::seconds>(
                currentTime - startTime).count())  << " s." << endl;
447             file << "Initial volume: " << initialVolume << endl;
448             file << "End volume: " << endVolume << endl;
449             file << "Volume change: " << volumeChange << " %" << endl;
450             file << "Interface error: " << L1Error << endl;
451             file << "Average area error: " << MError << endl;
452             file.close();
453         }
454 
455         return 0;
456     }
457 
```

A.2 initialization.h and initialization.cpp

These files contain code to create the initial signed distance field with reference from a sphere surface and a function to save the signed distance field to a .txt-file. initialization.h is given first and contains the header file, while initialization.cpp contains the whole implementation.

```cpp
#pragma once

#include <vector>
#include <iostream>
#include <array>
#include <cmath>
#include <fstream>
using namespace std;

#define PI 3.14159265

// class for a 3D point
class Point
{
public:
    double x;
    double y;
    double z;
    Point(double x1, double y1, double z1){x = x1; y = y1; z = z1;}
    Point(){x = 0; y = 0; z = 0;}
    Point operator+(Point const &p);
    void operator=(Point const &p);
};

// returns length between two points
double length(Point const &p0, Point const &p1);

// check if a point is within a sphere of center c and radius r
bool isInsideSphere(double r, Point c, Point p);

// returns the signed distance from a point to the surface of a sphere of center c and radius r
double signedDistanceSphere(double r, Point c, Point p);

// generates a signed distance field for all points in [xmin, xmax] * [ymin, ymax] * [zmin, zmax] with reference to a sphere of center c and radius r
void signedDistanceField(vector<double> &arr, vector<double> x, vector<double> y, vector<double> z, double r, Point c, int M, int N, int P);

// returns a vector of n indexes with equally spaced values from start to end
vector<double> linspace(double start, double end, int n);

// saves a signed distance field to .txt-file
void saveScalarField(string filename, vector<double> const &arr, vector<double> x, vector<double> y, vector<double> z, int M, int N, int P);

```

```cpp
#include "initialization.h"

// addition of two points
Point Point::operator+(Point const &p){
    Point temp;
    temp.x = x + p.x;
    temp.y = y + p.y;
    temp.z = z + p.z;
    return temp;
}

// a point is assigned the same coordinates as another point
void Point::operator=(Point const &p){
    x = p.x;
    y = p.y;
    z = p.z;
}

// returns length between two points
double length(Point const &p0, Point const &p1){
    return sqrt((p0.x - p1.x)*(p0.x - p1.x) + (p0.y - p1.y)*(p0.y - p1.y) + (p0.z - p1.z)*(p0.z - p1.z));
}

// check if a point is within a sphere of center c and radius r
bool isInsideSphere(double r, Point c, Point p){
    if ((length(c, p) - r) < 0){
        return true;
    } else {
        return false;
    }
}

// returns the signed distance from a point to the surface of a sphere of center c and radius r
double signedDistanceSphere(double r, Point c, Point p){

    if (isInsideSphere(r, c, p)){
        return -(r - length(p, c));
    }
    return length(p, c) - r;
}


// generates a signed distance field for all points in [xmin, xmax] * [ymin, ymax] * [zmin, zmax] with reference to a sphere of center c and radius r
void signedDistanceField(vector<double> &arr, vector<double> x, vector<double> y, vector<double> z, double r, Point c, int M, int N, int P){ // Fix
    for (int k = 0; k < P; ++k){
        for (int j = 0; j < N; ++j){
            for (int i = 0; i < M; ++i){
                arr.push_back(signedDistanceSphere(r, c, Point(x[i], y[j], z[k])));
            }
        }
    }
}

// returns a vector of n indexes with equally spaced values from start to end
vector<double> linspace(double start, double end, int n){

    vector<double> vec;

    if (n == 0) {
        return vec;
    }
    if (n == 1) {
      vec.push_back(start);
      return vec;
    }
```

```
66
67      double dx = (end - start)/(n - 1);
68
69      for(int i = 0; i < n - 1; ++i){
70          vec.push_back(start + dx * i);
71      }
72      vec.push_back(end);
73
74      return vec;
75  }
76
77  // saves a signed distance field to .txt-file
78  void saveScalarField(string filename, vector<double> const &arr, vector<double> x, vector
    <double> y, vector<double> z, int M, int N, int P){
79      ofstream file;
80      file.open(filename);
81      if (!file.is_open()){cerr << "could not open file." << endl;}
82
83      file << M << "," << N << "," << P << endl;
84      int count = 0;
85      for (int k = 0; k < P; ++k){
86          for (int j = 0; j < N; ++j){
87              for (int i = 0; i < M; ++i){
88                  file << x[i] << "," << y[j] << "," << z[k] << "," << arr[count] << "," <<
                     endl;
89                  ++count;
90              }
91          }
92      }
93      file.close();
94  }
95
```

A.3 particleLSM.h and particleLSM.cpp

These files contain code to set up the particles in the particle level set method, use these particles to correct the interface, and a function to save the particle coordinates to a .txt-file. particleLSM.h is given first and contains the header file, while particleLSM.cpp contains the whole implementation.

```cpp
#pragma once
#include <vector>
#include <random>
#include "initialization.h"
#include "schemes.h"

// particle class with coordinates, radius and a bool, where true is a positive particle, and false is a negative particle
class Particle
{
public:
    double x;
    double y;
    double z;
    double r;
    bool positive;
    Particle(double x1, double y1, double z1){x = x1; y = y1; z = z1;}
    Particle(){x = 0; y = 0; z = 0;}
    Particle(double x1, double y1, double z1, double r1, bool pos){x = x1; y = y1; z = z1; r = r1; positive = pos;}
};

// initializes particles in a cell where (x0, y0, z0) are the coordinates of the cell closest ot origo.
// dx, dy, dz are the grid spacing. X, Y, Z are all grid nodes in the computational domain.
// phi is the signed distance field. normal is the normal-vector. M, N, P is the number of grid nodes in each direction.
// numParticles are the number of particles of each type to be initialized in the cell.
vector<Particle> initializeParticles(double x0, double y0, double z0, double dx, double dy, double dz,
    vector<double> &X, vector<double> &Y, vector<double> &Z, vector<double> &phi,
    Derivative &normal,
    int M, int N, int P, int numParticles);

// returns the corrected values of the signed distance field for each corner of the cells.
// the xi, yj, zk values are the coordinates of the corners of the cell, where x0 < x1, y0 < y1, z0 < z1.
// the phiijk are the values of the signed distance fiield at (i,j,k)
vector<double> correctInterface(Particle p, double x0, double x1, double y0, double y1, double z0, double z1,
    double phi000, double phi100, double phi110, double phi010, double phi001, double phi101, double phi111, double phi011, double phip);

// returns the interpolated value at (x, y, z)
// the xi, yj, zk values are the coordinates of the corners of the cell, where x0 < x1, y0 < y1, z0 < z1.
// the phiijk are the values of the signed distance fiield at (i,j,k)
double trilinearInterpolation(double x, double y, double z, double x0, double x1, double y0, double y1, double z0, double z1,
    double c000, double c100, double c110, double c010, double c001, double c101, double c111, double c011);

// returns the normal vectors Nx, Ny, Nz for the signed distance field
Derivative normal(vector<double> &arr, double dx, double dy, double dz, double M, double N, double P);

// saves the coordinates of all the particles to a .txt-files
void plotParticles(string filename, vector<Particle> particles);

```

```cpp
#include "particleLSM.h"

// initializes particles in a cell where (x0, y0, z0) are the coordinates of the cell closest ot origo.
// dx, dy, dz are the grid spacing. X, Y, Z are all grid nodes in the computational domain.
// phi is the signed distance field. normal is the normal-vector. M, N, P is the number of grid nodes in each direction.
// numParticles are the number of particles of each type to be initialized in the cell.
vector<Particle> initializeParticles(double x0, double y0, double z0, double dx, double dy, double dz,
    vector<double> &X, vector<double> &Y, vector<double> &Z, vector<double> &phi, Derivative &normal,
    int M, int N, int P, int numParticles){

    vector<Particle> particles;
    std::random_device rd; // obtain a random number from hardware
    std::mt19937 gen(rd()); // seed the generator
    std::uniform_int_distribution<> distr(0, 100); // define the range
    double rmin = 0.1*min(dx, min(dy, dz)); // minimum particle radius
    double rmax = 0.5*max(dx, max(dy, dz)); // maximum particle radius
    double bmin = rmin;
    double bmax = 3.0*max(dx, max(dy, dz));
    double lambda = 1.0;
    double itmax = 15; // max iterations in the attraction step
    for (int p = 0; p < numParticles*2; ++p){
        int positive = p%2; // even negative, odd positive

        // random coordinate in a cell
        double x = x0 + dx*distr(gen)/100.0;
        double y = y0 + dy*distr(gen)/100.0;
        double z = z0 + dz*distr(gen)/100.0;
        double phip;

        // index position of the particle
        int i = (int)(x/dx);
        int j = (int)(y/dy);
        int k = (int)(z/dz);

        if (i < 0 || j < 0 || k < 0 || i >= M || j >= N || k >= P){continue;}

        phip = trilinearInterpolation(x, y, z, X[i], X[i+1], Y[j], Y[j+1], Z[k], Z[k+1],
            phi[i+j*N+k*P*P],
            phi[i+1+j*N+k*P*P],
            phi[i+1+(j+1)*N+k*P*P],
            phi[i+(j+1)*N+k*P*P],
            phi[i+j*N+(k+1)*P*P],
            phi[i+1+j*N+(k+1)*P*P],
            phi[i+1+(j+1)*N+(k+1)*P*P],
            phi[i+(j+1)*N+(k+1)*P*P]);

        double phiGoal = positive*(bmin + (bmax - bmin)*distr(gen)/100.0) - (1-positive)*(bmin + (bmax - bmin)*distr(gen)/100.0);

