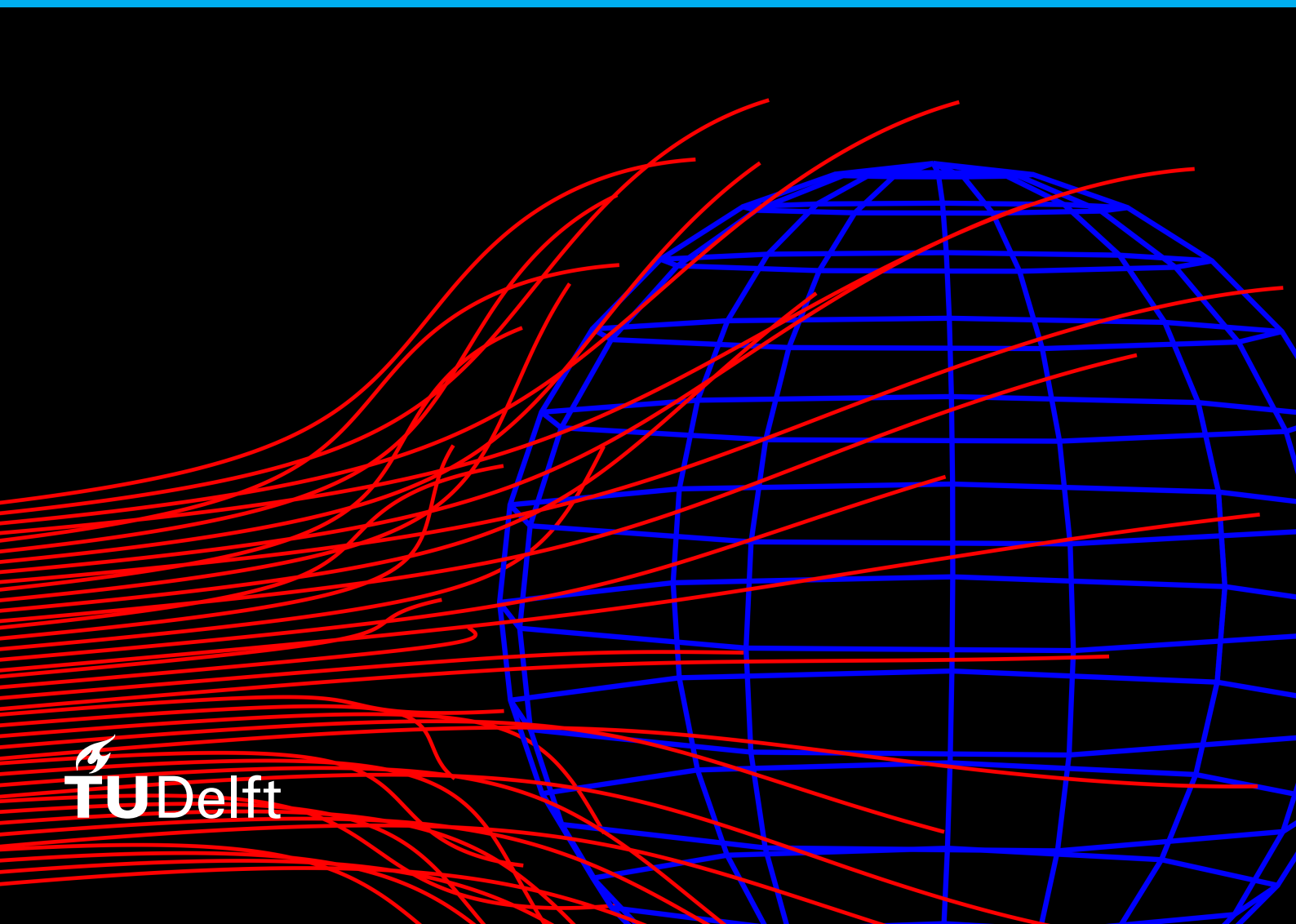
A Lagrangian passive scalar solver for mass transport in electrolytes and coupling to the particle-resolved Bluebottle

Code Development, Testing & Validation

S.S. Hemamalini



TU Delft

A Lagrangian passive scalar solver for mass transport in electrolytes and coupling to the particle-resolved Bluebottle

Code Development, Testing & Validation

by

S.S. Hemamalini

to obtain the degree of Master of Science

at the Delft University of Technology,

to be defended publicly on Monday, December 13, 2021 at 14:15.

Student number:	5071984	
Project duration:	February 1, 2021 – December 13, 2021	
Thesis committee:	Dr. Lorenzo Botto,	TU Delft, supervisor
	Dr. Johan Padding,	TU Delft
	Dr. Rene Pecnik,	TU Delft
	Dr. Luis Portela,	TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Acknowledgements

Introspectively, I have always known that a logical problem excites me unlike anything else so much so that I would consider solving Project Euler problems and writing code as one of my good hobbies. To be able to visualise fluid dynamics on a computer is a roller coaster you have to ride to understand. From understanding the phenomena in a mathematical form through intuitive yet complex equations, to replicating our understanding as a code, and at the end finding out some small piece of the puzzle is still missing — it is perhaps, I would say, the hardest logical problem, maybe even a never-ending problem. And so lies my affinity to coding and CFD. This thesis is me doing what I do best and what I like best. If something can come out of it, I couldn't be happier.

I would like to thank my thesis supervisor **Dr. Lorenzo Botto** for giving me an opportunity to work on what I would consider the perfect project — fundamental code development for a complex problem. I really enjoyed the journey from the start to end of my thesis. I have learned a lot from his expertise on understanding the physics and his grasp on the mathematics behind a physical phenomena.

I would also like to thank **Dr. Luis Portela** for his guidance on the course *Computational Multiphase Flow*. It was the first time I had written a fully working code from scratch and his lessons honed my skills in the field and served as an excellent booster to my thesis and my future.

Next, I would like to thank my friends at the Delft University of Technology — **Aswin**, **Ravi** and **Venkat**. I will always miss the days when we solved coursework assignments in Roland Holstlaan 193 together, and the group cooking, and the cycling to new places. I would also like to thank **Sneha** for being a constant companion in my Masters journey. Without them, the topsy-turvy world of COVID lockdowns wouldn't have been as colorful.

I would also like to thank **Lily** for providing me an opportunity to be a TA and to teach Python. I am grateful for her hospitality and support.

Next, I would like to thank **Dr. Madhu Ganesh** and **Dr. Krishnan** back home who laid the early stones in my path and were instrumental in shaping me for my Masters. I still remember those early days of learning CFD and running Ansys Fluent non-stop. I am also thankful to my friends back home (**Arrvind**, **Subhash**, **Nived**) who have been there for me right from the start of my career.

Finally, I would like to thank my **Amma** for her emotion and care. All of my achievements will bear your name, Amma, (I am literally called Mr. Hemamalini here) and what you have given me cannot be quantified. I am also indebted to my family back home — **my Thatha, my Paati, Ganesh Mama & Ramesh Mama** — they will always be with me, mentally if not physically, and I know that I will always be with them likewise.

S.S. Hemamalini
Delft, December 2021

And we count these moments. These moments when we dare to aim higher, to break barriers, to reach for the stars, to make the unknown known. We count these moments as our proudest achievements. But we lost all that. Or perhaps we've just forgotten that we are still pioneers. And we've barely begun. And that our greatest accomplishments cannot be behind us, that our destiny lies above us.

- Cooper, Interstellar (2014)

Abstract

Water electrolysis is a popular energy storage technique used in tandem with many renewable sources to convert the generated energy into storeable hydrogen. The efficiency of water electrolyzers is greatly affected by overpotential losses. Bubble evolution is a unique characteristic of flows in water electrolysis. The evolution of bubbles alter electrokinetics close to the electrodes and vary ionic mass transport. The localized flow features close to a bubble are often attributed to the variation in the current density and subsequently, the electrolyzer efficiency. For the purpose of modelling flows close to a bubble better, a Lagrangian method for the simulation of passive tracers is developed and programmed to be coupled to the flow field of *Bluebottle*, an open-source particulate multiphase flow solver that uses the PHYSALIS algorithm.

The dynamics of the tracers are modelled using a simplified Langevin equation. In the present work, the migration flux is omitted and priority is given to convection and diffusion with the objective of establishing a foundation for the simulation of ionic mass transport. Brownian motion is described using a random displacement term. The coupling with the flow field is achieved using trilinear interpolation. The domain boundaries in regards to tracer dynamics are modelled as either a rigid wall pair or as a periodic boundary pair. Specular reflection is programmed for the former ensuring elastic collision of a tracer with the domain boundary. For the latter, the tracer position is altered so as to place the tracer in the opposite side of the domain in the axis of intrusion. Particles are assumed to be non-penetrative and hence, specular reflection is implemented at the surface of each particle. Since the *tracer* module is coupled one-way with the flow field and executed after a *Bluebottle* time-step, a subroutine is developed to push the tracers out of a particle radially if a tracer is located inside a particle after a *Bluebottle* time-step. To ensure particle interaction is ensured across periodic boundaries, a subroutine is developed that places the particle in an apparent location that enables particle-tracer interaction.

The module execution time is found to be linearly proportional to the number of particles and the number of tracers and consumes roughly 10% of a *Bluebottle* iteration execution time in nominal tests. The module is tested to ensure the Brownian displacement term obeys diffusion statistics and also to ensure that the trilinear interpolation works as intended. The numerically enforced no-penetration boundary at particle surfaces is also tested and observed to prevent intrusion of tracers.

The tracer module is then used to stochastically simulate mass transport across a particulate suspension in a stagnant and a sheared flow field. The Sherwood number Sh determined from the tracer module is found to agree well with the expected experimental and numerical results of Wang et al. (2009). The tracer module is also compared to a scalar field solver of *Bluebottle*. The tracer module is observed to capture features of the flow field quite well. However, the transient tracer positions upon conversion to a transient continuous concentration field exhibits noise due to the discrete nature of the tracers. Hence, transient comparisons with a continuous field in terms of absolute magnitude requires a large number of tracers.

Recommendations for improvement of the code is provided. The present work is intended to be followed up with the addition of migration flux to the equations of motion for the tracers through the solution of an additional equation for velocity of the tracer using a force equivalence of Coulomb's law and Stokes' drag law. Future challenges that will be encountered in the development of an accurate ionic mass transport solver is briefly discussed.

Contents

1 Introduction

The world at present is facing an energy crisis fueled by a growing threat of climate change. The use of non-renewable resources such as petroleum and natural gas is slowly waning and allowing sustainable, renewable resources to become the primary source of energy [28]. As a result, the research into renewable energy production and sustainable alternatives to non-renewable energy is the highest it has ever been. Major renewable resources, though, are at a disadvantage compared to non-renewable resources that the availability of resources is highly seasonal and fluctuating. Resources such as solar energy fluctuate on a daily timescale whereas resources such as wind energy fluctuate on a seasonal timescale [39]. This fluctuation requires a method for mitigation without any major losses to the harvested energy. A simple way of distributing the fluctuating energy input into a constant stable output is using energy storage systems as buffer to receive surplus energy and to provide energy during deficit [25, 27]. Several storage systems are already used both at the location of the energy production such as wind farms and nuclear power plants, and also at the domestic level.

In the industry, there are several methods to achieve energy conversion and subsequent storage. A relatively new method that is used at present at the location of energy generation is water electrolysis [9, 36]. The energy harvested from the renewable energy sources are fed to a water electrolyzer that splits water into oxygen and hydrogen. Therefore, the harvested energy is stored in the form of chemical bonds of oxygen and hydrogen. This stored energy is regenerated through combustion of hydrogen and oxygen and electricity is generated. Water electrolysis is advantageous compared to other energy storage methods that it is a fully clean and green method of energy storage [36]. Water electrolysis does not involve harmful pollutants and only requires water as an input resource. The hydrogen produced has a deep impact in several sectors other than energy storage. As the combustion of hydrogen leaves no residue, the mainstream use of hydrogen is highly impactful in the decarbonisation of the industry [29]. However, the addition of a water electrolyzer and a regenerator brings forth an additional component in the energy loop which might potentially decrease the efficiency of energy generation [3]. There are a multitude of advanced research foci to improve this efficiency, thereby increasing the energy throughput.

Water electrolyzers commonly used in the industry are generally of two types – alkaline water electrolyzer and Polymer-Electrolyte-Membrane (PEM) water electrolyzer. Alkaline water electrolyzers contain an alkaline solution of potassium hydroxide or sodium hydroxide as the electrolyte. PEM electrolyzers contain a Solid-Polymer-Electrolyte (SPE) that conducts hydrogen ions (protons) from one electrode to the other. Commercially, alkaline electrolyers are used in stacks of several hundreds of electrodes placed in orders of hundreds of micrometers apart [44]. One of the primary foci

of the research community is the study of ionic mass transport across the two electrodes of the alkaline water electrolyzer [2, 4, 10, 14, 52]. Heat and overpotential losses are some of the major losses affecting an alkaline water electrolyzer and these are dependent on the flow of the ions through the electrolyte [3]. In a conventional electrolyzer, ions are accelerated through the electrolyte solution through diffusive, convective and migrative fluxes. However, in the case of a water electrolyzer, the motion is complex as the production of hydrogen and oxygen gases in the form of bubbles impede this motion in remarkable ways. The presence of bubbles add an additional convective element to the mass transport of the ions but hinder their motion by acting as obstacles [3, 14, 17, 18]. A deeper look into the effect of bubbles on water electrolysis and gas-evolving electrodes is provided and discussed in the following section.

1.1. Literature Review

1.1.1. Effect of bubbles on electrochemical processes

The production of bubbles at the electrodes has been extensively studied and a theoretical model highlighting the dependencies and the implications of bubble evolution was developed by Vogt [56]. Dukovic and Tobias qualitatively distinguished the effects of bubble evolution on the various potential losses seen in electrolytic processes [17]. The total potential loss ΔU_L can be described as the sum of different overpotentials as:

$$\Delta U_L = \eta_{ac} + \eta_{ohm} + \eta_{conc} \tag{1.1}$$

The formation of bubbles have been described to have complex relationships with all of the overpotentials stated in Equation 1.1.

Activation overpotentials (η_{ac}) denote the energy required to transfer charges from the electrode to the electrolyte at the surface of the electrode. Bubbles typically attach to the surface of the electrode and detach only upon reaching a critical radius [20]. This results in bubbles covering the electrode and reducing the electrocatalytic area. Since the area of contact between the electrode and the electrolyte is decreased, more energy is required to transfer the same charges from the electrode to the electrolyte. This results in an increase in activation overpotential of the respective electrode. Dukovic and Tobias devised a relation between the activation overpotential and the bubble coverage [17].

Concentration overpotential (η_{conc}) denotes the resistance to the electrochemical reaction taking place at the electrode surface due to the saturation level of the products close to the electrode surface. Upon supersaturation of the products, the reaction thermodynamically reverses and favors the formation of the reactants from the products. This is typically seen in reactions with stagnant electrolytes where the ionic mass transfer is purely diffusion-driven. Bubbles cause turbulent mixing close to the electrode and aid in the motion of ions. Hence, bubbles are known to reduce potential losses through concentration overpotential. In cases of low current densities ($< 100 \text{mA/cm}^2$) where concentration overpotential dominates the potential losses, formation of bubbles reduce the losses and improves current density at the electrode [19].

Ohmic losses (η_{ohm}) denote the potential drop due to the resistance faced by an ion while moving from one electrode to the other. The electrolyte contributes the most to the Ohmic resistance and the potential drop often follows the Ohm's law with the current density and the conductivity of the electrolyte. However, the presence of bubbles reduce the conductivity of the electrolyte since they effectively act as obstacles for the flow of ions. Due to the localisation of the bubbles in the electrolyte, the effect of bubbles on Ohmic overpotential is only known empirically. Ohmic losses due to bubbles dominate the potential losses at higher current densities [3]. Most industrial processes

are carried out at very high current densities. Hence, in many such industrial cases, Ohmic losses are the primary cause of loss in potential and thereby, loss in input energy. Ohmic overpotentials have been reported to be avoided by using a forced flow of the electrolyte over the electrode surface, reducing the average void fraction close to the electrode surface and inducing turbulent mixing [7].

Studies have shown that forced flow of electrolyte over the surface of the electrode vastly improve the potential losses by minimising the bubble diameter, decreasing the bubble coverage and reducing high temperature gradients [7, 24]. Several unconventional techniques exist, such as ultrasonic bubble removal, magnetic field-driven convection, to either impede the formation of bubbles or decrease the critical diameter for bubble detachment, thereby improving electrochemical efficiency [30, 32, 34].

Since the effect of bubbles on electrolytic processes is highly complex, an understanding of the flow near a gas-evolving electrode is essential.

1.1.2. Flows near gas-evolving electrodes

The presence of bubbles alter the dynamics of flow close to the electrode. The volume close to the electrode can be represented by different zones of relevance to the current work as the growth window, the diffusion zone and the clear electrolyte, figuratively presented in Figure 1.1 [53].

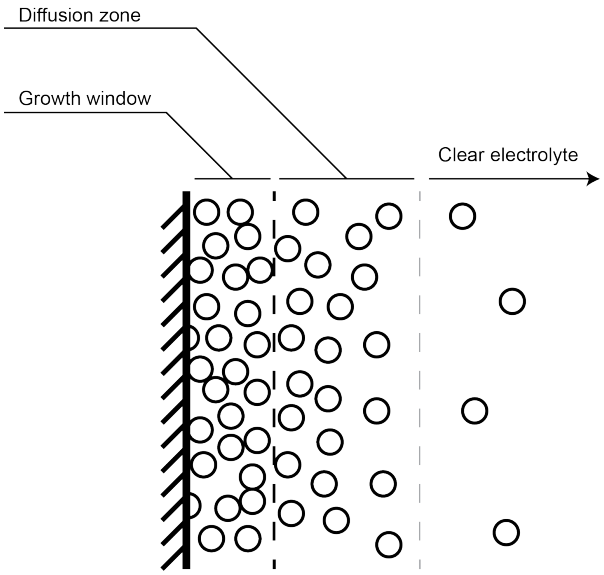


Figure 1.1: Description of flows near gas-evolving electrode

The formation of bubbles depends on the level of supersaturation of the gas dissolved in the electrolytic medium. Following Henry's law, thermodynamic inequilibrium at supersaturation zones allow for nucleation of bubbles at nucleation sites such as sharp edges on the surface of the electrode, or impurities suspended on the solution.

Bubbles attached to the surface of the electrode detach once they reach a critical size. Considerable research has been done to empirically correlate the critical diameter of a bubble to the current density at the electrode [16, 20, 33]. Bubbles continue to grow as long as they are close to the electrode, where the level of supersaturation allows for bubble growth [6]. Chandran et al. [11] presented their findings on the growth window of bubbles for an horizontally oriented electrode and found a bimodal distribution of bubble diameter on account of a balance between buoyant force and diffusive growth in the time spent by the bubble in the growth window.

The high concentration of bubbles in the growth window pushes some of the bubbles away from the electrode through lateral migration. The region adjacent to the growth window where the void fraction gradually reduces with distance from the electrode is termed as the *diffusion zone*. Due to the presence of bubbles, the flow does not follow a typical shear flow but is impeded by the presence of bubbles into following a flow profile similar to a Poiseuille flow close to the electrode as seen in Figure 1.2. When the gap between the electrodes is sufficiently small, the velocity profile resembles more closely to a Poiseuille profile [35]. This was also seen in the results of Dahlkild [14].

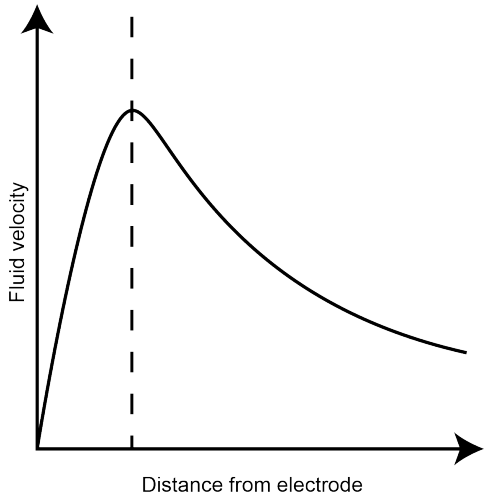


Figure 1.2: Expected velocity profile for the continuous phase

Very few bubbles migrate far into the bulk of the electrolyte. This region is typically driven by the background flow of the electrolyte. If there is no forced flow of the electrolyte, the time-averaged continuous phase velocity in the bulk of the electrolyte can be assumed a value of zero. If the electrolysis process takes place in a closed container, circulation zones can be seen in the bulk [24].

Generally, flows near gas-evolving electrodes are considered to be two-phase, with the liquid being the continuous phase and the gas being the dispersed phase. Depending on the type of modelling used for the dispersed phase, modelling of such flows can be of two types — Euler-Euler and Euler-Lagrange. Sufficient numerical studies have been done using both the methods. Many of the studies involving the Euler-Euler method are modelled using the mixture method which requires closure for the effective diffusivity of the mixture. Dahlkild [14] used empirical correlations in his model. Aldas et al. [2] assumed a laminar model in his numerical simulation and found that bubbles exhibit a localized turbulence in their wake that must be taken into account. Bideau et al. [4] modelled an additional dispersion force to the Euler-Euler model and found good agreement with the experimental results. Mandin et al. [35] modelled the flow near a vertical gas-evolving electrode using an Euler-Lagrange approach by considering bubbles as particules. They made two assumptions for treating bubbles as rigid spheres — the bubble diameter is very small ($d_p \approx 10^{-4}$ m) so that deformative stresses can be neglected and, the void fraction close to a gas-evolving electrode is sufficiently small so that coalescence of bubbles is rare. Following these assumptions, Mandin et al. [35] proceeded with a Langrangian simulation of bubbly flows with an horizontal force term for the dispersion of the bubbles along the vertical axis and obtained comparable qualitative results. Aldas et al. [2] used a laminar Euler-Euler model and reported an underestimation in the results with the hypothesis of localized bubble turbulence. This was also observed by Boissoneau et al. [6] in their experiments. Mat [37] included local turbulence in their two-fluid model and obtained agreeable results with experiments. Mandin et al. [35] concluded that a Computational Fluid Dynamic (CFD) model for the accurate representation of localized bubble turbulence needs to be developed.

Abdelouahed et al. [1] conducted experimental studies on vertically-oriented gas-evolving electrodes and found that the bubble diameter increases as a function of the height from the bottom of the electrode. This may be attributed to the fact that bubbles close to the electrode may grow in size due to the supersaturation levels of gas close to the electrode. They also reported a bubble size distribution with a mean size of 120 µm and a maximum velocity of 20mm/s for a current density of 200mA/cm^2. These values of bubble diameter and velocity have been used in further studies. It was also reported that the growth of the bubbles was not due to coalescence but predominantly due to diffusion [51].

From the perspective of gas-evolving electrolytic flows, the primary research objective can be thus stated:

Objective 1: Develop a numerical code that can be utilized for fully-resolved ionic mass transport simulations at the gas-evolving electrodes in the future, with an emphasis on localized bubble-induced flow.

1.1.3. Ionic mass transport in electrolytic flows

It is also prudent to know about the modelling of ionic mass transport and the current density in flows near gas-evolving electrodes. Classically, the current density **j** is solved as a vector field using the Ohm's law:

$$\mathbf{j} = \kappa \mathbf{E} = -\kappa \, \overline{\nabla} \phi \tag{1.2}$$

and the conservation of current equation:

$$\nabla \cdot \mathbf{j} = 0 \tag{1.3}$$

where **E** denotes the electric field vector space, ϕ the electric potential field and κ the electrical conductivity. In a homogeneous electrolyte solution, the value of κ can be assumed to be a constant. However, in the presence of bubbles, κ varies depending on the local void fraction. Computational simulations of gas-evolving electrodes that model the gaseous phase as a continuous phase (Euler-Euler methods) employ empirical relations that estimate the effective conductivity as a function of void fraction [7, 48]. The Bruggeman's relation:

$$\kappa = \kappa_0 \, (1 - \epsilon)^{3/2} \tag{1.4}$$

is used frequently for flows with a maximum void fraction of 0.12 having close agreement to experimental results [8]. The boundary conditions for the current density at the electrode surface is given by the decomposition of voltage:

$$\Delta U = E_0 + \eta_{ac} + \eta_{ohm} + \eta_{conc} \tag{1.5}$$

with E_0 denoting the equilibrium potential for the chemical reaction in the electrolyzer. The primary current distribution is given by the equilibrium potential and the ohmic potential loss η_{ohm} which is adequate for reproducing current distribution across the bulk of the electrolyte [35]. However, for a better modelling of the current density field close to the electrode, the activation overpotential η_{ac} is modelled using the Butler-Volmer relation. The activation overpotential η_{ac} contributes to the secondary current distribution. The concentration overpotential η_{conc} contributes to the ternary current distribution close to the electrode and is difficult to model without considering the transport phenomena of the ions close to the electrode. The coupling of the current density and the ionic concentration field is nonlinear.

The transport phenomena of an ionic specie i is modelled by the Nernst-Planck equation:

$$\begin{aligned} \mathbf{N}_i &= C_i\mathbf{v} - D_i\overline{\nabla}C_i - \frac{Z_iFD_i}{RT}C_i\nabla\phi \\ \frac{\partial C_i}{\partial t} &+ \overline{\nabla}\cdot\mathbf{N_i} = 0 \end{aligned} \tag{1.6}$$

where the terms in the right-hand-side of the first equation indicate advective, diffusive and migrative flux respectively. Here, the concentration C_i is assumed as a scalar field and solved numerically. In electrolyzers with very close gap between the electrodes, the migrative flux is estimated to be approximately twice as strong as the diffusive flux, thereby increasing the ionic current [22]. Suzuki et al. showed that the migrative flux is powerful up to a length scale termed as Debye length [49]. The Debye length is inversely proportional to the ionic concentration close to the electrodes and hence, a higher concentration potential would mean a smaller Debye length. At lengths larger than the Debye length, the gradient in electric potential drops to zero and the flow is purely convection- and diffusion-driven [58]. For this reason, most industrial water electrolyzers have a very small gap between the electrodes that increase the effect of migrative flux. Flows close to gas-evolving electrodes are also particularly unique with the fact that bubble entrainment and dispersion close to the electrodes provide an additional convective flux close to the electrodes that increases the mass flux of the ions. Forced convection of the electrolytic medium further increases the convective flux and increases the efficiency of the electrolytic process [40].

In the presence of a background flow, the convective flux may dominate the diffusive flux and the mass transport varies drastically. Wang et al. conducted an experimental and a numerical study on mass transfer in a sheared flow of suspensions of monodisperse neutrally-buoyant spheres [57]. At moderately high shear rates given by the Péclet number Pe, mass transfer was observed to improve with increasing particle volume fraction. This is attributed to the enhanced mixing effects provided by the particulate phase. They used a limiting current technique to measure the mass transport between two cylindrical electrodes with a weak electrolyte. This measurement technique is a standard technique to quantify convective-diffusive mass transport. At limiting current, the effect of migrative flux is negligible and the mass transport is dominated purely by convection and diffusion [57].

1.2. Stochastic modelling of mass transport

For the construction of a simple model, the migrative flux term of the ionic mass transport is currently ignored. The resulting dynamics would be convective-diffusive and would resemble ionic mass transport in a limiting current electrolyte bulk [57]. Diffusive mass transport can be modelled using two approaches — a continuous Eulerian approach and a discrete stochastic Lagrangian approach. The former is enabled by the Fick's law of diffusion:

$$\frac{\mathrm{d}C}{\mathrm{d}t} = D\nabla^2 C \tag{1.7}$$

where D is the diffusivity of the medium. In the presence of a flow field $\mathbf{u}$, the Fick's law can be generalised to a convection-diffusion equation for concentration c as:

$$\frac{\mathrm{d}C}{\mathrm{d}t} = D\nabla^2 C + \mathbf{u}\cdot\overline{\nabla}C \tag{1.8}$$

The above equation is the same as the Nernst-Planck equation (Equation 1.6) without the migrative flux term. Stochastic modelling is advantageous over the assumption of a continuous field for reasons such as discrete modelling of fluxes and a better representation of the randomness of the microscale. Stochastic models of concentration simulates each molecule individually using Langevin

dynamics. In many cases where the medium is a fluid, the molecules are assumed to be independent of each other. The interactions between molecules that create Brownian motion are modelled probabilistically as a Boltzmann distribution as a function of fluid temperature, despite the fact that they are deterministic in nature. Stochastic modelling of mass transport can be used to statistically visualise mass transport in large as well as small scales [43]. The Langevin equation:

$$\begin{aligned} m\frac{\partial \mathbf{v}(\mathbf{x},t)}{\partial t} &= \mathbf{F}(\mathbf{x},t) + \bar{\xi}_{\mathbf{v}}(t) \\ \mathbf{v} &= \frac{\partial \mathbf{x}(\mathbf{t})}{\partial t} \end{aligned} \tag{1.9}$$

where m is the mass of the molecule, $\mathbf{v}(\mathbf{x},t)$ the velocity of the molecule at an instant in time t and $\mathbf{x}(t)$ the position of the molecule in space at time t, described by Paul Langevin in 1908 [31], provides a mathematical framework for the stochastic modelling of dynamics of molecules in a fluid, particularly Brownian motion — the random motion of a molecule in a fluid through collisions with other molecules. The terms on the right side of the first equation in Equation 1.9 indicate the forces on the molecule, $F(\mathbf{x},t)$, the force on the molecule by virtue of its position $\mathbf{x}$ and time t, and a random force $\bar{\xi}(t)$ (with the subscript denoting the equation). The force on the molecule $\mathbf{F}(\mathbf{x}.t)$ is determined from the flow field and it is deterministic from the flow parameters. The random force $\bar{\xi}(t)$ is stochastic and non-deterministic.

