GPU-based simulation of dense suspensions: coupling the DEM and the LBM in standard C++

Maggio-Aprile, Raphaël Anthony

TRAVAIL DE MASTER

GPU-based simulation of dense suspensions: coupling the DEM and the LBM in standard C++

Encadrant: Prof. Jonas LÄTT

Co-encadrant: Dr. Christophe Guy COREIXAS

Candidat: Raphaël MAGGIO-APRILE

Juillet, 2023

Contents

Acronyms

AoS array of structures

BGK Bhatnagar-Gross-Krook

CDF Computational Fluid Dynamics

DEM Discrete Element Method

LBE Lattice Boltzmann Equation

LBM Lattice Boltzmann Method

OOP object-oriented programming

RBC red blood cell

SoA structure of arrays

STL Standard Template Library

TRT Two Relaxation Time

1 Introduction

Blood flow simulations play a crucial role in understanding the complex behavior of blood in the circulatory system. The main goal of this research project is to improve how we simulate blood flow using GPUs. The composition of blood is approximately 55% plasma, 45% red blood cells (RBCs), and less than 1% platelets and white blood cells. There are multiple challenges in simulating blood. In particular, the presence of a large number of RBCs suspended in a fluid (the plasma) and of a high volume fraction of particles within the fluid (the percentage of RBCs in the blood is called hematocrit and can vary between different values) necessitates sophisticated simulation techniques. Typically, we simulate the RBCs (and eventually the platelets) using a Discrete Element Method (DEM) that runs on the CPU, while fluid dynamics simulation of the plasma is done on the GPU using the Lattice Boltzmann Method (LBM). However, the problem with this approach is that it requires a lot of communication between the CPU and the GPU, which slows down the simulation. Existing DEM frameworks, such as LIGGGHTS [1], offer valuable tools for simulating particulate systems. However, these frameworks are object-oriented and lack efficient deployment on GPUs. To overcome this limitation, we aim to adopt a data-oriented approach, developing a portable code that is optimized for GPU architectures. By doing so, we can take full advantage of the powerful computational abilities and parallel processing capabilities of GPUs, resulting in better performance and efficiency for blood flow simulations. To ensure that our code can be easily used on different platforms, we have implemented it in standard C++ using parallel algorithms from the C++17 standard. The code can then be easily deployed on the GPU in a seamless manner using an appropriate compiler such as nvc++ from Nvidia. It is important to understand that our goal in this work is not to immediately outperform existing state-of-the-art methods, but to lay a solid foundation for future improvements and explore the challenges and potential solutions of performing all the simulations solely on the GPU. For these reasons, we have chosen to represent the RBCs as simple spheres, and their interactions will be described by a spring-dashpot model. The fluid will be simulated using the LBM with the Two Relaxation Time (TRT) model, while the coupling will be based on the partially saturated bounce-back method. Through this exploration, we expect to gain valuable insights into the complexities of GPU parallel computing and develop specific approaches to effectively simulate all aspects of blood flow using this powerful hardware. By utilizing the immense computational power of GPUs and following standard C++ practices, our aim is to create an innovative framework that enables more efficient and comprehensive blood flow simulations. Through the development and validation of our simulation framework (our C++ framework will be made available as open source in the near future), we aim to contribute to the advancement of blood flow simulations, opening new possibilities for understanding and studying the complexities of blood flow dynamics on a larger scale.

2 Literature review

The simulation of blood flow is a topic that has been and continues to be extensively studied. It is closely related to the simulation of granular media suspended in a fluid. Without attempting to provide an exhaustive overview of all the research in this field, we will discuss some works that are quite representative of the current state of research and upon which the present work heavily relies.

A Model for Red Blood Cells in Simulations of Large-scale Blood Flows, 2011

In this paper, Melchionna [2] presents a model for simulating blood flows using a combination of the LBM and the DEM. The model focuses on approximating RBCs as oblate ellipsoids. His model also take into account tank treading, which refers to the sliding motion of the RBC membrane under shearing flow. According to this work, it is an important factor to obtain accurate simulations near walls, as it influences the orientation of the RBCs. However, in his model, the RBC membrane is not explicitly tracked. Instead, the tank treading effect is considered by incorporating a tunable factor during the computation of the torque. The model also accounts for the higher viscosity of the internal fluid within the RBCs compared to the surrounding fluid. This is particularly important when the volume fraction of RBCs is high, since this viscosity contrast leads to a significant increase in the apparent viscosity with high volume fractions. To address this, the LBM is modified by introducing a local enhancement of the fluid viscosity within the RBCs shapes. Volume interactions are incorporated using the Gay-Berne potential, which is a modified Lennard-Jones potential suitable for ellipsoids. To confine the RBCs within a domain, spheres are placed at each mesh point of the walls to create a container within the simulation. A two-way coupling scheme is employed to account for the local interactions between RBCs and the surrounding fluid. For each cell inside the RBC, local forces and torques exerted by the fluid on the solids are considered. Additionally, an opposite reaction of the fluid is taken into account by adding a force term to the fluid update. The model demonstrates a relationship between relative viscosity and hematocrit levels for a laminar flow inside a channel close to experimental results. The author was able to demonstrate that it is possible to accurately simulate large-scale blood flows approximating RBCs as oblate ellipsoids, provided that certain parameters specific to RBCs, such as tank treading and the viscosity difference between the internal fluid of the red blood cell and the fluid representing the plasma, are taken into account in the model. However, specific implementation and performance results are not provided in the available notes and further details regarding these points would be beneficial to evaluate its practical applicability.

Computer Simulation Study of Collective Phenomena in Dense Suspensions of Red Blood Cells under Shear, 2012

This study of Krüger [3] focuses on the numerical investigation of simulations of RBCs in a shear flow. The simulation employs a combination of the LBM for the simulation of the plasma and finite element method to represent individually the deformable RBCs, incorporating the immersed boundary method to handle the interaction between the fluid and the deformable particles. The RBCs

are modeled as deformable particles, considering their deformability and tank-treading motion. The study favors a large number of particles over high accuracy with few particles to capture the collective phenomena in dense suspensions. For the immersed boundary method, the surface of the solid is discretized into multiple points, and interpolation is applied to approximate the surface position within each fluid cell. This method can accommodate any shape and is particularly suitable for deformable particles. However, challenges arise with the immersed boundary method, such as the sticking membrane problem caused by membrane nodes being too close to each other, which is especially the cases with high membrane density and thus with high volume fractions of RBCs. To address this, a repulsive force is introduced. They manage to simulate dense suspensions in shear flow with volume fractions up to 65%, considering 1000 particles and performing 10^5 steps, which requires substantial computational resources. The collective alignment resulting from tank-treading and the deformability contributes to the correct behavior of RBCs, resulting in a shear thickening behavior in agreement with the experimental observations. When rigid particles are used, shear thickening is also observed, but it does not align with the experiments. The study challenges the commonly reported reasons for shear thinning in blood found in the literature, namely deformation and tank-treading, emphasizing that these factors are not independent. According to the authors, the deformability of RBCs plays a crucial role in the shear thickening behavior of the blood, highlighting significant differences compared to rigid body simulations.

An efficient four-way coupled lattice Boltzmann - discrete element method for fully resolved simulations of particle-laden flows, 2020

In their study, Rettinger and Rüde [4] investigate a coupling approach combining the LBM and the DEM for fully resolved simulations. The coupling of the LBM and DEM simulation is done through a no-slip boundary condition for the cells at the solid surface. To improve the accuracy of the coupling, a method called the central linear interpolation scheme is used to better take into account the surface of the solid. However, this coupling approach initially faces challenges with fluid oscillations. To address the oscillation problem, a new LBM model based on the TRT model is employed. The TRT model is modified to control the bulk viscosity and effectively dampen the oscillations, resulting in improved stability and accuracy of the simulations. In terms of particle interaction, a lubrication-based approach is utilized for close particles with gaps smaller than the grid size. This unresolved gap is taken into account by introducing a lubrication term to simulate the behavior between the particles. For particle-particle interactions, a spring-dashpot model is employed, capturing the forces and dynamics between the particles accurately. Their proposed LBM/DEM framework is subjected to various tests to validate its accuracy. Their approach allows for the accurate simulation of a liquid containing suspended particles by utilizing the central linear interpolation method for the coupling and by addressing the oscillation issue through the use of a new LBM model based on the TRT model.

Rheology of mobile sediment beds in laminar shear flow: effects of creep and polydispersity, 2022

In their study, Rettinger et al. [5] investigate the modeling and simulation of sediment beds, which exhibit similarities to blood flows in certain aspects, such as high volume fraction and a large number of particles. The sediment beds are composed of particles represented by spheres of different sizes (polydisperse). The simulation approach combines the LBM and the DEM to study the rheology of sediment beds under shear flow. The LBM employs the TRT method, while the DEM utilizes a spring-dashpot model to capture the behavior of the particles. The coupling between LBM and DEM is based on the momentum exchange method and a method that interpolate surface information of the spheres. The fluid cells inside the simulation that interact with the sediment bed are removed, and the surface information is incorporated through interpolation. However, challenges arise when the surfaces of the particles are too close, leading to unresolved fluid forces. To address this, a lubrication term is introduced to account for the fluid forces between the closely spaced surfaces. The results of the simulations demonstrate that the monodisperse case yields similar outcomes to experimental observations. Additionally, the study explores the impact of polydispersity on the rheological behavior of the sediment beds.

Models, algorithms and validation for opensource DEM and CFD-DEM, 2012

In this study, Kloss et al. [1] design a framework combining the DEM and computational fluid dynamics. This paper actually describes the coupling mechanism between the simulation software OpenFOAM (for the fluid simulation) and LIGGGHTS (for DEM simulation). The study explores different modeling approaches (resolved and unresolved simulations) and validation techniques within this framework. The DEM simulation is based on spherical particles that are modeled using a spring-dashpot model based on the Hertzian contact theory (however, LIGGGHTS allows other solid shapes such as superquadrics and triangle meshes). For resolved cases, the immersed boundary method is employed. The solver for the DEM component is implemented in C++ and incorporates parallelization using the Message Passing Interface (MPI). The computational domain is decomposed into sub-domains, with each process responsible for solving one domain. A ghost region is assigned to each process, representing a portion of the neighboring regions containing particles likely to interact with the current particles of the domain. Forces on these particles are computed and communicated between iterations to the other processes. This approach allows for scalability on clusters of CPUs. Load balancing techniques are also considered, taking into account the sparsity of each sub-domain to optimize the utilization of available processes. Overall, their method allows for an efficient coupling of fluid simulation and solid simulation, particularly with scalability at large scales on CPU clusters.

Fast computation of accurate sphere-cube intersection volume, 2017

The computation of the intersection volume between a solid and a fluid cell is an important component of various methods that combine the LBM and the DEM. In their study, Jones and Williams [6] address the need for fast and accurate sphere-cube intersection volume computation. The authors review the accuracy and performance of various methods for this task, focusing on spheres where their radius is greater than twice the grid cell size (which is the case when the simulation is

resolved). The paper explores different mapping algorithms, including Monte-Carlo sampling, sub-division sampling, edge-intersection averaging, and other methods. The Monte-Carlo sampling and the sub-division sampling are very similar since both methods rely on taking a certain number of points within the cell and calculating the ratio between the number of points contained within the sphere and the total number of points (in Monte Carlo, these points are randomly generated, while in the sub-sampling method, these points are pre-defined). For the edge-intersection averaging method, each edge of the cube is analyzed to determine if an intersection point exists with the sphere. If an intersection point exists, the length of the edge inside the solid is recorded. If no intersection point exists, the distance is either considered as zero or one time the grid spacing (depending on if the edge is fully outside or inside the cube). The intersection volume is calculated as the sum of the lengths of the edges divided by the product of the number of edges and the grid cell size. The authors propose a new method based on linear approximation, which is the sum of the distance from the grid cell center to the particle surface and a precalculated term that depends on the radius of the sphere. This approximation is computed once per radius and is determined for the simple case where the closest point to the sphere lies in the middle of one side of the cube. Comparing different methods, the accuracy of Monte-Carlo and sub-division methods depends on the number of points used, while the linear approximation method offers a constant level of accuracy. The proposed linear approximation method is found to be 73% faster than the next fastest method, Monte-Carlo at equivalent accuracy. Importantly, the linear approximation method still maintains relatively high accuracy even when the collision occurs at a different angle than the case used to compute the pre-defined term of the linear approximation. In summary, Jones and Williams [6] present a fast and accurate method for computing sphere-cube intersection volumes in LBM/DEM simulations. The linear approximation method proves to be significantly faster than existing methods while maintaining a reliable level of accuracy. The findings demonstrate the potential for efficient computations in simulations involving numerous sphere-cube intersections.

2.1 Key insights from the literature

From these various sources, it is evident that several points are necessary to accurately simulate the behavior of blood flows. The first point is the importance of modeling RBCs deformation [3], potentially involving a full surface representation using the finite element method [3]. Accounting for the relative movement of the membrane, referred to as tank threading, seems necessary to achieve desired behaviors like shear thickening [3, 2]. However, the work of Melchionna [2] suggests that utilizing a simple potential like the Gay-Berne potential to represent the RBCs and modeling tank threading without explicitly tracking surface position could yield satisfactory results. Considering the viscosity difference between the interior and exterior of the RBCs also seems to play a significant role [2]. In our case, we have chosen to represent the RBCs as simple spheres, where the interactions are governed by a spring-dashpot model (see section 3.3.1), partly based on the model used by Kloss et al. [1]. Our model also does not account for the viscosity difference between the inside and outside of

RBCs. A more faithful representation of RBCs, considering their shape, their deformability, and the tank threading, should be explored in future work. For the coupling of the LBM and the DEM, although the immersed boundary method, as used by Krüger [3], seems the most reliable for accurately representing the surface, issues related to its usage (e.g., the sticking membrane problem) and its relative complexity (requiring meshing to represent the surface) lead us to opt for a simpler method: the partially saturated method (see section 3.4). This method involves computing the intersection volume between a sphere and a cube. The work of Jones and Williams [6] demonstrates the possibility of calculating this volume relatively accurately and with a reasonably low computational cost by using a linear approximation based on the distance between the surface of the sphere and the cube, along with a pre-calculated term dependent on the radius of the sphere. In our case (see section 3.4.1), we have chosen to use a linear function that relies solely on the distance between the surface of the sphere and the cube (the fluid cell). Therefore, it will be necessary to investigate how to calculate this term more precisely in a future work.

3 Methodology

3.1 AoS and SoA

Before even describing the theoretical foundations of the models we will use for our simulations, it is necessary to introduce the concept of a structure of arrays, as it forms the basis for most of the data structures we will employ in the various algorithms presented. Array of structures (AoS) and structure of arrays (SoA) are two different (and opposite) way to organize data in computer programming when we need to deal with a sequence of elements characterized by different quantities. For example, in this work we will need to work with multiples particles characterized by different quantities such as the position, the velocity, etc. AoS is a very intuitive data layout because it corresponds to the layout of a structure or a class in object-oriented programming (OOP). This approach is therefore simple to implement and to understand because all values related to a specific "object concept" are contained inside the class and we can use all the classic design patterns from the OOP. However, it is not optimized for memory access and cache efficiency when we need to access a property of multiples instances since the data is not contiguous in memory.

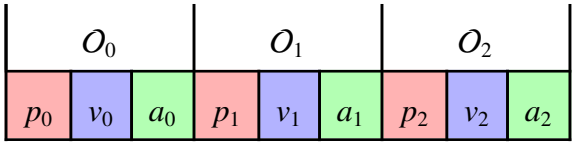


Figure 1: Example for the AoS layout where we have three objects O_0, O_1 and O_2 with the properties p, v and a.

Example 3.1: AoS in c++

Implementation of the example given in Figure 1.

```cpp
#include <iostream>

struct O {
    double p;
    double v;
    double a;
};

int main()
{
    O aos[3] = {{1.1, 1.2, 1.3}, {2.1, 2.2, 2.3}, {3.1, 3.2, 3.3}};
    std::cout << aos[1].p; // access properties p of O_1, output 2.1
}
```

SoA stores the properties of the multiple elements in different arrays (one per property). They are

more difficult to use because we have multiples arrays instead of one and it also become more difficult to encapsulate. However, each property is now contiguous in memory and thus memory access and cache efficiency is optimized when we need to access a property of multiples elements. For this reason, using SoA instead of AoS is one of the foundations of a data-oriented application.