        // attraction step
        for (int it = 0; it < itmax; ++it){

            double Nxp = trilinearInterpolation(x, y, z, X[i], X[i+1], Y[j], Y[j+1], Z[k], Z[k+1],
                normal.x[i+j*N+k*P*P],
                normal.x[i+1+j*N+k*P*P],
                normal.x[i+1+(j+1)*N+k*P*P],
                normal.x[i+(j+1)*N+k*P*P],
                normal.x[i+j*N+(k+1)*P*P],
                normal.x[i+1+j*N+(k+1)*P*P],
                normal.x[i+1+(j+1)*N+(k+1)*P*P],
                normal.x[i+(j+1)*N+(k+1)*P*P]);
            x = x + lambda*(phiGoal - phip)*Nxp;

```

```
63              double Nyp = trilinearInterpolation(x, y, z, X[i], X[i+1], Y[j], Y[j+1], Z[k
                ], Z[k+1],
64                  normal.y[i+j*N+k*P*P],
65                  normal.y[i+1+j*N+k*P*P],
66                  normal.y[i+1+(j+1)*N+k*P*P],
67                  normal.y[i+(j+1)*N+k*P*P],
68                  normal.y[i+j*N+(k+1)*P*P],
69                  normal.y[i+1+j*N+(k+1)*P*P],
70                  normal.y[i+1+(j+1)*N+(k+1)*P*P],
71                  normal.y[i+(j+1)*N+(k+1)*P*P]);
72              y = y + lambda*(phiGoal - phip)*Nyp;
73
74              double Nzp = trilinearInterpolation(x, y, z, X[i], X[i+1], Y[j], Y[j+1], Z[k
                ], Z[k+1],
75                  normal.z[i+j*N+k*P*P],
76                  normal.z[i+1+j*N+k*P*P],
77                  normal.z[i+1+(j+1)*N+k*P*P],
78                  normal.z[i+(j+1)*N+k*P*P],
79                  normal.z[i+j*N+(k+1)*P*P],
80                  normal.z[i+1+j*N+(k+1)*P*P],
81                  normal.z[i+1+(j+1)*N+(k+1)*P*P],
82                  normal.z[i+(j+1)*N+(k+1)*P*P]);
83              z = z + lambda*(phiGoal - phip)*Nzp;
84
85              i = (int)(x/dx);
86              j = (int)(y/dy);
87              k = (int)(z/dz);
88
89              if (i < 0 || j < 0 || k < 0 || i >= M || j >= N || k >= P){break;}
90
91              phip = trilinearInterpolation(x, y, z, X[i], X[i+1], Y[j], Y[j+1], Z[k], Z[k+
                1],
92                  phi[i+j*N+k*P*P],
93                  phi[i+1+j*N+k*P*P],
94                  phi[i+1+(j+1)*N+k*P*P],
95                  phi[i+(j+1)*N+k*P*P],
96                  phi[i+j*N+(k+1)*P*P],
97                  phi[i+1+j*N+(k+1)*P*P],
98                  phi[i+1+(j+1)*N+(k+1)*P*P],
99                  phi[i+(j+1)*N+(k+1)*P*P]);
100
101             if ((positive && (phip >= bmin && phip <= bmax)) || (!positive && (phip <= -
                bmin && phip >= -bmax))){
102                 double r;
103                 if (sign(phip)*phip > rmax){
104                     r = rmax;
105                 } else if (sign(phip)*phip < rmin){
106                     r = rmin;
107                 } else {
108                     r = sign(phip)*phip;
109                 }
110                 particles.push_back(Particle(x, y, z, r, positive));
111                 break;
112             } else {
113                 lambda = lambda/2.0;
114             }
115         }
116     }
117     return particles;
118 }
119
120 // returns the corrected values of the signed distance field for each corner of the
    cells.
121 // the xi, yj, zk values are the coordinates of the corners of the cell, where x0 < x1,
    y0 < y1, z0 < z1.
122 // the phiijk are the values of the signed distance fiield at (i,j,k)
123 vector<double> correctInterface(Particle p, double x0, double x1, double y0, double y1,
    double z0, double z1,
124     double phi000, double phi100, double phi110, double phi010, double phi001, double
```

```
        phi101, double phi111, double phi011, double phip){
125
126         // distance from particle surface to cell corners
127         double phip000 = sign(phip)*(p.r - sqrt(pow(x0 - p.x, 2) + pow(y0 - p.y, 2) + pow(z0
            - p.z, 2)));
128         double phip100 = sign(phip)*(p.r - sqrt(pow(x1 - p.x, 2) + pow(y0 - p.y, 2) + pow(z0
            - p.z, 2)));
129         double phip110 = sign(phip)*(p.r - sqrt(pow(x1 - p.x, 2) + pow(y1 - p.y, 2) + pow(z0
            - p.z, 2)));
130         double phip010 = sign(phip)*(p.r - sqrt(pow(x0 - p.x, 2) + pow(y1 - p.y, 2) + pow(z0
            - p.z, 2)));
131         double phip001 = sign(phip)*(p.r - sqrt(pow(x0 - p.x, 2) + pow(y0 - p.y, 2) + pow(z1
            - p.z, 2)));
132         double phip101 = sign(phip)*(p.r - sqrt(pow(x1 - p.x, 2) + pow(y0 - p.y, 2) + pow(z1
            - p.z, 2)));
133         double phip111 = sign(phip)*(p.r - sqrt(pow(x1 - p.x, 2) + pow(y1 - p.y, 2) + pow(z1
            - p.z, 2)));
134         double phip011 = sign(phip)*(p.r - sqrt(pow(x0 - p.x, 2) + pow(y1 - p.y, 2) + pow(z1
            - p.z, 2)));
135
136         vector<double> phi_p;
137         vector<double> phi_m;
138         vector<double> phi;
139
140         phi_p.push_back(max(phi000, phip000));
141         phi_p.push_back(max(phi100, phip100));
142         phi_p.push_back(max(phi110, phip110));
143         phi_p.push_back(max(phi010, phip010));
144         phi_p.push_back(max(phi001, phip001));
145         phi_p.push_back(max(phi101, phip101));
146         phi_p.push_back(max(phi111, phip111));
147         phi_p.push_back(max(phi011, phip011));
148
149         phi_m.push_back(min(phi000, phip000));
150         phi_m.push_back(min(phi100, phip100));
151         phi_m.push_back(min(phi110, phip110));
152         phi_m.push_back(min(phi010, phip010));
153         phi_m.push_back(min(phi001, phip001));
154         phi_m.push_back(min(phi101, phip101));
155         phi_m.push_back(min(phi111, phip111));
156         phi_m.push_back(min(phi011, phip011));
157
158         for (int i = 0; i < phi_p.size(); ++i){
159             if (abs(phi_p[i]) <= abs(phi_m[i])){
160                 phi.push_back(phi_p[i]);
161             } else if (abs(phi_p[i]) > abs(phi_m[i])){
162                 phi.push_back(phi_m[i]);
163             }
164         }
165         return phi;
166     }
167
168     // returns the interpolated value at (x, y, z)
169     // the xi, yj, zk values are the coordinates of the corners of the cell, where x0 < x1,
        y0 < y1, z0 < z1.
170     // the phiijk are the values of the signed distance fiield at (i,j,k)
171     double trilinearInterpolation(double x, double y, double z, double x0, double x1, double
        y0, double y1, double z0, double z1,
172         double c000, double c100, double c110, double c010, double c001, double c101, double
            c111, double c011){
173
174         double xd = (x - x0)/(x1 - x0);
175         double yd = (y - y0)/(y1 - y0);
176         double zd = (z - z0)/(z1 - z0);
177
178         double c00 = c000*(1 - xd) + c100*xd;
179         double c01 = c001*(1 - xd) + c101*xd;
180         double c10 = c010*(1 - xd) + c110*xd;
181         double c11 = c011*(1 - xd) + c111*xd;
```

```
182
183         double c0 = c00*(1 - yd) + c10*yd;
184         double c1 = c01*(1 - yd) + c11*yd;
185
186         return c0*(1 - zd) + c1*zd;
187     }
188
189     // returns the normal vectors Nx, Ny, Nz for the signed distance field
190     Derivative normal(vector<double> &arr, double dx, double dy, double dz, double M, double
        N, double P){
191         vector<double> Nx;
192         vector<double> Ny;
193         vector<double> Nz;
194         for (int k = 0; k < P; ++k){
195             for (int j = 0; j < N; ++j){
196                 for (int i = 0; i < M; ++i){
197                     if (i==0 || i==(M-1) || j==0 || j==(N-1) || k==0 || k==(P-1)){
198                         Nx.push_back(0);
199                         Ny.push_back(0);
200                         Nz.push_back(0);
201                         continue;
202                     }
203                     double phix = (arr[(i+1)+j*N+k*P*P] - arr[(i-1)+j*N+k*P*P])/(2*dx);
204                     if (phix == 0){
205                         phix = (arr[(i+1)+j*N+k*P*P] - arr[i+j*N+k*P*P])/(dx);
206                     }
207                     double phiy = (arr[i+(j+1)*N+k*P*P] - arr[i+(j-1)*N+k*P*P])/(2*dy);
208                     if (phiy == 0){
209                         phiy = (arr[i+(j+1)*N+k*P*P] - arr[i+j*N+k*P*P])/(dy);
210                     }
211                     double phiz = (arr[i+j*N+(k+1)*P*P] - arr[i+j*N+(k-1)*P*P])/(2*dz);
212                     if (phiz == 0){
213                         phiz = (arr[i+j*N+(k+1)*P*P] - arr[i+j*N+k*P*P])/(dz);
214                     }
215                     Nx.push_back(phix/abs(phix));
216                     Ny.push_back(phiy/abs(phiy));
217                     Nz.push_back(phiz/abs(phiz));
218                 }
219             }
220         }
221         return Derivative{Nx, Ny, Nz};
222     }
223
224     // saves the coordinates of all the particles to a .txt-files
225     void plotParticles(string filename, vector<Particle> particles){
226         ofstream file;
227         file.open(filename);
228         if (!file.is_open()){cerr << "could not open file." << endl;}
229
230         int count = 0;
231         for (int i = 0; i < particles.size(); ++i){
232             file << particles[i].x << "," << particles[i].y << "," << particles[i].z << ","
                << endl;
233         }
234         file.close();
235     }
236
```

A.4 schemes.h and schemes.cpp

These files contain all the numerical schemes used in the implemented particle level set method. schemes.h is given first and contains the header file, while schemes.cpp contains the whole implementation.