In the case of fluid molecules, the Langevin equation can be simplified using Stokes drag as the primary force on the molecules to:

$$\frac{\mathrm{d}\mathbf{x}(t)}{\mathrm{d}t} = \mathbf{v} + \bar{\xi}_{\mathbf{x}}(t) \tag{1.10}$$

Equation 1.10 can be used to describe transient convection-diffusion in a large time-scale [43]. Equation 1.10 can be discretised to form an explicit Euler model for the dynamics of a tracer molecule as:

$$\Delta\mathbf{x} = \mathbf{v}\Delta t + \bar{\xi}_{\mathbf{x}}(\Delta t) \tag{1.11}$$

with Δt representing the time-step. The magnitude of the random motion term to describe Brownian motion of the molecules is related to the diffusivity of the fluid through the Stokes-Einstein result:

$$\begin{aligned} \left|\xi^{\mathbf{x}}(\Delta t)\right| &= \sqrt{6D\Delta t} \\ D &= \frac{kT}{6\pi\mu a} \end{aligned} \tag{1.12}$$

where k is the Boltzmann constant, T the temperature of the fluid, μ the viscosity of the fluid and a the radii of the molecules. Equation 1.12 also satisfies the result for the mean squared displacement for Brownian motion in 3 dimensions [43].

Thus, the second research objective can be stated as:

Objective 2: Compare a stochastic Lagrangian simulation of mass transport with an Eulerian approach and list the benefits and drawbacks of a stochastic mass transport model.

In the presence of a second medium, the diffusive mass transport exhibits a similarity to electrical conductance [38]. The effective diffusivity of a two-phase fluid mixture can be expressed by an analogy to the dielectric constant of a capacitor with a dielectric medium. Maxwell developed the mathematical relation for the effective dielectric constant in the case where the dielectric medium is made up of a dispersion of spheres with a different dielectric constant than the fluid [38]. Deen provided a heat transfer analog of Maxwell's relation and hence, the effective conductivity k_{mix} of a mixture of dispersed spherical particles is given as Equation 1.13 [15].

$$\frac{k_{\mathrm{mix}}}{k} = 1 - 3\phi\frac{1-\gamma}{2+\gamma} \tag{1.13}$$

where ϕ indicates the volume fraction of the spherical particles in the fluid, γ the ratio of thermal conductivities of the sphere compared to the fluid. The mass transfer analog used by Wang et al. [57] is:

$$D_{\text{eff}} = 1 - 3\phi\frac{1-\alpha}{2+\alpha} \tag{1.14}$$

where $\alpha = D_p M/D$, D_p being the diffusivity of tracers in the spheres, D the diffusivity of tracers in the fluid and M the solubility ratio of tracers in the sphere in regards to the fluid. A forced background flow such as a shear flow may increase the effective diffusivity with increasing shear rates. In a forced flow setting, the convective mass flux dominates the diffusive mass flux and contributes to the increased mass transfer rates. However, the mass transfer rate in the presence of a shear also increases with increasing particle volume fraction as observed by Wang et al. [57]. This is hypothesized to be the effect of localized particle-induced turbulence that causes an increased mass transport.

For the validation of the constructed model, the third research objective can thus be stated as:

Objective 3: Reproduce the mass transfer results of Wang et al. [57] for both a stagnant and a sheared suspension of particles using the constructed stochastic mass transport model.

1.3. The PHYSALIS algorithm

Prosperetti et al. developed a relatively new method for simulating particulate flows termed as PHYSALIS [42]. The PHYSALIS algorithm considers a coupling of a finite-difference Navier-Stokes solver and a spectral solver of spherical harmonics close to each particle. This algorithm is restricted to the simulation of spherical particles owing to its usage of spherical harmonics. Furthermore, it has strict restrictions that the particles do not coalesce or grow in size. Hence, PHYSALIS is well suited for simulations of sedimentation. PHYSALIS has been developed over the years to include simultaneous simulations of a scalar field [47].

The actual PHYSALIS algorithm and its advantages will be only briefly discussed since the algorithm itself is not the focus of the present work. However, the assumptions involved that make PHYSALIS well suited for simulations of bubbly flows are discussed in detail.

1.3.1. Steps of the algorithm

The crux of PHYSALIS lies in the creation of boundary conditions for the flow variables of velocity, pressure, viscosity and vorticity at the vicinity of the particles to be coupled with the flow solver. As stated before, PHYSALIS relies on spherical harmonic coefficients of any order, termed as *Lamb coefficients*, to deduce a solution close to the analytical solution. This is done without the need for mesh motion or high mesh resolutions at the interface. The steps involved in the algorithm are as follows:

1. A cage is created on the fluid mesh around each particle for all the flow variables. This is figuratively seen in Figure 1.3.
2. The hydrodynamic force on the particle is computed from the Stokes solution using spherical harmonics of the analytical Stokes solution at the particle surface.
3. The position, velocity and the rotation of the particle is updated from the computed forces.
4. The boundary conditions for the fluid solver at the cage are determined from the analytical Stokes solution for the relative flow at the surface of the particle using the analytical Stokes solution.

5. The Navier-Stokes equations are then solved for the domain with the enforced boundary conditions at the particle surface. These are typically executed with pressure-Poission solvers.

6. The velocity and pressure field adjacent to a particle is checked for convergence. If the residual is higher than the set convergence criteria, the updated boundary conditions are used for the next iteration. These iterations are termed as *Lamb iterations*.

7. Once convergence is attained, the flow field and the particles are updated.

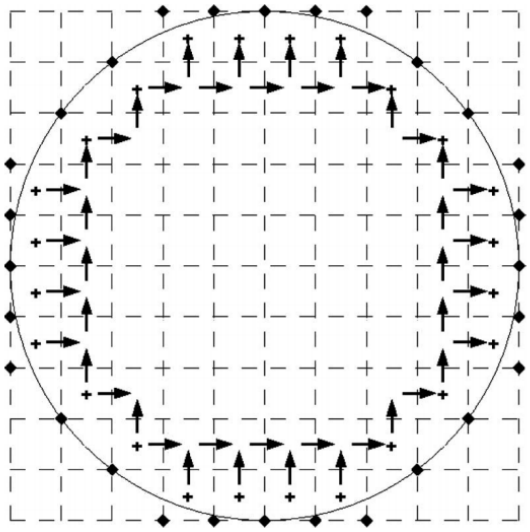


Figure 1.3: Cage around a sphere for various flow variables (⧫ - vorticity, + - pressure, → - horizontal velocity, ↑ - vertical velocity component [59])

1.3.2. Advantages over conventional methods

PHYSALIS poses several advantages over conventional particle-laden flow solvers. PHYSALIS negates the need for a high mesh resolution and mesh motion to obtain an accurate solution. The forces and the torque on the particles are computed with analytical precision without the need for higher order extrapolation techniques.

The need for the simulation of the second dispersed phase is also removed since in essence, only the computation of the spherical harmonic coefficients are done. This vastly reduces the number of equations to be solved by the fluid solver on comparison to other Euler-Lagrange models that rely on immersed boundaries and mesh motion. The number of simultaneous equations is also less than what is seen in Euler-Euler mixture or two-fluid models.

1.3.3. Motivation for use

There are certain limitations to using PHYSALIS, certain requirements that have to be met. The foremost limitation is that PHYSALIS is capable of simulating particle shapes for which Stokes solution exists. This restricts the particles to be either spheres (3D) or cylinders (2D). Following the values of bubble diameter and velocity given by Abdelouahed et al. [1], the Eötvös number of the bubbly flow can be estimated. An order of magnitude analysis shows that under such conditions, $\mathrm{Eo} \ll 1$. The particulate Reynolds number (Re_p) can also be estimated to be in the range $(1,100)$. According to Clift et al. [12], the corresponding values of Reynolds number and Eötvös number places the bubbles purely in the spherical regime with no surface deformation.

Another limitation that follows is the assumption that the particles are rigid spheres with a no-slip boundary at the surface. Bubbles typically posses a slip boundary condition at the surface due to the dispersed phase being a gas rather a rigid sphere. Clift et al. show that at low values of particulate Reynolds number (Re < 100), the Marangoni flow inside the bubble volume is very weak to enforce the slip boundary condition and the bubble behaves as a rigid sphere. At such cases, the dynamics of the bubble is assumed similar to that of a rigid sphere. Furthermore, the presence of impurities in the continuous phase, acting as surfactants, might transform the fluid boundary of the bubble into a rigid one. This might not always be the case at higher Reynolds numbers but this is an assumption that has been made in the present work.

The final limitation is that the particles do not coalesce and form a larger particle. According to Tanaka et al. [51], bubbles close to gas-evolving electrodes rarely coalesce due to the supersaturation of dissolved gases in the continuous phase. A lot of models [1, 26] have assumed this and obtained agreeable results. Hence, the coalescence of the bubbles have been neglected.

Hence, the fourth and fifth research objectives can be stated as:

Objective 4: Construct a stochastic "tracer" model and couple with the flow field and particles from *Bluebottle*.

Objective 5: Test and validate the *Bluebottle*-coupled tracer model with the flow field from *Bluebottle*.

The model thus constructed is referred to as the *tracer* model as the present work focusses on the convective and diffusive fluxes, thus implying that the molecules that are simulated stochastically are fluid-phase and would be indistinguishable from the flow field.

1.4. Objectives and Scope

The thesis aims to build a foundation for a stochastic mass transport model for the simulation of ionic mass transport near gas-evolving electrodes. The primary research objectives is compiled and presented below:

1. Develop a numerical code that can be utilized for fully-resolved ionic mass transport simulations at the gas-evolving electrodes in the future, with an emphasis on localized bubble-induced flow.
2. Compare a stochastic Lagrangian simulation of mass transport with an Eulerian approach and list the benefits and drawbacks of a stochastic mass transport model.
3. Reproduce the mass transfer results of Wang et al. [57] for both a stagnant and a sheared suspension of particles using the constructed stochastic mass transport model.
4. Construct a stochastic "tracer" model and couple with the flow field and particles from *Bluebottle*.
5. Test and validate the *Bluebottle*-coupled tracer model with the flow field from *Bluebottle*.

The immediate scope of the project lies in the development of a modular simulation tool for determining ionic mass transport statistics in water electrolysis. Furthermore, the developed code is intended to be extended to a stochastic model for the simulation of mass transport in any particulate multiphase setting.

2

Bluebottle

Bluebottle is a GPU implementation of the PHYSALIS algorithm. *Bluebottle* is written in CUDA and C to be run on computers with CUDA-compatible Nvidia GPUs. *Bluebottle* is open-source and the source tree can be found in Github (`https://github.com/groundcherry/bluebottle-3.0`). *Bluebottle* and the code for the simulation of tracers is executed in two GPU-enabled nodes of the Reynolds cluster at Delft University of Technology. Each node has 28 cores with 2 Intel®Xeon®E5-2680 v4 processors running at 2.4GHz with 64GB RAM and 125GB SSD storage. Each node also has connected a Nvidia Tesla K40m GPU coupled with 12GB DDR5 RAM. The Tesla K40m is CUDA-compatible. CUDA version 9.1 (v9.1.85) is installed in both the nodes. Ubuntu 16.04.7 LTS is installed on the cluster for operation via the Command-Line-Interface (CLI). The version of *Bluebottle* with the tracer implementation can be found in Github (`https://github.com/shyam97/bluebottle3`).

2.1. Installation of *Bluebottle*

Bluebottle relies on the dependencies CMake, CUDA, OpenMPI, HDF5 and CGNS. CMake is typically installed in many Linux distrubutions by default and enables the compiling of C scripts into executable object files with the command `make`. CUDA is a programming platform developed by Nvidia that is intended to make use of Graphical Processing Units (GPUs) for general purpose computing on GPUs (GPGPU). Processing on GPUs is faster than CPUs considering the massive parallelization capabilities of the GPUs [41]. OpenMPI is an open-source Message Passing Interface (MPI) that enables parallelization between different CPU and GPU processors. OpenMPI is generally used for CPU parallelization but the installation of OpenMPI can be mapped with a CUDA installation to be CUDA-aware. This enables parallelization and scaling of CUDA processes across multiple GPUs. HDF5 and CGNS are CFD-oriented data storage packages. HDF5 is an Hierarchical Data Format (HDF) is a data storage library that provides fast I/O processing and a compact, flexible data storage file format. As HDF5 is hierarchical, multiple parameters of the flow such as velocity, pressure and density can be stored as multi-dimensional arrays in a single file together. This allows easier file transfers and importing. CGNS (CFD General Notation System) is an industrial standard for the storage and importing of CFD data, approved by the American Institute of Aeronautics and Astronautics (AIAA). The file format `.cgns` is supported by many CFD tools such as Fluent, OpenFOAM and ParaView for importing and post-processing of CFD data.

With CMake and CUDA pre-installed in the GPU-enabled nodes of the cluster, OpenMPI, HDF5 and CGNS are installed locally in the user folder with CUDA support. The installation instructions for

the dependencies can be found at `https://github.com/groundcherry/bluebottle-3.0/blob/master/INSTALL`. The installation of *Bluebottle* is restricted by the constraints of the dependency versions as specified in the installation instructions file. For the sake of simplifying and automating the installation of *Bluebottle*, a shell script was written. The shell script can be found at `https://github.com/shyam97/bluebottle3/blob/master/install.sh`. The shell script requires that CMake and CUDA are pre-installed and prompts the user for the location of the CUDA folder. When all the dependencies are installed, the shell script automatically executes the `Makefile` in the root folder of *Bluebottle*. The `Makefile` contains compilation instructions for the source C and CUDA header and execution files of *Bluebottle* under the `src/` folder. Before the execution of `make`, the appropriate compute architecture for the GPU should be specified in the `Makefile` with the flag `arch`. The *Bluebottle* executable file is created upon succesful compilation in the `sim/` folder of the *Bluebottle* root folder.

Bluebottle has the directory tree structure as below.

```
base folder
├─ src
│  ├─ bluebottle.c
│  ├─ bluebottle.h
│  ├─ bluebottle.cu
│  └─ ...
└─ sim
   ├─ bluebottle (executable)
   ├─ input
   │  ├─ flow.config
   │  ├─ part.config
   │  ├─ decomp.config
   │  └─ record.config
   └─ output
```

The `input` folder of *Bluebottle* contains the configuration files for the case setup of a simulation. The flow parameters such as the values of physical parameters of density and viscosity, flow boundary conditions of pressure and velocity, initial parameter values in the domain, turbulence properties and the solver settings are specified in the `flow.config` file. The particle parameters such as the particle radius, initial position and velocity, density, material properties and type of motion (translation and rotation) can be specified in the `part.config` file. The domain extents and the mesh resolution and decomposition information is specified in the `decomp.config` file. The `record.config` is used to control the output of the flow parameters and particle information at specified intervals of flow time. Additionally, the configuration file required to restart the simulation from a point in time can be enabled in `record.config`. This is helpful when simulations are restricted by total computational time in the cluster.

2.2. Validation of *Bluebottle* installation

Bluebottle has already been validated using numerous rigorous particle simulation cases [5, 50, 59, 60]. In the current project, the installation of *Bluebottle* was validated with a simulation of sedimentation of a single sphere. The case setups are similar to the experiments and spectral CFD simulations given by ten Cate et al. [54]. The validation case was chosen for the following reasons — the results by ten Cate et al. [54] are highly accurate, the case setup is simple (single sphere sedimentation in a quiescent flow background), adding to the list of cases to be validated by *Bluebottle*.

A rectangular domain of length, breadth and height 100mm, 100mm and 160mm, respectively, is considered. This domain was discretised using a uniform Cartesian mesh of resolution 120x120x192 in the X-, Y- and Z- directions respectively. A single spherical particle of diameter 15mm is placed at a distance of 120mm from the bottom of the domain. The bottom wall of the domain is assumed a no-slip wall with a zero Dirichlet condition for velocity and a zero Neumann condition for pressure. The top boundary is modelled with zero Neumann conditions for velocity and pressure. Periodic boundary conditions are used for the lateral walls. The CFL criterion in *Bluebottle* is defined as

$$\Delta t = \frac{\mathrm{CFL}}{\sum_i \frac{u_{i,\max}}{\Delta x_i} + \frac{\nu}{\Delta x_i^2}} \tag{2.1}$$

where i indicates the three axes, $u_{i,\max}$ indicates the maximum velocity component of flow in the i axis and ν the kinematic viscosity of the fluid. In the laminar flow regime, the contribution of the viscous dissipation is smaller and hence, the formula for the time-step size reduces to the simple CFL criterion. The CFL value is set at 0.75 for the validation cases. The pressure convergence value is set at the default value of 1e−6 pressure units. The convergence criteria for the Lamb iterations is also set at the default value of 1e−2. No relaxation is specified for the flow variables. Gravitational acceleration of $9.81\mathrm{m/s^2}$ is specified in the -Z direction. The background flow is initialised in a quiescent state and and the initial particle velocity is set at zero. The particle is specified a density of $1120\mathrm{kg/m^3}$, according to the reference simulations by ten Cate et al [54]. The particle is allowed to translate but not rotate since the flow is axisymmetric and not expected to create torque. The fluid is specified with densities and viscosities matching the four reference cases for Reynolds numbers Re = (1.5, 4.1, 11.6, 31.9) as per the reference cases by ten Cate et al. [54]. On execution, the different cases ran at a time-step proportional to the Reynolds number assigned, according to the CFL condition. Simulations of higher Reynolds number completed in several hours while simulations of low Reynolds numbers took approximately two days to finish. The results of the particle position and velocity were stored in steps of 1e−2s. The velocity of the particle for each of the case against the progression of time is given in Figure 2.1. It can be seen that *Bluebottle* predicts the sedimentation velocity accurately for all the Reynolds numbers considered. The flow contour for Re = 31.9 at a time instant of $t = 0.6$s compared with the spectral results from ten Cate et al. is given in Figure 2.2. Hence, *Bluebottle* matches the experimental results of terminal velocity accurately despite having a coarser mesh and faster computation time. However, there is a certain discrepancy in the trajectory of the particle at the end of the simulation with *Bluebottle* predicting an earlier deceleration of the particle on reaching the bottom of the domain. A similar early deceleration is observed in all the cases and hence, the error is hypothesised to be a result of a coarser mesh for the domain and a coarser discretised cage for the particle.

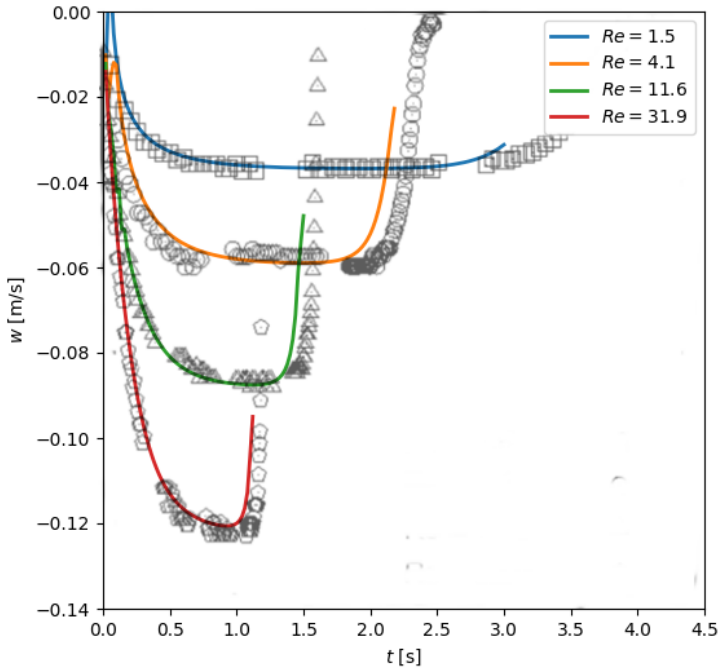


Figure 2.1: Velocity of sedimenting particle vs. time at different Reynolds numbers; solid lines - *Bluebottle* results, symbols - experimental values from ten Cate et al. [54]

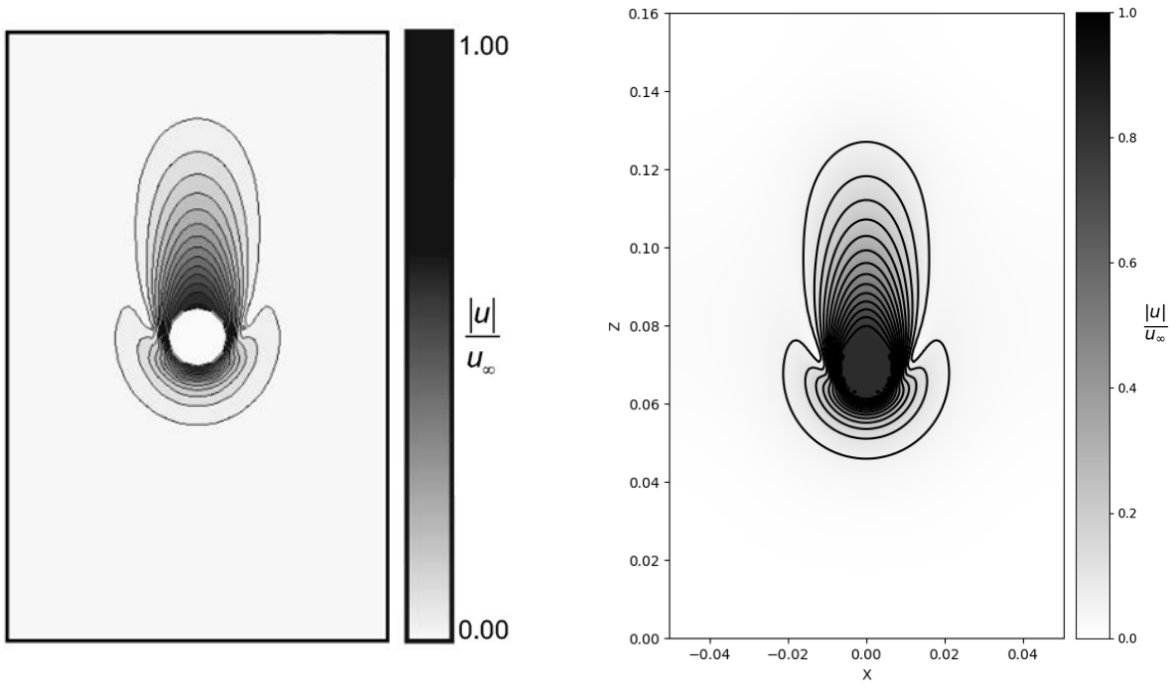


Figure 2.2: Flow contour for Re = 31.9 at $t = 0.6$s; left - *Bluebottle* result, right - spectral results by ten Cate et al.

3

Development of a fluid-phase tracer model

A fluid-phase tracer is an analog of a fluid molecule that undergoes advection with the flow and also molecular diffusion owing to its molecular scale. The current modelling of fluid-phase tracers is Lagrangian in nature and each tracer is modelled separately in its own frame of reference with no interaction from other tracers. Since fluid-phase tracers are also by definition a molecule of the continuous phase, they are not subjected to hydrodynamic forces acted upon by the continuous phase such as drag forces. Hence, a passive tracer "copies" the flow field at its location exactly with no relative velocity. However, at the scale of the tracer, Brownian motion also comes into play.

3.1. The *tracer* data type

Since the tracers are not individual particles but assumed to be molecules of the fluid-phase, only the positions of the tracers are stored. Any other properties pertaining to the fluid such as the velocity field, density and viscosity can be found by interpolation. Additionally, the array index of the tracer is stored for the purpose of debugging. Finally, for the implementation of a tracer distribution similar to Wang et al. [57], an additional variable named `tracertype` is used. This could be of use while simulating ionic flows where the cations and anions need to be differentiated. The definition of the tracer data type in C is presented in Appendix A.1.

3.2. The Langevin equation

The motion of the fluid-phase tracers is modelled using the following discretised Langevin equation for position:

$$\bar{\mathbf{r}}(t_n) = \bar{\mathbf{r}}(t_{n-1}) + \bar{\mathbf{v}}(\bar{\mathbf{r}}(t_{n-1}) \cdot \Delta t + \bar{\xi}\left[\sqrt{6D\Delta t}\right] \tag{3.1}$$

The Langevin equation, the explicit Euler scheme form presented in Equation 3.1, is generally an equation tracing the path taken by a Brownian particle. The first term on the right hand side of the equation is the original position $\bar{\mathbf{r}}$ of the particle. The second term denotes the advective displacement of the particle by the flow field $\bar{\mathbf{v}}$. The third term denotes the effect of Brownian force on the particle given by the fluid diffusivity D and the time-step (Δt) for the time integration.

The numerical implementation of the Langevin equation can be isolated into modules based on

each term of the equation:

$$\bar{\mathbf{r}}(t_n) = \underbrace{\bar{\mathbf{r}}(t_{n-1})}_{(1)} + \underbrace{\bar{\mathbf{v}}[\bar{\mathbf{r}}(t_{n-1}] \cdot \Delta t}_{(2)} + \underbrace{\bar{\xi}\left[\sqrt{6D\Delta t}\right]}_{(3)} \tag{3.2}$$

The first term (1) is straightforward and does not require computation since the location of the tracers are stored in a numerical array at all time-steps. The second term (2) can be calculated from the velocity field of the flow at the location of the tracer. Since the fluid domain is discretised, interpolation is required to compute the fluid velocity at the precise location of the tracer. The algorithm used for this purpose is discussed in detail in Section 3.2.1. The third term (3) is modelled considering the magnitude of Brownian motion and a random unit vector. The algorithm used for the calculation of the term is discussed in Section 3.2.2.

3.2.1. Interpolation in 3D space

The velocity field at the position of the tracer is calculated using trilinear interpolation. Trilinear interpolation is a first-order approximation method but is conveniently fast and easy to implement. Consider the tracer at location (x, y, z), in the bounding box given by the extents $[x_0, x_1]$, $[y_0, y_1]$ and $[z_0, z_1]$ bounded by eight mesh nodes, C_{zyx} in the X-, Y- and Z-directions in reverse. Equation 3.3 is used to determine the weights for the interpolation across the different axes. The value of the field at the location of the tracer can be deduced by progressive interpolation in X-, Y- and Z- axes as in Equations 3.4. The C code for the mathematical formulation, the `interpolator` function, is given in Appendix A.13.

$$w_x = \frac{x - x_0}{x_1 - x_0}; \quad w_y = \frac{y - y_0}{y_1 - y_0}; \quad w_z = \frac{z - z_0}{z_1 - z_0} \tag{3.3}$$

$$\begin{aligned} C_{00} &= C_{000}(1 - w_x) + C_{100}x_d \\ C_{01} &= C_{001}(1 - w_x) + C_{101}x_d \\ C_{10} &= C_{010}(1 - w_x) + C_{110}x_d \\ C_{11} &= C_{011}(1 - w_x) + C_{111}x_d \end{aligned} \tag{3.4}$$

$$\begin{aligned} C_0 &= C_{00}(1 - w_y) + C_{10}y_d \\ C_1 &= C_{01}(1 - w_y) + C_{11}y_d \end{aligned}$$

$$C = C_0(1 - w_z) + C_1 z_d$$

The trilinear interpolation relies on a bounding box of six nodal points. *Bluebottle* represents the three-dimensional grid in a one-dimensional form. Hence, the flattened array index of the corner of the bounding box containing the tracer is first determined using the `index_finder` function. The index determined, represented in code as `foundindex` is then used to find the Cartesian coordinate location of the bounding box nodes using the `grid_finder` function. The `grid_finder` function relies on the fact that a cell-centred grid is offset from the domain mesh grid by exactly half a cell width in each direction. The `foundindex` variable is also used to fetch values from the velocity field. These functions are presented in Appendix A.9.

If the tracer is located close to a wall such that the bounding box cannot be made of nodal points from the domain but using ghost cells and the boundaries, trilinear interpolation is not used. Instead, a simple previous node interpolation is used on all three axes. The `domain_checker` function,

presented in Appendix A.10, is used to determine the proximity of the tracer to the three axes of the rectangular domain and representing it using a single value. A prime number is assigned to each of the three axes – 2 for the X-axis, 3 for the Y-axis and 5 for the Z-axis in this case. The product of the prime numbers associated with each axis can be used to represent the proximity to all the domain extents in a single variable. For example, a value for the flag variable of 6 would mean that the tracer is close to the X- and Y-axes. Depending on the proximity to the domain extents, different methods of interpolation are followed. The previous node interpolation for a coarse grid is a severe drawback to the current implementation of flow field interpolation. A follow-up to the current project would include a higher order of interpolation in the domain and close to the domain boundaries.

The PHYSALIS algorithm does not exclude the mesh inside the particle cage from the fluid mesh. Instead, specific boundary conditions are applied in the cage. These boundary conditions are determined from the Stokes solution of flow close to a spherical surface. Hence, the flow field interpolation close to particle surfaces is proceeded as like any other region of the domain and the presence of a particle does not alter the interpolation method used in the current work. However, since the mesh near the particle surface is coarse than a mesh used in conventional multiphase modelling techniques like immersed boundary methods, accurate interpolation of the flow field may be compromised. Another follow-up to the current work would be to change the interpolation method for locations close to particle surfaces by incorporating the Stokes solution of the flow field at the tracer location and overriding trilinear interpolation.

3.2.2. The Brownian motion term

The magnitude of the Brownian displacement term is given by $\sqrt{6D\Delta t}$ with a random direction in 3D space. This can be easily implemented using spherical coordinates (r,θ,ϕ) and converted to Cartesian coordinates. The 3D direction components of the spherical coordinates – θ and ϕ – are randomized based on a uniform distribution in the interval $[0,2\pi)$ and converted using the transformation:

$$\bar{\xi} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \sqrt{6D\Delta t} \cdot \begin{bmatrix} \cos(\theta)\sin(\phi) \\ \sin(\theta)\sin(\phi) \\ \cos(\phi) \end{bmatrix} \tag{3.5}$$

The `randgen` function generates a random number in the range [-1,1]. The value from the `randgen` function is then used by the `randomizer` function in calculating a random displacement for each tracer using the mathematical formulation in Equation 3.5. The C code of the Brownian motion term is presented in Appendix A.6.

3.3. Interaction with particles and domain

With the Langevin equation for the motion of the tracers modelled, it is necessary to implement further algorithms to enable the interaction of the tracers with the spherical surfaces of the particles and with the domain boundaries.