	O_0	O_1	O_2
P	p_0	p_1	p_2
V	v_0	v_1	v_2
A	a_0	a_1	a_2

Figure 2: Example for the SoA layout. Here we have three different arrays P, V, A containing the properties of the objects O_0, O_1 and O_2.

Example 3.2: SoA in c++

Implementation of the example given in Figure 2.

```cpp
#include <vector>
#include <iostream>

struct O_soa {
    std::vector<double> p;
    std::vector<double> v;
    std::vector<double> a;
};

int main()
{
    O_soa soa = {{1.1, 2.1, 3.1}, {1.2, 2.2, 3.2}, {1.3, 2.3, 3.3}};
    std::cout << soa.p[1]; // access properties p of O_1, output 2.1
}
```

In our case, memory access is often the bottleneck. Therefore, using the SoA data layout can improve significantly our performances.

3.2 The lattice Boltzmann method

3.2.1 Basics

The Lattice Boltzmann method (LBM) originated from the Lattice Gas Automata (LGA) method. It has gained significant popularity as a method for simulating fluid flows in computational fluid dynamics (CFD). It offers an alternative to traditional methods that involve direct solving of the Navier-

Stokes equations[1]. The advantage of this method lies in its simplicity and ease of parallelization, which is of great importance to us. The method involves the discretization of space and time with a lattice structure. In each site of the lattice, we have quantities f_i called populations associated with discrete velocities[2] $\vec{c_i}$. These populations are often represented as a single vector, denoted as $\vec{f}$ where the components of the vector correspond to the different populations f_i. From these populations, we can obtain macroscopic quantities with physical meaning such as the mass density ρ and the momentum density $\vec{j} = \rho\vec{u}$ where $\vec{u}$ is the local velocity of the fluid in the site:

$$\rho = \sum_i f_i, \tag{1}$$

$$\vec{j} = \rho\vec{u} = \sum_i f_i \vec{c_i}. \tag{2}$$

Since the LBM does not work directly with macroscopic quantities and does not represent individual particles either, the method is commonly referred to as a mesoscopic approach, effectively bridging the gap between macroscopic methods, such as solving the Navier-Stokes equations, and microscopic methods like molecular dynamics. The evolution of the system is determined by the Lattice Boltzmann Equation (LBE), which describes how the populations f_i at time $t + \Delta t$ and position $\vec{r} + \Delta t\vec{c_i}$ evolve based on a specific collision operator Ω_i. The collision operator depends on the populations $\vec{f}$ at position $\vec{r}$ and time t. Δt is the time step of the simulation. The LBE is given by:

$$f_i(\vec{r} + \Delta t\vec{c_i}, t + \Delta t) = f_i(\vec{r}, t) + \Omega_i(f(\vec{r}, t)). \tag{3}$$

The LBM divides this evolution in two distinct steps: a collision step and a propagation step (also referred to as the streaming step). In the collision step, the populations within each cell interact with each other and relax towards a local equilibrium. During the propagation step, the populations are propagated or streamed to neighboring cells based on their associated discrete velocities. This step propagates the information throughout the lattice and reflects the advection of the physical quantities. We can then rewrite the LBE according to these two steps:

- the collision step:

$$f_i^{out}(\vec{r}, t) = f_i^{in}(\vec{r}, t) + \Omega_i(\vec{f^{in}}(\vec{r}, t)). \tag{4}$$

- the streaming step:

$$f_i^{in}(\vec{r} + \Delta t\vec{c_i}, t + \Delta t) = f_i^{out}(\vec{r}, t). \tag{5}$$

For better understanding, we call $\vec{f^{out}}$ the population after the collision step and $\vec{f^{in}} = \vec{f}$ the population after the streaming step. We sometimes say that $\vec{f^{in}}$ are the populations that "enter" the site and $\vec{f^{out}}$

[1] Although it has been shown that solving the Lattice Boltzmann equations also solves the Navier-stokes equations [7].

[2] The populations f_i are closely linked to the particle distribution function in kinetic theory and can be regarded as a discretized version (with respect to the velocity) of the latter [7].

the populations that "leave" the site, as if these populations would represent virtual particles that enter and leave the cells.

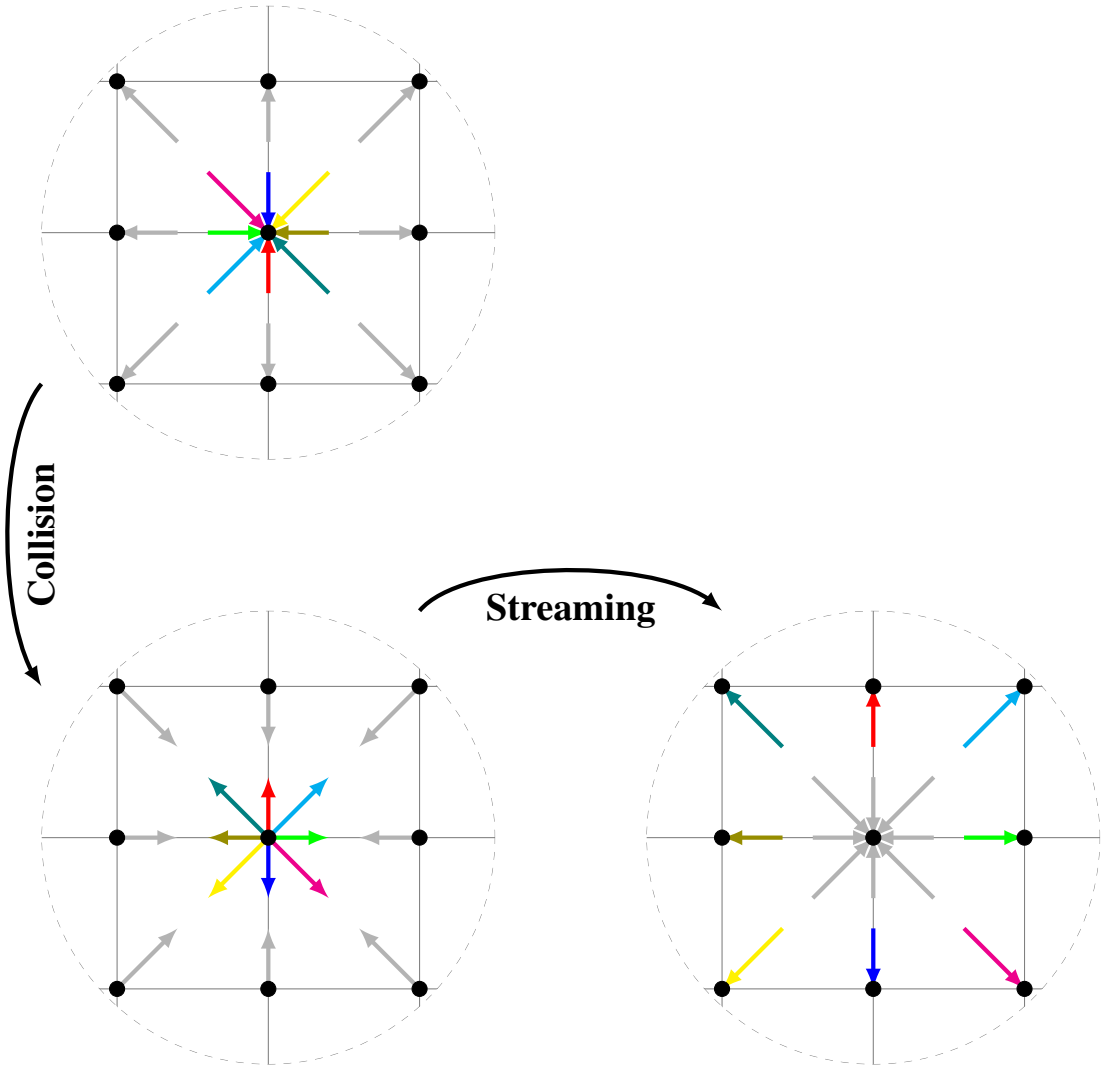


Figure 3: Collision and streaming steps.

3.2.2 Lattice structure

The lattice structure depends on the set of discrete velocities and the dimension of the problem. The naming convention for the lattice structure follows the format DQ, where D represents the number of dimensions and Q denotes the number of discrete velocities. For example, the D3Q19 model signifies that there is a three-dimensional lattice with 19 different discrete velocities (and thus 19 populations). When the number of discrete velocities is odd, it means that one of the velocity is null. The other velocities $\vec{c_i}$ all have an opposite velocity $\vec{c_i}$ such that $\vec{c_i} = -\vec{c_i}$ (for the null velocity, the opposite velocity is itself). Weights w_i are also assigned to discrete velocities to determine the contribution of each velocity direction. In three dimensions, there are other lattice structures available, such as the D3Q27 and D3Q15 models. The selection of the lattice structure depends on the specific problem we want to solve and often involves a tradeoff between memory consumption and the accuracy of

the simulation. For our work, we have chosen to utilize the D3Q19 model, which provides a good balance between memory requirements and accuracy. With the D3Q19 model, we have the following discrete velocities and weights:

Velocity	**Value**
$\vec{c_0}$	(-c, 0, 0)
$\vec{c_1}$	(0, -c, 0)
$\vec{c_2}$	(0, 0, -c)
$\vec{c_3}$	(-c, -c, 0)
$\vec{c_4}$	(-c, c, 0)
$\vec{c_5}$	(-c, 0, -c)
$\vec{c_6}$	(-c, 0, c)
$\vec{c_7}$	(0, -c, -c)
$\vec{c_8}$	(0, -c, c)
$\vec{c_9}$	(0, 0, 0)
$\vec{c_{10}}$	(c, 0, 0)
$\vec{c_{11}}$	(0, c, 0)
$\vec{c_{12}}$	(0, 0, c)
$\vec{c_{13}}$	(c, c, 0)
$\vec{c_{14}}$	(c, -c, 0)
$\vec{c_{15}}$	(c, 0, c)
$\vec{c_{16}}$	(c, 0, -c)
$\vec{c_{17}}$	(0, c, c)
$\vec{c_{18}}$	(0, c, -c)

Table 1: D3Q19 discrete velocities. Here $c = \frac{\Delta x}{\Delta t}$.

Weight	**Value**
w_0	1/18
w_1	1/18
w_2	1/18
w_3	1/36
w_4	1/36
w_5	1/36
w_6	1/36
w_7	1/36
w_8	1/36
w_9	1/3
w_{10}	1/18
w_{11}	1/18
w_{12}	1/18
w_{13}	1/36
w_{14}	1/36
w_{15}	1/36
w_{16}	1/36
w_{17}	1/36
w_{18}	1/36

Table 2: D3Q19 weights.

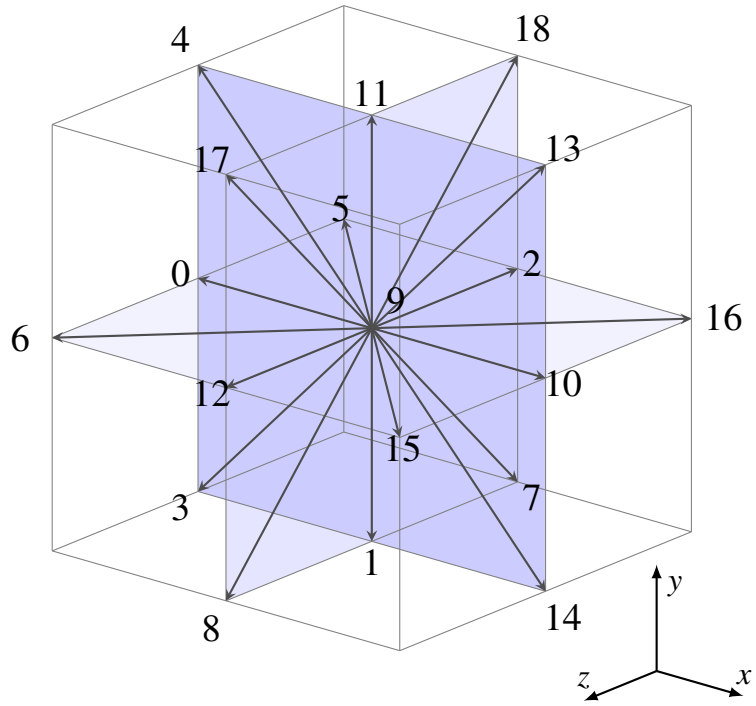


Figure 4: D3Q19 lattice structure.

The velocities $\vec{c_i}$ and weights w_i depend on the chosen structure and are selected to satisfy certain isotropy properties and preserve conservation laws. However, the specific details of how these velocities and weights are computed are beyond the scope of our discussion. More information on the computation of them can be found in Ref. [8, 7].

3.2.3 Collision operator

The collision operator is the fundamental component of the LBM that models the interactions between the particles in a fluid. The simplest collision operator used in the LBM is the Bhatnagar-Gross-Krook (BGK) operator, named after the authors who developed this method [9]. It depends on a single parameter known as the relaxation time τ that determines the rate at which the distribution function relaxes towards its local equilibrium state during the collision step of the LBM. By adjusting the value of the relaxation time we can change the behavior of the fluid to have different flow regimes from highly viscous (big τ) to highly turbulent flows (small τ). the BGK operator is given by the following formula:

$$\Omega_i^{BGK}\left(\vec{f^{in}}(\vec{r},t)\right) = -\frac{\Delta t}{\tau}\left(f_i^{in}(\vec{r},t) - f_i^{eq}(\rho,\vec{u})\right). \tag{6}$$

It is not uncommon to express equation (6) with a relaxation coefficient $\omega = \frac{1}{\tau}$. This is the notation we will use in this paper. In this case, equation (6) becomes:

$$\Omega_i^{BGK}\left(\vec{f^{in}}(\vec{r},t)\right) = -\omega\Delta t\left(f_i^{in}(\vec{r},t) - f_i^{eq}(\rho,\vec{u})\right), \tag{7}$$

where f_i^{eq} is the local equilibrium population and depends on the macroscopic quantities ρ and $\vec{u}$

given by equations (1) and (2). It is given by:

$$f_i^{eq}(\vec{r},t) = w_i\rho\left(1 + \frac{\vec{c_i}\cdot\vec{u}}{c_s^2} + \frac{(\vec{c_i}\cdot\vec{u})^2}{2c_s^4} - \frac{\vec{u}\cdot\vec{u}}{2c_s^2}\right), \tag{8}$$

where c_s is referred to as the speed of sound and is given by the following formula for the D3Q19 lattice [7]:

$$c_s^2 = \frac{1}{3}\frac{\Delta x^2}{\Delta t^2}, \tag{9}$$

where Δx is the lattice grid spacing. It is important to note that the moments of the equilibrium distribution function $\vec{f}^{eq}$ are equal to the moments of the distribution function $\vec{f^{in}}$:

$$\sum_i f_i^{eq} = \sum_i f_i^{in} = \rho, \tag{10}$$

$$\sum_i f_i^{eq}\vec{c_i} = \sum_i f_i\vec{c_i} = \vec{j} = \rho\vec{u}. \tag{11}$$

This property is important as it guarantees that the collision operator conserves the mass density ρ and momentum density $\vec{j}$ (thus $\sum_i \Omega_i^{BGK} = 0$ and $\sum_i \vec{c_i}\Omega_i^{BGK} = 0$). The value of τ can be determined based on the kinematic viscosity[3] of the fluid ν [7]:

$$\nu = c_s^2\left(\tau - \frac{\Delta t}{2}\right) = c_s^2\left(\frac{1}{\omega} - \frac{\Delta t}{2}\right). \tag{12}$$

One of the problem with the BGK operator is that the stability and the accuracy of the simulation partially depends on the kinematic viscosity (and thus on ω).