```cpp
#pragma once

#include <vector>
#include <array>
#include <tuple>
#include <functional>
#include <cmath>
#include "vectorUtilities.h"

using namespace std;

// a deriative vector with a value in each direction in 3D
struct Derivative {
    vector<double> x;
    vector<double> y;
    vector<double> z;
};

// first-order upwind scheme
Derivative upwind(vector<double> &phi, vector<double> AX, vector<double> AY, vector<double> AZ, int M, int N, int P, double dx, double dy, double dz);

// WENO scheme. Third-order accurate and fifth-order accurate in smooth regions
Derivative weno(vector<double> &phi, vector<double> AX, vector<double> AY, vector<double> AZ, int M, int const N, int const P, double dx, double dy, double dz);

// Godunov scheme used for the reinitialization equation
Derivative godunov(vector<double> &phi, vector<double> AX, vector<double> AY, vector<double> AZ, int M, int const N, int const P, double dx, double dy, double dz);

// first-order explicit Euler scheme used with the upwind scheme
void euler_upwind(vector<double> &phi, vector<double> AX, vector<double> AY, vector<double> AZ, int M, int N, int P, double dx, double dy, double dz, double dt);

// third-order TVDRK scheme used with the upwind scheme
void TVDRK3_upwind(vector<double> &phi, vector<double> AX, vector<double> AY, vector<double> AZ, int M, int N, int P, double dx, double dy, double dz, double dt);

// third-order TVDRK scheme used with the WENO scheme
void TVDRK3_weno(vector<double> &phi, vector<double> AX, vector<double> AY, vector<double> AZ, int M, int N, int P, double dx, double dy, double dz, double dt);

// first-order explicit Euler scheme used with the weno scheme
void euler_weno(vector<double> &phi, vector<double> AX, vector<double> AY, vector<double> AZ, int M, int N, int P, double dx, double dy, double dz, double dt);

// third-order TVDRK scheme used with the Godunov scheme to solve the reinitialization equation
void TVDRK3_godunov_reinit(vector<double> &phi, int M, int N, int P, double dx, double dy, double dz, double dt, vector<double> phi0);

// first-order explicit Euler scheme used with the Godunov scheme to solve the reinitialization equation
void euler_godunov_reinit(vector<double> &phi, int M, int N, int P, double dx, double dy, double dz, double dt, const vector<double> &phi0);

// sign function that returns 1 for a positive value, -1 for a negative value, and 0 for a value of 0
int sign(double num);

```

```cpp
#include "schemes.h"

// first-order upwind scheme
Derivative upwind(vector<double> &phi, vector<double> AX, vector<double> AY, vector<double> AZ, int M, int N, int P, double dx, double dy, double dz){

    vector<double> phix;
    vector<double> phiy;
    vector<double> phiz;

    for (int k = 0; k < P; ++k){
        for (int j = 0; j < N; ++j){
            for (int i = 0; i < M; ++i){

                if (i==0 || i==M-1 || j==0 || j==N-1 || k==0 || k==P-1){
                    phix.push_back(0);
                    phiy.push_back(0);
                    phiz.push_back(0);
                    continue;
                }

                if (AX[i + j*N + k*P*P] >= 0){
                    phix.push_back((phi[i + j*N + k*P*P] - phi[(i - 1) + j*N + k*P*P])/dx);
                } else if (AX[i + j*N + k*P*P] < 0){
                    phix.push_back((phi[(i + 1) + j*N + k*P*P] - phi[i + j*N + k*P*P])/dx);
                }

                if (AY[i + j*N + k*P*P] >= 0){
                    phiy.push_back((phi[i + j*N + k*P*P] - phi[i + (j - 1)*N + k*P*P])/dy);
                } else if (AY[i + j*N + k*P*P] < 0){
                    phiy.push_back((phi[i + (j + 1)*N + k*P*P] - phi[i + j*N + k*P*P])/dy);
                }

                if (AZ[i + j*N + k*P*P] >= 0){
                    phiz.push_back((phi[i + j*N + k*P*P] - phi[i + j*N + (k - 1)*P*P])/dz);
                } else if (AZ[i + j*N + k*P*P] < 0){
                    phiz.push_back((phi[i + j*N + (k + 1)*P*P] - phi[i + j*N + k*P*P])/dz);
                }
            }
        }
    }
    cout << AX.size() << " " << AY.size() << " " << AZ.size() << endl;
    return Derivative{phix, phiy, phiz};
}

// WENO scheme. Third-order accurate and fifth-order accurate in smooth regions
Derivative weno(vector<double> &phi, vector<double> AX, vector<double> AY, vector<double> AZ, int M, int const N, int const P, double dx, double dy, double dz){

    vector<double> phix;
    vector<double> phiy;
    vector<double> phiz;

    for (int k = 0; k < P; ++k){
        for (int j = 0; j < N; ++j){
            for (int i = 0; i < M; ++i){
                if (i == 0 || i == 1 || i==2 || i==(M-3) || i == (M-2) || i == (M-1)){
                    phix.push_back(0);
                    phiy.push_back(0);
                    phiz.push_back(0);
                    continue;
                }
                if (j == 0 || j == 1 || j==2 || j==(N-3) || j == (N-2) || j == (N-1)){
```

```
                    phix.push_back(0);
                    phiy.push_back(0);
                    phiz.push_back(0);
                    continue;
                }
                if (k == 0 || k == 1 || k==2 || k==(P-3) || k == (P-2) || k == (P-1)){
                    phix.push_back(0);
                    phiy.push_back(0);
                    phiz.push_back(0);
                    continue;
                }

                double v1;
                double v2;
                double v3;
                double v4;
                double v5;

                {
                if (AX[i + j*N + k*P*P] >= 0){
                    v1 = (phi[(i-2) + j*N + k*P*P] - phi[(i-3) + j*N + k*P*P])/dx;
                    v2 = (phi[(i-1) + j*N + k*P*P] - phi[(i-2) + j*N + k*P*P])/dx;
                    v3 = (phi[(i) + j*N + k*P*P] - phi[(i-1) + j*N + k*P*P])/dx;
                    v4 = (phi[(i+1) + j*N + k*P*P] - phi[(i) + j*N + k*P*P])/dx;
                    v5 = (phi[(i+2) + j*N + k*P*P] - phi[(i+1) + j*N + k*P*P])/dx;
                } else if (AX[i + j*N + k*P*P] < 0){
                    v1 = (phi[(i-1) + j*N + k*P*P] - phi[(i-2) + j*N + k*P*P])/dx;
                    v2 = (phi[(i) + j*N + k*P*P] - phi[(i-1) + j*N + k*P*P])/dx;
                    v3 = (phi[(i+1) + j*N + k*P*P] - phi[(i) + j*N + k*P*P])/dx;
                    v4 = (phi[(i+2) + j*N + k*P*P] - phi[(i+1) + j*N + k*P*P])/dx;
                    v5 = (phi[(i+3) + j*N + k*P*P] - phi[(i+2) + j*N + k*P*P])/dx;
                }
                double S1 = 13/12*(v1 - 2*v2 + v3)*(v1 - 2*v2 + v3) + 1/4*(v1 - 4*v2 + v3
                )*(v1 - 4*v2 + v3);
                double S2 = 13/12*(v2 - 2*v3 + v4)*(v2 - 2*v3 + v4) + 1/4*(v2 - v4)*(v2 -
                 v4);
                double S3 = 13/12*(v3 - 2*v4 + v5)*(v3 - 2*v4 + v5) + 1/4*(3*v3 - 4*v4 +
                v5)*(3*v3 - 4*v4 + v5);

                double epsilon = pow(10, -6)*max(max(max(max(v1*v1, v2*v2), v3*v3), v4*v4
                ), v5*v5) + pow(10, -99);

                double alpha1 = 0.1/((S1 + epsilon)*(S1 + epsilon));
                double alpha2 = 0.6/((S2 + epsilon)*(S2 + epsilon));
                double alpha3 = 0.3/((S3 + epsilon)*(S3 + epsilon));

                double omega1 = alpha1/(alpha1 + alpha2 + alpha3);
                double omega2 = alpha2/(alpha1 + alpha2 + alpha3);
                double omega3 = alpha3/(alpha1 + alpha2 + alpha3);

                double phix1 = v1/3 - 7*v2/6 + 11*v3/6;
                double phix2 = -v2/6 + 5*v3/6 + v4/3;
                double phix3 = v3/3 + 5*v4/6 - v5/6;

                phix.push_back(omega1*phix1 + omega2*phix2 + omega3*phix3);
                }

                {
                if (AY[i + j*N + k*P*P] >= 0){
                    v1 = (phi[i + (j-2)*N + k*P*P] - phi[i + (j-3)*N + k*P*P])/dy;
                    v2 = (phi[i + (j-1)*N + k*P*P] - phi[i + (j-2)*N + k*P*P])/dy;
                    v3 = (phi[i + j*N + k*P*P] - phi[i + (j-1)*N + k*P*P])/dy;
                    v4 = (phi[i + (j+1)*N + k*P*P] - phi[i + j*N + k*P*P])/dy;
                    v5 = (phi[i + (j+2)*N + k*P*P] - phi[i + (j+1)*N + k*P*P])/dy;
                } else if (AY[i + j*N + k*P*P] < 0){
                    v1 = (phi[i + (j-1)*N + k*P*P] - phi[i + (j-2)*N + k*P*P])/dy;
                    v2 = (phi[i + j*N + k*P*P] - phi[i + (j-1)*N + k*P*P])/dy;
                    v3 = (phi[i + (j+1)*N + k*P*P] - phi[i + j*N + k*P*P])/dy;
                    v4 = (phi[i + (j+2)*N + k*P*P] - phi[i + (j+1)*N + k*P*P])/dy;
```