3.3.1. Interaction with particles

Since the tracers are assumed to be fluid-phase, the particle surface is assumed to be impenetrable with a Neumann boundary of tracer concentration with value zero at the surface of the particle. The non-penetration boundary condition is enforced by assuming the particle surface to be rigid, and all collisions of the tracers with the particle to be elastic. Hence, tracers colliding with particles are assumed to be reflected in a specular fashion from the surface. The direction of the reflected ray is

given by the vector formulation of specular reflection from optics [13]:

$$\hat{\mathbf{d}}_s = \hat{\mathbf{d}}_i - 2(\hat{\mathbf{d}}_n \cdot \hat{\mathbf{d}}_i)\hat{\mathbf{d}}_n \tag{3.6}$$

where $\hat{\mathbf{d}}_s$ denotes the direction of the reflected ray, $\hat{\mathbf{d}}_i$ the direction of the incident ray and $\hat{\mathbf{d}}_n$ the direction of the normal at the point of incidence. The sign conventions of the vectors $\hat{\mathbf{d}}_i$, $\hat{\mathbf{d}}_s$ and $\hat{\mathbf{d}}_n$ are given in 2D form in Figure 3.1. Point C denotes the center of the spherical particle $\bar{\mathbf{x}}_p(t_{n+1})$,

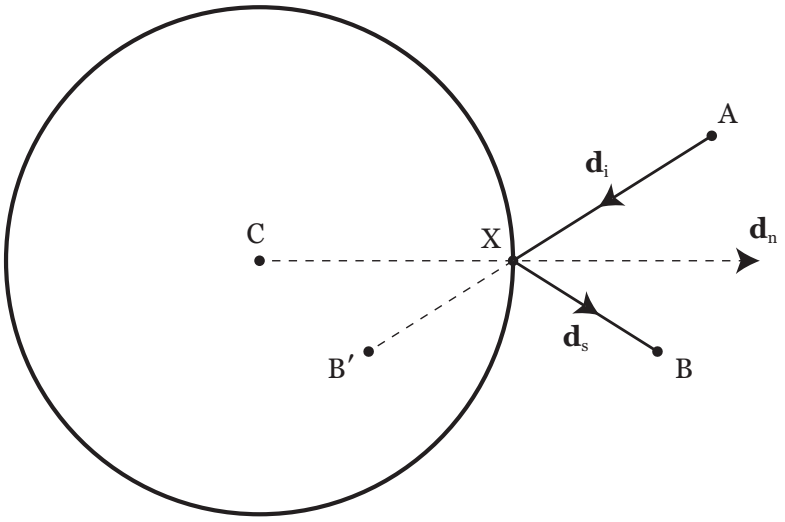


Figure 3.1: 2D illustration of vector formulation of specular reflection of tracers on particle surface

point A denotes the location of the tracer before displacement $\bar{\mathbf{r}}(t_n)$, point B denotes the location of the tracer after displacement $\bar{\mathbf{r}}(t_{n+1})$, point B′ indicates the location of the tracer after displacement if the particle was absent.

First, a check is made to determine if a tracer is inside a particle after displacement i.e., if the distance between a particle center $\bar{\mathbf{x}}_p(t_{n+1})$ and the tracer location $\bar{\mathbf{r}}(t_{n+1})$ is less than the particle radius a. If true, the specular reflection module is executed. A ray-marching algorithm described as follows is used to determine the point of incidence $\overline{\mathrm{X}}$ of the particle at the surface of the sphere.

1. Divide the tracer displacement $\overline{\mathrm{AB'}}$ into 10 equal parts.
2. Iterate over the parts and check if the point is inside the particle.
3. If the point is within the particle, go to the previous point. Divide the distance between the two points into 10 equal parts and repeat.
4. Continue until sufficient precision is attained.

Once the point of incidence (X in Figure 3.1) is known, the incident ray unit vector $\hat{\mathbf{d}}_i$ is known from the vector given by the incident point $\overline{\mathrm{X}}$ and the tracer location $\overline{\mathrm{A}}$ as:

$$\hat{\mathbf{d}}_i = \frac{\overline{\mathrm{AX}}}{|\overline{\mathrm{AX}}|} \tag{3.7}$$

The normal vector $\hat{\mathbf{d}}_n$ is given by the vector connecting the point of incidence $\overline{\mathrm{X}}$ and the center of the particle $\overline{\mathrm{C}}$ as:

$$\hat{\mathbf{d}}_n = \frac{\overline{\mathrm{CX}}}{|\overline{\mathrm{CX}}|} \tag{3.8}$$

The direction of the reflected ray $\hat{\mathbf{d}}_s$ is determined from Equation 3.6 and the magnitude of displacement along the reflected ray is determined from the relation $|\overline{\mathrm{XB}}| = |\overline{\mathrm{AB'}}| - |\overline{\mathrm{AX}}|$. Finally, the reflected location $\overline{\mathrm{B}}$ of the tracer is given by:

$$\overline{\mathrm{B}} = \overline{\mathrm{X}} + |\overline{\mathrm{XB}}| \cdot \hat{\mathbf{d}}_s \tag{3.9}$$

The code implementation, the `particlereflector` function, is presented in Appendix A.14.

When two particles are in close proximity, a single reflection step might place the tracer inside the adjacent particle. To solve this, the reflection subroutine is repeated until the tracer is not inside any particle. For the successive reflection steps, the point of incidence of the tracer with the particle $\overline{\mathrm{X}}$ is taken as the initial location of the tracer and the position after reflection from the previous step is taken as the final location. For this reason, the function `intersector` presented in Appendix A.14 is used to determine the point of incidence of the tracer at the particle surface.

As the simulation of tracers is a one-way coupling with the PHYSALIS solver, there is a possibility that a tracer will be placed inside a particle after a PHYSALIS solution and before the tracer iteration i.e, the tracer location $\bar{\mathbf{r}}(t_n)$ might be within the surface of a particle with center $\bar{\mathbf{x}}_p(t_n + 1)$. This is shown in Figure 3.2. The point P indicates the location of the tracer before the start of the tracer

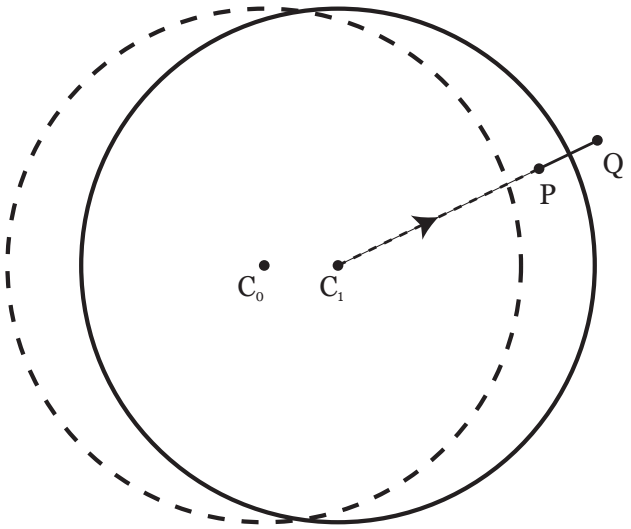


Figure 3.2: 2D illustration of pushing of tracer before the start of tracer simulation

simulation $\bar{\mathbf{r}}(t_n)$. Point C_0 indicates the location of the particle at the previous time-step $\bar{\mathbf{x}}_p(t_n)$ and C_1 indicates the location of the particle at the current time-step $\bar{\mathbf{x}}_p(t_n + 1)$. Hence, a subroutine, presented in Appendix A.15, is written to ensure that the tracers are pushed out of the spherical particle before the start of the tracer iteration.

A check is first made to determine if a tracer is inside a particle before the tracer iteration i.e., if the distance between a particle center $\bar{\mathbf{x}}_p(t_{n+1})$ and the tracer location $\bar{\mathbf{r}}(t_n)$ is less than the particle radius a. If true, the tracer pushing algorithm is executed. The vector $\overline{\mathrm{PC}_1}$ is computed and the unit vector in the direction $\hat{\mathrm{PC}}_1$ is also computed. The updated location of the tracer $\overline{Q}$ is determined as:

$$\overline{Q} = \overline{P} + 2 \cdot \left(a - |\overline{\mathrm{PC}_1}| \right) \cdot \hat{\mathrm{PC}}_1 \tag{3.10}$$

The algorithm is added to mitigate the errors due to one-way coupling where the tracer can passively get inside a particle through the motion of the particle. The magnitude of displacement through the subroutine is expected to be an order of magnitude smaller than the particle radius since the CFL criteria restricts displacement of the particle per time-step to a maximum of a grid cell length.

3.3.2. Interaction with periodic boundaries

In the case of periodic boundaries, a particle might have a part of its volume on one side of the domain with the rest of the volume on the opposite side. In this case, the algorithms for the tracer to interact with the particles may fail since the particle center might not be completely indicative of its volume. This is portrayed in Figure 3.3. In this case, even though the center of the particle (point C in Figure 3.3) is on the right side of the domain, the presence of a periodic boundary places a part of the volume of the spherical particle on the left side. This part volume has an apparent center C′. If the tracer displacement places the tracer at an updated position of B′, checking the distance of B′ from the actual particle center C will infer that the tracer is not inside the particle. Hence, the specular reflection module will not be executed. The error thus caused has been observed in domains where the ratio of axes length and particle radii are small. Hence, to mitigate this error and to ensure correct interaction with particles, a function named `periodic_maneuver` given in A.12 is developed.

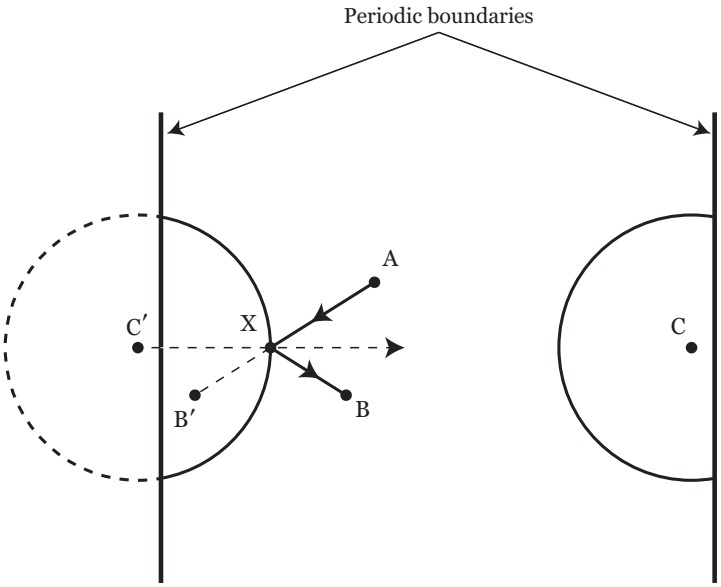


Figure 3.3: 2D illustration of interaction of particles across periodic boundaries

The function is appended at the start of all particle interaction functions including `particle_checker`, `particle_reflector`, `tracer_pusher` and `intersector`. The function takes a location and a distance as arguments and modifies an array of three flags corresponding to each axis. A conditional statement is used to detect if the location that is passed as the argument is within the distance that is passed as an argument from each domain extent. Depending on the proximity, the flags are given the values $\{-1, 0, 1\}$. The mathematical formulation for the condition, given location $\mathbf{x} = [x, y, z]$ and distance d in the X-direction is:

$$\text{flag}[0] = \begin{cases} -1 & \text{if } x < x_0 + d \\ 1 & \text{if } x > x_1 - d \\ 0 & \text{otherwise} \end{cases} \tag{3.11}$$

where x_0 and x_1 are the domain extents in the X-direction.

To give an example, if a particle of radius a is positioned close to the domain boundary in the +X direction and if the distance of the particle center to the domain boundary is less than the particle radius a (the case similar to Figure 3.3), the first flag value `flag[0]` is passed as 1. Similar checks are done for the Y- and Z-axes. These flags are also generated for every tracer. If the flag values for the

tracer and the particle are contradicting i.e, a product of -1, the particle center is relocated in that particular axis. The apparent center C′ is given by:

$$\overline{C'} = \overline{C} + \text{flag}_{\text{tracer}} \cdot [x_l, y_l, z_l] \tag{3.12}$$

where x_l, y_l and z_l denotes the domain lengths in X-, Y- and Z-directions respectively.

In a case similar to Figure 3.3, the point B′ would result in a X-direction flag of -1 as it is close to the -X domain boundary. Since the flag of tracer and the flag of the particle is contradicting, the particle will be moved to an apparent center C′ as in Figure 3.3. Since each tracer and each particle has a unique flag array combination, the repetitions of specular reflections are also accounted for.

3.3.3. Interaction with domain boundaries

The interaction of the tracers with domain external boundaries are modelled depending on the flow boundary the tracer comes into contact with. For the simulation of fluid flow in *Bluebottle*, each domain boundary pair can be treated as either solid walls or periodic boundaries.

In the case of solid walls, elastic collision of each tracer with the walls is assumed. Hence, the tracers are reflected in a specular fashion upon crossing a wall. As the domain boundaries are aligned to the axes, specular reflection is straightforward to implement. Consider specular reflection in the X-axis. Assuming $[x_0, x_1]$ as the extents of the domain in the X-direction and x as the X-coordinate of the tracer after displacement, a simple check is made to determine if $x \in [x_0, x_1]$. If this condition is not met, the conditional statement:

$$x' = \begin{cases} x_0 + (x_0 - x) & \text{if } x < x_0 \\ x_1 - (x - x_1) & \text{if } x > x_1 \end{cases} \tag{3.13}$$

specifies the new location x' after specular reflection at the wall. The same conditional statement is repeated for the Y- and Z-axes boundaries. In the specific case where a tracer might be adjacent to a corner of the cuboidal domain, displacement of the tracer might result in successive reflections with two or three adjacent walls. For this reason, the specular reflection conditions are repeated 3 times to ensure that all the reflections are accurately captured and all the tracers stay within the domain boundaries.

In the case of periodic boundaries, a tracer crossing a domain must exit out of the opposite boundary. Proceeding similar to the case of solid walls, the mathematical formulation for the enforcement of periodic boundaries at the walls of the domain in the X-direction is given by:

$$x' = \begin{cases} x_1 - (x_0 - x) & \text{if } x < x_0 \\ x_0 + (x - x_1) & \text{if } x > x_1 \end{cases} \tag{3.14}$$

with similar statements for the Y- and Z-directions. Similar to the solid walls, the condition is repeated thrice to ensure repeated entries are captured. The corresponding code, the `wallreflector` function, is presented in Appendix A.11.

In the case of boundary conditions similar to Wang et al. [57] where two "types" of tracers are modelled, a reflection off of one wall changes the tracers to the first type and reflections off of the opposite wall changes the tracers to the second type, irrespective of the type of tracers before reflection. A variable `tracertype` is used to denote the type of tracers - either "0" or "1". In the simulations, the mass transport is monitored in the Y-direction. Hence, the boundary condition by Wang et al. [57] is modelled as:

$$\text{tracertype} = \begin{cases} 0 & \text{if } y < y_0 \\ 1 & \text{if } y > y_1 \end{cases} \tag{3.15}$$

where y denotes the Y-coordinate of the tracer position and $[y_0, y_1]$ denotes the domain extents in the Y-direction.

A proof of concept is established for deleting tracers upon crossing a wall instead of reflecting back into the domain. Upon deletion, two subroutines are developed – one for idling the tracer at the same position outside the domain boundary and the other for deleting the tracer from memory altogether. The former is useful for tracking individual tracers and visualizing their path while the latter is useful for statistical cases where the final location of the tracer is relevant and the path of an individual tracer is of little use.

3.4. Initialisation conditions

The initialisation of the tracers is implemented so that tracers can be initialised depending on the flow and the particle setups. The function `tracerinit` used for this purpose is presented in Appendix A.3. In the present work, three types of initialisations have been used.

- For the validation of the tracer motion with a 2D flow around a cylinder, tracers were initialised with a uniform planar distribution at one of the faces of the domain boundary.
- For the simulation of a motion of a sphere through a wall of tracers, collision of two spheres, and comparison with the scalar solver of *Bluebottle*, tracers were initialised in a planar fashion perpendicular to the direction of motion of the spherical particle. The tracers were randomly distributed in a square-shaped subdomain of height and width a quarter of the domain size at the center of the domain.
- For the validation of mass transfer in stagnant and sheared suspensions, tracers were initialised randomly at the two opposite XZ-planes, since the direction of measurement of mass transfer is the Y-direction. Tracers were initialised at both the domain boundaries as two types of tracers were considered.

The only criterion for the initialisation of the tracers is that the tracers are part of the fluid domain and not the particle domain. Hence, a random distribution across the domain with particles is physically invalid. However, the tracer-pushing subroutine discussed in Section 3.3.1 will push the tracers out of a particle before commencing the tracer iterations.

3.5. Additional boundary conditions

The behavior of a tracer on collision with a wall is discussed in Section 3.3.3 to be of two types – a symmetry boundary or a periodic boundary. However, the two boundary conditions do not influence the absolute number of tracers at a boundary. In regards to tracer concentration with respect to electrokinetics, boundary conditions representing a constant number of tracers and a fixed rate of production of tracers have also been implemented. The functions driving these boundary conditions are presented in Appendix A.5.

The constant number of tracers boundary condition has been implemented by tracking the number of tracers within a user-defined distance threshold from a boundary. If this number of tracers decreases below the threshold, a new tracer is added to the tracer array. This is implemented using the function `maintainer`.

Implementing a fixed rate of production of tracers requires monitoring of the simulation time and time-step. New tracers can be added in proportion to the simulation time-step so that the rate of production of tracers can be kept constant. This is implemented using the function `ratekeeper`.

3.6. Flow of control

The flow of command for the *tracer* module is given in Figure 3.4.

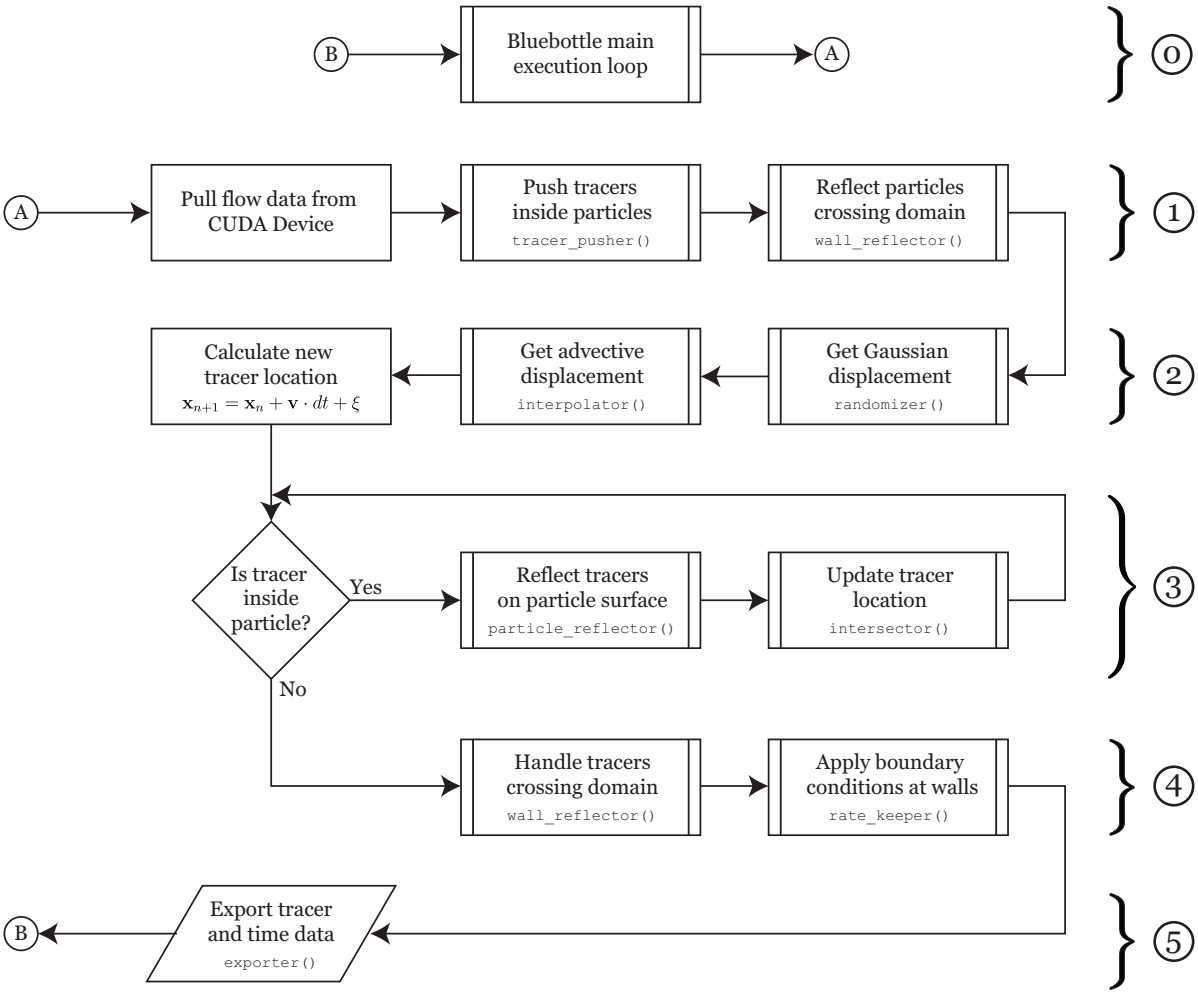


Figure 3.4: Schematic of tracer code and flow of control; circled numbers represent a submodule that can be combined in a representative fashion.

In the Figure 3.4, (0) represents operations outside the *tracer* module i.e, the *Bluebottle* main execution loop, (1) represents importing and clean-up of flow and particle data from the *Bluebottle* solution, (2) represents determination of displacement through advection and random motion, (3) represents interaction of tracers with particles, (4) represents interaction of tracers with domain boundaries, (5) represents export of tracer locations and transfer of command to *Bluebottle*.

An orderly fashion of execution results in high modularity of code where any submodule can be updated without affecting the execution of the other submodules.

3.7. Integration with *Bluebottle*

The C file `tracer.c` with the corresponding header file `tracer.h` is saved in the source folder `src` in the *Bluebottle* root directory. The tracer header file is linked in the main header `bluebottle.h`. The files are added to the `Makefile` to be compiled with the other source files with make. The tracer simulation initialisation function `tracer_start`, presented in Appendix A.2, is added to the PHYSALIS time integration loop at the end of the first time-step with a flag in place to prevent re-

initialisation in the successive time-steps. The main tracer execution function `tracer_execute`, presented in Appendix A.16, is placed after the flow field and the particle positions are updated by *Bluebottle*. The time-step `dt` is passed as an argument to the function.

3.7.1. One-way coupling

The tracer simulation is connected to the particulate flow simulation through the velocity field and the particle positions. These are, by default, global variables accessible to all files linked to `bluebottle.h`. For the simulation of tracers where the background flow and the particle positions are time invariant, the flow field needs to be accessed only once during the tracer simulation initialisation. Whereas, in the case of time varying flow field and moving particles, the flow field and the particle positions are accessed every time the tracer simulation is executed.

The velocity field array is stored with the ghost cells in a staggered grid arrangement in the GPU memory. Hence, the velocity field is pulled from the GPU to the host and converted into a cell-centered arrangement without the ghost cells before the start of the tracer simulation.

3.7.2. Input & Output

In the present work, the input variables such as the number of tracers, initial positions of the tracers, molecular diffusivity and the boundary conditions are input in the `tracer.c` file itself. Modifications to the code can be made to add the input parameters to a separate *config* file similar to the input parameters of the main particulate simulation.

The tracer locations and the type of tracer, in case of a boundary condition similar to Wang et al. [57], are exported in CSV format in the `output` folder of the simulation directory. The particle positions are also exported for post-processing. Additionally, the time-step number, the simulation time, the time-step and the number of particles in the domain are also saved for keeping a log of the number of exports and the timestamp of the exports. During initialisation of the tracer simulation, the domain extents in all the three axes are also saved for post-processing purposes.

In the event where the simulation has to be stopped and restarted from a certain time-step, the initialisation function `tracer_start` is provided with a Boolean flag variable termed `readfile` which if true, would load the tracer positions from an output file.

4

Testing & Validation of tracer implementation in Bluebottle

The *tracer* module developed consists of sub-modules that can be simulated individually as a specific case. To elaborate, if the flow field is quiescent, the resulting simulation would be that of molecular diffusion. On the other hand, if the diffusivity is assumed to be zero, the modelling of tracers must accurately represent streaklines of the flow. In the case of a steady flow field, the streaklines can be equated to streamlines of the flow.

In an Eulerian simulation, the phase field is used to distinguish the continuous phase from the discrete phase. The phase field is simultaneously solved with the flow equation with its own set of boundary conditions. However, in a Lagrangian setting, this boundary condition cannot be applied directly and hence, is numerically enforced using a set of algorithms to ensure the repulsion of tracers from the surface of the particle. Hence, the non-penetration boundary condition at the particle surface is to be tested.

4.1. Pure diffusion in 3D space

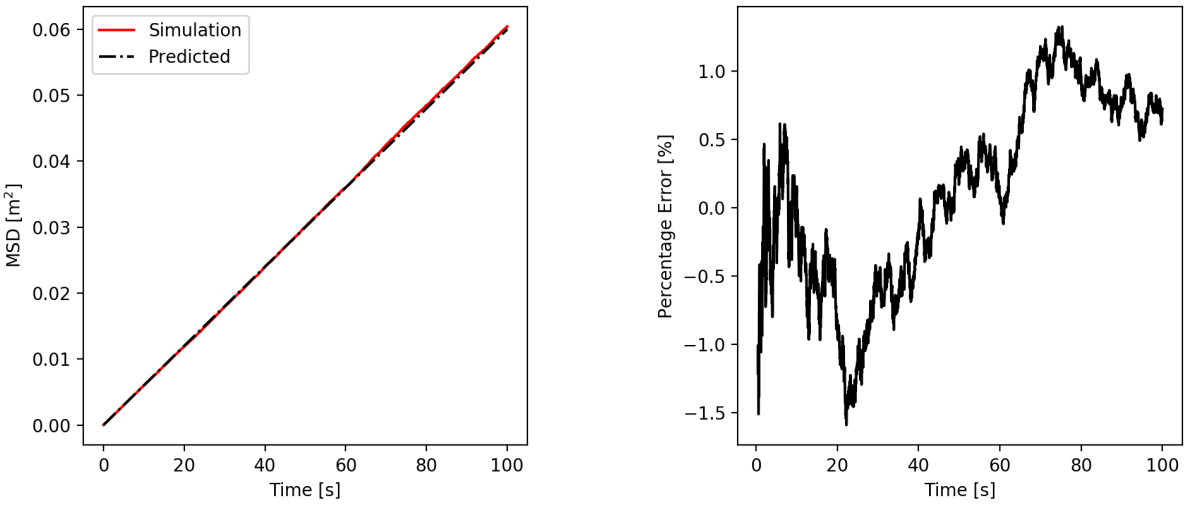
In order to test the diffusion term, the mean-squared-displacement (MSD) was determined for pure diffusion in 3D space with the background flow being quiescent and without the presence of particles. The MSD is given by:

$$\mathrm{MSD}(t) = \langle |\mathbf{r}(t) - \mathbf{r}_0|^2 \rangle \tag{4.1}$$

where $\mathbf{r}$ is the location of a tracer at time t and $\mathbf{r}_0$ is its initial location. In a three-dimensional Cartesian space, the mean-squared-displacement is statistically:

$$\mathrm{MSD}(t) = 6Dt \tag{4.2}$$

where D is the diffusivity of the particles. For the test simulation, 10000 tracers with a diffusivity of D = 1e−4 m^2/s are simulated. A domain size of $2 \times 2 \times 2$ m is used with a resolution of $128 \times 128 \times 128$ cells in the three directions. Since this test does not require the flow field nor interaction with particles, the `interpolator` module and the `particlereflector` functions are turned off. The time-step for the simulations is fixed at an arbitrary value of dt= 1e−3 sec and the simulation time at 100 seconds, well below the average time of 1666.6 seconds required for a tracer to cross the extents of the domain.



(a) Mean squared displacement – Simulation vs. Predicted

(b) Percentage error in MSD estimation

Figure 4.1: Comparison of mean squared displacement from simulation of tracers with predicted value

From Figure 4.1a, a close adherence to the predicted value of MSD can be seen. From Figure 4.1b, the percentage error of the value from the simulation is limited to <5%. Repeated simulations show varying error distribution versus time indicating that the error obtained can be attributed to statistical noise.

4.2. Streamlines in 2D flow about a cylinder

To test the interpolation term – the effect of flow on the tracers, the path of the tracers advected by a 2D flow about a cylinder is determined. Since a 2D domain cannot be directly simulated using *Bluebottle*, periodic boundary conditions are used in the Z-axis and the motion of the tracers is restricted to the XY-plane through the center of the particle. The particle was kept stationary and a flow of Re= 1 was simulated with a Dirichlet boundary for velocity and a Neumann boundary for pressure at the inlet and the outlet. The flow domain, with extents [-1,1] in all directions and a resolution of $128 \times 128 \times 128$ cells, was initialised with a channel flow field values. The tracers were initialised with zero diffusivity, after the background flow had reached a steady state, with equal spacing about the x-axis at $y = 0$ with a span of half of the domain. The simulation was carried out until all of the tracers reached the domain exit. Figure 4.2 shows a comparison between the streamlines of the flow as simulated by *Bluebottle* and the path traced by the tracers. The streamlines are plotted in such a way that they are displayed in between two tracer paths. The tracer module accurately follows the streamlines of the background flow. To test interpolation close to the surface of the particle, 200 tracers were simulated with the same domain resolution of *Bluebottle*. The tracers attach to the surface of the particle and do not adhere to the boundary layer on post-processing the path taken by the tracers. However, this error is caused by the coarser resolution of the *Bluebottle* mesh resulting in erroneous interpolation. Higher order methods of interpolation or the implementation of Stokes solution to determine the flow field close to the surface of the particle may overcome this error. However, in the current work, this is not implemented since a higher mesh resolution can easily mitigate the magnitude of such errors. The comparison between tracer pathlines and streamlines close to the surface of the particle is presented in Figure 4.3. The similarities between the streamlines and the tracer pathlines close to the surface of the particle are to be noted.