Another commonly used operator is the TRT operator [10]. It uses two parameters s^+ and s^- instead of one for the BGK operator. This operator is given by :

$$\Omega_i^{TRT} = -s^+\Delta t(f_i^+ - e_i^+) - s^-\Delta t(f_i^- - e_i^-), \tag{13}$$

where f_i^+ and f_i^- are referred to as the symmetric and antisymmetric components of the population f_i and e_i^+ and e_i^- as the symmetric and antisymmetric components of the equilibrium population f_i^{eq}.

$$f_i^+ = f_{\bar{i}}^+, \quad f_i^- = -f_{\bar{i}}^-, \quad f_i^+ + f_i^- = f_i, \tag{14}$$

$$e_i^+ = e_{\bar{i}}^+, \quad e_i^- = -e_{\bar{i}}^-, \quad e_i^+ + e_i^- = f_i^{eq}. \tag{15}$$

The population f_i^+ and f_i^- are given by:

$$f_i^{\pm} = \frac{f_i \pm f_{\bar{i}}}{2}. \tag{16}$$

[3] the kinematic viscosity is defined as the ratio of dynamic viscosity to the density of the fluid. Its SI units are square meters per second (m^2/s).

the equilibrium populations are given by:

$$e_i^+ = w_i\rho\left(1 + \frac{(\vec{c_i}\cdot\vec{u})^2}{2c_s^4} - \frac{\vec{u}\cdot\vec{u}}{2c_s^2}\right), \tag{17}$$

$$e_i^- = w_i\rho\frac{\vec{c_i}\cdot\vec{u}}{c_s^2}. \tag{18}$$

As we can see, by choosing $s^+ = s^-$, we obtain the BGK operator. The parameter s^+ is related to the kinematic viscosity ν in the same way that ω was with the BGK operator in formula (12) ($\nu = c_s^2\left(\frac{1}{s^+} - \frac{\Delta t}{2}\right)$). The parameter s^- is a free parameter and is chosen for the stability of the simulation. In fact, the stability of the simulation is linked to a parameter called the "magic parameter" that depends on s^+ and s^- [7]:

$$\Lambda = \left(\frac{1}{s^+\Delta t} - \frac{1}{2}\right)\left(\frac{1}{s^-\Delta t} - \frac{1}{2}\right). \tag{19}$$

One of the advantage of this method is that the stability of the simulation does not depend on ν anymore since we can freely choose the value of s^- based on the magic parameter [7]. The choice of the value of Λ depends on the properties we want to obtain. In our case, we choose $\Lambda = \frac{3}{16}$ because it gives us the property[4] that the physical position of the wall when using the bounce-back boundary condition (see section 3.2.5) is exactly at the midpoint between the fluid cell and the bounce-back cell [7, 11]. In fact, it is mainly due to this property that we use the TRT model in this work. With this parameter, we have that:

$$\Lambda = \frac{3}{16} \Leftrightarrow s^- = \frac{1}{\Delta t}\frac{8\cdot(2 - s^+\Delta t)}{8 - s^+\Delta t}. \tag{20}$$

3.2.4 Unit conversion

To simplify calculations and for ease of implementation, it is common in practice to normalize the time by Δt and the distance by Δx in the code (which is equivalent to set both the grid spacing and the time step to a value of 1). We want to establish a way to recover our physical units Δt and Δx from these normalized lattice units. The value of Δx will depends on the lattice size and Δt will be chosen to satisfy some stability constraints. We also use a dimensionless quantity that characterizes the flow regime of a fluid called the Reynolds number and generally simply denoted as Re. By using the Reynolds number, we can relate the physical units of a flow to the lattice units of the simulation. At low Reynolds numbers, we are in the regime of laminar flows, which are characterized by smooth fluid motion and at high Reynolds numbers, we are in the regime of turbulent flows which are more chaotic. It is defined as the ratio of the characteristic velocity u (for example the velocity of the inlet flow) times the characteristic length L (for example the diameter of a particle) to the the kinematic viscosity of the fluid ν:

[4] A comparison of different choices of Λ and their properties can be found in Ref. [7].

$$Re = \frac{uL}{\nu}. \tag{21}$$

Let $u_{LB} = \frac{\Delta t}{\Delta x}u$ be the characteristic velocity in lattice units, $L_{LB} = \frac{L}{\Delta x}$ the characteristic length in lattice units, $\nu_{LB} = \frac{\Delta t}{\Delta x^2}\nu$ the kinematic viscosity in lattice units, $c_{s_{LB}} = c_s\frac{\Delta t}{\Delta x}$ the speed of sound in lattice units and $\omega_{LB} = \omega\Delta t$ (we have $s^+_{LB} = \omega_{LB}$ for the TRT model) the relaxation coefficient in lattice units. Δx can be easily obtained from the L and L_{LB}:

$$\Delta x = \frac{L}{L_{LB}}. \tag{22}$$

To obtain Δt, we often fix u_{LB} to satisfy some numerical stability constraints[5]:

$$\Delta t = \frac{\Delta x \cdot u_{LB}}{u}. \tag{23}$$

ν_{LB} can then be found using the Reynolds number:

$$\begin{aligned} Re = \frac{uL}{\nu} = \frac{u_{LB}\frac{\Delta x}{\Delta t}L_{LB}\Delta x}{\nu_{LB}\frac{\Delta x^2}{\Delta t}} = \frac{u_{LB}L_{LB}}{\nu_{LB}} \\ \Leftrightarrow \nu_{LB} = \frac{u_{LB}L_{LB}}{Re}. \end{aligned} \tag{24}$$

Finally, we can use equation (12) to find the relaxation coefficient of the simulation:

$$\omega_{LB} = \omega\Delta t = \frac{c^2_{s_{LB}}}{\nu_{LB} - \frac{c^2_{s_{LB}}}{2}} = \frac{1}{\frac{\nu_{LB}}{c^2_{s_{LB}}} - \frac{1}{2}}. \tag{25}$$

With the D3Q19 lattice, we have $c^2_{s_{LB}} = \frac{1}{3}$, and thus:

$$\omega_{LB} = \frac{1}{3 \cdot \nu_{LB} + \frac{1}{2}}. \tag{26}$$

3.2.5 Boundary conditions

One of the key point of the LBM is the rules we apply at the boundaries of the simulated domain. There are various boundary conditions we can use, and they strongly depend on the desired simulation. In our case, we will only detail three boundary conditions: the periodic boundary condition, the bounce-back boundary condition and the outlet boundary condition. Let us start with the simplest one, the periodic boundary condition. The idea is simply to consider that the domain repeats infinitely. In practice, this means that the populations "leaving" the domain ($\overrightarrow{f^{out}}$ populations located in a cell at the domain boundary with periodic boundary condition and with a direction pointing outward) will

[5]More information on how to choose u_{LB} can be found in Ref. [12], but a good start can be $u_{LB} = 0.02$ [12]. In a similar manner, it is also possible to fix the value of ν_{LB} to find Δt.

be streamed to the opposite side of the domain (for example, populations "leaving" at the north of the domain will enter at the south and those "leaving" at the east will enter at the west, etc.). In terms of formulas, the streaming step is modified as follows:

$$\begin{aligned} f_i^{in}(\vec{r'}, t + \Delta t) &= f_i^{out}(\vec{r}, t) \\ \vec{r'} &= \begin{bmatrix} r_x + \Delta t\ c_{ix} & \mod N_x \Delta x \\ r_y + \Delta t\ c_{iy} & \mod N_y \Delta x \\ r_z + \Delta t\ c_{iz} & \mod N_z \Delta x \end{bmatrix} \end{aligned} \tag{27}$$

Where r_k and c_{ik} are the components of $\vec{r}$ and $\vec{c_i}$ and N_x, N_y, N_z are the number of lattice cells in the X, Y, and Z direction. In terms of implementation, the domain is generally considered periodic by default when no other boundary condition is added [6]. The second boundary condition we will discuss is the outlet condition. We use this condition when we assume that the fluid continues to flow beyond the boundaries of the domain. For example, we could simulate a flow in a pipe. In this case, we would like the outlet condition to mimic the behavior of an uninterrupted pipe. During the calculation of the collision operator, a certain number of populations f_i^{in} are unknown. As we assume that the values of the populations evolve continuously in the domain, we will simply use the values of the f_i^{in} from a neighboring cell to replace the missing populations values:

$$f_i^{in}(\vec{r}, t) = f_i^{in}(\vec{r} + \Delta t \vec{c_i}, t) = f_i^{out}(\vec{r}, t - \Delta t) \qquad \text{if } \vec{r} - \Delta t \vec{c_i} \text{ is outside the domain.} \tag{28}$$

The last boundary condition we will discuss is the bounce-back boundary condition. It mimics a non slipping wall and simply reflects the outgoing populations back into the domain:

$$f_{\bar{i}}^{in}(\vec{r}, t + \Delta t) = f_i^{out}(\vec{r}, t) \qquad \text{if } \vec{r} + \Delta t \vec{c_i} \text{ is outside the domain.} \tag{29}$$

It is very important to note that the physical position of the wall surface is not the position of the wall cell. In fact, the physical position of the wall is half-way between the fluid cell and the wall cell [3, 7].

[6] Keep in mind that in c++, the binary operator "%" for integers is not the same as the mathematical modulo but is the remainder of the integer division, which means that its behavior is different for negative numbers. In our case, adding the size of the domain before applying the remainder operation is sufficient to obtain the mathematical modulo.

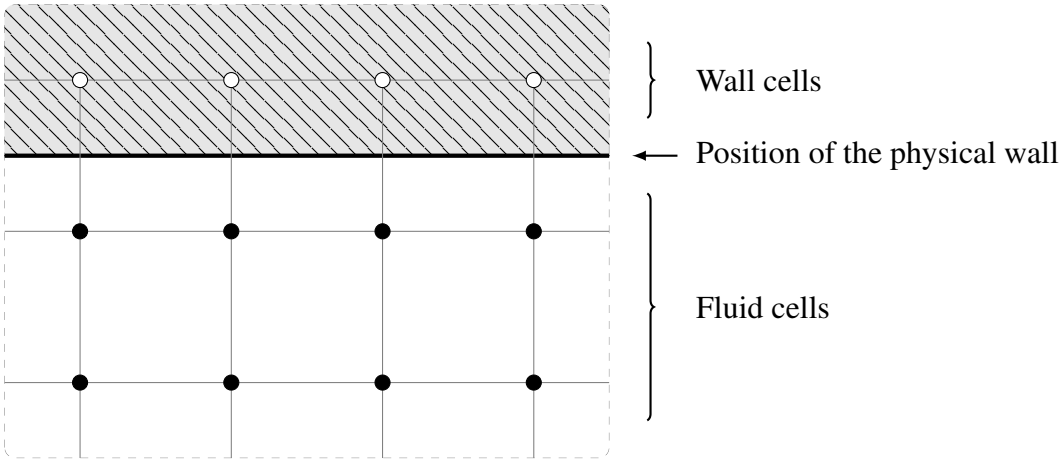


Figure 5: Half-way wall position for the bounce-back boundary condition.

We can also have a bounce-back boundary condition where the wall has a velocity tangential to its surface. This method is also called "momentum exchange" and is defined as [13, 3]:

$$f_{\bar{i}}^{in}(\vec{r}, t + \Delta t) = f_i^{out}(\vec{r}, t) - \frac{2}{c_s^2} w_i \vec{c_i} \cdot \vec{u_w} \qquad \text{if } \vec{r} + \Delta t \vec{c_i} \text{ is outside the domain,} \tag{30}$$

where $\vec{u_w}$ is the wall velocity.

3.2.6 The collision and streaming algorithm

In order to perform the collision and streaming steps in parallel without race conditions, we will use two different containers for the populations that we will swap at each iteration. One container will contain the input population at the current iteration and the other container will contain the new input populations for the next iteration. The idea of the algorithm is to perform the collision and streaming step at the same time and to use the symmetric properties of the equillibrium function in order to reduce the number of operations (equations (14) and (15)). This algorithm is strongly based on the work from Latt et al. [13]. We iterate over the cells in parallel. Inside this loop, we loop over the populations. We compute the new output population and we stream it to the container containing the input populations at the next iteration. By using symmetric properties, we can compute f_i^{out} and $f_{\bar{i}}^{out}$ at the same time. In order to simplify the notation, the following algorithms are given for a time step and a grid spacing equal to one ($\Delta t = \Delta x = 1$). The pseudocode for the main loop of the algorithm is the following:

Algorithm 3.1: Collision and streaming algorithm

Data
cellType: the list containing the types of the cells (fluid, bounce back, outlet)
NCells: the number of LBM cells
fin: the container of the input populations at the current iteration
fout: the container of the new input populations for the next iteration
w: list of the weights associated to the velocities
ω: the relaxation coefficient. For the TRT model, we have $\omega = s^+$

▷ *iterate over the LBM cells, the first loop can be paralellized* ◁
for cellIndex = $0, \ldots,$ NCells $- 1$ **do**
  if cellType[cellIndex] != bounceBack **then**
    ▷ *compute the cell coordinates* ◁
    xyz = getCellCoordinates(cellIndex)
    ▷ *compute the density and velocity* ◁
    ρ, $\vec{u}$ = getMacro(cellIndex)
    ▷ *compute the squared norm of the velocity* ◁
    $u^2 = \vec{u} \cdot \vec{u}$
    ▷ *iterate over the populations (only the first 8 since we will use symmetric properties to compute the opposite velocity* ◁
    for k = $0, \ldots, 8$ **do**
      ▷ *compute the collision* ◁
      $f_k^{out}, f_{\bar{k}}^{out}$ = collision(cellIndex, popIndex, *rho*, $\vec{u}$, u^2)
      ▷ *stream the populations* ◁
      stream(cellIndex, k, xyz, $f_k^{out}, f_{\bar{k}}^{out}$)
      stream(cellIndex, $\bar{k}$, xyz, $f_{\bar{k}}^{out}, f_k^{out}$)
    ▷ *resting population* ◁
    fout(cellIndex, 9) $= (1 - \omega) \cdot$ fin(cellIndex, 9) $+ \omega \cdot w[9] \cdot \rho \cdot (1 - 1.5 \cdot u^2)$
▷ *Swap the populations* ◁
swap(fin, fout)

The pseudocode for the collision operation is the following (only for the TRT method, but the method is similar for the BGK method):

Algorithm 3.2: LBM collision algorithm

Data

cellIndex: index of the LBM cell
k: index of population
rho: fluid density inside the cell
$\vec{u}$: fluid velocity inside the cell
u^2: squared norm of the velocity
c: list of the lattice velocities
fin: the container of the input populations at the current iteration

procedure COLLISION(cellIndex, k, ρ, $\vec{u}$, u^2)

$s^+ = 8 \cdot (2 - s^+)/(8 - s^+)$
▷ *dot product between the lattice velocity and the cell velocity* ◁
c_u = $\overrightarrow{\text{c[k]}} \cdot \vec{u}$
▷ *eq+ and eq- equilibrium populations* ◁
$e_k^+ = \text{w[k]} \cdot \rho \cdot (1 + 4.5 \cdot \text{c\_u}^2 - 1.5 \cdot u^2)$
$e_k^- = \text{w[k]} \cdot \rho \cdot 3 \cdot \text{c\_u}$
▷ *f+ and f- populations* ◁
$f_k^{in} = \text{fin(cellIndex, } k)$
$f_{\bar{k}}^{in} = \text{fin(cellIndex, } \bar{k})$
$f_k^+ = 0.5 \cdot (f_k^{in} + f_{\bar{k}}^{in})$
$f_k^- = f_k^+ - f_{\bar{k}}^{in}$
▷ *fout and fout_opp populations* ◁
$f_k^{out} = f_k^{in} - s^+ \cdot (f_k^+ - e_k^+) - s^- \cdot (f_k^- - e_k^-)$
$f_{\bar{k}}^{out} = f_{\bar{k}}^{in} - s^+ \cdot (f_k^+ - e_k^+) + s^- \cdot (f_k^- - e_k^-)$
return f_k^{out}, $f_{\bar{k}}^{out}$

The pseudocode for the streaming function is the following:

Algorithm 3.3: LBM streaming algorithm

Data
cellIndex: index of the LBM cell
cellType: the list containing the types of the cells (fluid, bounce back, outlet)
k: index of population
xyz: coordinates of the cell
fin: the container of the input populations at the current iteration
fout: the container of the new input populations for the next iteration

procedure STREAM(cellIndex, k, xyz, f_k^{out}, $f_{\bar{k}}^{out}$)
- ▷ *compute the neighbor cell coordinates* ◁
- xyz_n = getNeighborCellCoordinates(xyz, k)
- ▷ *compute the neighbor cell index* ◁
- cellIndex_n = getCellIndex(xyz_n)
- ▷ *stream the populations* ◁
- **if** cellType[cellIndex] == outlet and !isNeighborInDomain(cellIndex, cellIndex_n) **then**
  - ▷ *outlet* ◁
  - fout(cellIndex, $\bar{k}$) = $f_{\bar{k}}^{out}$
  - **if** cellType[cellIndex_n] == fluid **then**
    - ▷ *special case in the edge: outlet -> fluid with periodic boundary conditions* ◁
    - fout(cellIndex_n, k) = f_k^{out}
- **else if** cellType[cellIndex_n] == fluid or cellType[cellIndex_n] == outlet **then**
  - fout(cellIndex_n, k) = f_k^{out}
- **else if** cellType[cellIndex_n] == bounceBack **then**
  - ▷ *bounce back* ◁
  - fout(cellIndex, $\bar{k}$) = f_k^{out} + fin(cellIndex_n, k)

Therefore, we have an efficient and relatively simple method to compute the fluid part of the simulation in parallel on a GPU. However, keep in mind that there are other more efficient methods based on a single structure containing the populations [13].