```
                    v5 = (phi[i + (j+3)*N + k*P*P] - phi[i + (j+2)*N + k*P*P])/dy;
                }
                double S1 = 13/12*(v1 - 2*v2 + v3)*(v1 - 2*v2 + v3) + 1/4*(v1 - 4*v2 + v3
                )*(v1 - 4*v2 + v3);
                double S2 = 13/12*(v2 - 2*v3 + v4)*(v2 - 2*v3 + v4) + 1/4*(v2 - v4)*(v2 -
                 v4);
                double S3 = 13/12*(v3 - 2*v4 + v5)*(v3 - 2*v4 + v5) + 1/4*(3*v3 - 4*v4 +
                v5)*(3*v3 - 4*v4 + v5);

                double epsilon = pow(10, -6)*max(max(max(max(v1*v1, v2*v2), v3*v3), v4*v4
                ), v5*v5) + pow(10, -99);

                double alpha1 = 0.1/((S1 + epsilon)*(S1 + epsilon));
                double alpha2 = 0.6/((S2 + epsilon)*(S2 + epsilon));
                double alpha3 = 0.3/((S3 + epsilon)*(S3 + epsilon));

                double omega1 = alpha1/(alpha1 + alpha2 + alpha3);
                double omega2 = alpha2/(alpha1 + alpha2 + alpha3);
                double omega3 = alpha3/(alpha1 + alpha2 + alpha3);

                double phiy1 = v1/3 - 7*v2/6 + 11*v3/6;
                double phiy2 = -v2/6 + 5*v3/6 + v4/3;
                double phiy3 = v3/3 + 5*v4/6 - v5/6;

                phiy.push_back(omega1*phiy1 + omega2*phiy2 + omega3*phiy3);

                }

                {
                if (AZ[i + j*N + k*P*P] >= 0){
                    v1 = (phi[i + j*N + (k-2)*P*P] - phi[i + j*N + (k-3)*P*P])/dz;
                    v2 = (phi[i + j*N + (k-1)*P*P] - phi[i + j*N + (k-2)*P*P])/dz;
                    v3 = (phi[i + j*N + k*P*P] - phi[i + j*N + (k-1)*P*P])/dz;
                    v4 = (phi[i + j*N + (k+1)*P*P] - phi[i + j*N + k*P*P])/dz;
                    v5 = (phi[i + j*N + (k+2)*P*P] - phi[i + j*N + (k+1)*P*P])/dz;
                } else if (AZ[i + j*N + k*P*P] < 0){
                    v1 = (phi[i + j*N + (k-1)*P*P] - phi[i + j*N + (k-2)*P*P])/dz;
                    v2 = (phi[i + j*N + k*P*P] - phi[i + j*N + (k-1)*P*P])/dz;
                    v3 = (phi[i + j*N + (k+1)*P*P] - phi[i + j*N + k*P*P])/dz;
                    v4 = (phi[i + j*N + (k+2)*P*P] - phi[i + j*N + (k+1)*P*P])/dz;
                    v5 = (phi[i + j*N + (k+3)*P*P] - phi[i + j*N + (k+2)*P*P])/dz;
                }
                double S1 = 13/12*(v1 - 2*v2 + v3)*(v1 - 2*v2 + v3) + 1/4*(v1 - 4*v2 + v3
                )*(v1 - 4*v2 + v3);
                double S2 = 13/12*(v2 - 2*v3 + v4)*(v2 - 2*v3 + v4) + 1/4*(v2 - v4)*(v2 -
                 v4);
                double S3 = 13/12*(v3 - 2*v4 + v5)*(v3 - 2*v4 + v5) + 1/4*(3*v3 - 4*v4 +
                v5)*(3*v3 - 4*v4 + v5);

                double epsilon = pow(10, -6)*max(max(max(max(v1*v1, v2*v2), v3*v3), v4*v4
                ), v5*v5) + pow(10, -99);

                double alpha1 = 0.1/((S1 + epsilon)*(S1 + epsilon));
                double alpha2 = 0.6/((S2 + epsilon)*(S2 + epsilon));
                double alpha3 = 0.3/((S3 + epsilon)*(S3 + epsilon));

                double omega1 = alpha1/(alpha1 + alpha2 + alpha3);
                double omega2 = alpha2/(alpha1 + alpha2 + alpha3);
                double omega3 = alpha3/(alpha1 + alpha2 + alpha3);

                double phiz1 = v1/3 - 7*v2/6 + 11*v3/6;
                double phiz2 = -v2/6 + 5*v3/6 + v4/3;
                double phiz3 = v3/3 + 5*v4/6 - v5/6;

                phiz.push_back(omega1*phiz1 + omega2*phiz2 + omega3*phiz3);
                }
            }
        }
    }
```

```
188         return Derivative{phix, phiy, phiz};
189     }
190 
191     // Godunov scheme used for the reinitialization equation
192     Derivative godunov(vector<double> &phi, vector<double> AX, vector<double> AY, vector<
        double> AZ, int M, int const N, int const P, double dx, double dy, double dz){
193 
194         vector<double> phix;
195         vector<double> phiy;
196         vector<double> phiz;
197 
198         for (int k = 0; k < P; ++k){
199             for (int j = 0; j < N; ++j){
200                 for (int i = 0; i < M; ++i){
201 
202                     if (i == 0 || i == (M-1) || j == 0 || j == (N-1) || k == 0 || k == (P-1)
                        ){
203                         phix.push_back(1.0);
204                         phiy.push_back(1.0);
205                         phiz.push_back(1.0);
206                         continue;
207                     }
208 
209                     double phix_m = (phi[i + j*N + k*P*P] - phi[(i-1) + j*N + k*P*P])/dx;
210                     double phix_p = (phi[(i+1) + j*N + k*P*P] - phi[i + j*N + k*P*P])/dx;
211 
212                     double phiy_m = (phi[i + j*N + k*P*P] - phi[i + (j-1)*N + k*P*P])/dy;
213                     double phiy_p = (phi[i + (j+1)*N + k*P*P] - phi[i + j*N + k*P*P])/dy;
214 
215                     double phiz_m = (phi[i + j*N + k*P*P] - phi[i + j*N + (k-1)*P*P])/dz;
216                     double phiz_p = (phi[i + j*N + (k+1)*P*P] - phi[i + j*N + k*P*P])/dz;
217 
218                     if (AX[i + j*N + k*P*P] >= 0){
219                         phix.push_back(sqrt(max(max(phix_m, 0.0)*max(phix_m, 0.0), min(phix_p
                            , 0.0)*min(phix_p, 0.0))));
220                     } else if (AX[i + j*N + k*P*P] < 0){
221                         phix.push_back(sqrt(max(min(phix_m, 0.0)*min(phix_m, 0.0), max(phix_p
                            , 0.0)*max(phix_p, 0.0))));
222                     }
223 
224                     if (AY[i + j*N + k*P*P] >= 0){
225                         phiy.push_back(sqrt(max(max(phiy_m, 0.0)*max(phiy_m, 0.0), min(phiy_p
                            , 0.0)*min(phiy_p, 0.0))));
226                     } else if (AY[i + j*N + k*P*P] < 0){
227                         phiy.push_back(sqrt(max(min(phiy_m, 0.0)*min(phiy_m, 0.0), max(phiy_p
                            , 0.0)*max(phiy_p, 0.0))));
228                     }
229 
230                     if (AZ[i + j*N + k*P*P] >= 0){
231                         phiz.push_back(sqrt(max(max(phiz_m, 0.0)*max(phiz_m, 0.0), min(phiz_p
                            , 0.0)*min(phiz_p, 0.0))));
232                     } else if (AZ[i + j*N + k*P*P] < 0){
233                         phiz.push_back(sqrt(max(min(phiz_m, 0.0)*min(phiz_m, 0.0), max(phiz_p
                            , 0.0)*max(phiz_p, 0.0))));
234                     }
235                 }
236             }
237         }
238         return Derivative{phix, phiy, phiz};
239     }
240 
241     // first-order explicit Euler scheme used with the upwind scheme
242     void euler_upwind(vector<double> &phi, vector<double> AX, vector<double> AY, vector<
        double> AZ, int M, int N, int P, double dx, double dy, double dz, double dt){
243         auto [phix, phiy, phiz] = upwind(phi, AX, AY, AZ, M, N, P, dx, dy, dz);
244         phi = phi - dt*(AX*phix + AY*phiy + AZ*phiz);
245     }
246 
247     // third-order TVDRK scheme used with the upwind scheme
```

```
248 void TVDRK3_upwind(vector<double> &phi, vector<double> AX, vector<double> AY, vector<
    double> AZ, int M, int N, int P, double dx, double dy, double dz, double dt){
249
250     vector<double> n1;
251     vector<double> n2;
252     vector<double> n3_2;
253
254     {
255         auto [phix, phiy, phiz] = upwind(phi, AX, AY, AZ, M, N, P, dx, dy, dz);
256         n1 =  phi - dt*(AX*phix + AY*phiy + AZ*phiz);
257     }
258
259     {
260         auto [phix, phiy, phiz] = upwind(n1, AX, AY, AZ, M, N, P, dx, dy, dz);
261         n2 =  n1 - dt*(AX*phix + AY*phiy + AZ*phiz);
262     }
263
264     vector<double> n1_2 = 3/4*phi + 1/4*n2;
265
266     {
267         auto [phix, phiy, phiz] = upwind(n1_2, AX, AY, AZ, M, N, P, dx, dy, dz);
268         n3_2 =  n1_2 - dt*(AX*phix + AY*phiy + AZ*phiz);
269     }
270
271     phi = 1/3*phi + 2/3*n3_2;
272
273 }
274
275 // third-order TVDRK scheme used with the WENO scheme
276 void TVDRK3_weno(vector<double> &phi, vector<double> AX, vector<double> AY, vector<double
    > AZ, int M, int N, int P, double dx, double dy, double dz, double dt){
277
278     {
279     vector<double> n1;
280     vector<double> n2;
281     vector<double> n3_2;
282
283     {
284         auto [phix, phiy, phiz] = weno(phi, AX, AY, AZ, M, N, P, dx, dy, dz);
285         n1 =  phi - dt*(AX*phix + AY*phiy + AZ*phiz);
286     }
287
288     {
289         auto [phix, phiy, phiz] = weno(n1, AX, AY, AZ, M, N, P, dx, dy, dz);
290         n2 =  n1 - dt*(AX*phix + AY*phiy + AZ*phiz);
291     }
292
293     vector<double> n1_2 = 3.0/4*phi + 1.0/4*n2;
294
295     {
296         auto [phix, phiy, phiz] = weno(n1_2, AX, AY, AZ, M, N, P, dx, dy, dz);
297         n3_2 =  n1_2 - dt*(AX*phix + AY*phiy + AZ*phiz);
298     }
299
300     phi = 1.0/3*phi + 2.0/3*n3_2;
301
302     }
303
304     for(int i = 0; i < M; ++i){
305         for (int j = 0; j < N; ++j){
306             phi[2 + i*N + j*P*P] = phi[3 + i*N + j*P*P] - (phi[4 + i*N + j*P*P] - phi[3 +
                 i*N + j*P*P]);
307             phi[1 + i*N + j*P*P] = phi[2 + i*N + j*P*P] - (phi[3 + i*N + j*P*P] - phi[2 +
                 i*N + j*P*P]);
308             phi[0 + i*N + j*P*P] = phi[1 + i*N + j*P*P] - (phi[2 + i*N + j*P*P] - phi[1 +
                 i*N + j*P*P]);
309             phi[(M-3) + i*N + j*P*P] = phi[(M-4) + i*N + j*P*P] - (phi[(M-5) + i*N + j*P*
                P] - phi[(M-4) + i*N + j*P*P]);
310             phi[(M-2) + i*N + j*P*P] = phi[(M-3) + i*N + j*P*P] - (phi[(M-4) + i*N + j*P*
```