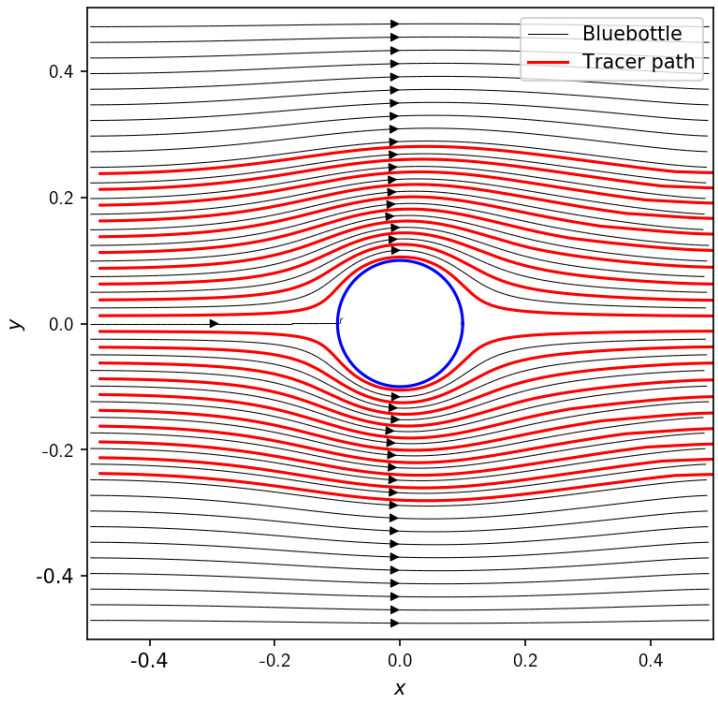


Figure 4.2: Streamlines from flow computed by *Bluebottle* vs. path traced by tracers for flow about a cylinder at Re = 1

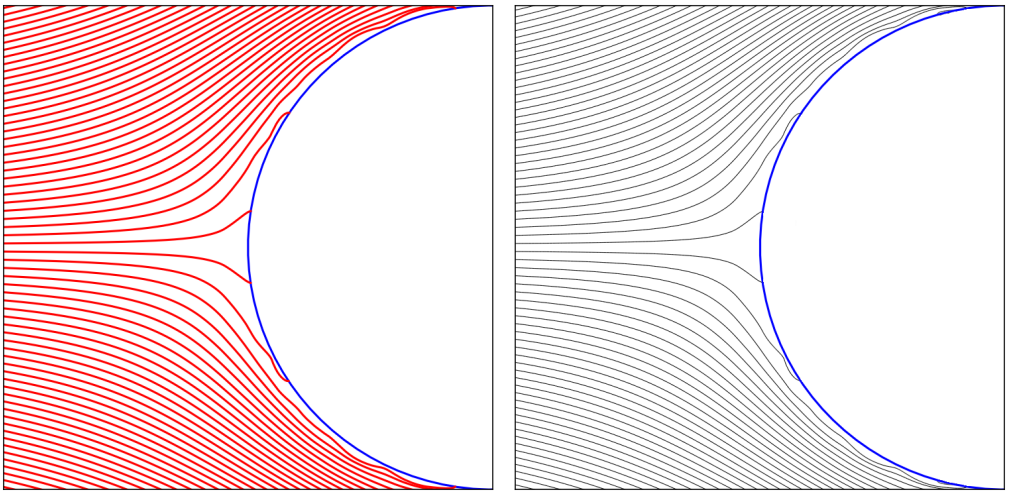


Figure 4.3: Tracer pathlines (left) vs. flow streamlines (right) close to the surface of the particle

4.3. Validation of non-penetration boundary at particle surfaces

The non-penetration boundary at the particle surface is numerically implemented using forced specular reflection and pushing of tracers. In order to test the implementation, the collision of particles with a wall of tracers with relative motion is simulated. In these cases, both the Brownian motion term and the interpolation term is used to determine tracer displacement. The tracers are initialised in a plane perpendicular to the direction of relative motion of the particle, with span of the initialisation region being half the domain length.

4.3.1. Motion of a sphere through a wall of tracers

In the first test case, a single moving particle of radius $a = 0.1$m is passed through a stationary wall of 2000 tracers with a Reynolds number of 10. The diffusivity of the tracers is set at $D = 1\text{e}{-3}\text{m}^2/\text{s}$. Figure 4.4 visualises the motion of the particle and the tracers. As the particle collides with the diffusing

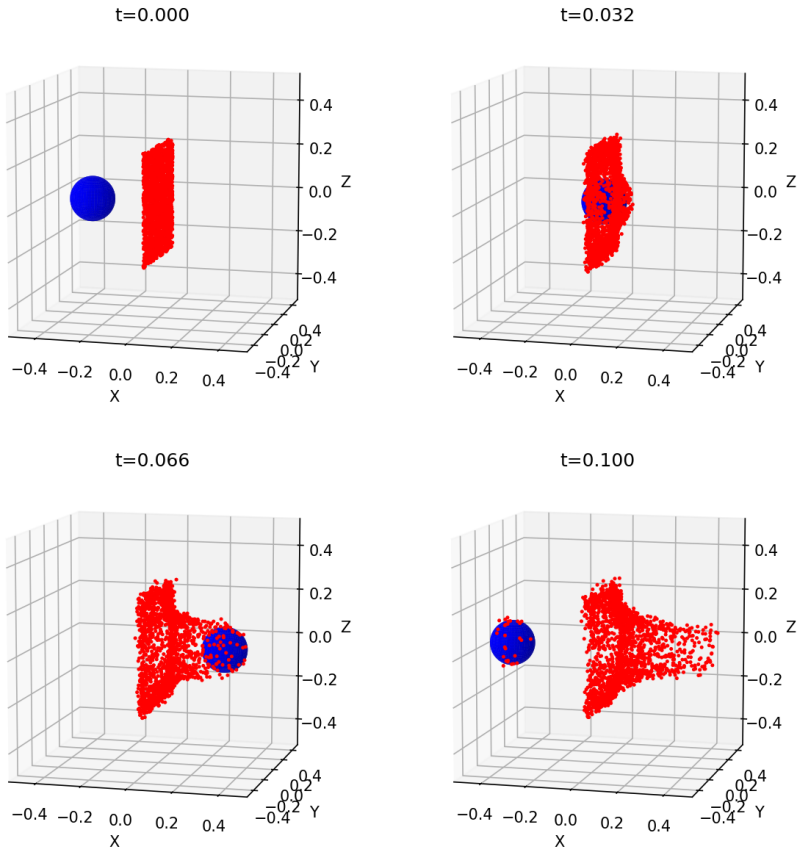


Figure 4.4: Visualisation of motion of a spherical particle through a wall of tracers at $\text{Re}_p = 10$ and tracer diffusivity $D = 1\text{e}{-3}\ \text{m}^2/\text{s}$

tracer wall, some tracers directly in the line of motion of the particle end up inside the particle after displacement. The specular reflection module is used to reflect the tracers out of the surface of the particle. This can be confirmed by monitoring the number of reflections a tracer undergoes every time-step. To ensure that the non-penetration boundary condition is not violated, the presence of a tracer inside the particle is also checked. In this case, no tracer intrusion is observed.

4.3.2. Head-on collision of two spheres

An extreme case for testing tracer intrusion is the simulation of collision of two particles with a wall of tracers located exactly at the plane of collision. This is significant since the specular reflection of a tracer on the surface of one particle could place the tracer inside the other particle. Although this scenario is rare in many simulations, it is an important test case to test the non-penetration boundary at particle surface. This test case served as a motivation for the implementation of successive reflection steps during the development of code as reported in Section 3.3.1.

Similar to the previous test case, the particles are of radii $a = 0.1$m and possess a Reynolds number of 10 with respect to the continuous phase. The particles are positioned symmetrically opposite

from the plane of collision and are given symmetrically opposing velocities at $t = 0$ so as to collide exactly at the center of the domain. The tracers are initialised similar to the previous case. The visualisation of the tracers and the particles is shown in Figure 4.5.

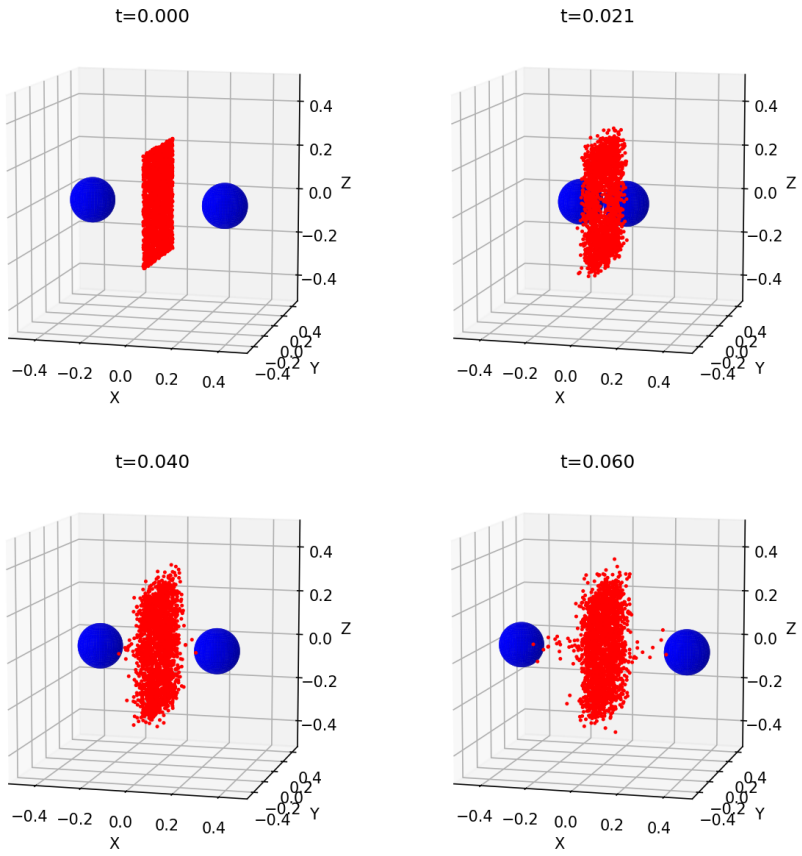


Figure 4.5: Visualisation of head-on collision of two spherical particles with Re_p=10 on a wall of tracers with tracer diffusivity D = 1e−2 m^2/s

The tracer implementation, along with *Bluebottle*, is written in C and the output is exported as CSV files and post-processed with Python v3.7.12 and Matplotlib v3.2.2. During the post-processing of initial runs, several tracers were found to intrude the particle with the intrusion being in the order of magnitude of 1e−6 m. On debugging and inspection, this order of magnitude was found to be related to the order of precision of the exported tracer location. When the tracer location was imported as a default `float` variable, 6 decimal places are exported, which directly corresponds to the encountered order of magnitude of intrusion. Upon increasing the export precision to 9 decimal places, the frequency of intrusion drastically reduces. Figure 4.6 shows the reducing frequency of particle surface intrusion upon exporting tracer and particle locations with precisions 6, 7 and 8 decimal places. With a precision of export of 9 decimal places, no intrusion was observed.

4.4. Effect on *Bluebottle* runtime

The addition of the tracer module to *Bluebottle* adds to the computational time of *Bluebottle*. The execution time of the tracer module is tested for varying number of particles and tracers using simple simulations of diffusion through a stagnant suspension. In these cases, the criteria that the suspension is stagnant is utilised and hence, the flow field is imported only once at the start of the

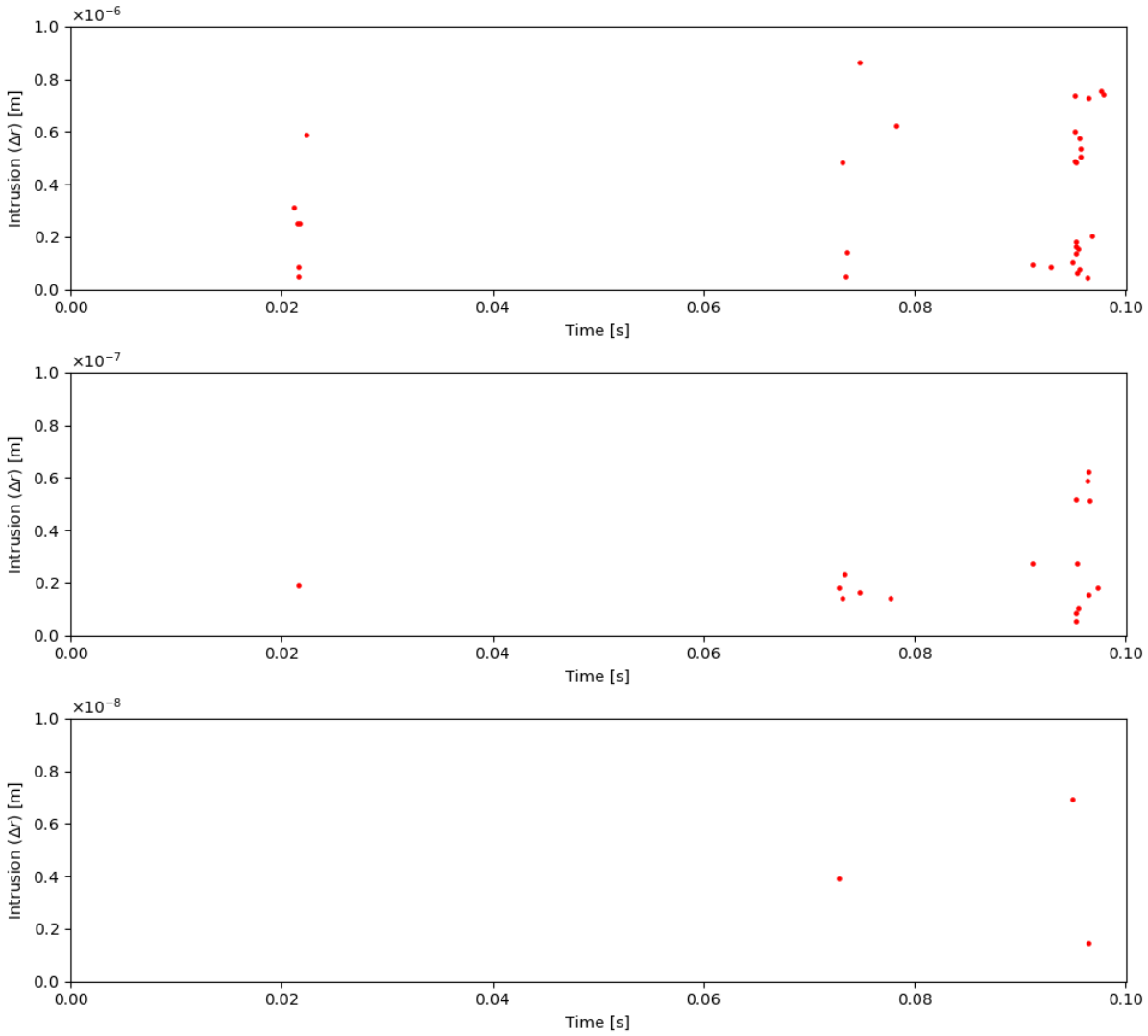


Figure 4.6: Intrusion of tracers inside particle surface with export precision of 6 decimal places (top), 7 decimal places (middle) and 8 decimal places (bottom); each red dot represents intrusion of tracer with the x-coordinate representing time of intrusion and y-coordinate representing distance of intrusion normal from particle surface

simulation. The Lamb iterations are stopped and only the tracers are simulated. For the comparison of the tracer execution time versus the number of particles, the number of tracers is kept fixed at 2000. Whereas, for the comparison of the tracer execution time versus the number of tracers, the number of particles is kept fixed at 15.

Figure 4.7 exhibits a linear relationship of the tracer module execution time with both the number of particles and number of tracers. However, the execution time has a stronger relationship with the number of particles with a factor of 3.793e−4 compared to a factor of 2.017e−5 for the number of tracers.

In the case where the flow field is imported every time-step, the relationship between the tracer execution time and the number of tracers varies. A sheared suspension of 15 particles is simulated for this case. Since the flow field is also varying, it is imported to the tracer module for flow interpolation every time iteration. Figure 4.8 showing the comparison between the tracer execution time and the number of tracers again exhibits a linear relationship of the tracer execution time against the number of tracers. However, a constant time of 0.1028s is present in the average execution time.

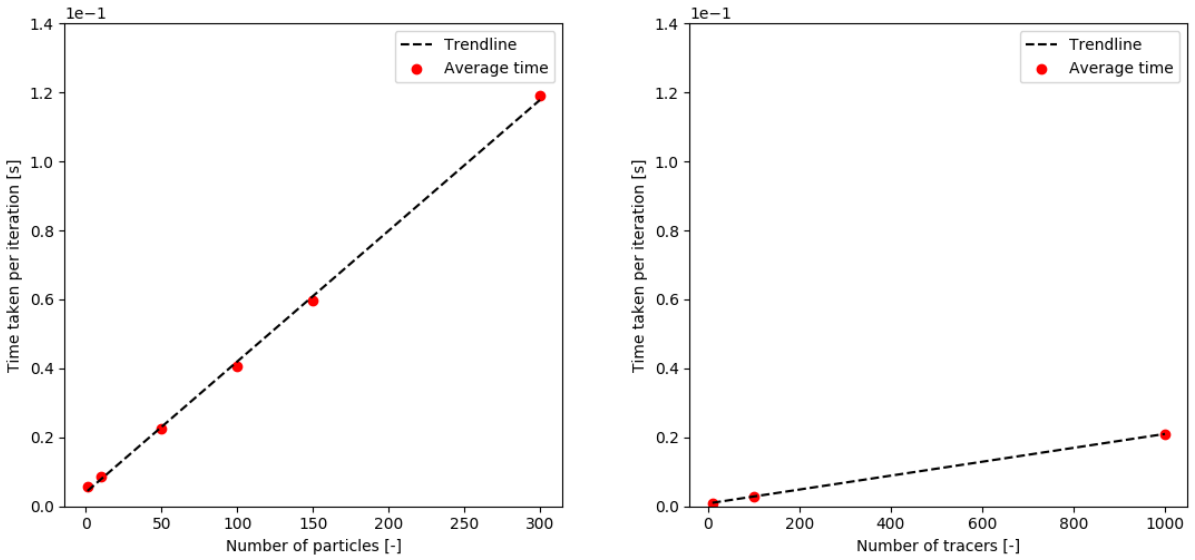


Figure 4.7: Variation in *tracer* module execution time vs. number of particles (left), number of tracers (right)

This corresponds to the time consumed in the importing of flow fields from CUDA device to host.

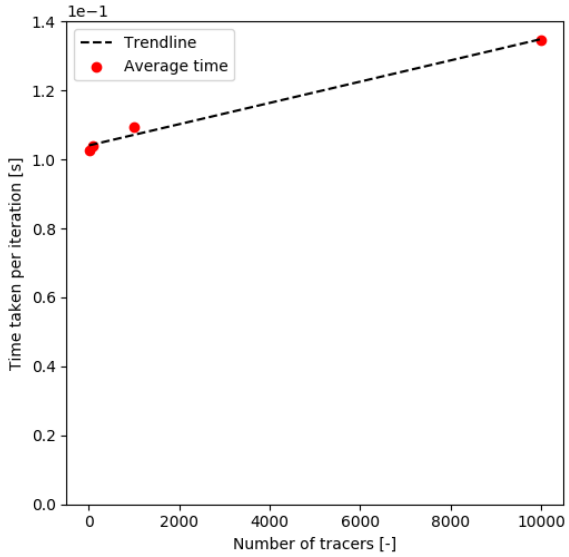


Figure 4.8: Variation in *tracer* module execution time vs. number of tracers

The tracer module execution time is then compared with the average time per Lamb iteration of *Bluebottle*. A simulation of sheared suspension with 15 particles and 2000 tracers is executed and the execution time is monitored. Figure 4.9 shows the comparison of execution time for a single Lamb iteration and the total tracer module. It can be seen that the time taken for the execution of the tracer module is approximately an order of magnitude lower than the execution time for a Lamb iteration. Also can be seen are the fluctuations in the time taken for the execution of the tracer module. This can be attributed to variations in the number of tracers that undergo specular reflection at the walls and particle surface.

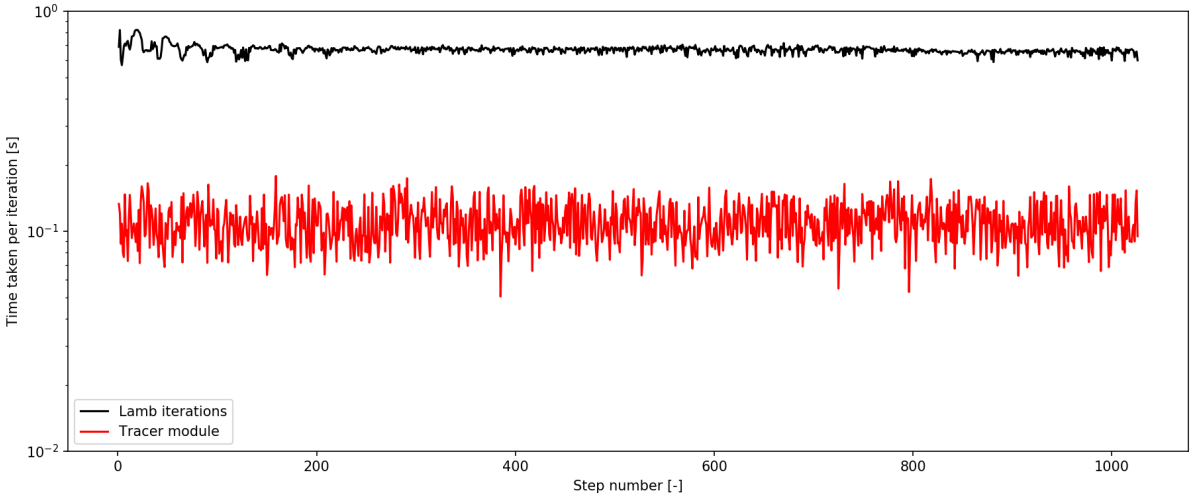


Figure 4.9: Variation in execution time for Lamb iteration and *tracer* module

5

Simulations of heat & mass transfer

One of the primary objective of the development of a *tracer* module is to test if a scalar field like the temperature field or concentration field can be accurately monitored and simulated in a stochastic manner. Hence, for this purpose, the *tracer* module is used to simulate passive fluid-phase tracers and convective-diffusive heat diffusion.

5.1. Estimation of Sherwood number for particulate suspensions

The simulation of particulate suspensions was executed using *Bluebottle* with a case setup similar to the simulations performed by Wang et al. [57]. A domain of size $0.8 \times 1 \times 0.8$ m was used with a mesh resolution of $100 \times 125 \times 100$ cells in the X-, Y- and Z-directions respectively. Mass transport was monitored in the Y-direction with the XZ-planes serving as solid walls. Hence, the length of the domain in the mass transport direction $H = 1$ m. The remaining boundaries were set as periodic boundaries for the flow. The spherical particles simulated are of uniform radii of $a = 0.1$ m. This results in an aspect ratio $H/a = 10$. The spherical particles are dispersed uniformly throughout the domain using the `seeder` utility of *Bluebottle*. In all the simulations, the default residual threshold values for the Lamb iterations of *Bluebottle* are used.

A two-tracer method similar to the implementation by Wang et al. [57] is used. The tracers are assumed to be of two "types" "0" and "1". The XZ-plane in the -Y direction (named the *south* plane) is assumed to be the plane of production of type "0" tracers and the XZ-plane in the +Y direction (named the *north* plane), type "1" tracers. Hence, any tracer of type "1" crossing the *south* plane is converted to a type "0" tracer and vice versa. This method of employing two "types" of tracers has the advantage that new tracers need not be created and tracers crossing a boundary can be reused. Mathematically, the condition can be represented as:

$$\begin{aligned} c_0(-y) &= 1; \quad c_0(+y) = 0 \\ c_1(-y) &= 0; \quad c_1(+y) = 1 \end{aligned} \tag{5.1}$$

where c_0 and c_1 represents the concentration of specie "0" and "1" respectively, normalized with the maximum concentration in the domain . $-y$ and $+y$ represents the domain boundaries in the y-direction.

In the absence of spherical particles, the time evolution of concentration follows Fick's law of diffusion:

$$\frac{\partial C}{\partial t} = D\frac{\partial^2 C}{\partial y^2} \tag{5.2}$$

which is similar to the Laplace equation for 1D heat diffusion at steady state. Hence, the time evolution of concentration is assumed to follow the same solution as the heat diffusion equation. At steady state, the profile for the concentration is expected to be the same as the temperature profile in 1D conduction:

$$\frac{\mathrm{d}C}{\mathrm{d}y} = \text{constant} \tag{5.3}$$

Substituting the Dirichlet boundary conditions, we obtain:

$$C(y) = C(y=-y) + \frac{C(y=+y) - C(y=-y)}{H} \cdot y \tag{5.4}$$

Simplifying further using normalized concentration c for the two species "0" and "1" in the place of concentration C, we get:

$$\begin{aligned} c_0(y) &= 1 - \frac{y}{H} \\ c_1(y) &= \frac{y}{H} \end{aligned} \tag{5.5}$$

Pure diffusion with no background flow and no spherical particles was simulated using the *tracer* module. The histogram of tracers was obtained for an interval of 50000 iterations after the completion of 200,000 iterations. The histogram for the tracer specie "0" and "1" is shown in the Figure 5.1.

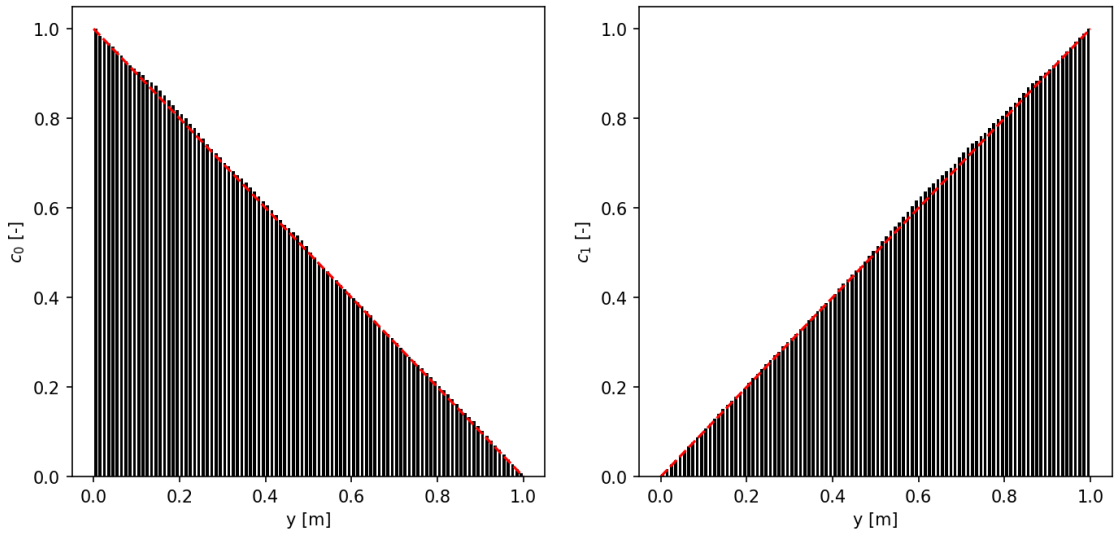


Figure 5.1: Histogram of normalized tracer concentration c_0 (left) and c_1 (right) for tracer specie "0" and "1" respectively; black bars indicate *tracer* module result as a histogram, red line indicates analytical solution given in Equation 5.5.

The rate of mass transport is given by the Sherwood number Sh and can be estimated using the following definition:

$$\mathrm{Sh} = \frac{hH}{D} \tag{5.6}$$

where $H = 1$ m is the domain size in the direction of monitoring mass transport, D is the diffusivity of the tracers which is modified depending on the Péclet number Pe of the flow given as:

$$\mathrm{Pe} = \frac{\Gamma a^2}{D} \tag{5.7}$$

where Γ is the shear rate of the walls. h is the mass transfer coefficient and is calculated as:

$$h = \frac{\langle Q \rangle}{A c_0} \tag{5.8}$$

For the estimation of $\langle Q \rangle$, the number of tracers that change from type "0" to type "1" and the opposite is monitored to represent the number of tracers crossing the boundary. A is the area of the plane perpendicular to the mass transport direction. For the given domain size, $A = 0.64$ m^2. The concentration driving force c_0 is then evaluated as n/V with $n = 2000$ being the number of tracers in the domain and $V = 0.64$ m^3 being the domain volume. The Python implementation for the calculation of Sh is presented in Appendix B.3.

5.1.1. Mass transport in stagnant suspensions

A stagnant suspension is expected to reduce the mass transfer of the tracers as the spherical particles will act as obstacles for the motion of the tracers. The theoretical model of the mass transfer rate given the volume fraction of the spherical particles is given by Deen [15] as an analog to Maxwell's relation of an effective dielectric constant as:

$$\mathrm{Sh} = 1 + 3\phi \frac{\alpha - 1}{\alpha + 2} \tag{5.9}$$

where $\alpha = D_p M/D$, D_p and D being the diffusivities of the tracers in the dispersed phase and the continuous phase respectively and M the solubility ratio of tracer concentration in the dispersed and the continuous phase. In the case of passive fluid-phase tracers, $D_p = 0, M = 0$ owing to the no-flux boundary at the particle surface. Hence, Equation 5.9 simplifies to:

$$\mathrm{Sh} = 1 - \frac{3\phi}{2} \tag{5.10}$$

Three values of volume fraction $\phi = \{0, 0.1, 0.2\}$ are simulated. The domain is initialised with a quiescent state and the particles are dispersed as per the required volume fraction. Since *Bluebottle* iterations consume 90% of the computation time, the domain is initialised and only the *tracer* module is executed until the simulation endtime. The flow field and the particles are imported from *Bluebottle* only once at the start of the simulation as the flow field is quiescent and steady. This conserves even more computational time.

The *tracer* module was simulated from a total simulation time of 5 times the time scale required for the bulk of tracers to cross the domain. This is estimated from the diffusivity D and the domain length H as H^2/D. The time-step for the simulation is user-defined since there is no coupling with *Bluebottle* other than the particle locations. The time-step is chosen in such a way that the magnitude of Brownian displacement $\sqrt{6D\Delta T}$ equates to a tenth of the particle radius i.e., $\Delta t = a^2/600D$. This assumption is used to ensure that the `tracer_pusher` subroutine does not result in non-physical dynamics of the tracers at the particle surface. Further case setup information is presented in Appendix B.1.

Initially, the *tracer* module was simulated without the `periodic_maneuver` subroutine and an offset in the results was observed. The offset decreased with increase in domain size from which discrepancies in wall effects were inferred. Then, the `periodic_maneuver` subroutine was devised and was used in all the forthcoming simulations.