3.3 The discrete element method

The DEM is used to simulate granular material such as riverbed, sand, red blood cells, etc. There are various ways to model the particles in a granular medium. Some methods, such as those in Ref. [3], fully model the particles surfaces using a deformable triangle mesh, while others approximate the surfaces using simpler geometric volumes (biconcave discs, superquadrics, ellipsoids, spheres, etc.) [2, 5, 14]. In our case, we are interested in simulating red blood cells and platelets. Although red blood cells are typically biconcave and deformable in shape, we will focus on the simplest case and represent them only as spheres. The goal of this work is primarily to provide a simple basis for simulating red blood cells on GPUs, and simulations with more realistic shapes should be explored in future work. It is worth noting that certain conditions, such as hereditary spherocytosis, result in spherical-shaped red blood cells, making a simulation where red blood cells are represented as spheres still relevant.

3.3.1 Pairwise interactions between solids

The model we will use is based on contact mechanics. Particles that collide have a certain contact surface and interact with each other by exerting forces and torques on the other particles. The motion of a particle i thus only depends on two quantities:

- The sum of the forces applied by the other particles:

$$\vec{F_i} = \sum_{ij, i \neq j} \vec{F_{ij}}. \tag{31}$$

- The sum of the torques applied by the other particles:

$$\vec{M_i} = \sum_{ij, i \neq j} \vec{M_{ij}}. \tag{32}$$

There is an interaction between two particles only if they are in contact. With spheres, this happens when the distance d between the centers of the spheres is smaller than the sum of their radii:

$$d < R_i + R_j. \tag{33}$$

The force exerted by a particle j on a particle i can be decomposed into two components:

$$\vec{F} = \vec{F_n} + \vec{F_t}. \tag{34}$$

- A component $\vec{F_n}$ normal to the contact surface
- A component $\vec{F_t}$ parallel to the contact surface (and thus tangential to the surface of the sphere).

Here and in the rest of this section, the force $\vec{F}$ refers to the force $\vec{F_{ij}}$ when indices are not specified. These forces will be calculated using a model called the spring-dashpot model [15]. In this model, the normal and tangential forces can be represented as a system consisting of a spring and a damper. The formula of the forces for this model is:

$$\vec{F} = \underbrace{\left(-k_n \, \vec{\delta_{n_{ij}}} - \gamma_n \, \vec{v_{n_{ij}}}\right)}_{\vec{F_n}} + \underbrace{\left(-k_t \, \vec{\delta_{t_{ij}}} - \gamma_t \, \vec{v_{t_{ij}}}\right)}_{\vec{F_t}}, \tag{35}$$

where k_n is the normal elastic coefficient, k_t is the tangential elastic coefficient, $\vec{\delta_{n_{ij}}}$ is the normal overlap vector, $\vec{\delta_{t_{ij}}}$ is the tangential overlap vector, γ_n is the normal damping coefficient, γ_t is the tangential damping coefficient, $\vec{v_{n_{ij}}}$ is the normal relative velocity vector, $\vec{v_{t_{ij}}}$ is the tangential relative

velocity vector[7]. The Forces $\vec{F_n}$ and $\vec{F_t}$ are therefore decomposed into two terms: a force term opposing displacement and a force term opposing velocity. Depending on the instant of collision (whether the particles are approaching or separating), these force terms may not necessarily be in the same direction:

$$\vec{F_n} = \underbrace{-k_n \ \vec{\delta}_{n_{ij}}}_{\text{against the displacement}} \quad \underbrace{-\gamma_n \ \vec{v}_{n_{ij}}}_{\text{against the velocity}} . \tag{36}$$

Additionally, the tangential overlap $\vec{\delta_{t_{ij}}}$ is truncated to satisfy the following condition that is ruled by a Coulomb's friction coefficient:

$$F_t \leq \mu F_n. \tag{37}$$

The physical interpretation is as follows: at the beginning of a the collision of two spheres with a relative tangential motion, the surfaces themselves do not move relative to each other, but the material of the spheres deforms and accumulates elastic potential energy. When the accumulated potential energy becomes too large ($F_t = \mu F_n$), the surfaces start to slide, and the frictional force described by Coulomb's law comes into play[8].

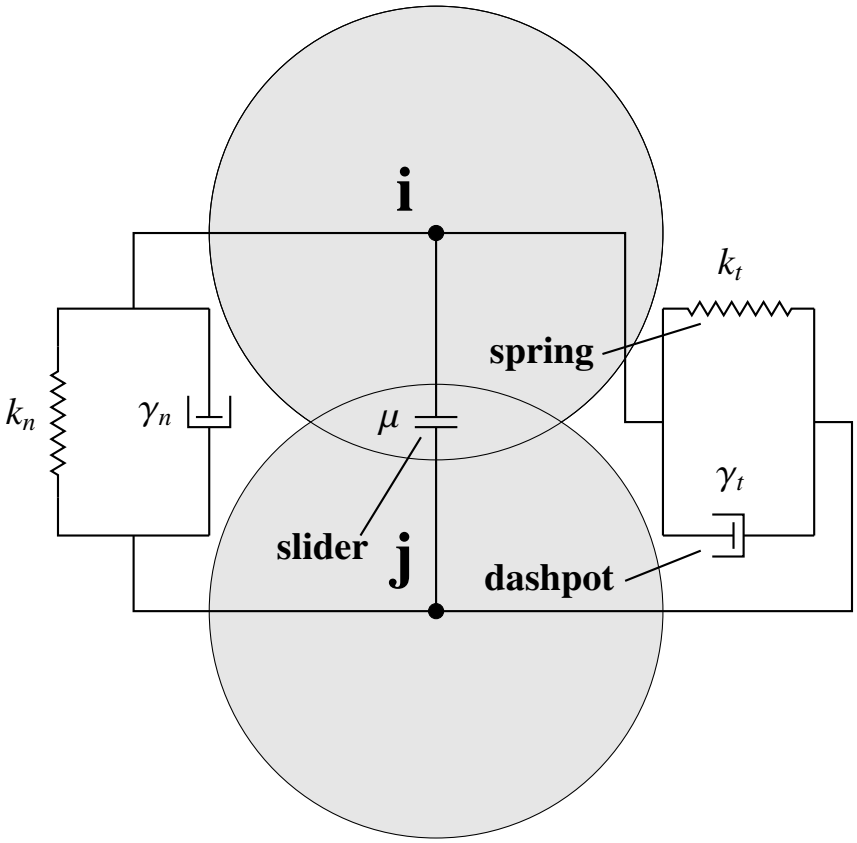


Figure 6: Spring-dashpot model.

The normal force is the simplest to calculate because the overlap depends only on the relative

[7] In this paper, the relative position and velocity between the particles indexed by "ij" are given in the form $\vec{r_i} - \vec{r_j}$ for the relative position and $\vec{v_i} - \vec{v_j}$ for the relative velocity.

[8] Note that this model does not take into account that in most cases, static friction has a higher coefficient of friction than kinetic friction.

position of particles i and j. To do this, we define the unit vector $\overrightarrow{e_{n_{ij}}}$ normal to the contact plane (it goes from j to i direction) and the scalar $\delta_{n_{ij}}$ representing the value of the overlap. Since we are working with spheres, the normal overlap is simply given by the difference between the sum of the radii of the spheres and the distance between their centers:

$$\delta_{n_{ij}} = R_i + R_j - \left\| \vec{r_i} - \vec{r_j} \right\|. \tag{38}$$

If $\delta_{n_{ij}} > 0$, the spheres are in contact. The unit vector for the normal direction is given by the normalized vector from j center to i center:

$$\overrightarrow{e_{n_{ij}}} = \frac{\vec{r_i} - \vec{r}_j}{\left\| \vec{r_i} - \vec{r_j} \right\|}. \tag{39}$$

The relation between $\delta_{n_{ij}}$ and $\overrightarrow{\delta_{n_{ij}}}$ is the following:

$$\delta_{n_{ij}} = -\overrightarrow{\delta_{n_{ij}}} \cdot \overrightarrow{e_{n_{ij}}}. \tag{40}$$

The normal relative velocity is given by:

$$v_{n_{ij}} = -(\vec{v_i} - \vec{v_j}) \cdot \overrightarrow{e_{n_{ij}}}, \tag{41}$$

where $\vec{v_i}$ and $\vec{v_j}$ are the velocities of the centers of mass of i and j. Therefore, the normal force $\vec{F_n}$ can be rewritten as $\vec{F_n} = F_n \cdot \overrightarrow{e_{n_{ij}}}$:

$$\vec{F_n} = \underbrace{\left(k_n\, \delta_{n_{ij}} + \gamma_n\, v_{n_{ij}}\right)}_{F_n} \cdot \overrightarrow{e_{n_{ij}}}. \tag{42}$$

The calculation of the tangential force $\vec{F_t}$ is slightly more complex because it does not only depend on the positions of the spheres at time a given time. Moreover, while in 2D it is straightforward to determine $\overrightarrow{e_{t_{ij}}}$ from $\overrightarrow{e_{n_{ij}}}$ [14] (there is only one possible direction), in 3D $\overrightarrow{e_{t_{ij}}}$ lies on a plane perpendicular to $\overrightarrow{e_{n_{ij}}}$, which we will refer to as the contact plane (see Figure 7).

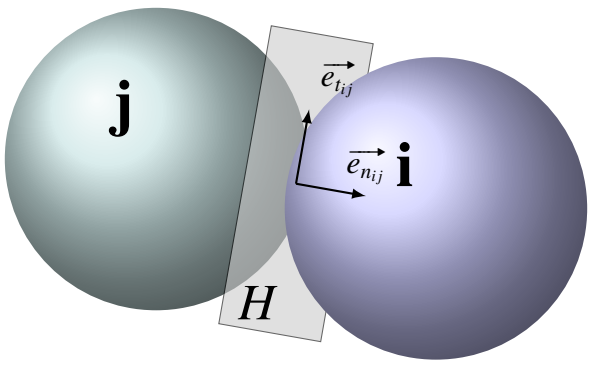
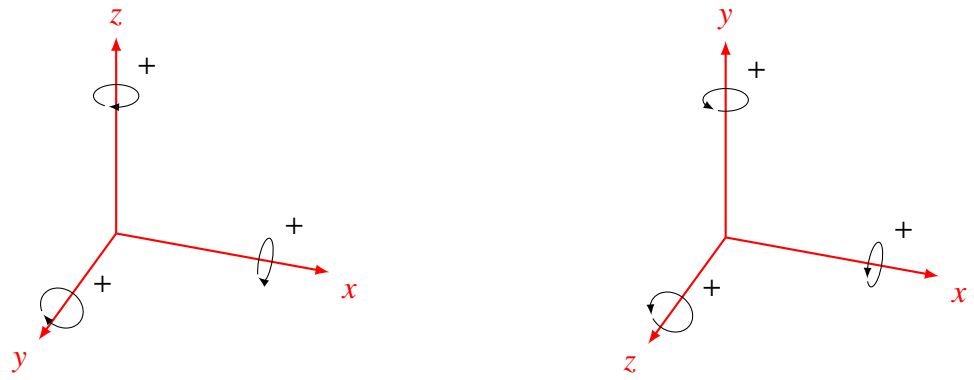


Figure 7: In 3D, $\vec{e_{t_{ij}}}$ can be anywhere on the contact plane H.

The tangential direction of the collision $\overrightarrow{e_{t_{ij}}}$ will not be explicitly computed. First we will compute the relative tangential velocity $\overrightarrow{v_{t_{ij}}}$. This corresponds to the relative velocity between the surfaces at the contact point projected on the contact surface. This is why we need to take into account the angular velocities of the spheres given by the vectors $\vec{\omega}_i$ and $\vec{\omega}_j$. Note that in this paper, we will use the right-hand coordinate system where the positive direction of rotation is in the counterclockwise (see Figure 8). With the right-handed coordinate system, the cross product, denoted by the symbol "×", is defined as follows:

$$\begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} \times \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} = \begin{bmatrix} a_y b_z - a_z b_y \\ a_z b_x - a_x b_z \\ a_x b_y - a_y b_x \end{bmatrix}. \tag{43}$$



(a) Left-handed coordinate system.

(b) Right-handed coordinate system.

Figure 8: The left-handed and the right-handed coordinate systems. The arrows indicate the direction of a positive rotation around the corresponding axis.

Taking into account the rotation of the spheres, the relative velocity between the surfaces of the spheres is given by:

$$\begin{aligned} \overrightarrow{v_{rel_{ij}}} &= (\vec{v}_i - \vec{v}_j) - R_i\vec{\omega}_i \times \overrightarrow{e_{n_{ij}}} - R_j\vec{\omega}_j \times \overrightarrow{e_{n_{ij}}} \\ &= (\vec{v}_i - \vec{v}_j) - (R_i\vec{\omega}_i + R_j\vec{\omega}_j) \times \overrightarrow{e_{n_{ij}}}. \end{aligned} \tag{44}$$

To get the tangential component if the relative velocity, we need to project the relative velocity on the contact plane, which is done by removing the normal component:

$$\begin{aligned} \overrightarrow{v_{t_{ij}}} &= \overrightarrow{v_{rel_{ij}}} - \overrightarrow{e_{n_{ij}}}(\overrightarrow{e_{n_{ij}}} \cdot \overrightarrow{v_{rel_{ij}}}) \\ &= \overrightarrow{v_{rel_{ij}}} + \overrightarrow{e_{n_{ij}}} v_{n_{ij}}. \end{aligned} \tag{45}$$

We will now investigate the computation of the tangential relative displacement $\overrightarrow{\delta_{t_{ij}}}$ in (35). Since the tangential displacement depends on the relative movement of the surfaces over time, we need to integrate the tangential relative velocity $v_{t_{ij}}$ over the duration of the collision:

$$\overrightarrow{\delta_{t_{ij}}}(t) = \int_{t_0}^{t} \overrightarrow{v_{t_{ij}}}(u)\, du, \tag{46}$$

where t_0 is the time at which the collision starts. We must therefore keep a "history" of the displacement between iterations because the collision may last several iterations. The method we will use to calculate this tangential displacement is the same as the one used in LIGGGHTS [16] and is detailed in the algorithm 3.4.

Algorithm 3.4: Computation of the tangential displacement over time

How can we do it in practice ?

- We need to keep in memory a 3D vector of the tangential displacement $\vec{\delta}_{t_{ij}}$ (represented in the following steps by the variable `tangHist`).

1. If there is no contact the vector `tangHist` is set to zero.
2. If there is a contact, we add $\vec{v}_{t_{ij}} \cdot \Delta t$ to the vector (Δt is the time step of the DEM simulation).
3. Since the previous contact planes are not necessarily the same, we need to re-project `tangHist` on the new contact plane:

$$\texttt{tangHist} = \texttt{tangHist} - \overrightarrow{e_{n_{ij}}}(\overrightarrow{e_{n_{ij}}} \cdot \texttt{tangHist})$$

4. If $\|k_t \cdot \texttt{tangHist}\| > \mu F_n$, the tangential force is limited by the Coulomb's law and `tangHist` is truncated according to the force μF_n and the elastic coefficient k_t:

$$\texttt{F\_max} = \mu \cdot F_n \cdot \texttt{tangHist}/\|\texttt{tangHist}\|$$
$$\texttt{tangHist} = -\texttt{F\_max}/k_t$$

Note: The contribution from the tangential velocity damping is not taken into account if the force is truncated.

We then use equation (35) to compute $\vec{F_t}$ and $\overrightarrow{F_{ij}}$. Thanks to Newton's third law, the force $\overrightarrow{F_{ij}}$ applied on i by j is opposed to the force of the same magnitude and opposite direction $\overrightarrow{F_{ji}}$:

$$\forall i, j \qquad \overrightarrow{F_{ij}} = -\overrightarrow{F_{ji}}. \tag{47}$$

The torque $\vec{M}_{ij}$ exerted by j on i can be simply obtained from the tangential force $\vec{F_t}$. This torque is given by:

$$\overrightarrow{M_{ij}} = -R_i \cdot \overrightarrow{e_{n_{ij}}} \times \overrightarrow{F_{t_{ij}}}. \tag{48}$$

Similarly, the torque exerted by j on i $\vec{M}_{ji}$ can also be calculated from $\overrightarrow{F_{t_{ij}}}$ (note that the sign remains the same).

$$\overrightarrow{M_{ji}} = -R_j \cdot \overrightarrow{e_{n_{ij}}} \times \overrightarrow{F_{t_{ij}}}. \tag{49}$$

In practice, it can be useful to store the torque divided by the radius of the spheres because this quantity is the same for both spheres involved in the collision.