```
            P] - phi[(M-3) + i*N + j*P*P]);
311         phi[(M-1) + i*N + j*P*P] = phi[(M-2) + i*N + j*P*P] - (phi[(M-3) + i*N + j*P*
            P] - phi[(M-2) + i*N + j*P*P]);
312
313         phi[j + (2)*N + i*P*P] = phi[j + (3)*N + i*P*P] - (phi[j + (4)*N + i*P*P] -
            phi[j + (3)*N + i*P*P]);
314         phi[j + (1)*N + i*P*P] = phi[j + (2)*N + i*P*P] - (phi[j + (3)*N + i*P*P] -
            phi[j + (2)*N + i*P*P]);
315         phi[j + (0)*N + i*P*P] = phi[j + (1)*N + i*P*P] - (phi[j + (2)*N + i*P*P] -
            phi[j + (1)*N + i*P*P]);
316         phi[j + (N-3)*N + i*P*P] = phi[j + (N-4)*N + i*P*P] - (phi[j + (N-5)*N + i*P*
            P] - phi[j + (N-4)*N + i*P*P]);
317         phi[j + (N-2)*N + i*P*P] = phi[j + (N-3)*N + i*P*P] - (phi[j + (N-4)*N + i*P*
            P] - phi[j + (N-3)*N + i*P*P]);
318         phi[j + (N-1)*N + i*P*P] = phi[j + (N-2)*N + i*P*P] - (phi[j + (N-3)*N + i*P*
            P] - phi[j + (N-2)*N + i*P*P]);
319
320         phi[j + i*N + (2)*P*P] = phi[j + i*N + (3)*P*P] - (phi[j + i*N + (4)*P*P] -
            phi[j + i*N + (3)*P*P]);
321         phi[j + i*N + (1)*P*P] = phi[j + i*N + (2)*P*P] - (phi[j + i*N + (3)*P*P] -
            phi[j + i*N + (2)*P*P]);
322         phi[j + i*N + (0)*P*P] = phi[j + i*N + (1)*P*P] - (phi[j + i*N + (2)*P*P] -
            phi[j + i*N + (1)*P*P]);
323         phi[j + i*N + (N-3)*P*P] = phi[j + i*N + (N-4)*P*P] - (phi[j + i*N + (N-5)*P*
            P] - phi[j + i*N + (N-4)*P*P]);
324         phi[j + i*N + (N-2)*P*P] = phi[j + i*N + (N-3)*P*P] - (phi[j + i*N + (N-4)*P*
            P] - phi[j + i*N + (N-3)*P*P]);
325         phi[j + i*N + (N-1)*P*P] = phi[j + i*N + (N-2)*P*P] - (phi[j + i*N + (N-3)*P*
            P] - phi[j + i*N + (N-2)*P*P]);
326
327     }
328 }
329 }
330
331 // first-order explicit Euler scheme used with the weno scheme
332 void euler_weno(vector<double> &phi, vector<double> AX, vector<double> AY, vector<double>
     AZ, int M, int N, int P, double dx, double dy, double dz, double dt){
333     auto [phix, phiy, phiz] = weno(phi, AX, AY, AZ, M, N, P, dx, dy, dz);
334     phi = phi - dt*(AX*phix + AY*phiy + AZ*phiz);
335
336     for(int i = 0; i < M; ++i){
337         for (int j = 0; j < N; ++j){
338             phi[2 + i*N + j*P*P] = phi[3 + i*N + j*P*P] - (phi[4 + i*N + j*P*P] - phi[3 +
                 i*N + j*P*P]);
339             phi[1 + i*N + j*P*P] = phi[2 + i*N + j*P*P] - (phi[3 + i*N + j*P*P] - phi[2 +
                 i*N + j*P*P]);
340             phi[0 + i*N + j*P*P] = phi[1 + i*N + j*P*P] - (phi[2 + i*N + j*P*P] - phi[1 +
                 i*N + j*P*P]);
341             phi[(M-3) + i*N + j*P*P] = phi[(M-4) + i*N - j*P*P] + (phi[(M-5) + i*N + j*P*
            P] - phi[(M-4) + i*N + j*P*P]);
342             phi[(M-2) + i*N + j*P*P] = phi[(M-3) + i*N - j*P*P] + (phi[(M-4) + i*N + j*P*
            P] - phi[(M-3) + i*N + j*P*P]);
343             phi[(M-1) + i*N + j*P*P] = phi[(M-2) + i*N - j*P*P] + (phi[(M-3) + i*N + j*P*
            P] - phi[(M-2) + i*N + j*P*P]);
344
345             phi[j + (2)*N + i*P*P] = phi[j + (3)*N + i*P*P] - (phi[j + (4)*N + i*P*P] -
            phi[j + (3)*N + i*P*P]);
346             phi[j + (1)*N + i*P*P] = phi[j + (2)*N + i*P*P] - (phi[j + (3)*N + i*P*P] -
            phi[j + (2)*N + i*P*P]);
347             phi[j + (0)*N + i*P*P] = phi[j + (1)*N + i*P*P] - (phi[j + (2)*N + i*P*P] -
            phi[j + (1)*N + i*P*P]);
348             phi[j + (N-3)*N + i*P*P] = phi[j + (N-4)*N + i*P*P] - (phi[j + (N-5)*N + i*P*
            P] - phi[j + (N-4)*N + i*P*P]);
349             phi[j + (N-2)*N + i*P*P] = phi[j + (N-3)*N + i*P*P] - (phi[j + (N-4)*N + i*P*
            P] - phi[j + (N-3)*N + i*P*P]);
350             phi[j + (N-1)*N + i*P*P] = phi[j + (N-2)*N + i*P*P] - (phi[j + (N-3)*N + i*P*
            P] - phi[j + (N-2)*N + i*P*P]);
351
352             phi[j + i*N + (2)*P*P] = phi[j + i*N + (3)*P*P] - (phi[j + i*N + (4)*P*P] -
```

```
                phi[j + i*N + (3)*P*P]);
353             phi[j + i*N + (1)*P*P] = phi[j + i*N + (2)*P*P] - (phi[j + i*N + (3)*P*P] -
                phi[j + i*N + (2)*P*P]);
354             phi[j + i*N + (0)*P*P] = phi[j + i*N + (1)*P*P] - (phi[j + i*N + (2)*P*P] -
                phi[j + i*N + (1)*P*P]);
355             phi[j + i*N + (N-3)*P*P] = phi[j + i*N + (N-4)*P*P] - (phi[j + i*N + (N-5)*P*
                P] - phi[j + i*N + (N-4)*P*P]);
356             phi[j + i*N + (N-2)*P*P] = phi[j + i*N + (N-3)*P*P] - (phi[j + i*N + (N-4)*P*
                P] - phi[j + i*N + (N-3)*P*P]);
357             phi[j + i*N + (N-1)*P*P] = phi[j + i*N + (N-2)*P*P] - (phi[j + i*N + (N-3)*P*
                P] - phi[j + i*N + (N-2)*P*P]);
358         }
359     }
360 }
361
362 // third-order TVDRK scheme used with the Godunov scheme to solve the reinitialization
    equation
363 void TVDRK3_godunov_reinit(vector<double> &phi, int M, int N, int P, double dx, double dy
    , double dz, double dt, vector<double> phi0){
364
365     vector<double> n1;
366     vector<double> n2;
367     vector<double> n3_2;
368
369     for (int k = 0; k < P; ++k){
370         for (int j = 0; j < N; ++j){
371             for (int i = 0; i < M; ++i){
372                 if (i==0 || i==(M-1) || i==1 || i==M-2 || i==2 || i==M-3 || j==0 || j==(N
                    -1) || j==1 || j==N-2 || j==2 || j==N-3 || k==0 || k==(P-1) || k==1 || k
                    ==P-2 || k==2 || k==P-3){
373                     n1.push_back(phi[i+j*N+k*P*P]);
374                     continue;
375                 }
376
377                 double a = (phi[i+j*N+k*P*P]-phi[(i-1)+j*N+k*P*P])/dx;
378                 double b = (phi[(i+1)+j*N+k*P*P]-phi[i+j*N+k*P*P])/dx;
379                 double c = (phi[i+j*N+k*P*P]-phi[i+(j-1)*N+k*P*P])/dy;
380                 double d = (phi[i+(j+1)*N+k*P*P]-phi[i+j*N+k*P*P])/dy;
381                 double e = (phi[i+j*N+k*P*P]-phi[i+j*N+(k-1)*P*P])/dz;
382                 double f = (phi[i+j*N+(k+1)*P*P]-phi[i+j*N+k*P*P])/dz;
383
384                 double G;
385                 if (phi0[i+j*N+k*P*P] > 0){
386                     G = sqrt(max(max(a,0.0)*max(a,0.0), min(b,0.0)*min(b,0.0))
387                         + max(max(c,0.0)*max(c,0.0), min(d,0.0)*min(d,0.0))
388                         + max(max(e,0.0)*max(e,0.0), min(f,0.0)*min(f,0.0))) - 1;
389                 } else if (phi0[i+j*N+k*P*P] < 0){
390                     G = sqrt(max(min(a,0.0)*min(a,0.0), max(b,0.0)*max(b,0.0))
391                     + max(min(c,0.0)*min(c,0.0), max(d,0.0)*max(d,0.0))
392                     + max(min(e,0.0)*min(e,0.0), max(f,0.0)*max(f,0.0))) - 1;
393                 }
394                 n1.push_back(phi[i + j*N + k*P*P] - dt*sign(phi0[i+j*N+k*P*P])*G);
395             }
396         }
397     }
398
399     for (int k = 0; k < P; ++k){
400         for (int j = 0; j < N; ++j){
401             for (int i = 0; i < M; ++i){
402                 if (i==0 || i==(M-1) || i==1 || i==M-2 || i==2 || i==M-3 || j==0 || j==(N
                    -1) || j==1 || j==N-2 || j==2 || j==N-3 || k==0 || k==(P-1) || k==1 || k
                    ==P-2 || k==2 || k==P-3){
403                     n2.push_back(n1[i+j*N+k*P*P]);
404                     continue;
405                 }
406
407                 double a = (n1[i+j*N+k*P*P]-n1[(i-1)+j*N+k*P*P])/dx;
408                 double b = (n1[(i+1)+j*N+k*P*P]-n1[i+j*N+k*P*P])/dx;
409                 double c = (n1[i+j*N+k*P*P]-n1[i+(j-1)*N+k*P*P])/dy;
```