Figure 5.2 shows the results of the simulation against the theoretical model described in Equation 5.10. In the three simulations executed, an approximately constant positive offset in Sh of 0.03 is observed. The error in the resulting values for the three values of particle volume fraction ϕ is constant. Moreover, the error persists for a particle volume fraction $\phi = 0$. Thus, it can be inferred that

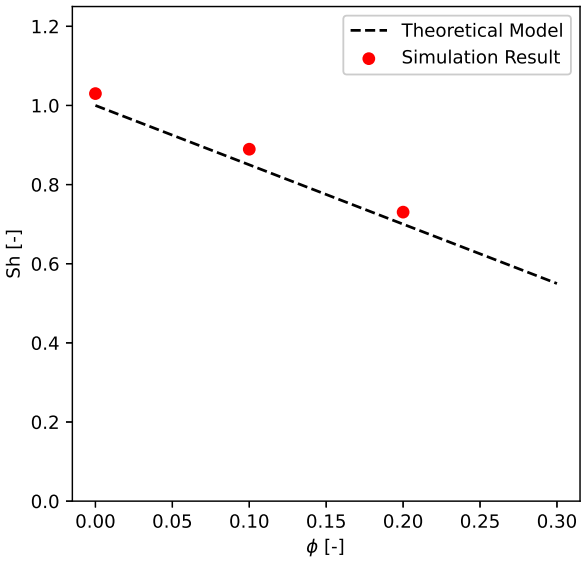


Figure 5.2: Simulation & Theoretical Sherwood number (Sh) vs. particle volume fraction (ϕ) for a stagnant suspension

the cause of the error is not particle interaction effects. The contribution of the interpolation term can also be neglected since the current simulation does not utilise interpolation functions. The simulations were run for varying diffusivity values and similar results were observed, thus ruling out the possibility that the Brownian term is the cause of the error. Hence, in the current work, the origin of the constant error is unknown and a better evaluation of the case is recommended as a follow-up.

5.1.2. Mass transport in sheared suspensions

A sheared suspension is expected to increase mass transfer of the tracers owing to the convective mass transport of the tracers with the flow field. The Péclet number defines the rate of shear with regards to molecular diffusion of the fluid. It is given by Equation 5.7. The *north* and *south* planes of the domain are given a uniform velocity of $\pm\Gamma H/2$, H being the domain size in the Y-direction.

For the simulations of sheared suspensions, the *tracer* module is coupled with the flow field computed by *Bluebottle*. Hence, the time-step for the *tracer* module is the same as the time-step used by *Bluebottle*. The CFL number for the *Bluebottle* simulation was fixed at 0.5 as it was found to offer better simulation time by providing a good balance of time-step and number of iterations per time-step. The *Bluebottle* simulation was initialised with a flow profile resembling shear across the two solid walls.

Wang et al. [57] recommended a simulation until a total strain Γt of 850 to achieve a steady state of flow field and mass transport. An equivalent simulation in *Bluebottle* would consume a large computational time. Hence, for the purpose of speed, the tracers were initialised by running a simulation of a stagnant suspension with the generated particle location, with a simulation time similar to Section 5.1.1. The final location of the tracers were then used as the initial location for the simulation of sheared suspensions. This is performed since the distribution of tracers after the simulation of a stagnant suspension (see Figure 5.1) is closer to the expected steady state distribution of tracers in a sheared suspension [57] than a random distribution. This method of initialisation cuts the required simulation runtime short by approximately ten times and a comparable value of Sh is obtained after approximately 75000 timesteps for the simulations discussed.

Since each simulation took approximately two weeks to be completed, only a specific case for validation with results from Wang et al. are considered - the particle Reynolds number is fixed at Re = 4 and the Péclet number is varied as Pe = {100, 200, 300}. Convergence in Sh is not checked simultaneously during the simulation and is only verified during post-processing (see Appendix C.1). The Sherwood number Sh is computed similar to Section 5.1.1 and the results are presented in Figure 5.3. The first 10000 timesteps are not considered for the estimation of Sh as a buffer to let the concentration distribution align with the shear flow profile. The results are compared with the empirical model proposed by Wang et al. [57] and also their numerical data. The results of Sh vs. Pe show a good agreement with the trend in the theoretical model. However, an underestimation in Sh of 14%, 12.2% and 1.4% for Pe= {100, 200, 300} respectively is observed. The error in the estimation of Sh can be attributed to a shorter time interval considered for the estimation of Sh i.e, the simulation could have been carried out longer for more precise results.

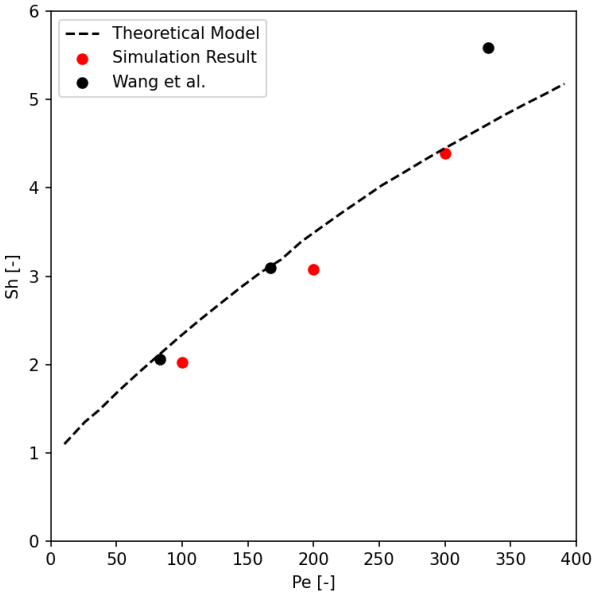


Figure 5.3: Simulation & Theoretical Sherwood number (Sh) vs. Péclet number (Pe) for sheared suspensions

Hence, with the simulations of stagnant and sheared suspensions, the *tracer* module is validated for the simulation of stochastic mass transport with the inclusion of convective and diffusive flux.

5.2. Comparison with an Eulerian convection-diffusion solver

Bluebottle also has an inbuilt scalar solver that solves the convection-diffusion equation for any scalar simultaneously with the multiphase flow equations. The scalar solver is an Eulerian solver that computes on the same grid as that of the fluid mesh. The Langevin equation used for the simulation of tracers is a stochastic analog of the convection-diffusion equation and hence, the scalar solver of *Bluebottle* can be compared with the statistical results of the *tracer* module.

A simulation with the same premise as that in Section 4.3.1 is used to visualise the behavior of the scalar field in comparison with the concentration field from tracer statistics – collision of a spherical particle with a wall of tracers and head-on collision of two spherical particles. Periodic boundary conditions were applied to all the faces of the domain for the scalar field and the field was initialised with a value of 1 in the same region as the initialisation of tracers - a plane in yz-direction of length and width equal to quarter of the domain size situated at the center of the domain in the x-direction, with the rest of the region being initialised with value 0. The particles are initialised with a value

of 0 for the scalar. The no-flux boundary at the particle surface for the scalar can be enforced by assuming $C_{p,p} \ll C_{p,f}$ where $C_{p,p}$ is the specific heat analog for the particle and $C_{p,f}$ for the fluid. However, it was observed that very low values of $C_{p,p}$ results in lower stability of the scalar solver. Hence, $C_{p,p}$ was chosen such that $C_{p,f} = 1000 C_{p,p}$. Figure 5.4 shows the comparison between the locations of the tracers and the scalar field. The time evolution of the simulation is presented in Appendix C.2.1. It is observed that the tracers accurately capture the profile of the scalar field. It is also noted that the scalar solver of *Bluebottle* has certain artefacts in the solution which result in alternating values on the wake of the particle. The *tracer* module is free from such artefacts.

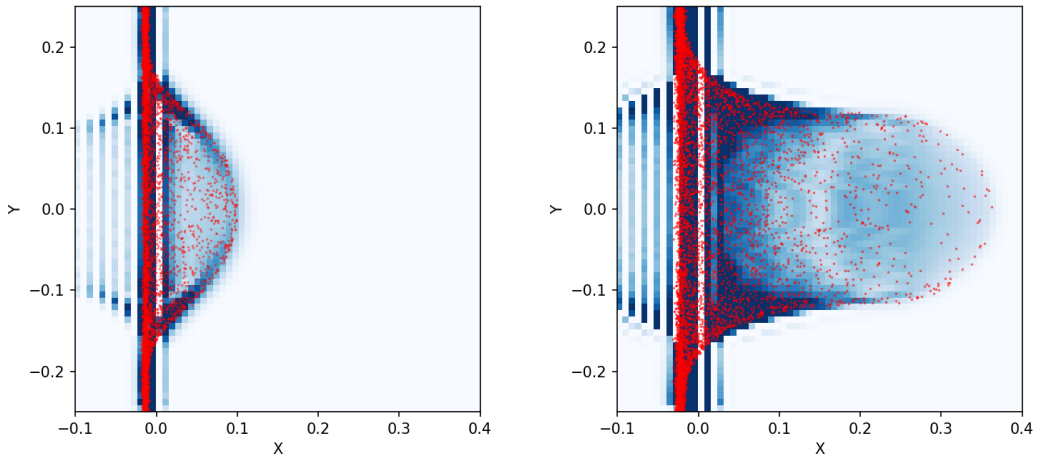


Figure 5.4: Comparison of tracer locations (red dots) and scalar field solution (blue contour) at $t = 0.02$s (left) and $t = 0.05$s (right) with $D = 1\text{e}{-4}$ m^2/s.

The estimation of a scalar field from the tracer locations is executed using binning of tracer locations into a grid similar to the mesh used for the scalar field solution. The Python implementation for the determination of the tracer concentration field from the tracer locations is presented in Appendix B.4. In this case, the number of tracers greatly influence the quality of results. This can be seen in Figure 5.5. Here, two spherical particles undergo a head-on collision with a wall of tracers and the scalar field at the point of collision. The simulation parameters are set similar to Section 4.3.2. The locations of the tracers are binned into a grid of the same size as that of the mesh used for the scalar field. The concentration field for the tracers is averaged from 20 grid nodes about the center of the domain in the Z-direction and are visualised in the XY-plane. Similar averaging is applied to the scalar field. It is observed that the concentration field from the averaging of tracer locations contains noise. Despite the noise, the profile resembles the scalar field and weak structures like adhesion of the scalar field to the particle surface is also captured quite well. The full time evolution of the scalar field and the tracer concentration field is presented in Appendix C.2.2.

Finally, the *tracer* module is compared to the scalar solver in terms of execution time. This comparison is presented in Figure 5.6. For the comparison, the simulation of sheared suspensions with 15 particles as in Section 5.3 is considered. The scalar field is initialised with opposing Dirichlet boundary conditions for the scalar on the rigid walls where mass transport is monitored. It can be seen that the computation time per iteration of the scalar solver is smaller than the time required to simulate one time-step for 2000 tracers. However, considering the scalar solution requires, on average, more than one iteration per time-step, the total execution time for the scalar solver is of approximately the same order of magnitude as the time consumed for a tracer iteration.

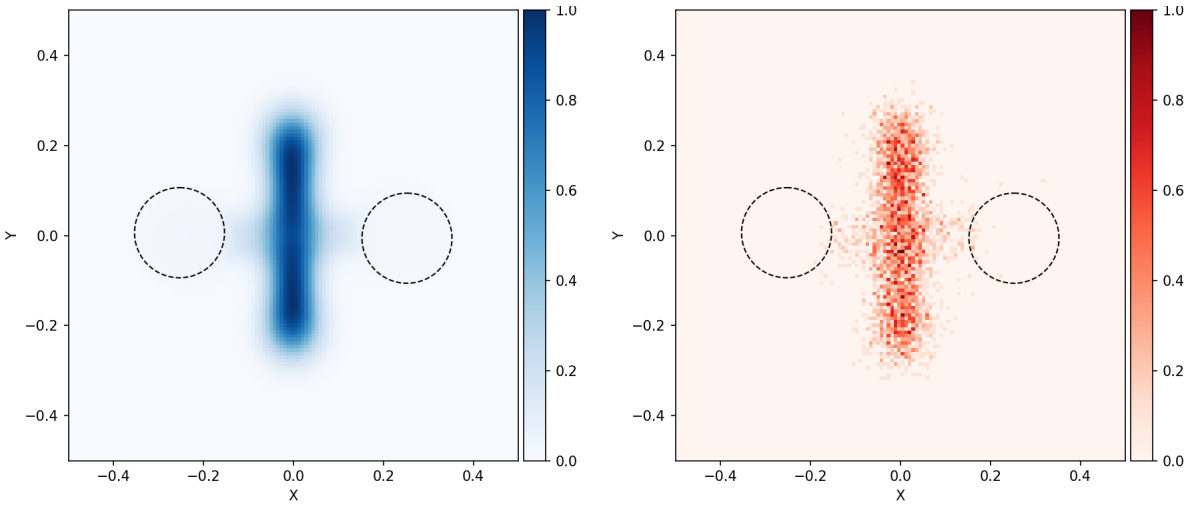


Figure 5.5: Scalar field (left) vs. tracer concentration field (right) at $t = 0.09$s, $D = 1\text{e}{-2}$ m^2/s; outlines of particles of $a = 0.1$m are shown in black dotted lines

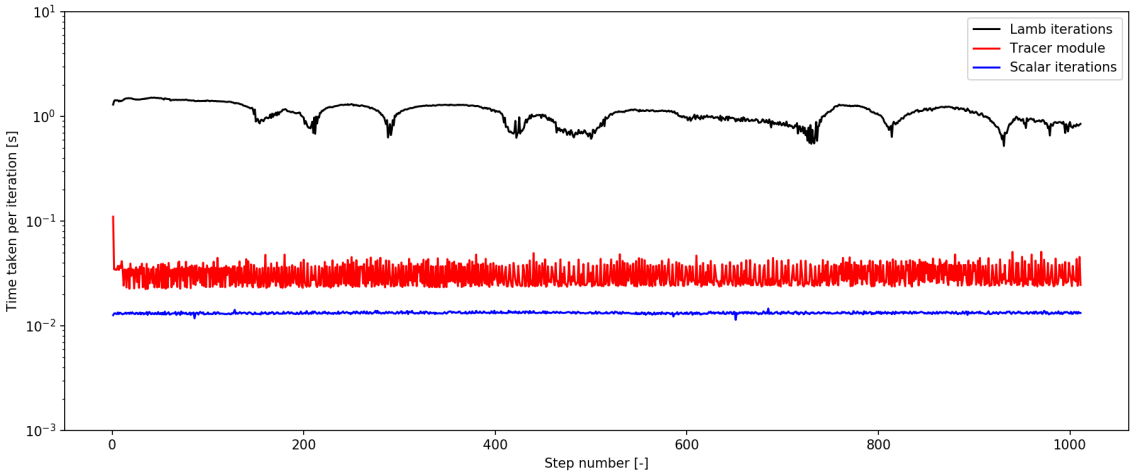


Figure 5.6: Variation in execution time per iteration for Lamb iteration vs. *tracer* execution vs. scalar solver iteration

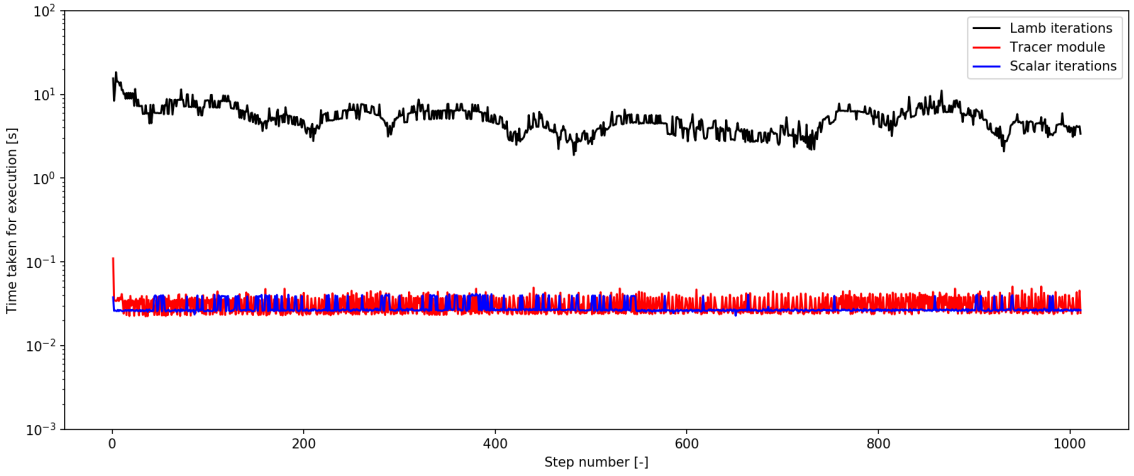


Figure 5.7: Variation in total execution time for Lamb iterations vs. *tracer* execution vs. scalar solver iterations

6

Summary & Conclusions

6.1. Installation of *Bluebottle* and the *tracer* module

The fork of *Bluebottle* containing the *tracer* module can be found at `https://github.com/shyam97/bluebottle3`. The `install.sh` file in the root directory is a shell script that can be used to install *Bluebottle* with the prerequisites being an installation of CUDA (>=9.0), CMake and a network connection. The flow field variables can be controlled using the `.config` files in the `/sim/input/` directory of *Bluebottle*. The parameters for the *tracer* module can be controlled from the function `tracerstart` in the file `/src/tracer.c`. Changes to the file `tracer.c` require that the *Bluebottle* executable is refreshed with the `make` command. The output files can be found in the directory `/sim/output/` and the post-processing tools can be found in the directory `/tools/python/`. The *tracer* module can be also be used coupled with other flow solvers provided that the flow field coupling and the *tracer* execution is handled properly.

6.2. Summary of the *tracer* module code

A Lagrangian method for simulation of mass transport via convective and diffusive motion of fluid-phase tracers is developed. This method is developed as a one-way coupling to *Bluebottle*, a particulate multiphase flow solver that uses the PHYSALIS algorithm. The tracers are simulated using a simplified form of the Langevin equation. The dynamics of a tracer can be isolated into diffusive and convective displacement, and interaction with the domain boundaries and particles. The different modules in code execution are summarised as follows.

- The diffusive term of the tracer displacement is calculated using a Gaussian random displacement with magnitude $\sqrt{6\Delta t}$.
- The convective displacement is modelled using a simple one-way coupling with the flow field at the present time-step for the tracers. The flow field solution is imported from *Bluebottle* and interpolated in a trilinear fashion to determine the field values at the location of each tracer.
- The domain boundaries are modelled either as solid walls that provide specular reflection upon collision of a tracer or as periodic boundaries that redirect the tracers to the opposite side of the domain upon intrusion. Boundary conditions that resemble the conventional Eulerian boundary conditions such as constant value (Dirichlet) and constant flux (Neumann)

boundary conditions are also developed to be used close to boundaries. Furthermore, a boundary condition similar to the one developed by Wang et al. where two "types"of tracers are used to estimate bidirectional mass flux is also implemented.

- The no penetration boundary at the particle surface is numerically implemented. Specular reflection algorithms are used to enfore this boundary condition if the tracer displacement leads to a possible intrusion of the particle surface. In the event where the particle displacement engulfs a tracer, a fail-safe method to push the tracers out of the particle is devised and implemented.
- To preserve particle interactions close to periodic boundaries, an algorithm to temporarily place the particle so as to be detected by the particle interaction algorithms is developed and implemented. This algorithm computes a set of flags depending on the proximity to domain boundaries that are unique to every tracer-particle pair.
- The code is developed to run immediately after every flow field solution step of *Bluebottle*. In cases where the flow field is expected to be quiescent or steady state, the *tracer* module can also be programmed to run for the whole simulation without the execution of the *Bluebottle* solver.
- The tracer locations are exported and imported as .CSV files. Post-processing of exported data is executed using Python and Matplotlib.

The runtime for the *tracer* module within *Bluebottle* is tested. The computation time for the *tracer* module is observed to be an order of magnitude smaller compared to the flow field iterations of *Bluebottle*. A large part of the tracer computation time is estimated to be from the flow field interpolation. Furthermore, the computational time of the *tracer* module is

- independent of the mesh size and resolution for the dynamics of the tracers. However, the interpolation subroutine is dependent on the mesh resolution.
- independent of the time-step size. However, forcing a larger time-step would result in the violation of the enforced no-flux boundary at the particle surface and would result in non-physical dynamics of the tracers.
- directly proportional to the number of particles with a linear proportionality.
- directly proportional to the number of tracers with a linear proportionality.

6.3. Summary of code validation

The individual modules of the *tracer* module were validated. The results are summarised as below.

- The mean-squared-displacement statistics of pure diffusion indicate agreement with the theoretical model.
- The coupling to the flow field is tested with a simple 2D flow about a cylinder. The pathlines generated by tracers distributed equally about the diametrical axis of the cylinder in the flow direction matches the streamlines of the flow accurately.
- The no penetration boundary is tested with a simulation of a spherical particle colliding with a wall of tracers and a head-on collision of two spherical particles with a wall of tracers at the point of collision. No intrusions were observed in the former case. In the latter case, intrusions corresponding to the export precision of tracer locations were observed.

The *tracer* module is validated with simulations of mass transport across stagnant and sheared suspensions. These simulations served as feedback for the implementation of several subroutines to improve the precision in capturing particle and domain extent interactions. The estimated Sherwood number matches the expected results well.

The *tracer* module is also compared with an Eulerian scalar solver that is inbuilt in *Bluebottle*. The *tracer* module captures flow profiles and features accurately. However, the statistical noise present in the stochastic simulation greatly affects space- and time-averaged results. This noise can only be controlled by a very high number of tracers.

6.3.1. Scope for improvement of current code

The following modifications can be performed to the present code.

- The current *tracer* module is executed fully on the CPU without the use of CUDA. This means that the flow field needs to be imported from CUDA device every time-step for the execution of the *tracer* module. This import step takes approximately 0.1 seconds per time-step in the execution of the *tracer* module. To minimise this time consumption, the *tracer* module can be formatted to run on the CUDA device. This will not only erase the computation time for the flow field import but also increase the speed of *tracer* module execution.
- The interpolation of the flow field close to the walls can be modified to enable a higher order method of interpolation.
- Stokes equations for flow around a sphere can be used to improve the interpolation of the flow field close to the surface of a particle.
- The input and output parameters to the *tracer* module can be merged with the config files of *Bluebottle* to ensure a cleaner method of execution.
- The discrepancy in the value of Sh for stagnant particulate suspensions discussed in Section 5.1.1 needs to be addressed.

6.4. Scope for future work

The current work aims to develop a computational tool for the simulation of passive tracers. The dynamics of the tracers are modelled using the Langevin equation for position (Equation 3.1). In the current work, the diffusive and the convective fluxes are modelled. Good agreement was obtained with prior experimental and numerical results for convection-diffusion. The *tracer* module finds its use as is in the following applications.

1. The tracers are assumed to have no relative velocity with the fluid. Hence, the tracers can be used as fluid-phase tracers in simulations of convective-diffusive mass transport in particulate suspensions.

2. The *tracer* module can be used to simulate scalar fields in particulate flows stochastically using a Lagrangian approach. A validation case for the simulation of tracers to represent a scalar field is presented and the tracers are observed to capture structures and profiles of the scalar field quite well. The flux of a scalar field can also be captured discretely and cumulatively time-averaged to determine the average flux. As simulations in *Bluebottle* are restricted to simulations of particulate multiphase flows with spherical particles, a specific use case for the *tracer* module is the simulation of a scalar field in fluidized beds or packed beds of spheres.

3. The good agreement of experimental results of Wang et al. [57] provides a proof of concept for the simulation of mass transport of weak electrolytes in electrolytic processes. The limiting current measurement technique [21, 45, 46] is a standard mass transfer measurement technique where a weak electrolyte is used to measure mass transport with an electrolytic solution that is saturated with a strong electrolyte. The strong electrolyte screens the electric field in the bulk of the electrolyte, thereby reducing the migrative flux of the weak electrolyte [45]. However, the same technique cannot be applied for water electrolyzers directly as Haverkort et al. [22] showed the migration contribution to the total flux of ions in a water electrolyzer to be stronger than the diffusive flux. The contribution of the electric potential varies depending on the distance of the ion from the electrode [49] and at larger distances, the electrostatic force decreases drastically. The Debye length provides a length scale above which the migration effects reduces considerably [58]. In macroscale systems with electrode gap considerably larger than the Debye length of the electrolyte, the migration flux is restricted to a thin space near the electrode and the primary flux of the ions through the bulk is diffusion [58]. In these specific cases, the *tracer* module can be used as is with the assumption that the contribution of migrative flux on the motion of ions is negligible.

To simulate ionic mass transport close to the electrode, the migration flux needs to be modelled. This can be done using an electric field-mediated force that requires the knowledge of either the electric field or the potential gradient at individual locations in the domain. Coulomb's force equivalence

$$\mathbf{F}_c = \mathbf{E}q \tag{6.1}$$

can then be used with the Stokes drag

$$\mathbf{F}_d = -\zeta|\mathbf{u}-\mathbf{v}|\cdot\hat{\mathbf{v}} \tag{6.2}$$

to estimate the net force on the ion as

$$m\frac{\partial \mathbf{v}}{\partial t} = \mathbf{E}q - \zeta|\mathbf{u}-\mathbf{v}|\cdot\hat{\mathbf{v}} \tag{6.3}$$

relative to the fluid phase. Here, $\mathbf{E}$ represents the electric field, q the charge of the ion, m the mass of the ion, $\mathbf{u}$ the velocity of the fluid phase at the location of the ion, $\mathbf{v}$ the velocity of the ion and $\hat{\mathbf{v}}$ the unit vector in the direction of $\mathbf{v}$. ζ is the Stokes drag coefficient of an ion given by

$$\zeta = 6\pi\mu r \tag{6.4}$$

where μ is the dynamic visocity of the fluid phase and r is the radius of the ion assumed to be spherical.

In this case, two equations - the velocity equation given by Equation 6.3 and the displacement equation given by Equation 3.1 - needs to be modelled. Hence, the velocity of the ions should be stored as an array in the tracer datatype. Modelling of electrostatic force and an addition of an electrostatic force term to the Langevin equation makes the equation a stochastic analog of the Nernst-Planck equation 1.6. Hence, upon addition of the electrostatic force term, the scalar field associated with the tracers in the presence of an electric field can be validated by simulations of the Nernst-Planck equation. The concentration of ions at the surface of a bubble can be assumed to be the saturation concentration [23]. This requires the development of relevant boundary conditions for tracer concentration at the surface of the bubble. In this case, not all ions would be reflected in a specular fashion at the bubble surface. An increase in the concentration of ions close to a bubble will result in supersaturation at the bubble surface and subsequently, growth of the bubble [3, 55]. As the diameter of particles simulated by *Bluebottle* cannot be changed during the simulations, diffusive growth of bubbles cannot be modelled using this method.

Bibliography

[1] Lokmane Abdelouahed, Rainier Hreiz, Souhila Poncin, Gérard Valentin, and Francois Lapicque. Hydrodynamics of gas bubbles in the gap of lantern blade electrodes without forced flow of electrolyte: Experiments and CFD modelling. *Chemical Engineering Science*, 111:255–265, may 2014. doi: 10.1016/j.ces.2014.01.028.

[2] K. Aldas, N. Pehlivanoglu, and M. Mat. Numerical and experimental investigation of two-phase flow in an electrochemical cell. *International Journal of Hydrogen Energy*, 33(14):3668–3675, jul 2008. doi: 10.1016/j.ijhydene.2008.04.047.

[3] Andrea Angulo, Peter van der Linde, Han Gardeniers, Miguel Modestino, and David Fernández Rivas. Influence of bubbles on the energy conversion efficiency of electrochemical reactors. *Joule*, 4(3):555–579, mar 2020. doi: 10.1016/j.joule.2020.01.005.

[4] Damien Le Bideau, Philippe Mandin, Mohamed Benbouzid, Myeongsub Kim, Mathieu Sellier, Fabrizio Ganci, and Rosalinda Inguanta. Eulerian two-fluid model of alkaline water electrolysis for hydrogen production. *Energies*, 13(13):3394, jul 2020. doi: 10.3390/en13133394.

[5] J. J. Bluemink, D. Lohse, A. Prosperetti, and L. Van Wijngaarden. A sphere in a uniformly rotating or shearing flow. *Journal of Fluid Mechanics*, 600:201–233, mar 2008. doi: 10.1017/s0022112008000438.

[6] P. Boissonneau and P. Byrne. An experimental investigation of bubble-induced free convection in a small electrochemical cell. *Journal of Applied Electrochemistry*, 30(7):767–775, 2000. doi: 10.1023/a:1004034807331.

[7] B. E. Bongenaar-Schlenter, L. J. J. Janssen, S. J. D. Van Stralen, and E. Barendrecht. The effect of the gas void distribution on the ohmic resistance during water electrolytes. *Journal of Applied Electrochemistry*, 15(4):537–548, jul 1985. doi: 10.1007/bf01059295.

[8] D. A. G. Bruggeman. Berechnung verschiedener physikalischer konstanten von heterogenen substanzen. i. dielektrizitätskonstanten und leitfähigkeiten der mischkörper aus isotropen substanzen. *Annalen der Physik*, 416(7):636–664, 1935. doi: 10.1002/andp.19354160705.

[9] Alexander Buttler and Hartmut Spliethoff. Current status of water electrolysis for energy storage, grid balancing and sector coupling via power-to-gas and power-to-liquids: A review. *Renewable and Sustainable Energy Reviews*, 82:2440–2454, 2018.

[10] Barbaros Cetin and Dongqing Li. Effect of joule heating on electrokinetic transport. *ELECTROPHORESIS*, 29(5):994–1005, mar 2008. doi: 10.1002/elps.200700601.

[11] Prasad Chandran, Shamit Bakshi, and Dhiman Chatterjee. Study on the characteristics of hydrogen bubble formation and its transport during electrolysis of water. *Chemical Engineering Science*, 138:99–109, dec 2015. doi: 10.1016/j.ces.2015.07.041.

[12] Roland Clift, John R Grace, and Martin E Weber. Bubbles, drops, and particles. 2005.

[13] Peter Comninos. *Mathematical and computer programming techniques for computer graphics*. Springer Science & Business Media, 2010.