3.3.2 Hertzian contact model

The elastic and damping coefficients $(k_n, k_t, \gamma_n, \gamma_t)$ can be obtained from the material properties of the solids [14]:

- The Young's modulus E

The Young's modulus represents the stiffness of a solid material (resistance to longitudinal deformation when a compressing or stretching force is applied) within the elastic limit of the material. Its SI unit is pascal (Pa).

- The Poisson's ratio υ

  The Poisson's ratio is the ratio of the lateral strain to the longitudinal strain when a force is applied on the material along the longitudinal axis. It describes how the solid material will expand when stretched and contract when compressed[9] within the elastic limit of the material.

- The shear modulus G

  The shear modulus measures the resistance of a solid material to shear deformation when a shearing force is applied. It is closely linked to the Young's modulus and the Poisson's ratio[10]. Its SI unit is pascal (Pa).

At the end of the 18th century, Heinrich Hertz was the first to derive formulas that link the deformation to the force exerted between two spheres [18]. The following formulas allow obtaining the damping and spring coefficients from the physical properties of the materials of the spheres i and j [14, 19][11]:

$$k_n = \frac{4}{3} Y^* \sqrt{R^* \delta_n}, \qquad k_t = 8 G^* \sqrt{R^* \delta_n}, \tag{50}$$

$$\gamma_n = -2 \sqrt{\frac{5}{6}} \beta \sqrt{S_n m^*} \geq 0, \qquad \gamma_t = -2 \sqrt{\frac{5}{6}} \beta \sqrt{S_t m^*} \geq 0, \tag{51}$$

[9] Note that for some materials it is possible that a quantity like the Poisson's ratio depends on the direction of the applied force. In our case, we will be working with isotropic materials for which these quantities are independent of the direction.

[10] The relation between the shear modulus, the Young's modulus and the Poisson's ratio for isotropic materials is given by $E = 2G(1 + \upsilon)$ [17].

[11] We have chosen to keep the same notation as in the LIGGGTHS documentation [16].

where [20, 16]:

$$S_n = 2E^* \sqrt{R^* \delta_n} \tag{52}$$

$$S_t = 8G^* \sqrt{R^* \delta_n} \tag{53}$$

$$\beta = \frac{\ln(\epsilon)}{\sqrt{\ln^2(\epsilon) + \pi^2}} \tag{54}$$

$$Y^* = \left(\frac{1 - \upsilon_i^2}{E_i} + \frac{1 - \upsilon_j^2}{E_j} \right)^{-1} \tag{55}$$

$$G^* = \left(\frac{2(2 - \upsilon_i)(1 + \upsilon_i)}{E_i} + \frac{2(2 - \upsilon_j)(1 + \upsilon_j)}{E_j} \right)^{-1} \tag{56}$$

$$R^* = \left(\frac{1}{R_i} + \frac{1}{R_j} \right)^{-1} \tag{57}$$

$$m^* = \left(\frac{1}{m_i} + \frac{1}{m_j} \right)^{-1} \tag{58}$$

where δ_n is the distance between the surfaces of the spheres given by equation (38), E_i and E_j are the Young's modulus of the spheres, υ_i and υ_j are the Poisson's ratio, m_i and m_j are the masses, and R_i and R_j are the radii. The parameter ϵ is the coefficient of restitution of the collision. Formally, it corresponds to the ratio of relative velocity of the spheres after the collision to the relative velocity of the spheres before the collision. The coefficient of restitution is within the interval $[0, 1]$. When it equals 1, the collision is perfectly elastic, and the damping terms γ_n and γ_t are zero. When epsilon is smaller than 1, energy loss occurs during the collision, and it is no longer perfectly elastic. In our model, the coefficient of restitution is a fixed and velocity independent parameter. It can only be taken within the interval $(0, 1]$ which means that it does not allow for a coefficient of restitution equal to 0 (completely inelastic collision), but we can approach it by choosing a small value for ϵ. In practice, the coefficient of restitution is often chosen somewhat arbitrarily and serves primarily as a meta-parameter that can be finely tuned to make the simulation behavior align with our expectations[12]. If we desire to simulate collisions between solids that dissipate a portion of the collision energy differently, we can consider using different coefficients of restitution for each particle and simply take the arithmetic mean of the coefficients of the two solids during an interaction [14].

3.3.3 Interactions with static obstacles

It is often convenient to introduce fixed obstacles into the simulation. In our case we will introduce two types of obstacle:

1. A plane
2. A cylinder

[12]Note that in this model, we use the same coefficient of restitution for both the normal and tangential damping of the collision. It is entirely possible to use two different coefficients to represent more complex interactions [20].

Like the other solids, each of these obstacles will have physical properties (Young's modulus, Poisson ration, etc.). Although these obstacles are fixed in space, a plane obstacle is allowed to have a velocity collinear to its plane and a cylinder can have a velocity in the direction along its main axis as well as an angular velocity around this axis.

Interaction with a plane

Let H_j be a plane. A point (x, y, z) lies in the plane H_j if it satisfies the following equation:

$$ax + by + cz + d = 0, \tag{59}$$

where $\vec{n} = \begin{bmatrix} a \\ b \\ c \end{bmatrix}$ is a vector normal to the plane and d is a constant.
In order to simplify the computations, we impose that $\|\vec{n}\| = 1$. Let i be a sphere with position $\vec{r_i}$, radius R_i, velocity $\vec{v_i}$ and angular velocity ω_i. We want to find the distance between i and the plane to compute the force exerted by the plane on the sphere.

The "signed distance" D can be obtained by projecting $\vec{r}_i$ onto $\vec{n}$ and move it by d:

$$D = \vec{n} \cdot \vec{r}_i + d. \tag{60}$$

If D is positive, the sphere i is on the side of the plane given by the direction of $\vec{n}$. Otherwise it is on the other side. Thus, the normal overlap $\delta_{n_{ij}}$ is simply given by:

$$\delta_{n_{ij}} = R_i - |D|, \tag{61}$$

and the unit vector for the collision plane is given by:

$$\overrightarrow{e_{n_{ij}}} = \text{sgn}(D) \cdot \vec{n}. \tag{62}$$

The calculation of the force and torque exerted on i by the wall is then exactly the same as for the collision between two solids. It is simply necessary to consider that the obstacle has an infinite mass and an infinite radius in the formulas (57) and (58):

$$R^* = R_i, \tag{63}$$

$$m^* = m_i. \tag{64}$$

Interaction with an infinite cylinder

Let C_j be an infinite cylinder with a radius R_j which is centered on a line L_j that passes through the point $\vec{p} = \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix}$ and is collinear to the vector $\vec{d} = \begin{bmatrix} a \\ b \\ c \end{bmatrix}$. Again, we impose that $\left\|\vec{d}\right\| = 1$. The parametric equations of this line are, $\forall\, t \in \mathbb{R}$:

$$\begin{cases} x = & at + p_x \\ y = & bt + p_y \\ z = & ct + p_z \end{cases} \cdot \tag{65}$$

Let i be a sphere with position $\vec{r} = \vec{r_i}$, radius R_i and velocity $\vec{v_i}$. We want to find the distance between i and the surface of the cylinder to compute the force exerted on the sphere. Let $\vec{r'}$ be to closest point to $\vec{r_i}$ on the directional line L_j of the cylinder. Since $\vec{r'}$ lies on the line L_j, we have that $\exists\, t \in \mathbb{R}$ such that:

$$\vec{r'} = \vec{d} \cdot t + \vec{p}. \tag{66}$$

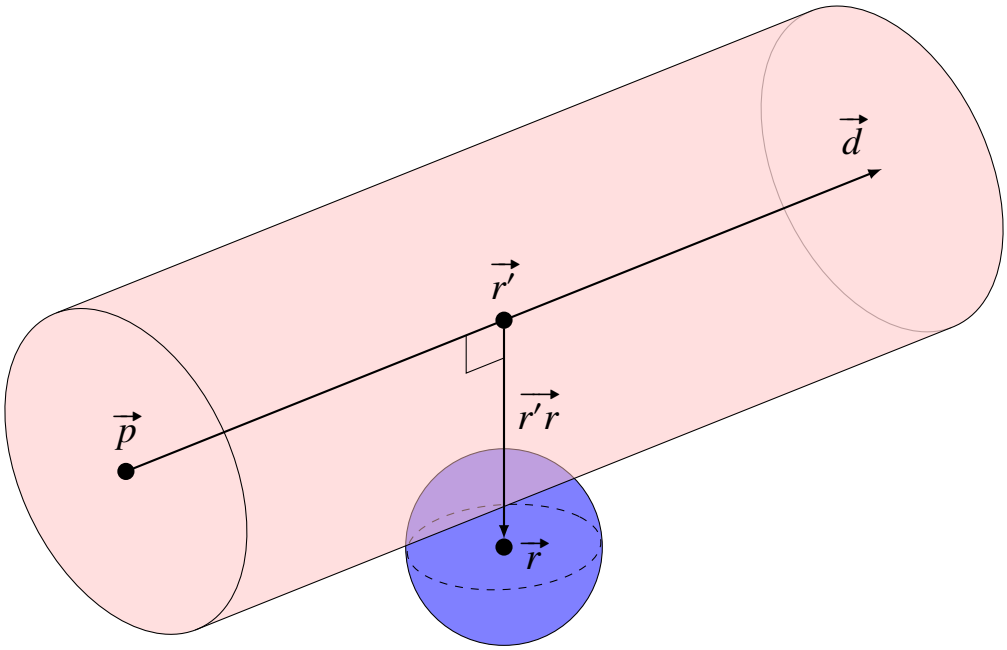


Figure 9: Distance to the cylinder center.

The vector $\vec{r'r} = \vec{r} - \vec{r'}$ is given by:

$$\vec{r'r} = \vec{r} - (\vec{d} \cdot t + \vec{p}) = \vec{r} - \vec{d} \cdot t - \vec{p}. \tag{67}$$

Since $\overrightarrow{r'r} \cdot \vec{d} = 0$, we have that:

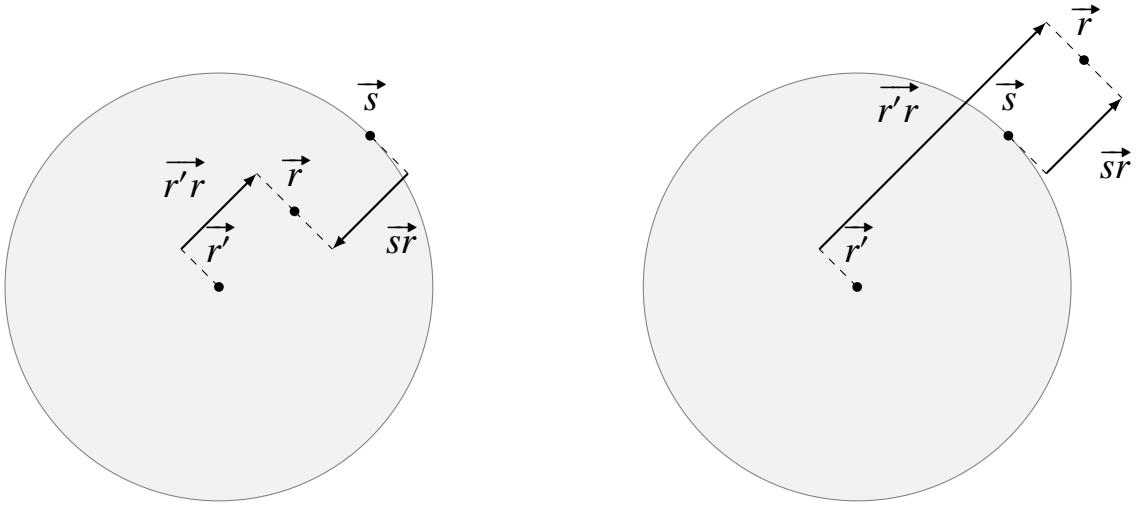
$$\begin{aligned} t &= \vec{d} \cdot (\vec{r} - \vec{p}) \\ \Rightarrow \overrightarrow{r'r} &= \vec{d} \cdot (\vec{d} \cdot (\vec{r} - \vec{p})) + \vec{r} - \vec{p}. \end{aligned} \tag{68}$$

Let $\vec{s}$ be the closest point to $\vec{r}$ on the surface of the cylinder. The vector $\overrightarrow{r's}$ that goes from $\vec{r'}$ to $\vec{s}$ is given by:

$$\overrightarrow{r's} = R_j \cdot \frac{\overrightarrow{r'r}}{\left\|\overrightarrow{r'r}\right\|}. \tag{69}$$

Then, the vector $\overrightarrow{sr}$ that goes from $\vec{s}$ to $\vec{r}$ is given by:

$$\begin{aligned} \overrightarrow{sr} &= \overrightarrow{r'r} - \overrightarrow{r's} \\ &= \overrightarrow{r'r} \cdot \left(1 - \frac{R_j}{\left\|\overrightarrow{r'r}\right\|}\right). \end{aligned} \tag{70}$$



(a) The sphere position $\vec{r}$ is inside the cylinder.

(b) The sphere position $\vec{r}$ is outside the cylinder.

Figure 10: The two possible collision configurations between a sphere and a cylinder.

Thus, the normal overlap $\delta_{n_{ij}}$ is simply given by:

$$\delta_{n_{ij}} = R_i - |\overrightarrow{sr}|, \tag{71}$$

and the unit vector for the collision plane is given by:

$$\overrightarrow{e_{n_{ij}}} = \frac{\overrightarrow{sr}}{\|\overrightarrow{sr}\|}. \tag{72}$$

Since the cylinder can have a velocity in the direction $\vec{d}$ and an angular velocity around its main

axis, we have to take it into account when we compute the relative velocity between the surface of i and j at the contact point. However, we cannot directly use the formula (44) because $\overrightarrow{r's}$ does not necessarily have the same direction as $\overrightarrow{e_{n_{ij}}}$ depending on whether the sphere is inside or outside the cylinder. The relative velocity between the surfaces is therefore:

$$\vec{v}_{rel_{ij}} = (\vec{v}_i - \vec{v}_j) - R_i\vec{\omega}_i \times \overrightarrow{e_{n_{ij}}} - \vec{\omega}_j \times \overrightarrow{r's}. \tag{73}$$

As with the plane, the calculation of the force exerted on i is exactly the same as for the collision between two solids. Again, we consider that the obstacle has an infinite mass and an infinite radius[13] (equations (63) and (64)).

To compute the torque exerted by the sphere i on the cylinder j, we cannot simply use the formula (49) since the direction of the torque depends on whether the collision takes place inside the cylinder or outside the cylinder. The formula is therefore:

$$\vec{M}_{ji} = -\overrightarrow{r's} \times \vec{F}_t. \tag{74}$$

3.3.4 Collision detection

The simplest way to detect collisions between solids is to iterate over all possible pairs of solids using a double loop and check if the overlap given by the formula (38) is positive. However, the time complexity of such an algorithm would be on the order of $O(N^2)$, where N is the number of spheres. Since the simulations we are performing typically involve a large number of particles, we need a method that can reduce the time complexity for collision detection and that can work in parallel. For this purpose, we have chosen to use a uniform grid that divides the domain into cubic cells. Each cell contains the identifiers of the solids that are inside it. Therefore, there is a potential collision between two solids only if these two solids are contained in the same cell. We also need to know the cells in which the solids are contained in order to update their positions in the uniform grid at each iteration. For efficiency reasons, we will not directly work with the exact shape of the solids. Instead, we will use the axis-aligned bounding box (AABB) that encloses the solid. This approach allows us to store only the maximum and minimum coordinates of the cells that contain the solid, avoiding the need to work with variable-sized lists.

[13] We dot not use the radius of the cylinder R_j to compute the effective radius in (57) because the contact surface between a cylinder and a sphere is not trivial. We assume that the radius is large enough so that the collision between the sphere and the cylinder is equivalent to the collision between the sphere and a plane tangent to the cylinder surface at the contact point.