```
410                     double d = (n1[i+(j+1)*N+k*P*P]-n1[i+j*N+k*P*P])/dy;
411                     double e = (n1[i+j*N+k*P*P]-n1[i+j*N+(k-1)*P*P])/dz;
412                     double f = (n1[i+j*N+(k+1)*P*P]-n1[i+j*N+k*P*P])/dz;
413
414                     double G;
415                     if (phi0[i+j*N+k*P*P] > 0){
416                         G = sqrt(max(max(a,0.0)*max(a,0.0), min(b,0.0)*min(b,0.0))
417                             + max(max(c,0.0)*max(c,0.0), min(d,0.0)*min(d,0.0))
418                             + max(max(e,0.0)*max(e,0.0), min(f,0.0)*min(f,0.0))) - 1;
419                     } else if (phi0[i+j*N+k*P*P] < 0){
420                         G = sqrt(max(min(a,0.0)*min(a,0.0), max(b,0.0)*max(b,0.0))
421                         + max(min(c,0.0)*min(c,0.0), max(d,0.0)*max(d,0.0))
422                         + max(min(e,0.0)*min(e,0.0), max(f,0.0)*max(f,0.0))) - 1;
423                     }
424                     n2.push_back(n1[i + j*N + k*P*P] - dt*sign(phi0[i+j*N+k*P*P])*G);
425                 }
426             }
427         }
428
429         vector<double> n1_2 = (3.0/4*phi + 1.0/4*n2);
430
431         for (int k = 0; k < P; ++k){
432             for (int j = 0; j < N; ++j){
433                 for (int i = 0; i < M; ++i){
434                     if (i==0 || i==(M-1) || i==1 || i==M-2 || i==2 || i==M-3 || j==0 || j==(N
                        -1) || j==1 || j==N-2 || j==2 || j==N-3 || k==0 || k==(P-1) || k==1 || k
                        ==P-2 || k==2 || k==P-3){
435                         n3_2.push_back(n1_2[i+j*N+k*P*P]);
436                         continue;
437                     }
438
439                     double a = (n1_2[i+j*N+k*P*P]-n1_2[(i-1)+j*N+k*P*P])/dx;
440                     double b = (n1_2[(i+1)+j*N+k*P*P]-n1_2[i+j*N+k*P*P])/dx;
441                     double c = (n1_2[i+j*N+k*P*P]-n1_2[i+(j-1)*N+k*P*P])/dy;
442                     double d = (n1_2[i+(j+1)*N+k*P*P]-n1_2[i+j*N+k*P*P])/dy;
443                     double e = (n1_2[i+j*N+k*P*P]-n1_2[i+j*N+(k-1)*P*P])/dz;
444                     double f = (n1_2[i+j*N+(k+1)*P*P]-n1_2[i+j*N+k*P*P])/dz;
445
446                     double G;
447                     if (phi0[i+j*N+k*P*P] > 0){
448                         G = sqrt(max(max(a,0.0)*max(a,0.0), min(b,0.0)*min(b,0.0))
449                             + max(max(c,0.0)*max(c,0.0), min(d,0.0)*min(d,0.0))
450                             + max(max(e,0.0)*max(e,0.0), min(f,0.0)*min(f,0.0))) - 1;
451                     } else if (phi0[i+j*N+k*P*P] < 0){
452                         G = sqrt(max(min(a,0.0)*min(a,0.0), max(b,0.0)*max(b,0.0))
453                         + max(min(c,0.0)*min(c,0.0), max(d,0.0)*max(d,0.0))
454                         + max(min(e,0.0)*min(e,0.0), max(f,0.0)*max(f,0.0))) - 1;
455                     }
456                     n3_2.push_back(n1_2[i + j*N + k*P*P] - dt*sign(phi0[i+j*N+k*P*P])*G);
457                 }
458             }
459         }
460
461         phi = 1.0/3*phi + 2.0/3*n3_2;
462
463     }
464
465     // first-order explicit Euler scheme used with the Godunov scheme to solve the
        reinitialization equation
466     void euler_godunov_reinit(vector<double> &phi, int M, int N, int P, double dx, double dy,
         double dz, double dt, const vector<double> &phi0){
467         vector<double> phiNew;
468         for (int k = 0; k < P; ++k){
469             for (int j = 0; j < N; ++j){
470                 for (int i = 0; i < M; ++i){
471                     if (i==0 || i==(M-1) || i==1 || i==M-2 || i==2 || i==M-3 || j==0 || j==(N
                        -1) || j==1 || j==N-2 || j==2 || j==N-3 || k==0 || k==(P-1) || k==1 || k
                        ==P-2 || k==2 || k==P-3){
472                         phiNew.push_back(phi[i+j*N+k*P*P]);
```

```
                    continue;
                }

                double a = (phi[i+j*N+k*P*P]-phi[(i-1)+j*N+k*P*P])/dx;
                double b = (phi[(i+1)+j*N+k*P*P]-phi[i+j*N+k*P*P])/dx;
                double c = (phi[i+j*N+k*P*P]-phi[i+(j-1)*N+k*P*P])/dy;
                double d = (phi[i+(j+1)*N+k*P*P]-phi[i+j*N+k*P*P])/dy;
                double e = (phi[i+j*N+k*P*P]-phi[i+j*N+(k-1)*P*P])/dz;
                double f = (phi[i+j*N+(k+1)*P*P]-phi[i+j*N+k*P*P])/dz;

                double G;
                if (phi0[i+j*N+k*P*P] > 0){
                    G = sqrt(max(max(a,0.0)*max(a,0.0), min(b,0.0)*min(b,0.0))
                        + max(max(c,0.0)*max(c,0.0), min(d,0.0)*min(d,0.0))
                        + max(max(e,0.0)*max(e,0.0), min(f,0.0)*min(f,0.0))) - 1;
                } else if (phi0[i+j*N+k*P*P] < 0){
                    G = sqrt(max(min(a,0.0)*min(a,0.0), max(b,0.0)*max(b,0.0))
                    + max(min(c,0.0)*min(c,0.0), max(d,0.0)*max(d,0.0))
                    + max(min(e,0.0)*min(e,0.0), max(f,0.0)*max(f,0.0))) - 1;
                }
                phiNew.push_back(phi[i + j*N + k*P*P] - dt*sign(phi0[i+j*N+k*P*P])*G);
            }
        }
    }
    phi = phiNew;
}

// sign function that returns 1 for a positive value, -1 for a negative value, and 0 for
a value of 0
int sign(double num){
    int res;
    if (num < 0){
        res = -1;
    } else if (num > 0){
        res = 1;
    } else if (num == 0){
        res = 0;
    }
    return res;
}

```

A.5 testCases.h and testCases.cpp

These files contain code to generate the velocity fields and the computation of the error measures introduced in section 4.6. testCases.h is given first and contains the header file, while testCases.cpp contains the whole implementation.