[14] Anders A. Dahlkild. Modelling the two-phase flow and current distribution along a vertical gas-evolving electrode. *Journal of Fluid Mechanics*, 428:249–272, feb 2001. doi: 10.1017/s0022112000002639.

[15] William Murray Deen. *Analysis of Transport Phenomena*, volume 2. Oxford University Press New York, 1998.

[16] Géraldine Duhar and Catherine Colin. Dynamics of bubble growth and detachment in a viscous shear flow. *Physics of Fluids*, 18(7):077101, jul 2006. doi: 10.1063/1.2213638.

[17] John Dukovic and Charles W. Tobias. The influence of attached bubbles on potential drop and current distribution at gas-evolving electrodes. *Journal of The Electrochemical Society*, 134(2): 331–343, feb 1987. doi: 10.1149/1.2100456.

[18] J. Eigeldinger and H. Vogt. The bubble coverage of gas-evolving electrodes in a flowing electrolyte. *Electrochimica Acta*, 45(27):4449–4456, sep 2000. doi: 10.1016/s0013-4686(00)00513-2.

[19] C. Gabrielli, F. Huet, and R.P. Nogueira. Fluctuations of concentration overpotential generated at gas-evolving electrodes. *Electrochimica Acta*, 50(18):3726–3736, jun 2005. doi: 10.1016/j.electacta.2005.01.019.

[20] Sean R. German, Martin A. Edwards, Qianjin Chen, Yuwen Liu, Long Luo, and Henry S. White. Electrochemistry of single nanobubbles. estimating the critical size of bubble-forming nuclei for gas-evolving electrode reactions. *Faraday Discussions*, 193:223–240, 2016. doi: 10.1039/c6fd00099a.

[21] DW Gibbons, RH Muller, and CW Tobias. Mass transport to cylindrical electrodes rotating in suspensions of inert microspheres. *Journal of the Electrochemical Society*, 138(11):3255, 1991.

[22] J.W. Haverkort and H. Rajaei. Electro-osmotic flow and the limiting current in alkaline water electrolysis. *Journal of Power Sources Advances*, 6:100034, dec 2020. doi: 10.1016/j.powera.2020.100034.

[23] F.J. Higuera. A model of the growth of hydrogen bubbles in the electrolysis of water. *Journal of Fluid Mechanics*, 927, sep 2021. doi: 10.1017/jfm.2021.778.

[24] Fumio Hine and Koichi Murakami. Bubble effects on the solution IR drop in a vertical electrolyzer under free and forced convection. *Journal of The Electrochemical Society*, 127(2):292–297, feb 1980. doi: 10.1149/1.2129658.

[25] SR Holm, H Polinder, JA Ferreira, P Van Gelder, and R Dill. A comparison of energy storage technologies as energy buffer in renewable energy sources with respect to power capability. In *IEEE young researchers symposium in electrical power engineering*, 2002.

[26] Rainier Hreiz, Lokmane Abdelouahed, Denis Fünfschilling, and François Lapicque. Electrogenerated bubbles induced convection in narrow vertical cells: PIV measurements and euler–lagrange CFD simulation. *Chemical Engineering Science*, 134:138–152, sep 2015. doi: 10.1016/j.ces.2015.04.041.

[27] Hussein Ibrahim, Adrian Ilinca, and Jean Perron. Energy storage systems—characteristics and comparisons. *Renewable and sustainable energy reviews*, 12(5):1221–1250, 2008.

[28] IEA. Global energy review 2021. *IEA (2021)*, 2021.

[29] Hydrogen Irena. A renewable energy perspective. *IRENA, Abu Dhabi*, 2019.

[30] Md H. Islam, Odne S. Burheim, and Bruno G. Pollet. Sonochemical and sonoelectrochemical production of hydrogen. *Ultrasonics Sonochemistry*, 51:533–555, mar 2019. doi: 10.1016/j.ultsonch.2018.08.024.

[31] Paul Langevin. Sur la théorie du mouvement brownien. *Compt. Rendus 146*, pages 530–533, 1908.

[32] Sheng-De Li, Cheng-Chien Wang, and Chuh-Yung Chen. Water electrolysis in the presence of an ultrasonic field. *Electrochimica Acta*, 54(15):3877–3883, jun 2009. doi: 10.1016/j.electacta.2009.01.087.

[33] Yifan Li, Gaoqiang Yang, Shule Yu, Zhenye Kang, Jingke Mo, Bo Han, Derrick A. Talley, and Feng-Yuan Zhang. In-situ investigation and modeling of electrochemical reactions with simultaneous oxygen and hydrogen microbubble evolutions in water electrolysis. *International Journal of Hydrogen Energy*, 44(52):28283–28293, oct 2019. doi: 10.1016/j.ijhydene.2019.09.044.

[34] Ming-Yuan Lin, Lih-Wu Hourng, and Chan-Wei Kuo. The effect of magnetic force on hydrogen production efficiency in water electrolysis. *International Journal of Hydrogen Energy*, 37(2):1311–1320, jan 2012. doi: 10.1016/j.ijhydene.2011.10.024.

[35] Philippe Mandin, Jérôme Hamburger, Sebastien Bessou, and Gérard Picard. Modelling and calculation of the current density distribution evolution at vertical gas-evolving electrodes. *Electrochimica Acta*, 51(6):1140–1156, nov 2005. doi: 10.1016/j.electacta.2005.06.007.

[36] Radenka Maric and Haoran Yu. Proton exchange membrane water electrolysis as a promising technology for hydrogen production and energy storage. *Nanostructures in energy generation, transmission and storage*, page 13, 2019.

[37] M Mat. A two-phase flow model for hydrogen evolution in an electrochemical cell. *International Journal of Hydrogen Energy*, 29(10):1015–1023, aug 2004. doi: 10.1016/j.ijhydene.2003.11.007.

[38] James Clerk Maxwell. *A treatise on electricity and magnetism*, volume 1. Oxford: Clarendon Press, 1873.

[39] F. M. Mulder. Implications of diurnal and seasonal variations in renewable energy generation for large scale energy storage. *Journal of Renewable and Sustainable Energy*, 6(3):033105, may 2014. doi: 10.1063/1.4874845.

[40] Niro Nagai, Masanori Takeuchi, and Tetsuya Furuta. Effects of bubbles between electrodes on alkaline water electrolysis efficiency under forced convection of electrolyte. In *Proceedings of 16th world hydrogen energy conference, Lyon*, pages 1–10, 2006.

[41] Nvidia. Cuda c++ programming guide, 2021. URL `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.

[42] A. Prosperetti and H.N. Ogũz. Physalis: A new o(n) method for the numerical simulation of disperse systems: Potential flow of spheres. *Journal of Computational Physics*, 167(1):196–216, feb 2001. doi: 10.1006/jcph.2000.6667.

[43] WB Russel, DA Saville, and WR Schowalter. Colloidal dispersions cambridge univ, 1989.

[44] Maximilian Schalenbach, Geert Tjarks, Marcelo Carmo, Wiebke Lueke, Martin Mueller, and Detlef Stolten. Acidic or alkaline? towards a new perspective on the efficiency of water electrolysis. *Journal of The Electrochemical Society*, 163(11):F3197–F3208, 2016. doi: 10.1149/2.0271611jes.

[45] J Robert Selman and Charles W Tobias. Mass-transfer measurements by the limiting-current technique. In *Advances in chemical engineering*, volume 10, pages 211–318. Elsevier, 1978.

[46] JR Selman and JC McClure. Limiting current to a vertical rotating rod electrode. *Journal of Electroanalytical Chemistry and Interfacial Electrochemistry*, 110(1-3):79–92, 1980.

[47] Adam J Sierakowski and Andrea Prosperetti. Resolved-particle simulation by the physalis method: Enhancements and new capabilities. *Journal of computational physics*, 309:164–184, 2016.

[48] L. Sigrist, O. Dossenbach, and N. Ibl. On the conductivity and void fraction of gas dispersions in electrolyte solutions. *Journal of Applied Electrochemistry*, 10(2):223–228, mar 1980. doi: 10.1007/bf00726089.

[49] Yohichi Suzuki and Kazuhiko Seki. Possible influence of the kuramoto length in a photocatalytic water splitting reaction revealed by poisson–nernst–planck equations involving ionization in a weak electrolyte. *Chemical Physics*, 502:39–49, mar 2018. doi: 10.1016/j.chemphys.2018.01.006.

[50] S Takagi, H.N Ogũz, Z Zhang, and A Prosperetti. PHYSALIS: a new method for particle simulation. *Journal of Computational Physics*, 187(2):371–390, may 2003. doi: 10.1016/s0021-9991(03)00077-9.

[51] Yoshinori Tanaka, Kenji Kikuchi, Yasuhiro Saihara, and Zempachi Ogumi. Bubble visualization and electrolyte dependency of dissolving hydrogen in electrolyzed water using solid-polymer-electrolyte. *Electrochimica acta*, 50(25-26):5229–5236, sep 2005. doi: https://doi.org/10.1016/j.electacta.2005.01.062.

[52] G. Y. Tang, C. Yang, C. J. Chai, and H. Q. Gong. Modeling of electroosmotic flow and capillary electrophoresis with the joule heating effect: the nernst-planck equation versus the boltzmann distribution. *Langmuir*, 19(26):10975–10984, dec 2003. doi: 10.1021/la0301994.

[53] Amir Taqieddin, Roya Nazari, Ljiljana Rajic, and Akram Alshawabkeh. Review—physicochemical hydrodynamics of gas bubbles in two phase electrochemical systems. *Journal of The Electrochemical Society*, 164(13):E448–E459, 2017. doi: 10.1149/2.1161713jes.

[54] A. ten Cate, C. H. Nieuwstad, J. J. Derksen, and H. E. A. Van den Akker. Particle imaging velocimetry experiments and lattice-boltzmann simulations on a single sphere settling under gravity. *Physics of Fluids*, 14(11):4012–4025, nov 2002. doi: 10.1063/1.1512918.

[55] Peter van der Linde, Álvaro Moreno Soto, Pablo Peñas-López, Javier Rodríguez-Rodríguez, Detlef Lohse, Han Gardeniers, Devaraj van der Meer, and David Fernández Rivas. Electrolysis-driven and pressure-controlled diffusive growth of successive bubbles on microstructured surfaces. *Langmuir*, 33(45):12873–12886, oct 2017. doi: 10.1021/acs.langmuir.7b02978.

[56] H. Vogt. The rate of gas evolution of electrodes—i. an estimate of the efficiency of gas evolution from the supersaturation of electrolyte adjacent to a gas-evolving electrode. *Electrochimica Acta*, 29(2):167–173, feb 1984. doi: 10.1016/0013-4686(84)87043-7.

[57] Luying Wang, Donald L Koch, Xiaolong Yin, and Claude Cohen. Hydrodynamic diffusion and mass transfer across a sheared suspension of neutrally buoyant spheres. *Physics of Fluids*, 21 (3):033303, mar 2009. doi: https://doi.org/10.1063/1.3098446.

[58] Yifei Wang, S. R. Narayanan, and Wei Wu. Field-assisted splitting of pure water based on deep-sub-debye-length nanogap electrochemical cells. *ACS Nano*, 11(8):8421–8428, jul 2017. doi: 10.1021/acsnano.7b04038.

[59] Z. Zhang and A. Prosperetti. A second-order method for three-dimensional particle simulation. *Journal of Computational Physics*, 210(1):292–324, nov 2005. doi: 10.1016/j.jcp.2005.04.009.

[60] Z. Z. Zhang, L. Botto, and A. Prosperetti. Microstructural effects in a fully-resolved simulation of 1,024 sedimenting spheres. In *Fluid Mechanics and Its Applications*, pages 197–206. Springer Netherlands. doi: 10.1007/1-4020-4977-3_20.

A

C code for tracer simulation

The following are the functions and files used in the *tracer* module. The latest version of the code, if updated, can be found in `https://github.com/shyam97/bluebottle3/tree/master/src` as `tracer.c` and `tracer.h`.

A.1. Header file

```
#include "bluebottle.h"
#include <sys/stat.h>
// #include <float.h>
// #include <math.h>
// #include <stdio.h>
// #include <stdlib.h>
// #include <stddef.h>
// #include <string.h>
// #include <time.h>
// #include <mpi.h>
// #include <sys/time.h>

#ifndef _TRACER_H
#define _TRACER_H

#define TRACER_DIR "tracer"
#define DEBUG_DIR "debug"
#define PARTICLEINFO_DIR "particleinfo"

void tracer_start(void);
void tracer_execute(double dt);
void tracer_exit(void);

void tracer_init(int i);
double randgen(void);
double *randomizer(double *randvec, double diffusivity, double tstep);
double distance(double *loc, double x, double y, double z);
double *tracer_pusher(double *loc, double x, double y, double z, double r, int *flags);
int wall_checker(double *loc);
int domain_checker(double *loc);
double *wall_reflector(int i, double *loc);
int particle_checker(double *loc, int *flags);
void periodic_maneuver(double *loc, int *flags, int max_a);
double *reflector(double *loc1, double *loc2, double x,
```

```
  double y, double z, double r, int *flags);
double *intersector(double *loc1, double *loc2, double x,
    double y, double z, double r, int *flags);
double index_finder(double *loc);
double grid_finder(double index, int axis, double gridloc);
double *interpolator(double *loc, double *uout, double *vout, double *wout);
void exporter(void);
void tracer_delete(int deleteid);
void tracer_add(void);
void ratekeeper(void);
void maintainer(void);

typedef struct tracer {
  int tracerid;
  double pos[3];
  int tracertype;
} tracer;

extern tracer *tracer_array;
extern real *ucc;
extern real *vcc;
extern real *wcc;
extern long int tracercheck;
extern double diffusivity;
extern double tstep;
extern int foundindex;
extern double interpvel[3];
extern int tracernum;
extern double locx[3];
extern int flags[3];
extern double dist_temp;
extern int idflag;
extern int farwalltreatment;
extern int fixednumber;
extern float fixedrate;
extern float timecheck;
extern int tracertreatment;
extern int tracerdebug;
extern int exportfreq;
extern int exportcount;
extern int readfile;
extern double max_radius;
extern int steady_flag[2];

#endif
```

A.2. Initialisation and finalisation of tracer simulation

```
/*------------------------------------------------------------------------*/
/* Initialize tracer simulation and arrays */
/*------------------------------------------------------------------------*/

void tracer_start(void)
{
  printf("\nInitializing tracer arrays... ");
  readfile = 0;
  tracercheck = 0;
  exportfreq = 10000;
  exportcount = 0;
  diffusivity = 1E-4;
```

```
13    farwalltreatment = 0; //0 for continuing motion, 1 for delete, 2 for stopping at end
14    tracertreatment = 0; // 0 for 2type, 1 for fixednumber, 2 for fixedrate
15    int steady_flag[] = {0,0}; // 0,0 for steady, 0,1 for shear
16
17    fixednumber = 500;
18
19    fixedrate = 0;
20    fixedrate2 = 5;
21    timecheck = ttime + fixedrate;
22
23    srand((unsigned int)time(NULL));
24
25    tracernum = 2000;
26    tracer_array = (tracer*) malloc(tracernum * sizeof(tracer));
27
28    if (tracertreatment==2 && fixedrate2>0) {
29          tracernum = 0;
30    }
31
32    max_radius = -1;
33    for (int i=0; i<NPARTS; i++) {
34      if (parts[i].r > max_radius) { max_radius = parts[i].r; }
35    }
36    //printf("%f", max_radius);
37
38    if (readfile == 0){
39      for (int i=0; i<tracernum; i++) {
40        tracer_init(i);
41    }
42    }
43    else if (readfile ==1){
44      char fname[30];
45      FILE * fp;
46      sprintf(fname, "%s/%s/tracer.csv", ROOT_DIR, INPUT_DIR);
47      fp = fopen (fname, "r");
48
49      for (int i=0;i<tracernum;i++){
50          fscanf(fp, "%d, %lf, %lf, %lf, %d", &tracer_array[i].tracerid,
51          &tracer_array[i].pos[0], &tracer_array[i].pos[1], &tracer_array[i].pos[2],
52          &tracer_array[i].tracertype);
53      }
54      fclose(fp);
55      printf("\nTracer locations imported successfully.\n");
56    }
57
58    struct stat st = {0};
59    char buf[CHAR_BUF_SIZE];
60    sprintf(buf, "%s/%s/%s", ROOT_DIR, OUTPUT_DIR, TRACER_DIR);
61    if (stat(buf, &st) == -1) {
62      mkdir(buf, 0700);
63    }
64
65    sprintf(buf, "%s/%s/%s", ROOT_DIR, OUTPUT_DIR, PARTICLEINFO_DIR);
66    if (stat(buf, &st) == -1) {
67      mkdir(buf, 0700);
68    }
69
70    // sprintf(buf, "%s/%s/%s", ROOT_DIR, OUTPUT_DIR, DEBUG_DIR);
71    // if (stat(buf, &st) == -1) {
72    //     mkdir(buf, 0700);
73    // }
```

```

  FILE *rundata;
  char rundataname[100];
  sprintf(rundataname, "%s/%s/domaindata.csv",ROOT_DIR, OUTPUT_DIR);
  rundata = fopen(rundataname,"w+");
  fprintf(rundata,"%f,%f,%f,%f,%f,%f\n", dom[rank].xs, dom[rank].xe, dom[rank].ys,
          dom[rank].ye, dom[rank].zs, dom[rank].ze);
  fclose(rundata);

  exporter();
  printf("done.\n");

}

/*----------------------------------------------------------------------------*/
/* Remove tracer array from memory */
/*----------------------------------------------------------------------------*/

void tracer_exit(void)
{
  free(tracer_array);
  if (steady_flag[1]==0) {
    free(ucc);
    free(vcc);
    free(wcc);
  }
}
```

A.3. Initialisation of individual tracer

```
/*----------------------------------------------------------------------------*/
/* Initialise each tracer*/
/*----------------------------------------------------------------------------*/

void tracer_init(int i)
{

  tracer_array[i].tracerid = i;

  tracer_array[i].pos[0] = dom[rank].xs + dom[rank].xl/2 + dom[rank].xl*randgen()/2;
  //tracer_array[i].pos[0] = dom[rank].xs + (i%2)*(dom[rank].xe - dom[rank].xs);
  // tracer_array[i].pos[0] = dom[rank].xs + 2*dom[rank].dx;
  // tracer_array[i].pos[0] = xlocation[i];

  // tracer_array[i].pos[1] = dom[rank].ys + (dom[rank].yl/2) + (i-9.5)*(dom[rank].yl/40);
  //tracer_array[i].pos[1] = dom[rank].ys + (i%2)*(dom[rank].ye - dom[rank].ys);
  tracer_array[i].pos[1] = dom[rank].ys + dom[rank].yl*(1+randgen())/2;
  // tracer_array[i].pos[1] = ylocation[i];

  tracer_array[i].pos[2] = dom[rank].zs + dom[rank].zl*(1+randgen())/2;
  // tracer_array[i].pos[2] = dom[rank].zs;
  // tracer_array[i].pos[2] = zlocation[i];

  tracer_array[i].tracertype = i%2;
}
```

A.4. Tracer boundary conditions

```
/*------------------------------------------------------------------------------*/
/* Maintain number of particles close to inlet */
/*------------------------------------------------------------------------------*/
void maintainer(void)
{
    int tracercount = 0;
    float safespace = dom[rank].dx;

    for (int a=0; a<tracernum; a++) {
    if (tracer_array[a].pos[0] < dom[rank].xs + safespace) {
        tracercount += 1;
    }
    }

    if (tracercount < fixednumber) {
    tracer_add();
    }
}

/*------------------------------------------------------------------------------*/
/* Keep on adding tracers at fixed rate */
/*------------------------------------------------------------------------------*/
void ratekeeper(void)
{
    if (ttime >= timecheck) {
    tracer_add();
    timecheck += fixedrate;
    }
}
```

A.5. Addition & Deletion of tracer from tracer array

```
/*------------------------------------------------------------------------------*/
/* Delete a tracer from current use */
/*------------------------------------------------------------------------------*/

void tracer_delete(int deleteid)
{
  for (int d=deleteid; d<tracernum-1; d++) {
    tracer_array[d].tracerid = tracer_array[d+1].tracerid;
    tracer_array[d].tracertype = tracer_array[d+1].tracertype;

    for (int e=0; e<3; e++) {
      tracer_array[d].pos[e] = tracer_array[d+1].pos[e];
    }
  }

  tracernum -= 1;
}

/*------------------------------------------------------------------------------*/
/* Add new tracer to the simulation */
/*------------------------------------------------------------------------------*/

void tracer_add(void)
{
  tracer_init(tracernum);
  tracernum+=1;
```

```
}
```

A.6. Calculating the Brownian term

```
/*------------------------------------------------------------------------*/
/* Generate random number from -1 to 1 */
/*------------------------------------------------------------------------*/

double randgen(void)
{
  double randomnum = rand();
  randomnum = fmod(randomnum,1e6);
  randomnum = randomnum/5e5;
  randomnum = randomnum - 1;
  return randomnum;
}

/*------------------------------------------------------------------------*/
/* Generate the Brownian motion term */
/*------------------------------------------------------------------------*/

double *randomizer(double *randvec, double diffusivity, double tstep)
{
  double magnitude = pow((6*diffusivity*tstep),0.5);
  double angle1, angle2;
  angle1 = PI * randgen();
  angle2 = PI * randgen();

  double randx = magnitude * cos(angle1) * sin(angle2);
  double randy = magnitude * sin(angle1) * sin(angle2);
  double randz = magnitude * cos(angle2);

  randvec[0] = randx;
  randvec[1] = randy;
  randvec[2] = randz;

  return randvec;
}
```

A.7. Calculating distance between two points

```
/*------------------------------------------------------------------------*/
/* Calculate distance between two points */
/*------------------------------------------------------------------------*/

double distance(double *loc, double x, double y, double z)
{
    dist_temp = pow((loc[0] - x),2) + pow((loc[1]-y),2) + pow((loc[2]-z),2);
    dist_temp = pow(dist_temp,0.5);
    return dist_temp;
}
```

A.8. Checking tracer proximity to walls and particles

```
/*------------------------------------------------------------------------*/
/* Check if tracer is inside a particle */
/*------------------------------------------------------------------------*/

```

```
5  int particle_checker(double *loc, int *flags) {
6    int flag=0;
7    for (int i=0;i<NPARTS;i++) {
8
9      int flagsp[3];
10     double locp[3];
11     locp[0] = parts[i].x;
12     locp[1] = parts[i].y;
13     locp[2] = parts[i].z;
14
15     periodic_maneuver(locp, flagsp, parts[i].r);
16
17     if (flags[0]*flagsp[0]==-1) {
18       locp[0] = flags[0]*dom[rank].xe + locp[0] - flags[0]*dom[rank].xs;
19     }
20
21     if (flags[1]*flagsp[1]==-1) {
22       locp[1] = flags[1]*dom[rank].ye + locp[1] - flags[1]*dom[rank].ys;
23     }
24
25     if (flags[2]*flagsp[2]==-1) {
26       locp[2] = flags[2]*dom[rank].ze + locp[2] - flags[2]*dom[rank].zs;
27     }
28
29     if (distance(loc, locp[0], locp[1], locp[2]) < parts[i].r) {
30       flag=1;
31       break;
32     }
33   }
34   return flag;
35 }
36
37 /*---------------------------------------------------------------------------*/
38 /* Check if tracer is close to a boundary */
39 /*---------------------------------------------------------------------------*/
40
41 int wall_checker(double *loc)
42 {
43   double min_bound, max_bound;
44   int count = 1;
45
46   min_bound = dom[rank].xs + dom[rank].xl/(2*dom[rank].xn);
47   max_bound = dom[rank].xe - dom[rank].xl/(2*dom[rank].xn);
48
49   if (loc[0] <= min_bound || loc[0] >= max_bound) {
50     count *= 2;
51   }
52
53   min_bound = dom[rank].ys + dom[rank].yl/(2*dom[rank].yn);
54   max_bound = dom[rank].ye - dom[rank].yl/(2*dom[rank].yn);
55
56   if (loc[1] <= min_bound || loc[1] >= max_bound) {
57     count *= 3;
58   }
59
60   min_bound = dom[rank].zs + dom[rank].zl/(2*dom[rank].zn);
61   max_bound = dom[rank].ze - dom[rank].zl/(2*dom[rank].zn);
62
63   if (loc[2] <= min_bound || loc[2] >= max_bound) {
64     count *= 5;
65   }
```

```
66
67    return count;
68  }
```

A.9. Location of bounding box node

```
1   /*--------------------------------------------------------------------------*/
2   /* Find the 1D index of the cell-centered grid given tracer location */
3   /*--------------------------------------------------------------------------*/
4
5   double index_finder(double *loc) {
6     double xi, yi, zi;
7     xi = floor((loc[0] - dom[rank].xs - dom[rank].dx/2)/dom[rank].dx);
8     yi = floor((loc[1] - dom[rank].ys - dom[rank].dy/2)/dom[rank].dy);
9     zi = floor((loc[2] - dom[rank].zs - dom[rank].dz/2)/dom[rank].dz);
10    if (xi<0) { xi =0; }
11    if (yi<0) { yi =0; }
12    if (zi<0) { zi =0; }
13    foundindex = xi + yi * dom[rank].xn
14    + zi * dom[rank].xn * dom[rank].yn;
15    return foundindex;
16  }
17
18  /*--------------------------------------------------------------------------*/
19  /* Find the location of the cell-centered grid given the 1D index */
20  /*--------------------------------------------------------------------------*/
21
22  double grid_finder(double index, int axis, double gridloc) {
23
24    double xxi = fmod(index , (double) dom[rank].xn);
25    double yyi = fmod(((index - xxi)/dom[rank].xn) , (double) (dom[rank].yn));
26    double zzi = ((index - xxi - yyi*dom[rank].xn))/(dom[rank].xn*dom[rank].yn);
27
28    if (axis==0) {
29      gridloc = (xxi+0.5)*dom[rank].dx + dom[rank].xs;
30      return gridloc;
31    }
32
33    else if (axis==1) {
34      gridloc = (yyi+0.5)*dom[rank].dy + dom[rank].ys;
35      return gridloc;
36    }
37
38    else if (axis==2) {
39      gridloc = (zzi+0.5)*dom[rank].dz + dom[rank].zs;
40      return gridloc;
41    }
42
43    else {
44      return 0;
45    }
46  }
```

A.10. Checking tracer exclusion from cell-centered grid

```
1   /*--------------------------------------------------------------------------*/
2   /* Check if tracer is out of a boundary */
3   /*--------------------------------------------------------------------------*/
4
```

```
int domain_checker(double *loc)
{
  double min_bound, max_bound;
  int count = 1;

  min_bound = dom[rank].xs;
  max_bound = dom[rank].xe;

  if (loc[0] < min_bound || loc[0] > max_bound) {
    count *= 2;
  }

  min_bound = dom[rank].ys;
  max_bound = dom[rank].ye;

  if (loc[1] < min_bound || loc[1] > max_bound) {
    count *= 3;
  }

  min_bound = dom[rank].zs;
  max_bound = dom[rank].ze;

  if (loc[2] < min_bound || loc[2] > max_bound) {
    count *= 5;
  }

  return count;
}
```

A.11. Specular reflection of tracers at walls

```
/*--------------------------------------------------------------------------*/
/* Move a tracer if it crosses a wall */
/*--------------------------------------------------------------------------*/

double *wall_reflector(int i, double *loc)
{
  double x = loc[0];
  double y = loc[1];
  double z = loc[2];

  if (x < dom[rank].xs) {
    //x = 2*dom[rank].xs - x;
    x = dom[rank].xe - (dom[rank].xs - x);
    // tracer_array[i].tracertype = 0;
    // printf("Tracer %d reflected xs.\n", i);
  }

  if (x > dom[rank].xe) {

    if (farwalltreatment==0){
    // x = 2*dom[rank].xe - x;
    x = dom[rank].xs + (x - dom[rank].xe);
    // tracer_array[i].tracertype = 1;
    // tracer_array[i].tracertype = 1;
    }
    if (farwalltreatment==1) {
      exporter();
      tracer_delete(i);
    }
```

```
30
31      if (farwalltreatment==2) {
32        tracer_array[i].tracertype = 1;
33      }
34
35      // printf("Tracer %d reflected xe.\n", i);
36    }
37
38    if (y < dom[rank].ys) {
39      y = 2*dom[rank].ys - y;
40      //y = dom[rank].ye - (dom[rank].ys - y);
41      tracer_array[i].tracertype = 0;
42      // printf("Tracer %d reflected ys.\n", i);
43    }
44
45    if (y > dom[rank].ye) {
46      y = 2*dom[rank].ye - y;
47      //y = dom[rank].ys + (y - dom[rank].ye);
48      tracer_array[i].tracertype = 1;
49      // printf("Tracer %d reflected ye.\n", i);
50    }
51
52    if (z < dom[rank].zs) {
53      // z = 2*dom[rank].zs - z;
54      z = dom[rank].ze - (dom[rank].zs - z);
55      // printf("Tracer %d reflected zs.\n", i);
56    }
57
58    if (z > dom[rank].ze) {
59      // z = 2*dom[rank].ze - z;
60      z = dom[rank].zs + (z - dom[rank].ze);
61      // printf("Tracer %d reflected ze.\n", i);
62    }
63
64    loc[0] = x;
65    loc[1] = y;
66    loc[2] = z;
67
68    return loc;
69  }
```

A.12. Allowing interaction with particles across periodic boundaries

```
1  /*------------------------------------------------------------------------*/
2  /* Flag particles and tracers that are close to a boundary */
3  /*------------------------------------------------------------------------*/
4
5  void periodic_maneuver(double *loc, int *flags, double max_a) {
6
7    if (loc[0] < dom[rank].xs + max_a){ flags[0] = -1;  }
8    else if (loc[0] > dom[rank].xe - max_a) { flags[0] = 1;  }
9    else { flags[0] = 0;}
10
11   if (loc[1] < dom[rank].ys + max_a){ flags[1] = -1;  }
12   else if (loc[1] > dom[rank].ye - max_a) { flags[1] = 1;  }
13   else { flags[1] = 0;}
14
15   if (loc[2] < dom[rank].zs + max_a){ flags[2] = -1;  }
16   else if (loc[2] > dom[rank].ze - max_a) { flags[2] = 1;  }
17   else { flags[2] = 0;}
```

```

}
```