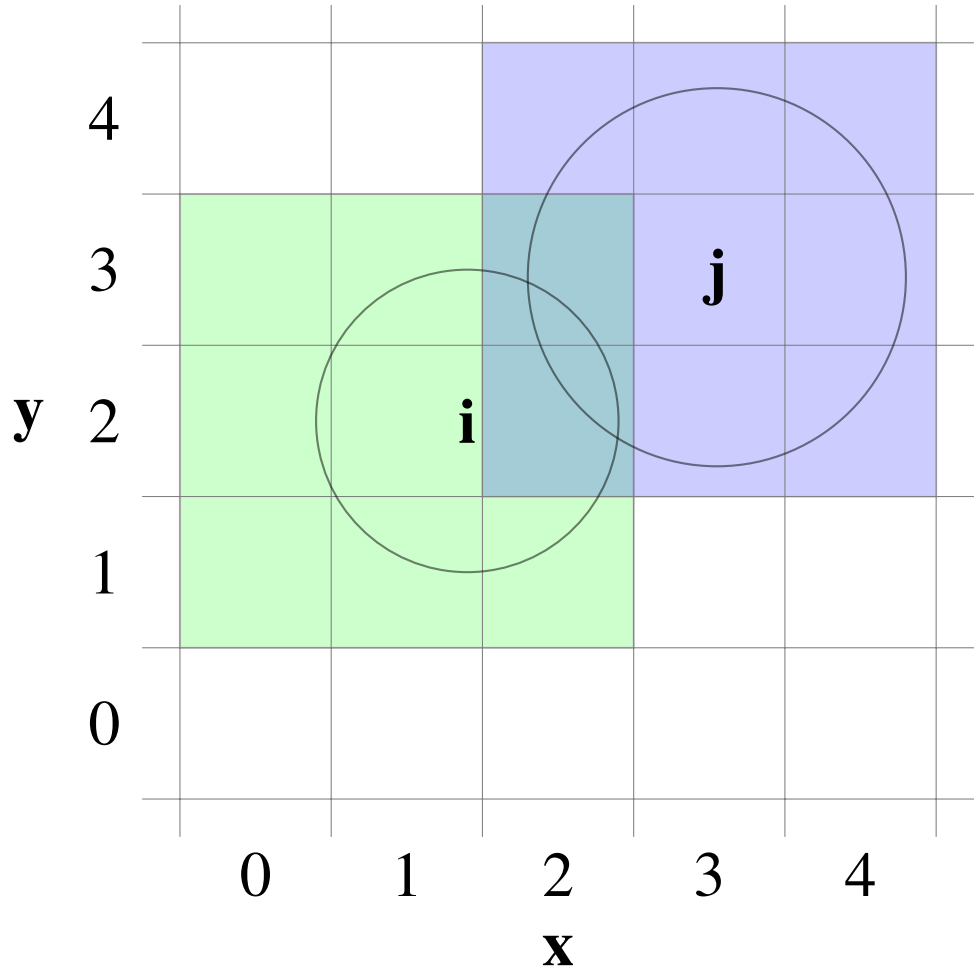


Figure 11: Example for the uniform grid in 2D. The spheres i and j are both in the cells $(2, 2)$ and $(2, 3)$. The sphere i is not physically present in the cells $(0, 1)$ and $(0, 3)$ but since we use an axis-aligned bounding box and only keep in memory the maximum and minimum coordinates of the sphere in the grid, i will be considered to be inside those cells during the collision detection step.

Our approach to compute the minimum and maximum coordinates of solids in the uniform grid only works for solids such as spheres, walls will be processed separately. For each dimension, we determine the lower bound by subtracting the radius from the position of the center of the sphere and multiplying it by the ratio of the size of the grid to the size of the domain. The upper bound is then calculated as the lower bound plus twice the radius multiplied by the distance ratio. In the case of a periodic domain, we ensure that the lower bound remains within the domain. The upper bound is updated using the same method as before and may extend outside the domain. Finally, the obtained values are truncated to obtain the corresponding cells in the domain.

Algorithm 3.5: Computation of the minimum and maximum coordinates

Data
gridSize: the size of the grid for each dimension
domainSize: the size of the domain for each dimension
position: the position of the sphere in the domain
radius: the radius of the sphere

▷ *compute the lower bound* ◁
ratio = gridSize[i]/domainSize[i]
for $i = 0, 1, 2$ **do** ▷ *iterate over the 3 dimensions*
    lowerBound = (position[i] - radius) * ratio
    lowerBound = lowerBound mod gridSize[i] ▷ *mathematical modulo*
    upperBound = lowerBound + 2 * radius * ratio
    lowerBound = int(lowerBound)
    upperBound = int(upperBound)

Adding solids

Since we know how to compute the minimum and maximum coordinates of a sphere in the uniform grid, the way we add it to the simulation is straightforward. First, we compute the minimum and maximum coordinates of the sphere using its radius and position. Then, we iterate over the cells within this bounding box and add the sphere to each respective cell. This process is performed only once at the beginning of the simulation. On the other hand, walls in the simulation are big and static and require a different approach. There are no minimum or maximum coordinates associated with walls. Instead, we iterate over all the cells of the domain and test if each cell intersects with the surface of the wall. This is determined based on the vector distance between the wall surface and the center of the cell, which can be computed using the appropriate equations ((60) and (70)). The surface of the wall is inside the cell if, for each dimension, the absolute value of the corresponding component of the the vector distance is smaller than half the size of a cell (the size of a cell for each dimension is given by the size of the domain in this dimension divided by the number of cells in the same dimension): `abs(distance_vect[i]) < cellSize[i] * 0.5`. To efficiently identify the cells containing the wall, we can perform this filtering step in parallel. However, similar to adding spheres, the process of adding walls is also performed only once at the beginning of the simulation, and therefore has no significant impact on the overall performance of the simulation.

Computation of the collision pairs

From the list of solids in each cell, we can easily calculate the list of potential collisions within each cell. To avoid redundancy, we will only keep the collisions (i, j) where $i < j$. If a cell contains N solids, there are potentially $\frac{N(N-1)}{2}$ collisions in the cell[14]. To obtain the final list of collisions, we simply sort the collisions in lexicographic order and remove duplicates (although in practice,

[14] this can be used to allocate memory for the collision array. If we have N_c cells and a maximum of $N_{s_{max}}$ solids per cell, there will be a maximum of $N_c \frac{N_{s_{max}}(N_{s_{max}}-1)}{2}$ potential collisions.

removing duplicates is not necessary because they can be easily detected later since the list will be sorted). Sorting this list of collisions will be necessary for the following operations. The sorting operation is the basis of many parallel algorithms. Its advantage is that it can be easily performed in parallel using the well-known radix sort algorithm [21], which is notably used to perform sorting operations on GPUs (for parallel sorting in C++, see the section 3.5 on C++17 parallel algorithms). The pseudocode for the collision detection is provided in algorithm 3.6 and an example is given in Figure 12.

Algorithm 3.6: Computation of the collision pairs

Data
solidsInCells: the list of solids in the cells
NbSolidsInCells: the number of solids inside each cells
collisionPairs: the collisions pairs
NCells: the number of cells
NmaxSolids: maximum number of solids inside a cell
NmaxColl = $\frac{\text{NmaxSolids}(\text{NmaxSolids}-1)}{2}$: maximum number of collisions inside a cell

▷ *iterate over the cells, the first loop can be paralellized* ◁
for cellIndex = $0, \ldots,$ NCells $-$ 1 **do**
    collisionIndex = cellIndex $\cdot$ NmaxColl
    for $i = 0, \ldots,$ NbSolidsInCells[cellIndex] $-$ 1 **do**
        solidI = solidsInCells[cellIndex][i]
        for $j = i + 1, \ldots,$ NbSolidsInCells[cellIndex] $-$ 1 **do**
            solidJ = solidsInCells[cellIndex][j]
            if solidI < solidJ **then**
                collisionPairs[collisionIndex] = (solidI, solidJ)
            else
                collisionPairs[collisionIndex] = (solidJ, solidI)
            collisionIndex++
▷ *sort and remove the duplicates in the list of collisions. This can be parallelized* ◁
collisionPairs = removeDuplicates(sort(collisionPairs))

c_0	c_1	c_2	c_3	c_4
A	B	A	G	A
B	D	E		B
C		F		
		G		

(a) The solids inside each cells.

c_0	c_1	c_2	c_3	c_4
(A,B)	(B,D)	(A,E)		(A,B)
(A,C)		(A,F)		
(B,C)		(A,G)		
		(E,F)		
		(E,G)		
		(F,G)		

(b) The potential collisions in each cells.

(A,B)	(E,G)			
(A,C)	(F,G)			
(A,E)				
(A,F)				
(A,G)				
(B,C)				
(E,F)				

(c) The sorted and duplicate-free list of collisions.

Figure 12: Example for the computation of the collision pairs. Here solids are represented by letters instead of numbers for better readability.

3.3.5 Moving the particles in the uniform grid

In sequential, the process is rather easy: we simply iterate over the particles and we replace them in the corresponding cells according to the new bounding box. In parallel, we have a problem of race condition because we cannot update multiple cells at the same time. The solution is to add a constraint on the particles velocities. We suppose that the particles cannot move more than the size of one cell in each direction xyz at each iteration. The advantage of using this constraint is that the particles entering a cell were necessary present in the neighboring cells at the last iteration or were already inside the cell. The idea is thus to iterate over the cells instead of the solids and to compute the new content of the cell based on the solids present in the neighboring cells at the last iteration. To do this, we use two grid structures that we will swap at each iteration:

- One containing the cells states at the last iteration that we will not modify.

- Another one that will contain the new arrangement of particles in the cells (in which we will be writing to).

Before iterating over the cells, we first need to compute the new minimum and maximum coordinates for each sphere in the simulation according to their new positions (of course, this can be done in parallel). The loop over the cells is performed in parallel. For each cell, we first need to clear the content of the current cell in the new grid that will receive the new data (In practice, this is very fast because we only need to set the number of elements in the cell to zero). Then, we need to iterate over the solids contained in the current cell and in the neighboring cells. If the solid is inside the cell, we write the particle id in the same cell of the new grid, otherwise we do nothing. We can easily determine if the solid is inside the cell by using the minimum and maximum coordinates. Since the walls are static in the domain, we do not need to move them. We simply need to copy their id's into the new cell if they were already present in the current cell.

Algorithm 3.7: Moving the particles in the uniform grid

```
Data
    newGrid: the list of solids in the cells
    OldGrid: the list of solids in the cells from the last iteration
    NCells: the number of cells

updateBoundingBoxes() ▷ update in parallel the minimum and maximum coordinates of the cells containing the solids
▷ iterate over the cells, the first loop can be paralellized ◁
for cellIndex = 0, ..., NCells − 1 do
    clearCell(newGrid, cellIndex) ▷ clear the cell in the new grid
    neighboringCells = getNeighboringCells(cellIndex) ▷ indices of the neighboring cells, including the current one
    for all neighboringCell in neighboringCells do
        solids = getSolids(oldGrid, neighboringCell)
        for all solid in solids do
            min, max = getBoundingBox(solid)
            ▷ static solid do not move ◁
            if isStatic(solid) then
                ▷ add it to the cell if not already inside ◁
                if cellIndex == neighboringCell then
                    addToCell(newGrid, cellIndex, solid) ▷ add the solid to the cell
                ▷ if the solid was not already added and if the cell is in the bounding box of the solid ◁
            else if solid not in getSolids(newGrid, cellIndex) AND isCellInBoundingBox(solid, min, max) then
                addToCell(newGrid, cellIndex, solid) ▷ add the solid to the cell
```

3.3.6 Recovering the collision history (tangential displacement)

We already said that a collision lasts generally more than one iteration and that we need to keep in memory this result. The new list of collisions may not be the same, so it is necessary to reconstruct the list of tangential displacements associated with these new collision pairs. To do this, we will keep in memory the collision pairs and the collision history at the previous iteration. We will also have a list of ranges for each solid i that indicates the indices of the first and the last collision pairs in which

the solid i is in the left part (since the collisions are sorted in lexicographic order, all the collision pairs where i is the first element of the pair are contiguous in the list of collisions). This list of ranges will be computed in parallel when we iterate over the collision pairs to compute the associated forces and torques. To compute it, we simply need to check if the first element of the previous collision pair is different than the current one to compute the index of the first collision and to check if the first element of the next collision pair is different than the current one for the last collision index. If a collision is new, we want to set its tangential displacement to zero and if the collision already existed at the last iteration, we set the new tangential displacement to this value. In fact, we only copy the value, the tangential displacement will be truly updated during the computation of the forces. To obtain the new collision history, we iterate in parallel over the new collision pairs. Let (i, j) be the current pair. Since we have the collision pairs form the previous iteration, we simply perform a binary search. We do not perform the search on the full list because we know the range of the collisions from the previous iteration where i is the left element of the pair. Thus, we can restrict the search for the current pair (i, j) within this range. If the pair (i, j) is found, we set the new tangential history to the previous value, otherwise we set it to zero. The pseudocode for this algorithm is the following:

Algorithm 3.8: Recovering the collision history

```
Data
    collisionPairsOld: the collisions pairs at the previous iteration
    collisionPairsNew: the new collisions pairs
    tangentialDisplacementOld: the list of tangential displacements at the previous iteration
    tangentialDisplacementNew: the new tangential displacements
    rangesOld: the ranges of collisions for each solid at the previous iteration

▷ iterate over the new collision pairs and find the old collision pairs, this can be done in parallel ◁
for all collisionPair in collisionPairsNew do
    collisionIndexNew = getCurrentIndex() ▷ index of the new collision
    i, j = collisionPair
    ▷ get the range for the solid i ◁
    begin, end = rangesOld[i]
    ▷ test if the range exists ◁
    if begin == rangeDefaultValue) then ▷ default value indicating that the range is empty
        ▷ if the range does not exist, we set the tangential displacement to 0 ◁
        tangentialDisplacementNew[collisionIndexNew] = 0⃗
    else
        ▷ use a binary search to find the collision pair in the old collision pairs according to the range ◁
        collisionIndexOld = binarySearchInOld((i, j), begin, end)
        ▷ if the collision pair is found, we copy the old tangential displacement ◁
        if isValid(collisionIndexOld) then
            tangentialDisplacementNew[collisionIndexNew] = TangentialDisplacementOld[collisionIndexOld]
        else
            ▷ if the collision pair is not found, we set the tangential displacement to 0 ◁
            tangentialDisplacementNew[collisionIndexNew] = 0⃗
```

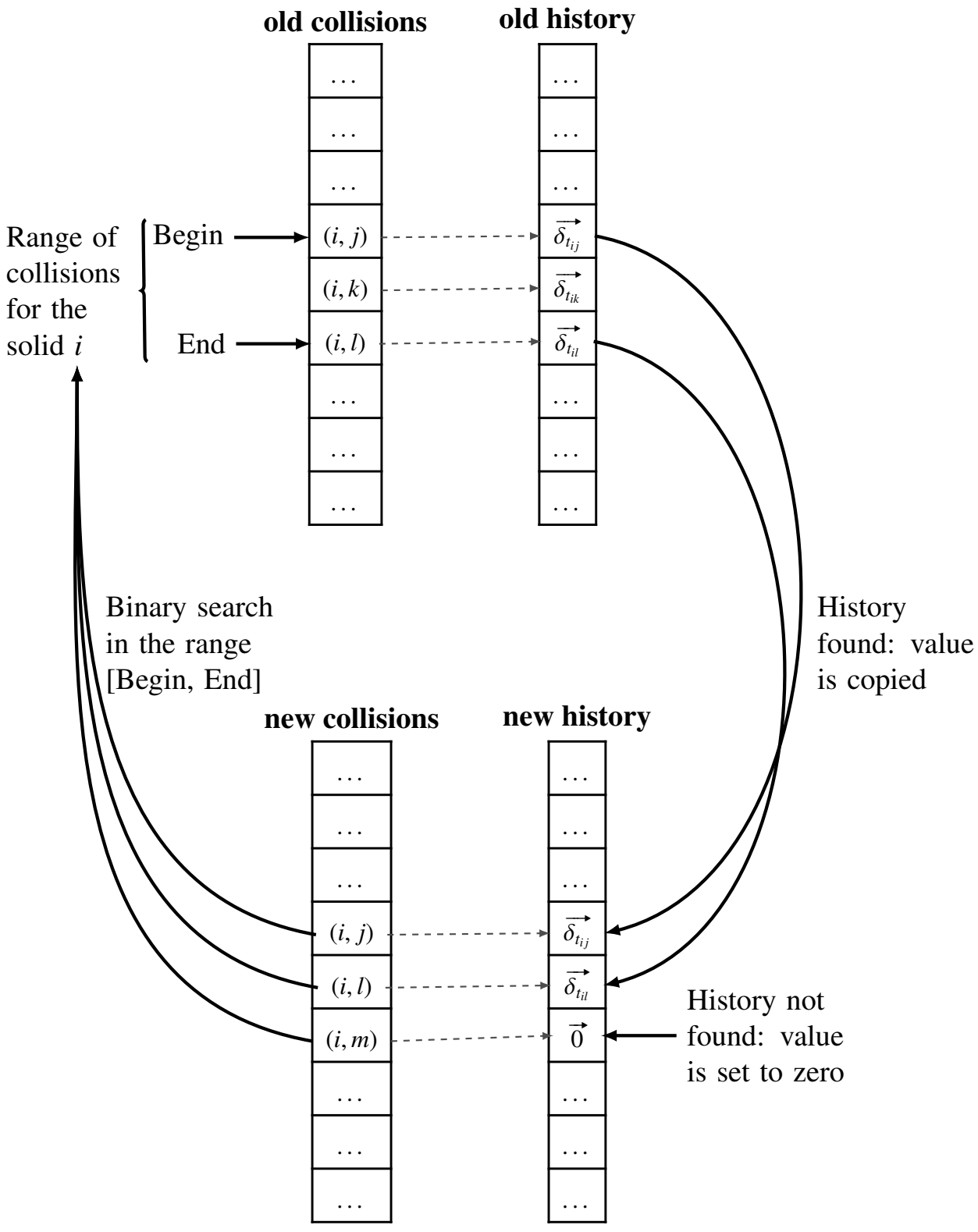


Figure 13: Example for the algorithm that recovers the tangential displacement.