```cpp
#pragma once

#include <vector>
#include <tuple>
#include <cmath>

#include "initialization.h"
#include "schemes.h"

using namespace std;

#define PI 3.14159265

// velocity vector with one value for each direction in 3D
struct Velocity {
    vector<double> x;
    vector<double> y;
    vector<double> z;
};

// velocity field for deformation in 3D. Taken from LeVeque (1996)
Velocity vortexVelocity(int M, int N, int P, vector<double> X, vector<double> Y, vector<double> Z, double t, double T);

// velocity field for deformation in 2D. Taken from Morgan and Waltz (2017)
Velocity shearedSphereVelocity(int M, int N, int P, vector<double> X, vector<double> Y, vector<double> Z, double t, double T);

// simple velocity field with u = v = w = 1 for all grid nodes
Velocity simpleVelocity(int M, int N, int P);

// returns the volume of the domain bounded by the zero contour in a signed distance field
double volume(vector<double> &phi, double dx, double dy, double dz);

// returns the surface area of the domain bounded by the zero contour in a signed distance field
double surfaceArea(vector<double> &phi, double dx, double dy, double dz, double M, double N, double P);

// error measure of the interface error
double interfaceError(vector<double> &phi0, vector<double> &phi, double dx, double dy, double dz, double M, double N, double P);

// error measure of the average mass error
double massError(vector<double> &phi, double dx, double dy, double dz, double M, double N, double P);

```

```cpp
#include "testCases.h"

// velocity field for deformation in 3D. Taken from LeVeque (1996)
Velocity vortexVelocity(int M, int N, int P, vector<double> X, vector<double> Y, vector<double> Z, double t, double T){

    vector<double> U;
    vector<double> V;
    vector<double> W;

    for (int k = 0; k < P; ++k){
        for (int j = 0; j < N; ++j){
            for (int i = 0; i < M; ++i){
                U.push_back(2*sin(PI*X[i])*sin(PI*X[i])*sin(2*PI*Y[j])*sin(2*PI*Z[k])*cos(PI*t/T));
                V.push_back(-sin(2*PI*X[i])*sin(PI*Y[j])*sin(PI*Y[j])*sin(2*PI*Z[k])*cos(PI*t/T));
                W.push_back(-sin(2*PI*X[i])*sin(2*PI*Y[j])*sin(PI*Z[k])*sin(PI*Z[k])*cos(PI*t/T));
            }
        }
    }
    return Velocity {U, V, W};
}

// velocity field for deformation in 2D. Taken from Morgan and Waltz (2017)
Velocity shearedSphereVelocity(int M, int N, int P, vector<double> X, vector<double> Y, vector<double> Z, double t, double T){

    vector<double> U;
    vector<double> V;
    vector<double> W;

    for (int k = 0; k < P; ++k){
        for (int j = 0; j < N; ++j){
            for (int i = 0; i < M; ++i){
                U.push_back(sin(PI*X[i])*cos(PI*Y[j])*cos(PI*t/T));
                V.push_back(-cos(PI*X[i])*sin(PI*Y[j])*cos(PI*t/T));
                W.push_back(0.0);
            }
        }
    }
    return Velocity {U, V, W};
}

// simple velocity field with u = v = w = 1 for all grid nodes
Velocity simpleVelocity(int M, int N, int P){
    vector<double> U = linspace(1, 1, M*N*P);
    vector<double> V = linspace(1, 1, M*N*P);
    vector<double> W = linspace(1, 1, M*N*P);
    return Velocity {U, V, W};
}

// returns the volume of the domain bounded by the zero contour in a signed distance field
double volume(vector<double> &phi, double dx, double dy, double dz){
    double epsilon = 1.5*dx;
    double V = 0;
    for (int i = 0; i < phi.size(); ++i){
        double H;
        if (phi[i] < -epsilon){
            H = 0.0;
        } else if (-epsilon <= phi[i] && phi[i] <= epsilon){
            H = 0.5 + phi[i]/(2*epsilon) + 1/(2*PI)*sin(PI*phi[i]/epsilon);
        } else if (epsilon < phi[i]){
            H = 1.0;
        }
        V += (1-H)*dx*dy*dz;
    }
```

```
64          return V;
65      }
66
67      // returns the surface area of the domain bounded by the zero contour in a signed
        distance field
68      double surfaceArea(vector<double> &phi, double dx, double dy, double dz, double M, double
         N, double P){
69          double A = 0;
70          double epsilon = 1.5*dx;
71          double phix;
72          double phiy;
73          double phiz;
74          for (int k = 0; k < P; ++k){
75              for (int j = 0; j < N; ++j){
76                  for (int i = 0; i < M; ++i){
77                      if (i==0 || i==(M-1) || j==0 || j==(N-1) || k==0 || k==(P-1)){
78                          continue;
79                      }
80                      double phix = (phi[(i+1)+j*N+k*P*P] - phi[(i-1)+j*N+k*P*P])/(2*dx);
81                      if (phix == 0){
82                          phix = (phi[(i+1)+j*N+k*P*P] - phi[i+j*N+k*P*P])/(dx);
83                      }
84                      double phiy = (phi[i+(j+1)*N+k*P*P] - phi[i+(j-1)*N+k*P*P])/(2*dy);
85                      if (phiy == 0){
86                          phiy = (phi[i+(j+1)*N+k*P*P] - phi[i+j*N+k*P*P])/(dy);
87                      }
88                      double phiz = (phi[i+j*N+(k+1)*P*P] - phi[i+j*N+(k-1)*P*P])/(2*dz);
89                      if (phiz == 0){
90                          phiz = (phi[i+j*N+(k+1)*P*P] - phi[i+j*N+k*P*P])/(dz);
91                      }
92                      double sigma;
93                      if (phi[i + j*N + k*P*P] < -epsilon){
94                          sigma = 0.0;
95                      } else if (-epsilon <= phi[i + j*N + k*P*P] && phi[i + j*N + k*P*P] <=
                        epsilon){
96                          sigma = (1/(2*epsilon) + 1/(2*epsilon)*cos(phi[i + j*N + k*P*P]*PI/
                            epsilon));
97                      } else if (epsilon < phi[i + j*N + k*P*P]){
98                          sigma = 0.0;
99                      }
100                     A += (sigma)*sqrt(phix*phix + phiy*phiy + phiz*phiz)*dx*dy*dz;
101                 }
102             }
103         }
104         return A;
105     }
106
107     // error measure of the interface error
108     double interfaceError(vector<double> &phi0, vector<double> &phi, double dx, double dy,
        double dz, double M, double N, double P){
109         double epsilon = 1.5*dx;
110         double A = surfaceArea(phi0, dx, dy, dz, M, N, P);
111         double L1 = 0;
112         for (int k = 0; k < P; ++k){
113             for (int j = 0; j < N; ++j){
114                 for (int i = 0; i < M; ++i){
115                     L1 += abs(1.0*(phi0[i + j*N + k*P*P] < 0) - 1.0*(phi[i + j*N + k*P*P] < 0
                        ))*dx*dy*dz;
116                 }
117             }
118         }
119         return L1/A;
120     }
121
122     // error measure of the average mass error
123     double massError(vector<double> &phi, double dx, double dy, double dz, double M, double N
        , double P){
124         double error = 0;
125         for (int k = 0; k < P; ++k){
```

```
126             for (int j = 0; j < N; ++j){
127                 for (int i = 0; i < M; ++i){
128                     error += abs(1.0*(phi[i + j*N + k*P*P] < 0))*dx*dy*dz;
129                 }
130             }
131         }
132         return error;
133     }
134
```

A.6 vectorUtilities.h and vectorUtilities.cpp

These files contain code used for various vector operations. vectorUtilities.h is given first and contains the header file, while vectorUtilities.cpp contains the whole implementation.

```cpp
#pragma once

#include <vector>
#include <iostream>
#include <cmath>
#include <limits>
using namespace std;

// multiplies the elements of two vectors
vector<double> operator*(vector<double> const &vec1, vector<double> const &vec2);

// divides the elements of one vector with another vector
vector<double> operator/(vector<double> const &vec1, vector<double> const &vec2);

// adds the elements of two vectors
vector<double> operator+(vector<double> const &vec1, vector<double> const &vec2);

// multiplies the elements of a vector with a scalar
vector<double> operator*(double const &scalar, vector<double> const &vec);

// multiplies the elements of a vector with a scalar
vector<double> operator*(vector<double> const &vec, double const &scalar);

// divides the elements of a vector with a scalar
vector<double> operator/(vector<double> const &vec, double const &scalar);

// divides a scalar with the elements of a vector
vector<double> operator/(double const &scalar, vector<double> const &vec);

// subtracts the elements of one vector with another vector
vector<double> operator-(vector<double> const &vec1, vector<double> const &vec2);

// adds the elements of a vector with a scalar
vector<double> operator+(vector<double> const &vec, double const &scalar);

// adds the elements of a vector with a scalar
vector<double> operator+(double const &scalar, vector<double> const &vec);

// subtracts the elements of a vector with a scalar
vector<double> operator-(vector<double> const &vec, double const &scalar);

// subtracts a scalar with the elements of a vector
vector<double> operator-(double const &scalar, vector<double> const &vec);

// takes the absolute value of all elements of a vector
vector<double> vectorAbs(vector<double> const &vec);

// returns the maximum value of all elements in a vector
double vectorMax(vector<double> const &vec);

// takes the square root of all elements of a vector
vector<double> vectorSqrt(vector<double> const &vec);

```

```cpp
#include "vectorUtilities.h"

// multiplies the elements of two vectors
vector<double> operator*(vector<double> const &vec1, vector<double> const &vec2){
    vector<double> res;
    if (vec1.size() != vec2.size()){
        cerr << "Vectors multiplication with different sized vectors." << endl;
    }
    for (int i = 0; i < vec1.size(); ++i){
        res.push_back(vec1[i]*vec2[i]);
    }
    return res;
}

// divides the elements of one vector with another vector
vector<double> operator/(vector<double> const &vec1, vector<double> const &vec2){
    vector<double> res;
    if (vec1.size() != vec2.size()){
        cerr << "Vectors division with different sized vectors." << endl;
    }
    for (int i = 0; i < vec1.size(); ++i){
        res.push_back(vec1[i]/vec2[i]);
    }
    return res;
}

// adds the elements of two vectors
vector<double> operator+(vector<double> const &vec1, vector<double> const &vec2){
    vector<double> res;
    if (vec1.size() != vec2.size()){
        cerr << "Vectors addition with different sized vectors." << endl;
    }
    for (int i = 0; i < vec1.size(); ++i){
        res.push_back(vec1[i] + vec2[i]);
    }
    return res;
}

// multiplies the elements of a vector with a scalar
vector<double> operator*(double const &scalar, vector<double> const &vec){
    vector<double> res;
    for (int i = 0; i < vec.size(); ++i){
        res.push_back(scalar*vec[i]);
    }
    return res;
}

// multiplies the elements of a vector with a scalar
vector<double> operator*(vector<double> const &vec, double const &scalar){
    vector<double> res;
    for (int i = 0; i < vec.size(); ++i){
        res.push_back(vec[i]*scalar);
    }
    return res;
}

// divides the elements of a vector with a scalar
vector<double> operator/(vector<double> const &vec, double const &scalar){
    vector<double> res;
    for (int i = 0; i < vec.size(); ++i){
        res.push_back(vec[i]/scalar);
    }
    return res;
}

// divides a scalar with the elements of a vector
vector<double> operator/(double const &scalar, vector<double> const &vec){
    vector<double> res;
    for (int i = 0; i < vec.size(); ++i){
```