A.13. Spatial interpolation of velocity fields

```
/*--------------------------------------------------------------------------*/
/* Find the velocity of the fluid at the position of the tracer */
/*--------------------------------------------------------------------------*/

double *interpolator(double *loc, double *ucc, double *vcc, double *wcc) {

  double min_bound, max_bound;
  double gridloc=0;

  switch (wall_checker(loc)) {
    case 1:
    foundindex = index_finder(loc);
    double c000, c001, c010, c011, c100, c101, c110, c111, cx, cy, cz;

    cx = (loc[0] - grid_finder(foundindex,0, gridloc))/dom[rank].dx;
    cy = (loc[1] - grid_finder(foundindex,1, gridloc))/dom[rank].dy;
    cz = (loc[2] - grid_finder(foundindex,2, gridloc))/dom[rank].dz;

    c000 = ucc[foundindex];
    c001 = ucc[foundindex + 1];
    c010 = ucc[foundindex + dom[rank].xn];
    c011 = ucc[foundindex + dom[rank].xn + 1];
    c100 = ucc[foundindex + dom[rank].xn*dom[rank].yn];
    c101 = ucc[foundindex + dom[rank].xn*dom[rank].yn + 1];
    c110 = ucc[foundindex + dom[rank].xn*dom[rank].yn + dom[rank].xn];
    c111 = ucc[foundindex + dom[rank].xn*dom[rank].yn + dom[rank].xn + 1];

    double tempx1 = (1-cx)*c000 + cx*c001;
    double tempx2 = (1-cx)*c010 + cx*c011;
    double tempx3 = (1-cx)*c100 + cx*c101;
    double tempx4 = (1-cx)*c110 + cx*c111;
    double tempy1 = (1-cy)*tempx1 + cy*tempx2;
    double tempy2 = (1-cy)*tempx3 + cy*tempx4;
    interpvel[0]  = (1-cz)*tempy1 + cz*tempy2;

    c000 = vcc[foundindex];
    c001 = vcc[foundindex + 1];
    c010 = vcc[foundindex + dom[rank].xn];
    c011 = vcc[foundindex + dom[rank].xn + 1];
    c100 = vcc[foundindex + dom[rank].xn*dom[rank].yn];
    c101 = vcc[foundindex + dom[rank].xn*dom[rank].yn + 1];
    c110 = vcc[foundindex + dom[rank].xn*dom[rank].yn + dom[rank].xn];
    c111 = vcc[foundindex + dom[rank].xn*dom[rank].yn + dom[rank].xn + 1];

    tempx1 = (1-cx)*c000 + cx*c001;
    tempx2 = (1-cx)*c010 + cx*c011;
    tempx3 = (1-cx)*c100 + cx*c101;
    tempx4 = (1-cx)*c110 + cx*c111;
    tempy1 = (1-cy)*tempx1 + cy*tempx2;
    tempy2 = (1-cy)*tempx3 + cy*tempx4;
    interpvel[1]  = (1-cz)*tempy1 + cz*tempy2;

    c000 = wcc[foundindex];
    c001 = wcc[foundindex + 1];
    c010 = wcc[foundindex + dom[rank].xn];
```

```
56      c011 = wcc[foundindex + dom[rank].xn + 1];
57      c100 = wcc[foundindex + dom[rank].xn*dom[rank].yn];
58      c101 = wcc[foundindex + dom[rank].xn*dom[rank].yn + 1];
59      c110 = wcc[foundindex + dom[rank].xn*dom[rank].yn + dom[rank].xn];
60      c111 = wcc[foundindex + dom[rank].xn*dom[rank].yn + dom[rank].xn + 1];
61
62      tempx1 = (1-cx)*c000 + cx*c001;
63      tempx2 = (1-cx)*c010 + cx*c011;
64      tempx3 = (1-cx)*c100 + cx*c101;
65      tempx4 = (1-cx)*c110 + cx*c111;
66      tempy1 = (1-cy)*tempx1 + cy*tempx2;
67      tempy2 = (1-cy)*tempx3 + cy*tempx4;
68      interpvel[2]  = (1-cz)*tempy1 + cz*tempy2;
69
70      break;
71
72      case 2:
73
74      min_bound = dom[rank].xs + dom[rank].xl/(2*dom[rank].xn);
75      max_bound = dom[rank].xe - dom[rank].xl/(2*dom[rank].xn);
76
77      if (loc[0] <= min_bound) {
78        foundindex = index_finder(loc);
79        interpvel[0] = ucc[foundindex];
80        interpvel[1] = vcc[foundindex];
81        interpvel[2] = wcc[foundindex];
82      }
83
84      if(loc[0] >= max_bound) {
85        foundindex = index_finder(loc);
86        interpvel[0] = ucc[foundindex];
87        interpvel[1] = vcc[foundindex];
88        interpvel[2] = wcc[foundindex];
89      }
90      break;
91
92      case 3:
93
94      min_bound = dom[rank].ys + dom[rank].yl/(2*dom[rank].yn);
95      max_bound = dom[rank].ye - dom[rank].yl/(2*dom[rank].yn);
96
97      if (loc[0] <= min_bound) {
98        foundindex = index_finder(loc);
99        interpvel[0] = ucc[foundindex];
100       interpvel[1] = vcc[foundindex];
101       interpvel[2] = wcc[foundindex];
102     }
103
104     if(loc[0] >= max_bound) {
105       foundindex = index_finder(loc);
106       interpvel[0] = ucc[foundindex];
107       interpvel[1] = vcc[foundindex];
108       interpvel[2] = wcc[foundindex];
109     }
110     break;
111
112     case 5:
113
114     min_bound = dom[rank].zs + dom[rank].zl/(2*dom[rank].zn);
115     max_bound = dom[rank].ze - dom[rank].zl/(2*dom[rank].zn);
116
```

```
117       if (loc[0] <= min_bound) {
118         foundindex = index_finder(loc);
119         interpvel[0] = ucc[foundindex];
120         interpvel[1] = vcc[foundindex];
121         interpvel[2] = wcc[foundindex];
122       }
123
124       if(loc[0] >= max_bound) {
125         foundindex = index_finder(loc);
126         interpvel[0] = ucc[foundindex];
127         interpvel[1] = vcc[foundindex];
128         interpvel[2] = wcc[foundindex];
129       }
130       break;
131
132       case 6:
133       case 10:
134       case 15:
135       case 30:
136       foundindex = index_finder(loc);
137       interpvel[0] = ucc[foundindex];
138       interpvel[1] = vcc[foundindex];
139       interpvel[2] = wcc[foundindex];
140       break;
141     }
142
143     return interpvel;
144   }
```

A.14. Specular reflection of tracer on particle surface

```
1   /*--------------------------------------------------------------------------*/
2   /* Reflect a tracer on the surface of a particle */
3   /*--------------------------------------------------------------------------*/
4
5   double *reflector(double *loc1, double *loc2, double x,
6     double y, double z, double r, int *flags)
7   {
8
9     int flagsp[3];
10    double locp[3];
11    locp[0] = x;
12    locp[1] = y;
13    locp[2] = z;
14
15    periodic_maneuver(locp, flagsp, r);
16
17    if (flags[0]*flagsp[0]==-1) {
18      x = flags[0]*dom[rank].xe + x - flags[0]*dom[rank].xs;
19    }
20
21    if (flags[1]*flagsp[1]==-1) {
22      y = flags[1]*dom[rank].ye + y - flags[1]*dom[rank].ys;
23    }
24
25    if (flags[2]*flagsp[2]==-1) {
26      z = flags[2]*dom[rank].ze + z - flags[2]*dom[rank].zs;
27    }
28
29    if (distance(loc2, x, y, z) >= r) {
```

```
30      return loc2;
31    }
32
33    else {
34
35      double deltax = loc2[0] - loc1[0];
36      double deltay = loc2[1] - loc1[1];
37      double deltaz = loc2[2] - loc1[2];
38
39      double delta = 10;
40
41      double lucy[3];
42      double lucy_copy[3];
43      lucy[0] = loc1[0];
44      lucy[1] = loc1[1];
45      lucy[2] = loc1[2];
46
47      while (delta<100000){
48
49        lucy_copy[0] = lucy[0];
50        lucy_copy[1] = lucy[1];
51        lucy_copy[2] = lucy[2];
52
53        lucy[0] += deltax/delta;
54        lucy[1] += deltay/delta;
55        lucy[2] += deltaz/delta;
56
57        if (distance(lucy, x, y, z) <= r) {
58          lucy[0] = lucy_copy[0];
59          lucy[1] = lucy_copy[1];
60          lucy[2] = lucy_copy[2];
61          delta *= 10;
62        }
63      }
64
65      double dsmag = distance(loc2, lucy[0],lucy[1],lucy[2]);
66      double dnmag = distance(lucy, x, y, z);
67
68      double locp[3];
69      locp[0] = x;
70      locp[1] = y;
71      locp[2] = z;
72
73      double di[3], dn[3], dotvec;
74      dotvec = 0;
75
76      for (int i=0;i<3;i++) {
77        di[i] = (loc2[i] - lucy[i])/dsmag;
78        dn[i] = (lucy[i] - locp[i])/dnmag;
79        dotvec += di[i]*dn[i];
80      }
81
82      for (int i=0; i<3; i++) {
83        locx[i] = (di[i] - 2*dotvec*dn[i])*dsmag + lucy[i];
84      }
85
86      return locx;
87    }
88  }
89
90  double *intersector(double *loc1, double *loc2, double x,
```

```
91    double y, double z, double r, int *flags) {
92
93    int flagsp[3];
94    double locp[3];
95    locp[0] = x;
96    locp[1] = y;
97    locp[2] = z;
98
99    periodic_maneuver(locp, flagsp, r);
100
101   if (flags[0]*flagsp[0]==-1) {
102     x = flags[0]*dom[rank].xe + x - flags[0]*dom[rank].xs;
103   }
104
105   if (flags[1]*flagsp[1]==-1) {
106     y = flags[1]*dom[rank].ye + y - flags[1]*dom[rank].ys;
107   }
108
109   if (flags[2]*flagsp[2]==-1) {
110     z = flags[2]*dom[rank].ze + z - flags[2]*dom[rank].zs;
111   }
112
113   if (distance(loc2,x,y,z)>=r) {
114     return loc1;
115   }
116
117   else {
118
119   double deltax = loc2[0] - loc1[0];
120   double deltay = loc2[1] - loc1[1];
121   double deltaz = loc2[2] - loc1[2];
122
123   double delta = 10;
124
125   double lucy[3];
126   double lucy_copy[3];
127   lucy[0] = loc1[0];
128   lucy[1] = loc1[1];
129   lucy[2] = loc1[2];
130
131   while (delta<100000){
132
133     lucy_copy[0] = lucy[0];
134     lucy_copy[1] = lucy[1];
135     lucy_copy[2] = lucy[2];
136
137     lucy[0] += deltax/delta;
138     lucy[1] += deltay/delta;
139     lucy[2] += deltaz/delta;
140
141     if (distance(lucy, x, y, z) <= r) {
142       lucy[0] = lucy_copy[0];
143       lucy[1] = lucy_copy[1];
144       lucy[2] = lucy_copy[2];
145       delta *= 10;
146     }
147   }
148
149   locx[0] = lucy[0];
150   locx[1] = lucy[1];
151   locx[2] = lucy[2];
```

```
152
153   return locx;
154   }
155 }
```

A.15. Pushing tracers out of a particle

```
1  /*--------------------------------------------------------------------------*/
2  /* Push tracer radially out of a particle before start */
3  /*--------------------------------------------------------------------------*/
4
5  double *tracer_pusher(double *loc, double x, double y, double z, double r, int *flags)
6  {
7    int flagsp[3];
8    double locp[3];
9    locp[0] = x;
10   locp[1] = y;
11   locp[2] = z;
12
13   periodic_maneuver(locp, flagsp, r);
14
15   if (flags[0]*flagsp[0]==-1) {
16     x = flags[0]*dom[rank].xe + x - flags[0]*dom[rank].xs;
17   }
18
19   if (flags[1]*flagsp[1]==-1) {
20     y = flags[1]*dom[rank].ye + y - flags[1]*dom[rank].ys;
21   }
22
23   if (flags[2]*flagsp[2]==-1) {
24     z = flags[2]*dom[rank].ze + z - flags[2]*dom[rank].zs;
25   }
26
27   if (distance(loc, x, y, z) >= r) {
28     return loc;
29   }
30   else {
31     // printf("Tracer is inside a particle.\n");
32     // printf("Location of particle = (%f, %f, %f).\n",x,y,z);
33     // printf("Location of tracer before = (%f, %f, %f).\n",loc[0],loc[1],loc[2]);
34     // printf("Distance = %f, r=%f.",distance(loc,x,y,z),r);
35
36     // for (int i=0; i<3; i++) {
37     //   printf("%f, ", loc[i]);
38     // }
39
40     double locp[3];
41
42     locp[0] = x;
43     locp[1] = y;
44     locp[2] = z;
45
46     double dn[3];
47     double dnmag = distance(loc,x,y,z);
48
49     for (int i=0; i<3; i++) {
50       dn[i] = (loc[i] - locp[i])/dnmag;
51       locx[i] = loc[i] + 2*(r-dnmag)*dn[i];
52     }
53
```

```
54      // printf("Location of tracer after = (%f, %f, %f).\n",locx[0],locx[1],locx[2]);
55      // printf("Distance = %f, r=%f.",distance(loc,x,y,z),r);
56      if (distance(locx, x, y, z) < r) {
57        printf("Tracer is still inside a particle.\n");
58      }
59
60      return locx;
61    }
62  }
```

A.16. Main function

```
1   /*---------------------------------------------------------------------------*/
2   /* Main simulation */
3   /*---------------------------------------------------------------------------*/
4
5   void tracer_execute(double dt)
6   {
7
8     tracercheck++;
9
10    double randvec[3];
11    double *correction1;
12    double *correction2;
13    double *correction3;
14    double *correction4;
15    double *correction5;
16    double *correction6;
17    double correctionr[3];
18    double loc_t[3];
19    int flags[3];
20
21    if (steady_flag[0]==0 || steady_flag[1]!=0) {
22      steady_flag[0]=1;
23      cuda_dom_pull();
24      cuda_part_pull();
25
26      ucc = (real*) malloc(dom[rank].Gcc.s3 * sizeof(real));
27      vcc = (real*) malloc(dom[rank].Gcc.s3 * sizeof(real));
28      wcc = (real*) malloc(dom[rank].Gcc.s3 * sizeof(real));
29
30      for (int k = dom[rank].Gcc._ks; k <= dom[rank].Gcc._ke; k++) {
31        for (int j = dom[rank].Gcc._js; j <= dom[rank].Gcc._je; j++) {
32          for (int i = dom[rank].Gcc._is; i <= dom[rank].Gcc._ie; i++) {
33
34            int C = GCC_LOC(i - DOM_BUF, j - DOM_BUF, k - DOM_BUF,
35                            dom[rank].Gcc.s1, dom[rank].Gcc.s2);
36
37            int Cfx_w = GFX_LOC(i - 1, j, k, dom[rank].Gfx.s1b, dom[rank].Gfx.s2b);
38            int Cfx = GFX_LOC(i, j, k, dom[rank].Gfx.s1b, dom[rank].Gfx.s2b);
39            int Cfx_e = GFX_LOC(i + 1, j, k, dom[rank].Gfx.s1b, dom[rank].Gfx.s2b);
40            int Cfx_ee = GFX_LOC(i + 2, j, k, dom[rank].Gfx.s1b, dom[rank].Gfx.s2b);
41
42            int Cfy_s =  GFY_LOC(i, j - 1, k, dom[rank].Gfy.s1b, dom[rank].Gfy.s2b);
43            int Cfy = GFY_LOC(i, j, k, dom[rank].Gfy.s1b, dom[rank].Gfy.s2b);
44            int Cfy_n =  GFY_LOC(i, j + 1, k, dom[rank].Gfy.s1b, dom[rank].Gfy.s2b);
45            int Cfy_nn = GFY_LOC(i, j + 2, k, dom[rank].Gfy.s1b, dom[rank].Gfy.s2b);
46
47            int Cfz_b = GFZ_LOC(i, j, k - 1, dom[rank].Gfz.s1b, dom[rank].Gfz.s2b);
48            int Cfz = GFZ_LOC(i, j, k, dom[rank].Gfz.s1b, dom[rank].Gfz.s2b);
```

```
49            int Cfz_t = GFZ_LOC(i, j, k + 1, dom[rank].Gfz.s1b, dom[rank].Gfz.s2b);
50            int Cfz_tt = GFZ_LOC(i, j, k + 2, dom[rank].Gfz.s1b, dom[rank].Gfz.s2b);
51
52            ucc[C] = -0.0625*u[Cfx_w] + 0.5625*u[Cfx] + 0.5625*u[Cfx_e]
53                     -0.0625*u[Cfx_ee];
54            vcc[C] = -0.0625*v[Cfy_s] + 0.5625*v[Cfy] + 0.5625*v[Cfy_n]
55                     -0.0625*v[Cfy_nn];
56            wcc[C] = -0.0625*w[Cfz_b] + 0.5625*w[Cfz] + 0.5625*w[Cfz_t]
57                     -0.0625*w[Cfz_tt];
58          }
59        }
60      }
61    }
62
63    // FILE *debugcsv;
64    // char debugname[100];
65    // sprintf(debugname, "%s/%s/%s/tracer-%d.csv", ROOT_DIR, OUTPUT_DIR, DEBUG_DIR,
66    //          exportcount);
67    // debugcsv = fopen(debugname,"w+");
68
69    for (int i=0; i<tracernum; i++) {
70
71      if (farwalltreatment<2 || tracer_array[i].tracertype==0) {
72
73        // fprintf(debugcsv, "%d, %f, %f, %f\n", i, tracer_array[i].pos[0],
74        //       tracer_array[i].pos[1],tracer_array[i].pos[2]);
75
76
77        // 1. Tracer pusher
78
79        periodic_maneuver(tracer_array[i].pos, flags, max_radius);
80
81        while (particle_checker(tracer_array[i].pos, flags)==1) {
82          for (int j=0; j<NPARTS; j++) {
83            correction1 = tracer_pusher(tracer_array[i].pos,parts[j].x,
84              parts[j].y, parts[j].z, parts[j].r, flags);
85
86            for (int k=0; k<3; k++) {
87              tracer_array[i].pos[k] = correction1[k];
88            }
89          }
90
91          correction5 = wall_reflector(i,tracer_array[i].pos);
92
93          for (int j=0; j<3; j++) {
94            tracer_array[i].pos[j] = correction5[j];
95          }
96
97        }
98
99        // fprintf(debugcsv, "%f, %f, %f\n", tracer_array[i].pos[0],
100       //       tracer_array[i].pos[1],tracer_array[i].pos[2]);
101
102       // 2. Randomizer
103       correction2 = randomizer(randvec, diffusivity, dt);
104
105       for (int j=0; j<3; j++) {
106         loc_t[j] = tracer_array[i].pos[j] + correction2[j];
107       }
108
109       // fprintf(debugcsv, "%f, %f, %f\n", loc_t[0], loc_t[1],loc_t[2]);
```

```
110
111         // 3. Interpolator
112         correction3 = interpolator(tracer_array[i].pos, ucc, vcc, wcc);
113
114         for (int j=0; j<3; j++) {
115           loc_t[j] += correction3[j]*dt;
116         }
117
118         // fprintf(debugcsv, "%f, %f, %f\n", loc_t[0], loc_t[1],loc_t[2]);
119
120
121         for (int k=0;k<3;k++) {
122            correctionr[k] = tracer_array[i].pos[k];
123         }
124
125         // 4. Sphere reflector
126
127         {
128
129         periodic_maneuver(loc_t, flags, max_radius);
130
131         int stepcount=0;
132
133         while(particle_checker(loc_t, flags)==1) {
134
135           stepcount++;
136
137           if (stepcount>50) {
138             break;
139           }
140
141           for (int j=0; j<NPARTS; j++) {
142
143             correction4 = reflector(correctionr, loc_t, parts[j].x, parts[j].y,
144               parts[j].z, parts[j].r, flags);
145
146             correction6 = intersector(correctionr,loc_t, parts[j].x, parts[j].y,
147               parts[j].z, parts[j].r, flags);
148
149             for (int k=0; k<3; k++) {
150               correctionr[k] = correction6[k];
151               loc_t[k] = correction4[k];
152             }
153           }
154         }
155
156         for (int j=0;j<3;j++) {
157           tracer_array[i].pos[j] = loc_t[j];
158         }
159
160         // fprintf(debugcsv, "%d, %d\n", i, stepcount);
161
162         }
163
164         // fprintf(debugcsv, "%f, %f, %f\n", tracer_array[i].pos[0],
165         //       tracer_array[i].pos[1],tracer_array[i].pos[2]);
166
167         //5. Wall reflector
168         correction5 = wall_reflector(i,tracer_array[i].pos);
169
170         for (int j=0; j<3; j++) {
```

```
171           tracer_array[i].pos[j] = correction5[j];
172         }
173
174         // fprintf(debugcsv, "%f, %f, %f\n\n", tracer_array[i].pos[0],
175         //       tracer_array[i].pos[1],tracer_array[i].pos[2]);
176
177       }
178     }
179
180     if (tracertreatment==2) {
181           ratekeeper();
182     }
183
184     if (tracercheck%exportfreq==0 && ttime<1200) {
185       exportcount++;
186       exporter();
187     }
188
189     if (tracercheck%1==0 && ttime>=1200) {
190       exportcount++;
191       exporter();
192       printf("Time=%.2f, n=%ld.\n",ttime,tracercheck);
193     }
194
195     fclose(debugcsv);
196     // exporter();
197
198      if (steady_flag[1]!=0) {
199       free(ucc);
200       free(vcc);
201       free(wcc);
202      }
203   }
```

A.17. Exporting tracer & particle positions

```
1   /*--------------------------------------------------------------------------*/
2   /* Export tracer and particle data as csv */
3   /*--------------------------------------------------------------------------*/
4
5   void exporter(void)
6   {
7     FILE *tracercsv;
8     char filename[100];
9     sprintf(filename, "%s/%s/%s/tracer-%d.csv", ROOT_DIR, OUTPUT_DIR, TRACER_DIR,
10            exportcount);
11    tracercsv = fopen(filename,"w+");
12    for (int i=0; i<tracernum; i++) {
13      fprintf(tracercsv, "%d, %e, %e, %e, %d\n", tracer_array[i].tracerid,
14      tracer_array[i].pos[0], tracer_array[i].pos[1], tracer_array[i].pos[2],
15      tracer_array[i].tracertype);
16    }
17    fclose(tracercsv);
18
19    FILE *particleinfocsv;
20    char particlename[100];
21    sprintf(particlename, "%s/%s/%s/particle-%d.csv",ROOT_DIR, OUTPUT_DIR,
22            PARTICLEINFO_DIR, exportcount);
23    particleinfocsv = fopen(particlename,"w+");
24    for (int i=0; i<NPARTS; i++) {
```

```
25      fprintf(particleinfocsv, "%d, %e, %e, %e, %e\n", parts[i].N, parts[i].x,
26              parts[i].y, parts[i].z, parts[i].r);
27    }
28    fclose(particleinfocsv);
29
30    FILE *runinfo;
31    char runname[100];
32    sprintf(runname, "%s/%s/tracerinfo.csv", ROOT_DIR, OUTPUT_DIR);
33    runinfo = fopen(runname,"a+");
34    fprintf(runinfo, "%d, %e, %e, %d\n", exportcount, ttime, dt, nparts);
35    fclose(runinfo);
36
37  }
```

B

Pre-processing & Post-processing

B.1. Case setup - Stagnant suspensions

For the case of stagnant suspensions, since the flow field is quiescent, the flow parameters can be assumed to be arbitrary. The important parameters that are relevant to these cases are the domain size, the number of particles, the total simulation time and the time-step size.

Four values of particle volume fraction are to be simulated:

$$\phi = \{0, 0.1, 0.2\} \tag{B.1}$$

The particle radius is assumed to be $a = 0.1$ m and the domain length is assumed to be $10a$ in all directions. Thus, the domain size used in the simulations is $1 \times 1 \times 1$ m. The mesh properties are not relevant for this case as the *tracer* module does not require a Cartesian mesh if flow field interpolation is not required.

The number of particles required for each value of ϕ are:

$$n_p = \phi \times \frac{V_d}{V_p} \tag{B.2}$$

where V_d is the domain volume and V_p is the particle volume. Substituting the values for the domain volume and particle volume, we get:

$$n_p \approx \{0, 24, 48\} \text{ particles} \tag{B.3}$$

where the brackets indicate the number of particles for the three values of ϕ respectively.

Two values of diffusivity are simulated for each value of ϕ:

$$D = \{1e{-}4, 1e{-}3\} \text{ m}^2/\text{s} \tag{B.4}$$

The total simulation time is assumed to be 5 times the time-scale required for the tracer to cross the domain. Hence, the total simulation time is:

$$t = 5\frac{H^2}{D} = \{50000, 5000\}\, s \tag{B.5}$$

The time-step for these cases are determined by assuming the magnitude of the Brownian displacement term $\sqrt{6D\Delta t}$ to be a tenth of the particle radius a. Hence, we have the time-step size as:

$$\Delta t = \frac{\Delta x^2}{6D} = \frac{a^2}{600D} = \{0.1666, 0.0166\} \tag{B.6}$$

The brackets in Equations B.5 and B.6 indicate the respective values for the two values of diffusivity D.

B.2. Case setup - Sheared suspensions

For the case of sheared suspensions, the Sh vs. Pe at fixed Re by Wang et al. [57] is to be validated. For this case, the Reynolds number is fixed at:

$$\mathrm{Re} = 4 \tag{B.7}$$

The Péclet number is varied as:

$$\mathrm{Pe} = \{100, 200, 300\} \tag{B.8}$$

Also given by Wang et al. is the ratio of domain size to the particle diameter:

$$H/a = 10 \tag{B.9}$$

For the simulations, the particle is assumed to have a radii of $a = 0.1$m. Hence, we have the domain size in the mass transport measurement direction (Y-direction) as $H = 1$m. The domain sizes in the X- and Y-directions are defined to be 8 times the particle diameter. Hence, the domain size to be used in the simulations is $0.8 \times 1 \times 0.8$. The mesh in the Y-direction is assumed to have a resolution of 125. A further assumption of a uniform mesh gives the mesh resolution as $100 \times 125 \times 100$.

The diffusivity is user-defined as:

$$D = 1\mathrm{e}{-4}\ \mathrm{m}^2/\mathrm{s} \tag{B.10}$$

Hence, the shear Γ can be determined from Equation 5.7 as:

$$\Gamma = \frac{\mathrm{Pe}\ D}{a^2} = \{1, 2, 3\} \tag{B.11}$$

Thus, the value for the shear velocity of the -Y and +Y walls are given by:

$$\mathbf{u} = \begin{bmatrix} \pm\dfrac{\Gamma H}{2} \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} \pm\{0.5, 1, 1.5\} \\ 0 \\ 0 \end{bmatrix} \tag{B.12}$$

The values for the kinematic viscosity of the fluid is given from the Reynolds number as:

$$\nu = \frac{\Gamma a^2}{\mathrm{Re}} = \{0.0025, 0.005, 0.0075\}\ \mathrm{m}^2/\mathrm{s} \tag{B.13}$$

For the particular validation case, the particle volume fraction is fixed at:

$$\phi = 0.1 \tag{B.14}$$

Hence, we calculate the number of particles required as:

$$n_p = \phi \times \frac{V_d}{V_p} \tag{B.15}$$

where V_d is the domain volume and V_p is the particle volume. Substituting the values for the domain volume and particle volume, we get:

$$n_p \approx 15 \text{ particles} \tag{B.16}$$

The simulation time recommended by Wang et al. [57] is given by:

$$t = \frac{850}{\Gamma} = \{850, 425, 283.33\} \text{ s} \tag{B.17}$$

The values in the brackets in Equations B.11, B.12, B.13 and B.17 indicate the respective values for Pe = $\{100, 200, 300\}$.