3.3.7 Computation of the forces

The computation of the forces is straightforward: we iterate over the collisions and calculate the corresponding forces and torques using the formulas from the last section. We also update the as-

sociated tangential displacement during this step. This calculation has the advantage of being fully parallelizable since each collision is considered independently. For collisions between two spheres, we compute the torque divided by the radius since this value is the same for both solids. However, in the case of a collision between a sphere and a cylinder, the direction of the torque may vary depending on whether the collision occurs inside or outside the cylinder (see Figure 10). To address this issue, we define two distinct objects: an inner cylinder and an outer cylinder. By doing so, we only need to store the torque on the sphere divided by its radius. The torque on the cylinder will be updated later based on whether it is an outer or inner cylinder object. This approach allows us to simplify the calculations and manage the torque correctly in different collision scenarios. Of course, we do not have this issue if we do not store the torque exerted on the cylinder in memory. Updating the forces on particles in parallel can be more complex as a solid can be involved in multiple collisions. To handle this, we use the fact that collisions are sorted in lexicographic order (and consequently the associated lists of forces and torques). Indeed, The forces and torques where a solid i is in the left part of the collisions pairs are contiguous in memory due to the way the collision list is sorted. We already computed a list of ranges for each solid i that indicates the indices of the first and the last collision pairs in which the solid i is in the left part. To update the forces in parallel without race conditions, all the forces and torques applied to a solid appearing in the left part of the collision pairs are summed in the same kernel (the indices of the collision pairs are obtained from the list of ranges) and the result is used to update the values of force and torque on that solid. During the force computation, we can also construct a list of pairs $(j, collisionIndex)$ that maps the indices of the collisions to the index of the second solid in the collision pairs. This list needs to be sorted using a parallel sorting algorithm in lexicographic order. From this sorted list, we can then compute the list of ranges in parallel and we can easily update the forces and torques exerted on the second solids, similar to what was done for the solids in the first position of the pairs. This approach ensures that the forces and torques on the solids involved in the collisions are properly updated without race condition. The disadvantage is that the forces and torques summed by one thread are not contiguous in memory, but we chose this approach because it was quicker than sorting the forces and torques according to the second solids in the collision pairs (see Figure 14). The pseudocode for this algorithm is the following:

Algorithm 3.9: Applying the forces and the torques

Data

collisionPairsRangesFirstSolid: Range of collision in which a solid in the first position of a pair is involved. This is computed during the computation of the forces
collisionPairsRangesSecondSolid: Range of collision in which a solid in the first position of a pair is involved. This list will be computed during this step
collisionPairsNew: the new collisions pairs
collisionForce: the forces associated to a collision pairs
collisionTorqueR: the torques divided by the radius associated to a collision pairs
solidForce: the forces applied on the solids
solidTorqueR: the torques divided by the radius applied on the solids
secondSolidMap: list of pairs (j, c) that maps the id of a solid to the index of a collision pair c where j is in the second position of the collision pair. This list is constructed during the computation of the forces without being sorted
nbSolids: number of solids
nbCollisions: number of collisions

```
▹ apply the forces on the first solid in the collision. This loop can be run in parallel ◃
for i = 0, ..., nbSolids-1 do
    ▹ get the range ◃
    begin, end = collisionPairsRangesFirstSolid[i]
    ▹ test is the range exists ◃
    if begin != rangeDefault then
        ▹ If the range is valid, we sum the forces and the torques applied on the solid ◃
        for idx = begin, ..., end do
            ▹ apply the forces and torques on the first solid ◃
            solidForce[i] += collisionForce[idx]
            solidTorqueR[i] += collisionTorqueR[idx]
▹ sort according to the second solid in the collision. This can be done in parallel ◃
sort(secondSolidMap)
▹ compute the ranges for the second solid in the collision. This loop can be parallelized ◃
for idx = 0, ..., nbCollisions-1 do
    j, collisionIndex = secondSolidMap[idx]
    ▹ compute the range (according to the second solid in the collision) ◃
    ▹ if the object ID of the previous force is different than j (or if it is the first force), then we have the
      beginning of the range ◃
    if idx == 0 OR secondSolidMap[idx-1][0] != j then
        collisionPairsRangesSecondSolid[j][0] = idx
    ▹ if the object ID of the next force is different than j (or if it is the last force), then we have the end of the
      range ◃
    if idx == nbCollisions-1 OR secondSolidMap[idx+1][0] != j then
        collisionPairsRangesSecondSolid[j][1] = idx
▹ apply the forces on the second solid in the collision, this loop can be parallelized ◃
for j = 0, ..., nbSolids-1 do
    ▹ get the range ◃
    begin, end = collisionPairsRangesFirstSolid[j]
    ▹ test is the range exists ◃
    if begin != rangeDefault then
        ▹ If the range is valid, we sum the forces and the torques applied on the solid ◃
        for idx = begin, ..., end do
            ▹ apply the forces and torques on the first solid ◃
            solidForce[j] -= collisionForce[secondSolidMap[idx][1]] ▹ Sign of the force is reversed
            solidTorqueR[j] += collisionTorqueR[secondSolidMap[idx][1]] ▹ the sign of the torque is the same
              than for the first solid
    ▹ reset the ranges ◃
    collisionPairsRangesFirstSolid[j][0] = rangeDefault
    collisionPairsRangesFirstSolid[j][1] = rangeDefault
```

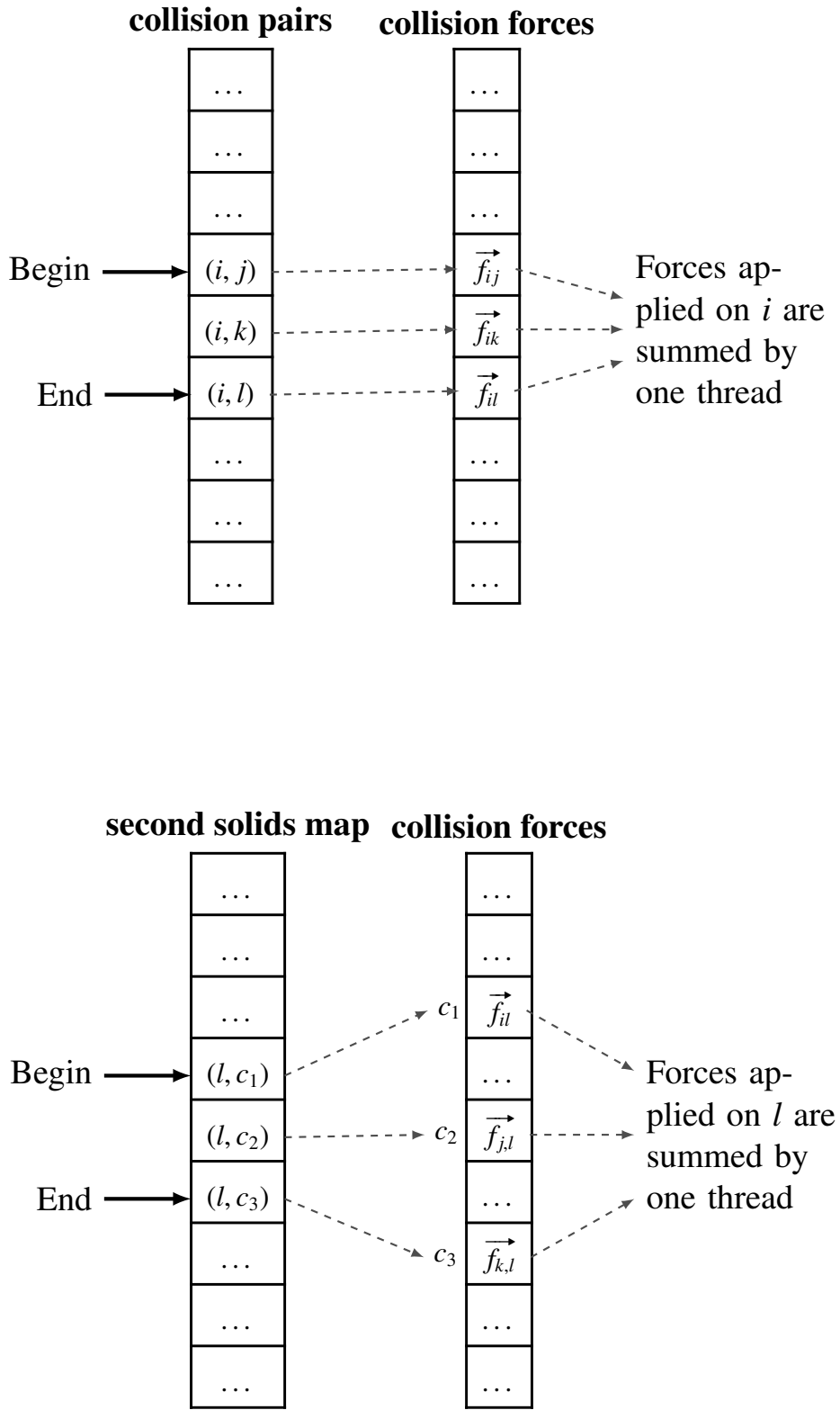


Figure 14: Example for the summation of the forces applied on one solid. The first case corresponds to the sum of the forces applied on a solid appearing in the first position of pairs and the second case corresponds to the sum of the forces applied on a solid appearing in the second position of pairs. The approach is similar for the torques.

The problem with this approach is that if one solid is involved in a large number of collisions compared to others, it can significantly slow down the simulation as we have to wait for one thread to process all its forces while other threads have finished their computation. This is particularly noticeable with walls, which are involved in numerous collisions. One solution would be to treat the walls separately, utilizing all threads to sum the forces on a wall. Another solution is simply to ignore the forces exerted by spheres on a wall (the wall forces on spheres are still considered). This is the solution we use in our tests when we do not need the forces exerted on the walls.

3.3.8 Integration scheme

The movement of a sphere i within the system is governed by the following equations of motion:

$$\vec{F_i} = m_i\vec{a_i} = m_i\frac{\mathrm{d}\vec{v_i}}{\mathrm{d}t} = \frac{\mathrm{d}^2\vec{r_i}}{\mathrm{d}t^2}, \tag{75}$$

$$\vec{M_i} = I_i\vec{\alpha_i} = I_i\frac{\mathrm{d}\vec{\omega_i}}{\mathrm{d}t}, \tag{76}$$

where m_i is the mass of the sphere, $I_i = \frac{2}{5}R_i^2 m_i$ the moment of inertia of the sphere[15] and α_i is the angular acceleration. One the the most commonly used method to integrate the equations of motions in molecular dynamics is the velocity Verlet algorithm [22]. It is used to compute the new position and velocity of the solids in the simulation according to the forces applied on the solids. Its global error is $O(\Delta t^2)$ for the position and the velocity. The steps of the method are the following:

$$\vec{r}(t+\Delta t) = \vec{r}(t) + \vec{v}(t)\Delta t + \frac{1}{2}\vec{a}(t)\Delta t^2, \tag{77}$$

$$\vec{v}(t+\Delta t) = \vec{v}(t) + \frac{\vec{a}(t) + \vec{a}(t+\Delta t)}{2}\Delta t. \tag{78}$$

The angular velocity is updated in a similar way:

$$\vec{\omega}(t+\Delta t) = \vec{\omega}(t) + \frac{\vec{\alpha}(t) + \vec{\alpha}(t+\Delta t)}{2}\Delta t. \tag{79}$$

The advantage of this method is that we need to compute the forces on the particles only once at each iterations. Another category of methods that we can use are the so called predictor-corrector methods, such as the Gear's fifth order method [19, 14]. These methods are based on two steps: one step where we predict the position and its higher order derivatives based on the current information that we have and another step where we correct these predictions based on the value of the computed acceleration (note that the forces are still computed once at each step). These methods are known to be more accurate and less sensitive to potential oscillation problems in the system. However, they require keeping track of multiple higher-order derivatives of the position. We will not provide further

[15] Since we are working with spheres, the moment of inertia is a scalar and we do not need to keep track of the orientations of the spheres. With more complex shapes, the moment of inertia becomes a tensor, and we need to keep track of the orientation of the sphere.

details on these methods because, in practice, we have not observed significant differences compared to the velocity Verlet method, perhaps partly because the time step required during coupling with the fluid is much smaller than the time step we would use without fluid coupling. More information on the different usable methods and their comparisons can be found in [19]. More precise methods exist but require multiple computations of the forces at each iteration and therefore make the coupling with the fluid much more difficult.

3.4 Coupling the LBM and the DEM

We have seen how to implement the DEM simulation as well as the fluid simulation with the LBM. Now, the task is to find a way to couple these two methods in order to have a solid-fluid simulation that can run entirely on a GPU. When coupling DEM and LBM simulations, there are generally two scenarios depending on the particle size relative to the fluid cell size: either the diameter of the particles is smaller than the fluid cell size Δx (in this case, the coupling is considered unresolved), or the diameter of the particles is several times larger than the fluid cell size (in this case, the coupling is considered resolved). In our case, we will use a resolved coupling. The interactions between the fluid and the spheres can be separated in two distinct parts: the forces exerted by the solids of the fluid and the forces exerted by the fluid on the solids. This problem is far from easy to solve because these forces depend, among other things, on the orientation of the solid surface within the fluid cells. Although several methods exist to represent these interactions by considering the orientation of the solid surface [3, 5], they are generally complicated to implement and to understand. This is why we have chosen to use a method that does not take into account the orientation of the solid surface but only the volume they occupy in each fluid cell. This method is called the "partially saturated method" in the LB community, and is very similar to the "volume penalization method" more commonly used in the CFD community (see, e.g., the work by Nguyen et al. [23]). It is particularly used when modeling porous media, but it can also be applied in our case in order to abstract the geometry of the surface. The idea behind this method is that during the collision step of the LBM, a proportion of the resulting populations is calculated as if there were only fluid in the cell (see the LBE (3)), while another proportion, which depends on the volume of the cell occupied by solids, interacts with the fluid. The new LBE for the partially saturated method is given by [24]:

$$\overrightarrow{f^{out}} - \overrightarrow{f^{in}} = \left(1 - \sum_k B_k\right) \cdot \overrightarrow{\Omega^c} + \sum_k B_k \cdot \overrightarrow{\Omega^s_k}, \tag{80}$$

where $\overrightarrow{\Omega^c}$ is the standard collision term given by equation (13) for the TRT model or by equation (7) if we want to use the BGK model. $\overrightarrow{\Omega^s}$ is the solid collision operator for the solid k and B_k is a coefficient related to the volume fraction of the fluid cell occupied by the solid k. The solid collision operator is

given by the following formula [24, 25]:

$$\Omega^s_{k,i} = \left[f^{in}_{\bar{i}}(x,t) - f^{eq}_{\bar{i}}(\rho,u)\right] - \left[f^{in}_{i}(x,t) - f^{eq}_{i}(\rho,u_k)\right], \tag{81}$$

where u_k is the surface velocity of the solid k [24, 25]. This velocity can easily be computed by using the distance between the center of mass of the solid k and the position of the fluid cell j in which the collision occurs:

$$\vec{u_k} = \vec{v_k} - \vec{\omega_k} \times (\vec{r_j} - \vec{r_k}), \tag{82}$$

where $\vec{r_k}$ is the center of mass of the solid k, $\vec{\omega_k}$ is the angular velocity of the sphere and $\vec{r_j}$ is the position of the fluid cell j[16]. The coefficient B_k is given by:

$$B_k = \frac{\beta_k(\tau - \frac{1}{2})}{(1-\beta_k) + (\tau - \frac{1}{2})}, \tag{83}$$

where β_k is the the so-called solid fraction (the proportion of the cell occupied by the solid k, see Figure 15) and $\tau = \frac{1}{s^+}$ is the relaxation time [24, 26]. The problem with equation (80) is that the term $\sum_k B_k$ can be greater than one, which can lead to incoherent results. To overcome this problem, we simply replace it by one if it is greater than one:

$$\overrightarrow{f^{out}} - \overrightarrow{f^{in}} = (1 - B_{tot}) \cdot \vec{\Omega^c} + \sum_k B_k \cdot \vec{\Omega^s_k} \tag{84}$$

$$B_{tot} = \min\left(\sum_k B_k, 1\right). \tag{85}$$

[16] In our case, we assume a "cell-centered" referential, meaning that is we have a 2D lattice of 10 cells by 10 cells, the position of the first cell is $(\frac{1}{2}\Delta x, \frac{1}{2}\Delta x)$.