```
70              res.push_back(scalar/vec[i]);
71          }
72          return res;
73      }
74
75      // subtracts the elements of one vector with another vector
76      vector<double> operator-(vector<double> const &vec1, vector<double> const &vec2){
77          vector<double> res;
78          if (vec1.size() != vec2.size()){
79              cerr << "Vectors subtraction with different sized vectors." << endl;
80          }
81          for (int i = 0; i < vec1.size(); ++i){
82              res.push_back(vec1[i] - vec2[i]);
83          }
84          return res;
85      }
86
87      // adds the elements of a vector with a scalar
88      vector<double> operator+(vector<double> const &vec, double const &scalar){
89          vector<double> res;
90          for (int i = 0; i < vec.size(); ++i){
91              res.push_back(vec[i]+scalar);
92          }
93          return res;
94      }
95
96      // adds the elements of a vector with a scalar
97      vector<double> operator+(double const &scalar, vector<double> const &vec){
98          vector<double> res;
99          for (int i = 0; i < vec.size(); ++i){
100             res.push_back(scalar + vec[i]);
101         }
102         return res;
103     }
104
105     // subtracts the elements of a vector with a scalar
106     vector<double> operator-(vector<double> const &vec, double const &scalar){
107         vector<double> res;
108         for (int i = 0; i < vec.size(); ++i){
109             res.push_back(vec[i]-scalar);
110         }
111         return res;
112     }
113
114     // subtracts a scalar with the elements of a vector
115     vector<double> operator-(double const &scalar, vector<double> const &vec){
116         vector<double> res;
117         for (int i = 0; i < vec.size(); ++i){
118             res.push_back(scalar - vec[i]);
119         }
120         return res;
121     }
122
123     // takes the absolute value of all elements of a vector
124     vector<double> vectorAbs(vector<double> const &vec){
125         vector<double> res;
126         for (unsigned int i = 0; i < vec.size(); ++i){
127             if (vec[i] < 0){
128                 res.push_back(-vec[i]);
129             } else {
130                 res.push_back(vec[i]);
131             }
132         }
133         return res;
134     }
135
136     // returns the maximum value of all elements in a vector
137     double vectorMax(vector<double> const &vec){
138         double max = numeric_limits<double>::lowest();
```

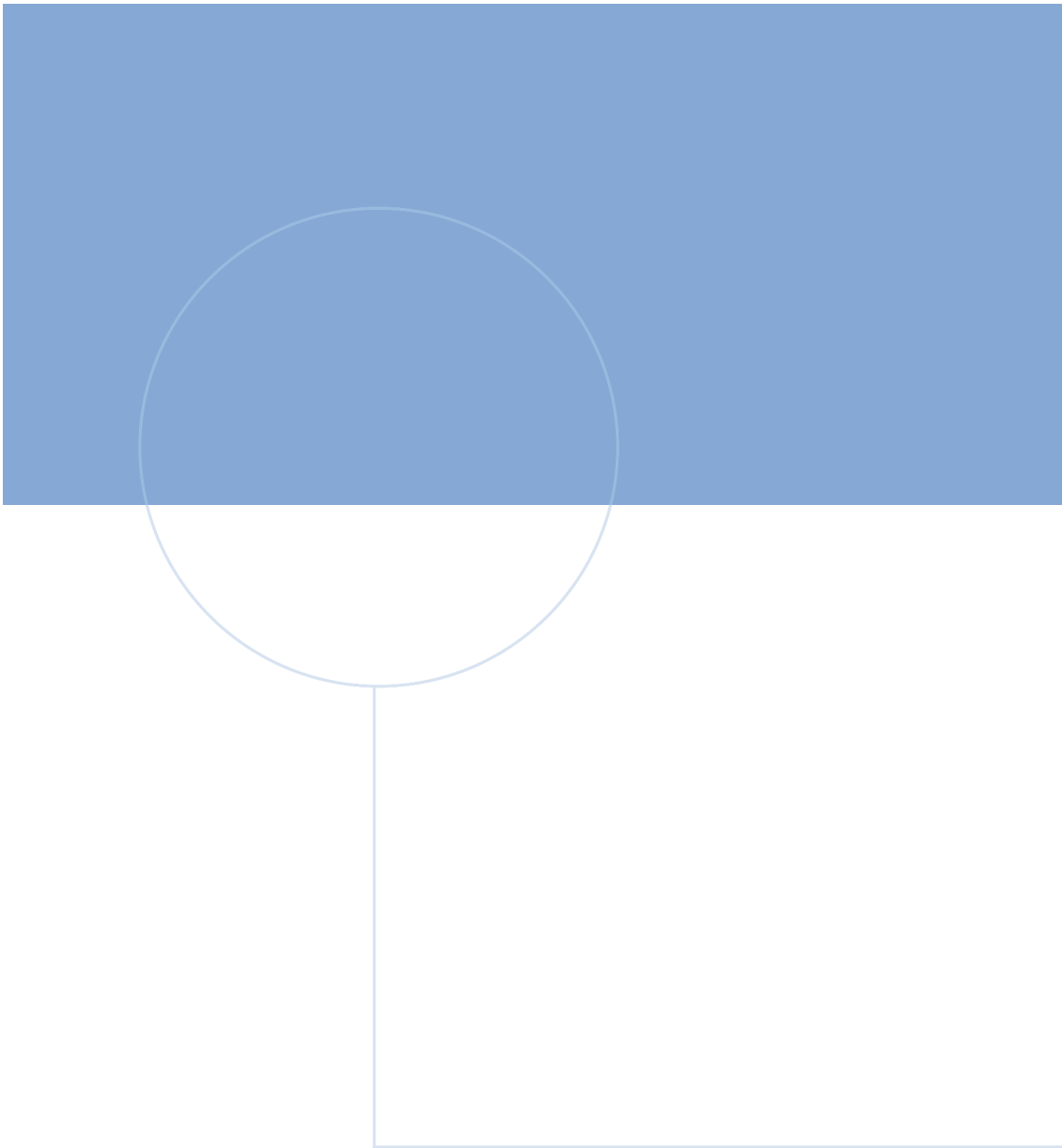
```
139         for (unsigned int i = 0; i < vec.size(); ++i){
140             if (isfinite(vec[i]) && (vec[i] > max)){
141                 max = vec[i];
142             }
143         }
144         return max;
145     }
146 
147     // takes the square root of all elements of a vector
148     vector<double> vectorSqrt(vector<double> const &vec){
149         vector<double> res;
150         for (unsigned int i = 0; i < vec.size(); ++i){
151             res.push_back(sqrt(vec[i]));
152         }
153         return res;
154     }
155 
```

A.7 plotter.py

This file contains the Python code to find the interface with the marching cubes algorithm using the scikit-image Python library, and also code to plot the interface and the particles used in the particle level set method.

```
from matplotlib import projections
import matplotlib.pyplot as plt
import numpy as np
from skimage import measure # scikit-image library
from mpl_toolkits.mplot3d.art3d import Poly3DCollection

# reads a .txt-file and returns the signed distance field and the number of nodes in each direction
def readFile(filename):
    f = open(filename)

    firstLine = f.readlines()[0].split(',')
    f.close()
    m = int(firstLine[0])
    n = int(firstLine[1])
    p = int(firstLine[2])

    f = open(filename)
    phi = np.zeros((m,n,p))
    lines = f.readlines()[1:]
    count = 0
    for k in range(p):
        for j in range(n):
            for i in range(m):
                phi[i,j,k] = float(lines[count].split(',')[3])
                count += 1
    return phi, m, n, p

# takes a signed distance field and the number of nodes in each direction
# uses the marching cubes algorithm to find the zero-contour and plots this contour
def getSurface(volume, m, n, p, level=0, plot=True, filename='fig'):
    verts, faces, normals, values = measure.marching_cubes(volume, level) # marching cubes algorithm
    size = max(m, n, p)
    if plot:
        fig = plt.figure()
        ax = fig.add_subplot(projection='3d')
        ax.set_box_aspect([1,1,1])
        mesh = Poly3DCollection(verts[faces]/size)
        mesh.set_edgecolor('k')
        ax.add_collection3d(mesh)
        plt.tight_layout()
        ax.set_xlim(0, 1)
        ax.set_ylim(0, 1)
        ax.set_zlim(0, 1)
        ax.view_init(elev=0., azim=0)
        ax.set_xlabel('x', fontsize=14, style='italic')
        ax.set_ylabel('y', fontsize=14, style='italic')
        ax.set_zlabel('z', fontsize=14, style='italic')
        plt.savefig(filename + '.pdf', dpi=900, format='pdf',bbox_inches='tight')
        plt.close()

# plots the particles from the particle level set method
def plotParticle(filename):
    f = open(filename + '.txt')

    x = []
    y = []
    z = []

    for line in f.readlines():
        parsedLine = line.split(',')
        x.append(float(parsedLine[0]))
        y.append(float(parsedLine[1]))
        z.append(float(parsedLine[2]))

    fig = plt.figure()
```

```
66        ax = fig.add_subplot(projection='3d')
67        ax.scatter(x,y,z)
68        plt.xlabel('x', fontsize=14, style='italic')
69        plt.ylabel('y', fontsize=14, style='italic')
70        plt.savefig(filename + '.pdf', dpi=900, format='pdf',bbox_inches='tight')
71        plt.close()
72
73    # main function for plotting
74    def main():
75        plt.rcParams['font.family'] = 'serif'
76        plt.rcParams['font.serif'] = ['Times New Roman']
77        path = 'figures/'
78
79        f = open(path + 'plotTimes.txt')
80        # plots signed distance field for all time steps recorded in plotTimes.txt
81        for line in f.readlines():
82            phi, m, n, p = readFile(path + line[:-1] + '.txt')
83            getSurface(phi, m, n, p, 0, True, path + line[:-1])
84
85        g = open(path + 'plotTimesParticle.txt')
86        # plots particles for all time steps recorded in plotTimesParticle.txt
87        for line in g.readlines():
88            plotParticle(line[:-1])
89
90    if __name__=='__main__':
91        main()
92
```



NTNU

Norwegian University of
Science and Technology