B.3. Estimation of Sherwood number

The Sherwood number Sh is calculated from Equation 5.6 cumulatively using the following Python script.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import os, csv

dir_path = os.path.dirname(os.path.dirname(os.path.realpath(__file__)))
dir_path = dir_path + "/sim/output/"
img_dir = dir_path + 'images'
if not os.path.exists(img_dir):
  os.makedirs(img_dir)

tracerpath = dir_path + "tracer/"
particlepath = dir_path + "particleinfo/"

ntracers = len(np.array(pd.read_csv(tracerpath + "tracer-0.csv",header=None)))

timedata = np.array(pd.read_csv(dir_path + "tracerinfo.csv",header=None))
n = len(timedata)

extents = np.array(pd.read_csv(dir_path+'domaindata.csv',header=None))[0]

wallarea = (extents[1]-extents[0])*(extents[5]-extents[4])
domvol = (extents[1]-extents[0])*(extents[3]-extents[2])*(extents[5]-extents[4])
concforc = ntracers/(domvol)
H = (extents[3]-extents[2])
D = 1e-4

sherwoodlog=[]
xcount = []
countcu = 0

for i in range(10000,n-1):
    xcount.append(timedata[i,1])
```

```
    data1 = np.array(pd.read_csv(tracerpath + "tracer-%d.csv" %i,header=None))
    data2 = np.array(pd.read_csv(tracerpath + "tracer-%d.csv" %(i+1), header=None))

    for j in range(ntracers):
        type_now = data1[j,4]
        type_nex = data2[j,4]

        if type_now==0 and type_nex==1:
            countcu += 0.5

        if type_now==1 and type_nex==0:
            countcu += 0.5

    massflux = countcu/(xcount[-1] - xcount[0])
    h = massflux/(wallarea*concforc)
    sherwood = h*H/D

    if i%100 == 0:
        print(i,"/",n, "\tTime=%.4f" %timedata[i,1],"\tSh=%.4f" %sherwood,\
        "\tCount=%d" %(countcu*2))

header = ["Cumulative Count", "Time of Count", "Mass Flux", "Wall Area",\
"Number of Tracers", "Domain Volume",
           "Driving Force", "h", "H", "D", "Sherwood"]
data = [countcu, xcount[-1] - xcount[0], massflux, wallarea, ntracers, domvol, \
concforc, h, H, D, sherwood]

with open(img_dir+'/sherwood.csv', 'w', encoding='UTF8', newline='') as f:
    writer = csv.writer(f)
    writer.writerow(header)
    writer.writerow(data)
```

B.4. Determination of tracer concentration field

The tracer concentration field is determined from the tracer locations through binning. The following Python code is used to generate a concentration field with the same mesh resolution as the scalar field.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import os,sys
from mpl_toolkits.axes_grid1 import make_axes_locatable

req = float(sys.argv[1])

dir_path = os.path.dirname(os.path.dirname(os.path.dirname(\
    os.path.realpath(__file__))))
dir_path = dir_path + "/sim/output/"

img_dir = dir_path + 'images/'
if not os.path.exists(img_dir):
  os.makedirs(img_dir)

tracerpath = dir_path + 'tracer/'
```

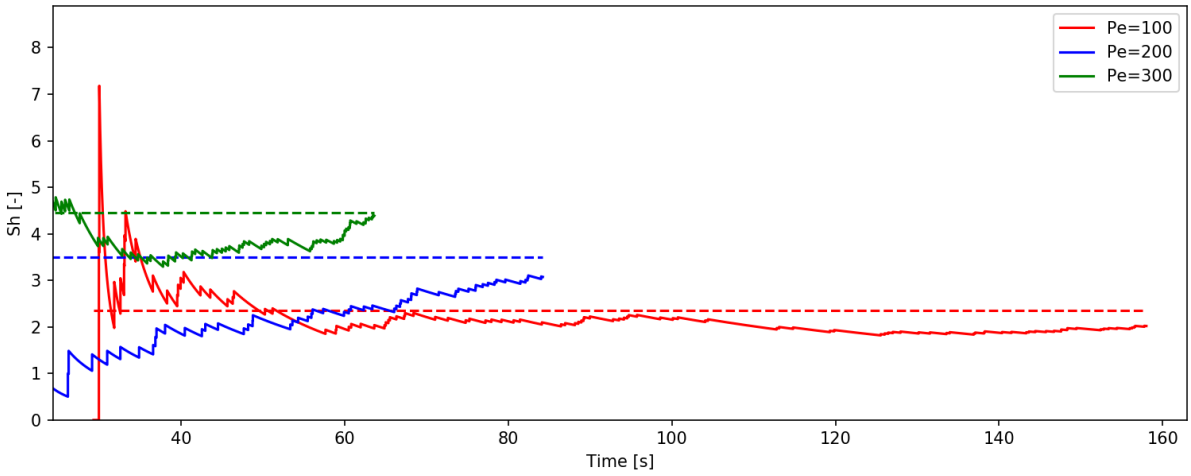
```
18 particlepath = dir_path + 'particleinfo/'
19
20 req = float(sys.argv[1])
21 tracerinfo = np.array(pd.read_csv(dir_path+'tracerinfo.csv',header=None))
22 times = tracerinfo[:,1]
23 timesclose = np.abs(times - req)
24 index = np.argmin(timesclose)
25
26 tracerarray = np.array(pd.read_csv(tracerpath+'tracer-%d.csv' %index, header=None))
27 tracernumbers = int(tracerarray[:,0][-1] + 1)
28 tracer_locx = tracerarray[:,1]
29 tracer_locy = tracerarray[:,2]
30 tracer_locz = tracerarray[:,3]
31 extents = np.array(pd.read_csv(dir_path+'domaindata.csv',header=None))[0]
32
33 particlearray = np.array(pd.read_csv(particlepath+'particle-%d.csv' %index, \
34     header=None))
35 p1x = particlearray[0,1]
36 p1y = particlearray[0,2]
37 p2x = particlearray[1,1]
38 p2y = particlearray[1,2]
39 c1x = 0.1*np.sin(np.linspace(0,2*np.pi,num=100)) + p1x
40 c1y = 0.1*np.cos(np.linspace(0,2*np.pi,num=100)) + p1y
41 c2x = 0.1*np.sin(np.linspace(0,2*np.pi,num=100)) + p2x
42 c2y = 0.1*np.cos(np.linspace(0,2*np.pi,num=100)) + p2y
43
44 n = 128
45 conc_array=np.zeros((n,n))
46 dx = 1/n
47
48 for i in range(tracernumbers):
49   if np.abs(tracer_locz[i])<64*dx:
50     xindex = int((tracer_locx[i] + 0.5)/dx)
51     yindex = int((tracer_locy[i] + 0.5)/dx)
52     conc_array[xindex,yindex] +=1
53
54 conc_array = conc_array.T
55 conc_array = conc_array / np.amax(conc_array)
56
57 plt.figure(num=1,figsize=(5,5),dpi=150)
58 ax=plt.gca()
59 im = ax.imshow(conc_array,cmap='Reds',vmin=0,extent=extents[:4], alpha=1)
60 plt.plot(c1x,c1y,'k--',c2x,c2y,'k--',linewidth=1)
61 plt.xlabel('X')
62 plt.ylabel('Y')
63 divider = make_axes_locatable(ax)
64 cax = divider.append_axes("right", size="5%", pad=0.05)
65 plt.colorbar(im, cax=cax)
66 plt.savefig(img_dir+'tracer-%.2f.png' %req)
```

C

Additional simulation results

C.1. Convergence of Sherwood number in sheared suspensions

The figure below show the convergence of Sherwood number Sh in simulations of sheared suspensions at Pe = {100, 200, 300}. The dotted lines indicate the expected Sh number from the theoretical model proposed by Wang et al. [57].



C.2. Time evolution of scalar field and tracer concentration field

C.2.1. Collision of a spherical particle on a wall of tracers and scalar field

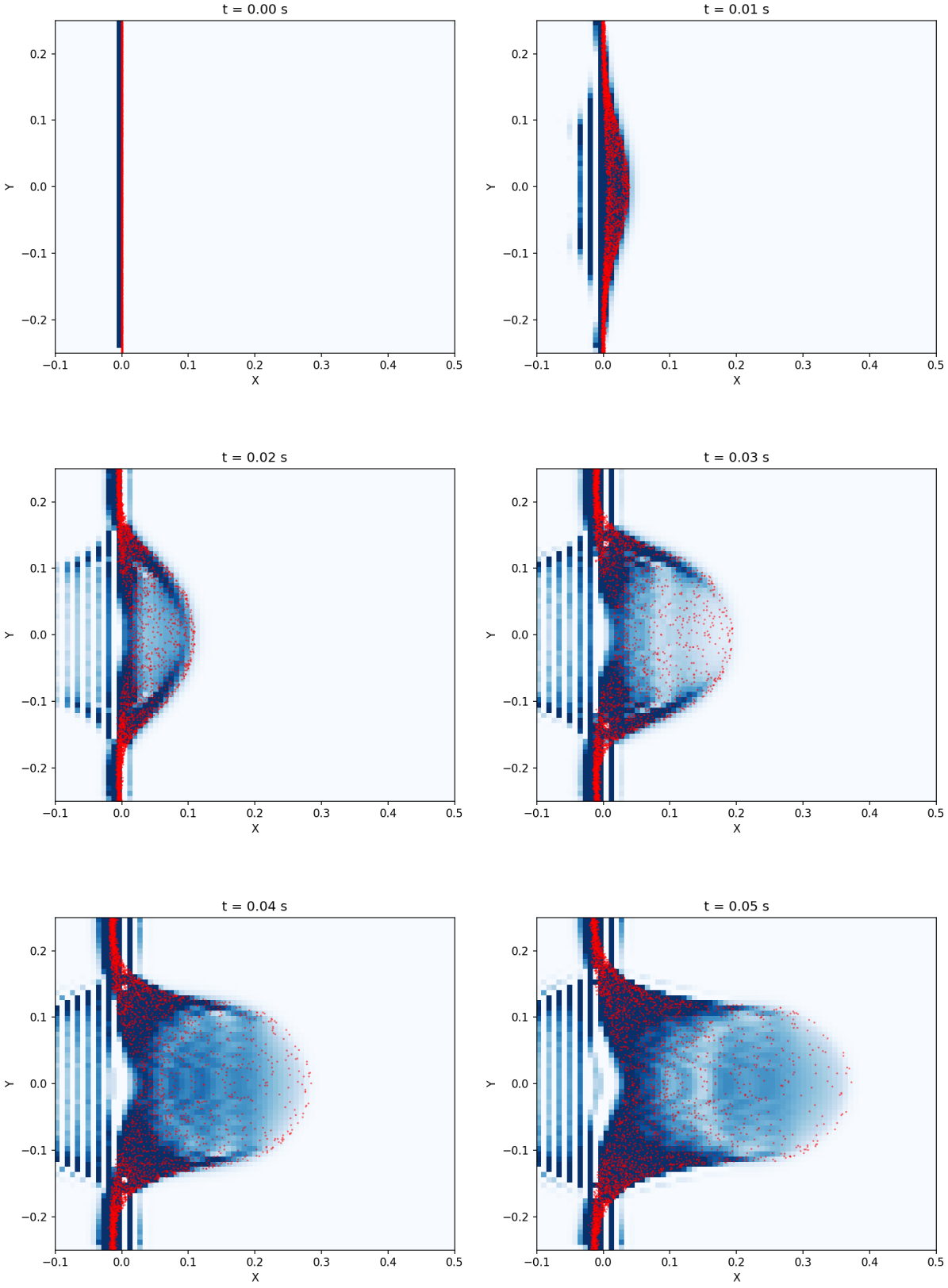
The images below show the visualisation of the collision of a spherical particle on a wall of tracers and a scalar field region of value 1. The scalar concentration field (non-smoothened) is shown in blue and the tracer locations are shown in red. The tracers and the scalar field is filtered to show values within 20 mesh cells from the center of the domain in the Z-direction. The scalar field is also normalized with the maximum value in the filtered region. The colorbar is omitted from the plots as the plots serve to validate the time evolution of tracers and the scalar field qualitatively.

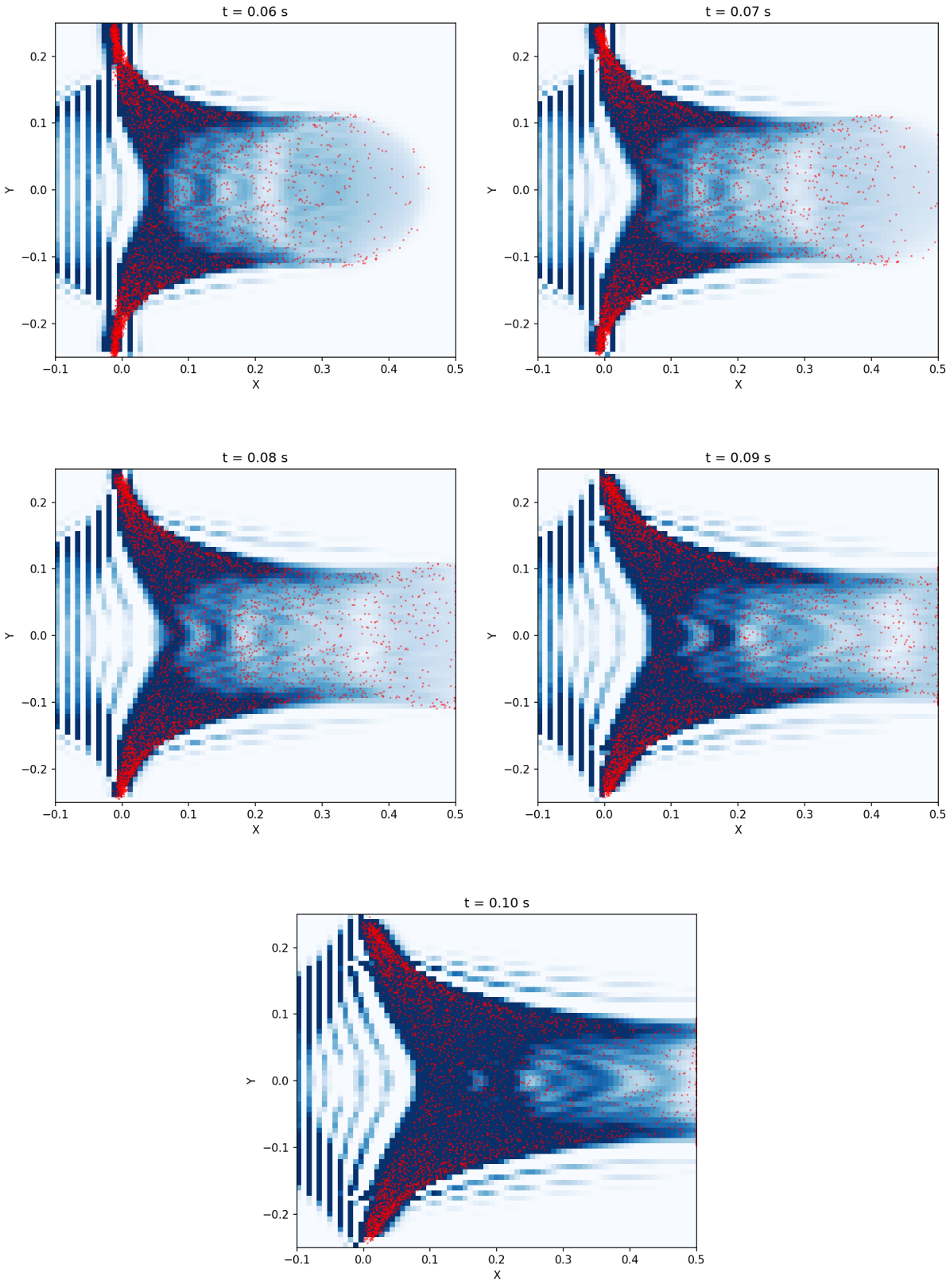
Some of the remarkable observations in the following plots are:

- Numerical artefacts in the scalar field to the left of the initialisation region. This is hypothesised to be caused by the sharp gradient in the value of the scalar and also the low diffusivity

value preventing diffusion and subsequent reduction of the sharp gradient.

- The extents of the tracer locations matches the profile of the scalar field very accurately.





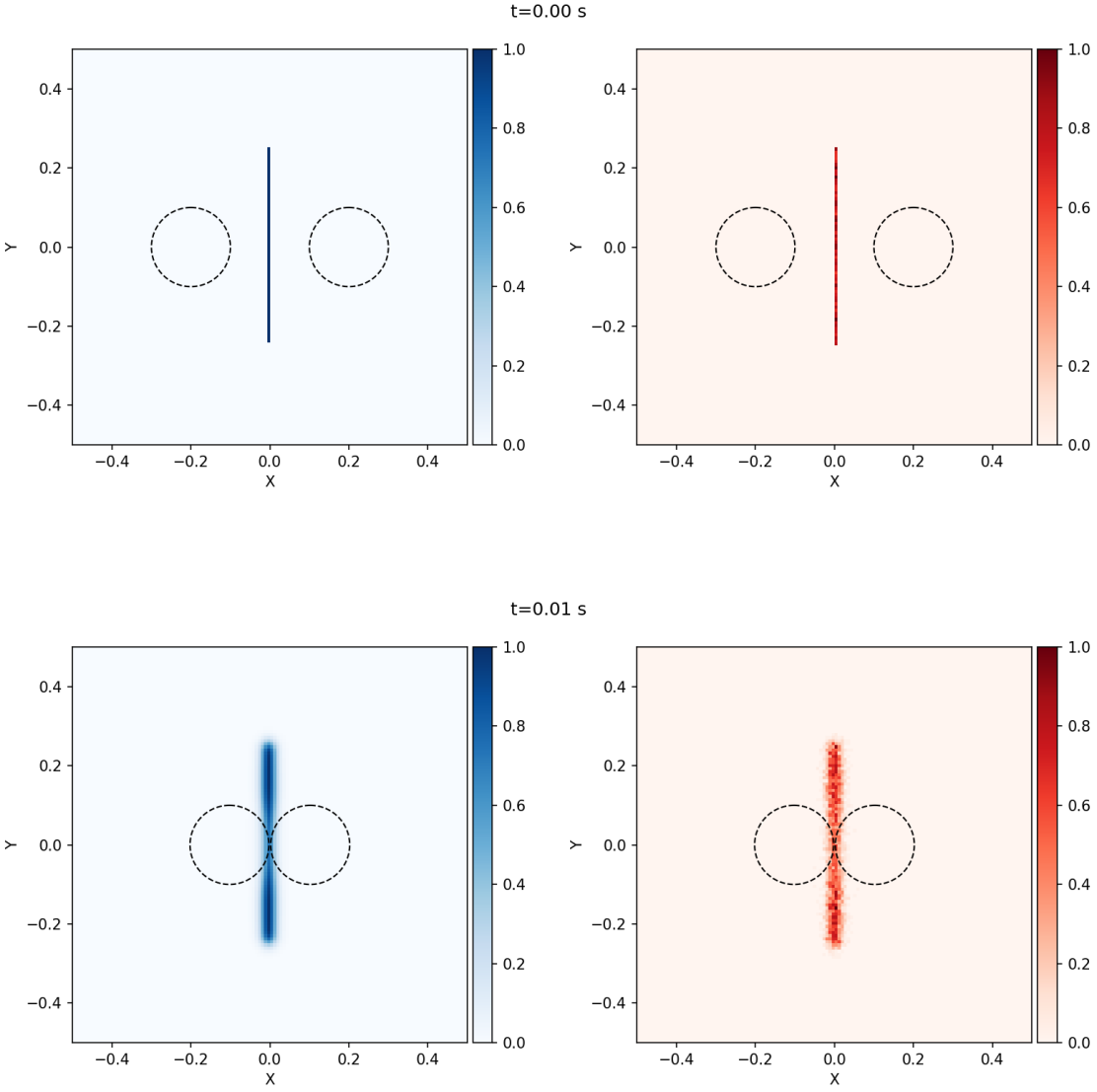
C.2.2. Collision of two spherical particles on wall of tracers and a scalar field

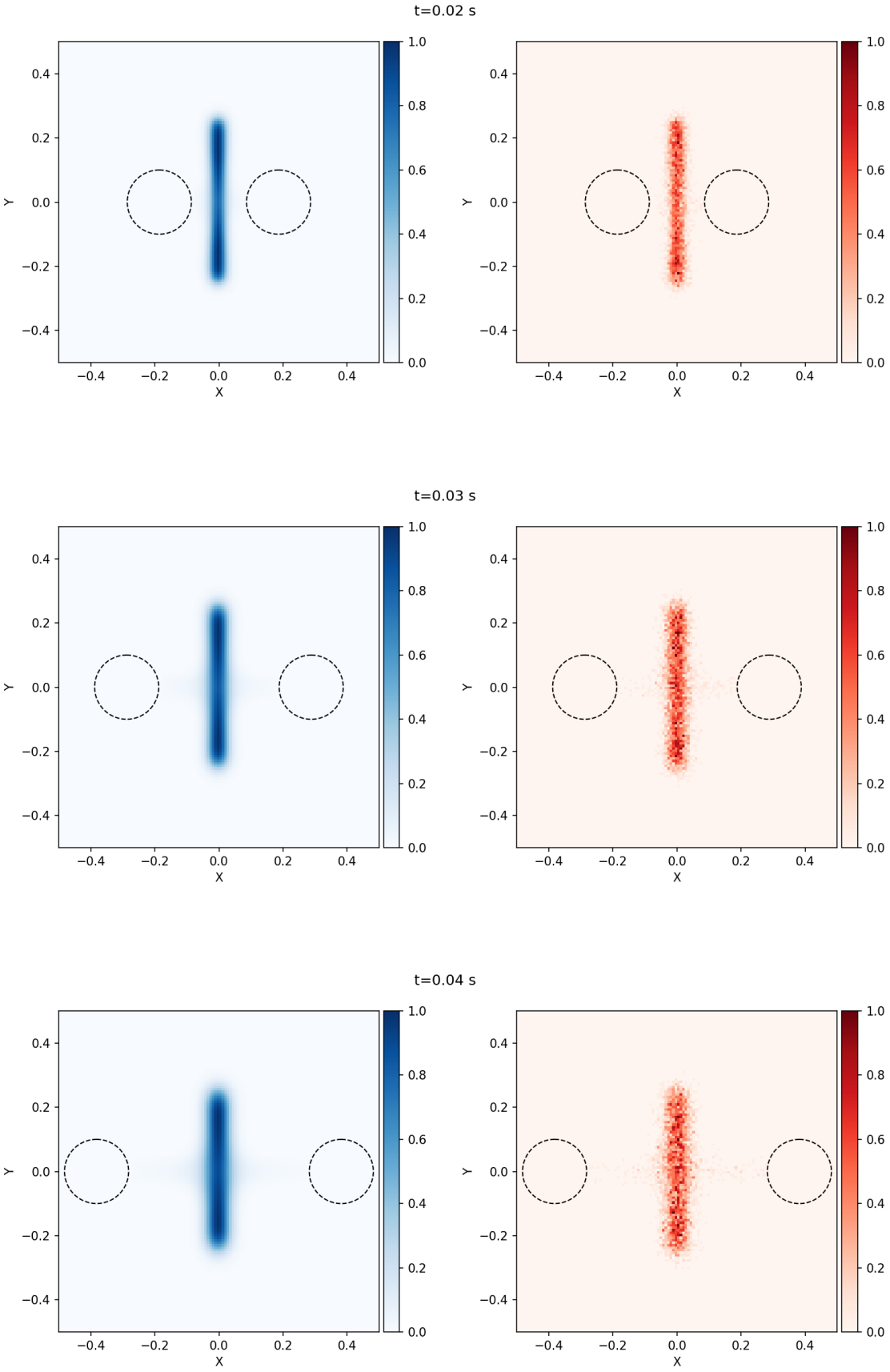
The images below show the visualisation of collision of two spherical particles on a wall of tracers and a scalar field region of value 1. The plots in left show the scalar field averaged and normalized

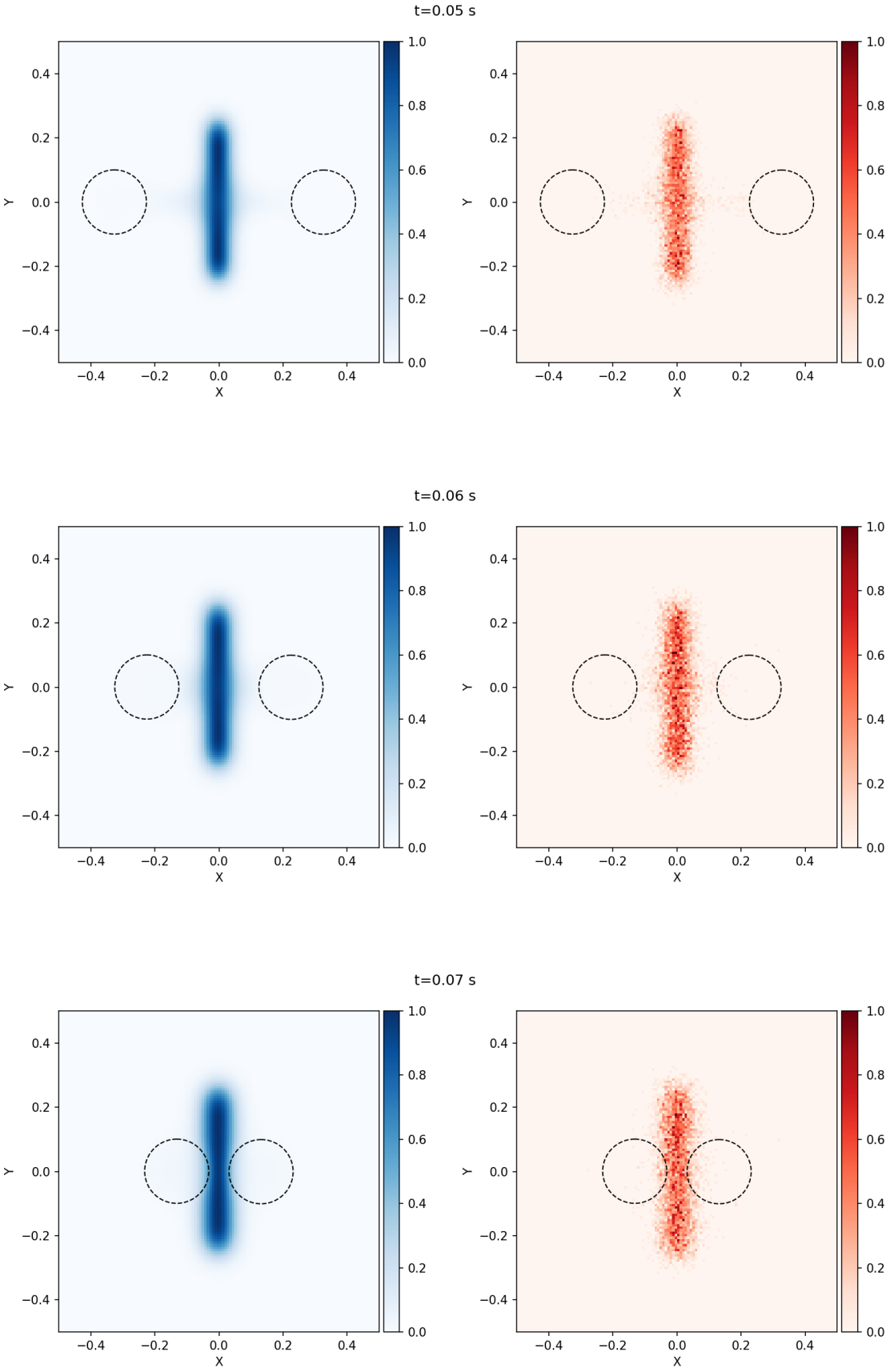
across the domain in the Z-direction. The plots in right show the concentration field determined from tracer location through binning to the same resolution as that of the scalar field. The particles are shown as outlines in black. Both the fields are non-smoothened.

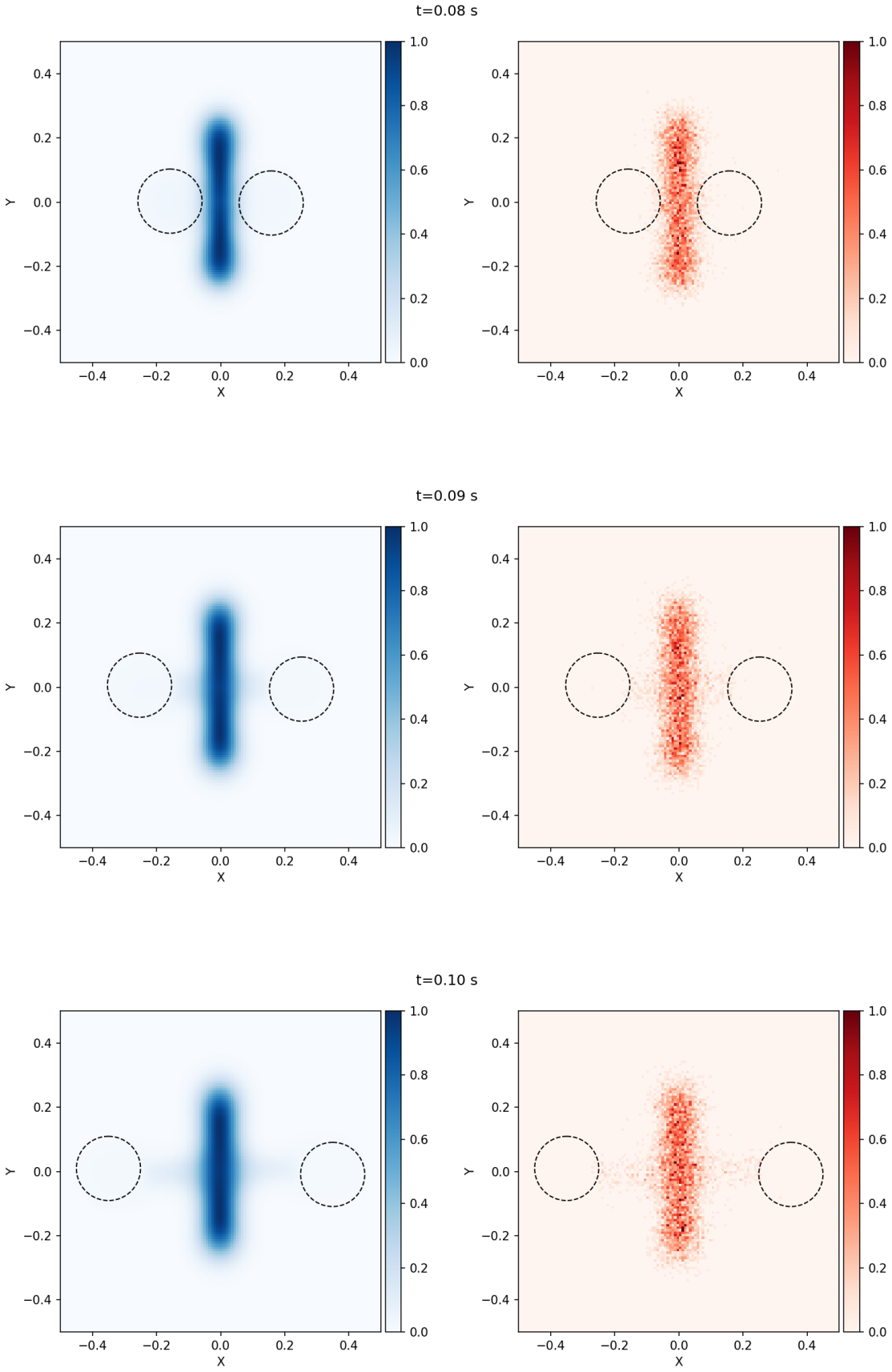
Some of the remarkable features observed in the following images are:

- The presence of noise in the tracer concentration field is to be noted. The noise can be mitigated either by time-averaging or through the simulation of a large number of tracers per unit volume. In this case, time-averaging is not possible as the comparison is transient.




t=0.02 s
t=0.03 s
t=0.04 s
X
Y
−0.4
−0.2
0.0
0.2
0.4
0.0
0.2
0.4
0.6
0.8
1.0


t=0.05 s
t=0.06 s
t=0.07 s
X
Y
−0.4
−0.2
0.0
0.2
0.4
0.0
0.2
0.4
0.6
0.8
1.0


t=0.08 s
t=0.09 s
t=0.10 s
X
Y