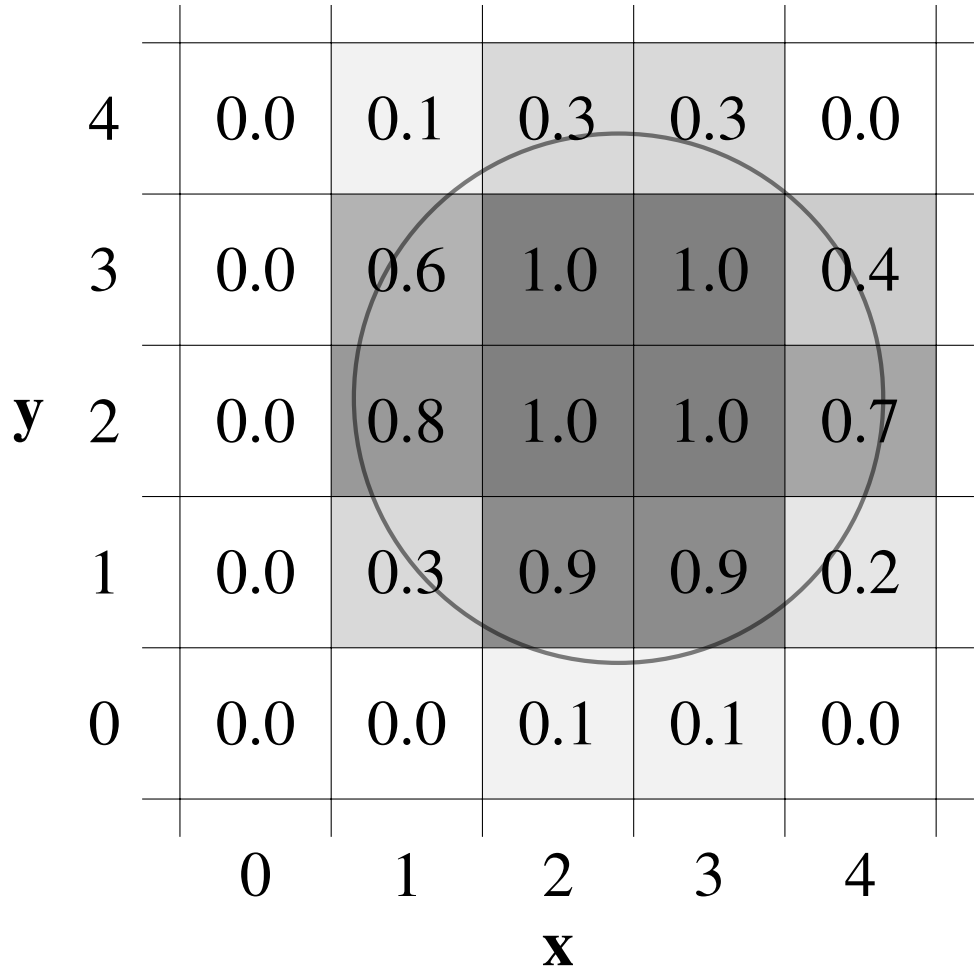


Figure 15: Example in 2D of a sphere in a fluid lattice with different solid fractions for each fluid cell.

The forces and torques exerted by the fluid on a solid k also depends on the coefficient B_k and is given by [24, 26]:

$$\overrightarrow{F_{sf}} = -\frac{\Delta x^3}{\Delta t} \sum_j \left[B_k(\vec{r_j}) \sum_i \Omega^s_{k,i}(\vec{r_j}) \vec{c_i} \right], \tag{86}$$

$$\overrightarrow{M_{sf}} = -\frac{\Delta x^3}{\Delta t} \sum_j \left[B_k(\vec{r_j}) \cdot \left((\vec{r_j} - \vec{r_k}) \times \sum_i \Omega^s_{k,i}(\vec{r_j}) \vec{c_i} \right) \right], \tag{87}$$

where j is the index of the cells contained inside (or partially inside) the solid, $\vec{c_i}$ are the velocities of the lattice and $\Omega^s_{k,i}(\vec{r_j})$ represent the component i of the solid collision term for the solid k inside the cell j.

3.4.1 Computation of the solid fraction

There are several ways to calculate the solid fraction. One way is, for example, to use the Monte Carlo method [6]. We generate a certain number of random points within the cell, and the solid

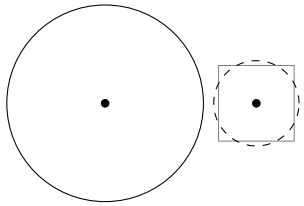
fraction is given by the ratio between the total number of points and those that are contained within the solid. Another similar approach is to use a certain number of fixed points distributed uniformly and, as before, calculate the ratio between the total number of points and the number of points within the solid. The disadvantage of these two methods is that the accuracy depends on the number of points, and it is necessary to calculate the distance for each of these points to determine whether the point is inside or outside the solid. More recently, Jones and Williams [6] have shown that it is possible to obtain excellent results by approximating the solid fraction for spheres purely linearly using the distance between the surface of the sphere and the center of the fluid cell, as well as another precalculated term that depends on the radius of the spheres (thus, only one distance calculation is necessary for the solid fraction calculation). In our case, due to time constraints and because the calculation of the solid fraction is not the main focus of this paper, we have chosen to approximate the solid fraction solely based on the distance between the surface of the sphere and the center of the fluid cell. Our formula to approximate the solid fraction is the following:

$$\beta_k = \begin{cases} 0 & \text{if} \quad \delta_{sc} < 0 \\ 1 & \text{if } \delta_{sc} > 2R_c \\ \frac{\delta_{sc}}{2R_c} & \text{else} \end{cases}. \tag{88}$$

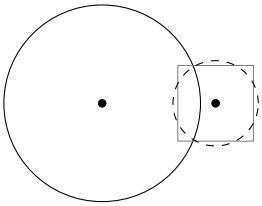
$$\delta_{sc} = R_k + R_c - \left\| \vec{r_k} - \vec{r_j} \right\|, \tag{89}$$

$$R_c = \Delta x \sqrt[3]{\frac{3}{4\pi}}, \tag{90}$$

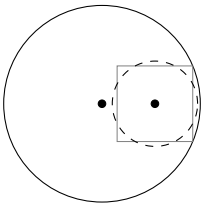
where R_k is the radius of the solid sphere, $\vec{r_k}$ is the center of mass of the solid k and $\vec{r_j}$ is the position of the fluid cell j. The calculation of δ_{sc} is actually just the overlap between the solid sphere k and another virtual sphere with a radius of R_c centered at the center of the fluid cell j. Since the volume of a fluide cell is Δx^3, the value of R_c is chosen such that the virtual sphere centered on the fluid cell has the same volume as the cell in question.



(a) $\delta_{sc} < 0, \beta_k = 0$



(b) $0 < \delta_{sc} < 2R_c, \beta_k = 0.323$



(c) $\delta_{sc} > 2R_c, \beta_k = 1$

Figure 16: An example in 2D for the calculation of the solid fraction β_k based on different configurations (in 2D, we choose $R_c = \Delta x \sqrt{\frac{1}{\pi}}$ so that the surface of the disk centered on the fluid cell is identical to the surface of the fluid cell of Δx^2.

Although this formula is not entirely correct and makes the computation of β_k non-isotropic, it allows for some continuity in the evolution of the system, which is always better than using only $\beta_k = 0$ or $\beta_k = 1$ depending on whether the cell is entirely contained in the solid or not. It would also be interesting to see the results using the formula for the volume of intersection between two spheres to compute the solid fraction, as an analytical formula exists for this calculation.

3.4.2 Coupling algorithm

Given that the coefficients B_k are used in the calculation of the forces on the solids and in the fluid update, it is interesting to compute them only once and store them in a structure. We will use two structures: a structure containing B_{tot} for each cell and another structure containing the B_k coefficients for each solid in each cell. To keep the size of this second structure fixed, we set the maximum number of solids that can be present in a fluid cell at the same time (in practice, it is extremely rare to have more than two solids in the same cell, as the cell size is typically several times smaller than the diameters of the solid spheres). We also need to have a structure containing the number of solids in each fluid cell. The solid collision term $\overrightarrow{\Omega_k^s}$ is also present in the fluid update formula and in the computation of the forces on the sphere. Unfortunately, due to the large number of lattice velocities (19 velocities for the D3Q19 lattice) and the large number of fluid cells, it is not practical to store this value. This is not a real problem because these terms will be recomputed only for specific cells. The coupling steps are as follows:

1. Iterate in parallel over the LBM cells and compute the B_k coefficients (fixed maximum number of B_k per cells) and B_{tot}.

2. Iterate over the solids in parallel and compute $\overrightarrow{\Omega_k^s}$ for all cells containing the solid. Update the forces and torques on the solids according to equations (86) and (87). If there is only one solid

in the cell, we can update the fluid with $\overrightarrow{f^{out}} = \overrightarrow{f^{in}} + (1 - B_{tot}) \cdot \overrightarrow{\Omega^c} + B_k \cdot \overrightarrow{\Omega_k^s}$ (if there is multiple solids in the cell, we cannot update the cell otherwise we would have a race condition).

3. Iterate in paralllel over the fluid cells where there is 0 or more than 1 solid inside. If there is no solid inside the fluid cell, the cell is update according to the standard LBE (3). If there is more than one solid inside the cell, we need to recompute $\overrightarrow{\Omega_k^s}$ and update the cell according to equation (84).

Although the solid collision term $\overrightarrow{\Omega_k^s}$ may be calculated twice in some cases, it is only recomputed for fluid cells that contain more than 2 solids simultaneously. In practice, this step is not the most time-consuming. In fact, the biggest weakness of this algorithm is that it is highly sensitive to the number of fluid cells occupied by a solid. During step 2, where we iterate in parallel over the solids, the iteration over the fluid cells contained within a single solid is performed by a single thread. Therefore, this algorithm is efficient when dealing with a large number of solids represented by a relatively small number of fluid cells. In this work, we did not investigate how the coupling can be performed with different types of walls. Only the case of planar walls placed at the domain boundaries was studied. In this case, for the LBM simulation, we just need to add a bounce-back boundary condition with the desired velocity at the boundary where the wall is located. In the DEM simulation, the wall will have the same velocity than in the LBM simulation and will be positioned halfway between the bounce-back cell and the fluid cell (this correspond to the physical location of the wall for the bounce-back boundary condition). The pseudocodes for these three steps of the coupling are the following:

Algorithm 3.10: Coupling - Step 1

Data

solidPosition: positions of the solids in the simulation.
solidRadius: radii of the solids in the simulation.
B_tot: total solid fraction coefficient in each cell.
B: solid fraction weights Bk for each solid in each cell.
BsolidIds: ids of the solids in each fluid cell
nbSolidPerCell: number of solid in each fluid cell
nbCells: number of fluid cells
maxNbSolidPerCell: maximum number of solids in a fluid cell

▹ *iterate over the LBM cells in parallel to compute coefficients Bk and B_tot* ◃
for cellIndex = 0, . . . , nbCells-1 **do**
  ▹ *position of the center of the fluid cell* ◃
  $\vec{r_j}$ = getCellCenter(cellIndex)
  ▹ *initialize total B coefficient* ◃
  B_tot[cellIndex] = 0
  init the solid ids to a value to indicate that the solid fraction is empty
  for k = 0, . . . , maxNbSolidPerCell - 1 **do**
    BsolidIds[cellIndex][k] = emptyValue
  solidsInCell = getSolidsInCell(cellIndex)
  nbSolidsCurrentCell = 0
  for all solidIndex in solidsInCell **do**
    $\vec{r_k}$ = solidPosition[solidIndex]
    $R_c = \Delta x \sqrt[3]{\frac{3}{4\pi}}$
    R_k = solidRadius[solidIndex]
    $\delta_{sc} = R_k + R_c - \left\|\vec{r_k} - \vec{r_j}\right\|$
    if $\delta_{sc} > 0$ **then**
      if $\delta_{sc} < 2R_c$ **then**
        $\beta_k = \frac{\delta_{sc}}{2R_c}$
        B[cellIndex][nbSolidsCurrentCell] = $\frac{\beta_k(\tau-\frac{1}{2})}{(1-\beta_k)+(\tau-\frac{1}{2})}$
      else
        B[cellIndex][nbSolidsCurrentCell] = 1
      BsolidIds[cellIndex][nbSolidsCurrentCell] = solidIndex
      B_tot[cellIndex] += B[cellIndex][nbSolidsCurrentCell]
      nbSolidsCurrentCell++
  ▹ *B_tot cannot be greater than 1* ◃
  B_tot[cellIndex] = min(1, B_tot[cellIndex])
  nbSolidPerCell[cellIndex] = nbSolidsCurrentCell

Algorithm 3.11: Coupling - Step 2

Data

solidPosition: Positions of the solids in the simulation.
solidRadius: radii of the solids in the simulation.
solidVelocity: velocities of the solids in the simulation.
solidAngularVelocity: angular velocities of the solids in the simulation.
B_tot: total solid fraction coefficient in each cell.
B: solid fraction weights Bk for each solid in each cell.
BsolidIds: Ids of the solids in each fluid cell
nbSolidPerCell: number of solid in each fluid cell
nbCells: number of fluid cells
nbSolids: number of solids
solidForce: the forces applied on the solids
solidTorqueR: the torques divided by the radius applied on the solids

▷ *Compute the solid fraction coefficients for all cells* ◁
▷ *iterate over the solids in parallel* ◁
for solidIndex = 0, . . . , nbSolids-1 **do**
  R_k = solidRadius[solidIndex]
  $\vec{r_k}$ = solidPosition[solidIndex]
  $\vec{v_k}$ = solidVelocity[solidIndex]
  $\vec{\omega_k}$ = solidAngularVelocity[solidIndex]
  ▷ *Iterate over the cells contained in the solid* ◁
  cells = getCellsInSolid(solidIndex)
  for all cellIndex in cells **do**
    ▷ *Get the center position of the cell* ◁
    $\vec{r_j}$ = getCellCenter(cellIndex)
    ▷ *Find the Bk coefficient computed earlier* ◁
    B_k = getBk(cellIndex, solidIndex)
    ▷ *macroscopic values of the fluid cell* ◁
    ρ, $\vec{u}$ = getMacro(cellIndex)
    $\vec{u_k} = \vec{v_k} - \vec{\omega_k} \times (\vec{r_j} - \vec{r_k})$
    ▷ *Compute the solid collision term according to the corresponding equation* ◁
    $\vec{\Omega_k^s}$= computeCollisionSolidTerm(cellIndex, ρ, $\vec{u}$, $\vec{u_k}$)
    ▷ *perform the collision and stream algorithm if there is only one solid* ◁
    if nbSolidPerCell[cellIndex] == 1 **then**
      $\vec{f^{out}} = \vec{f^{in}} + (1 - B_{tot}) \cdot \vec{\Omega^c} + B_k \cdot \vec{\Omega_k^s}$
    ▷ *update force and torque/r on the solid* ◁
    solidForce[solidIndex] += $-\frac{\Delta x}{\Delta t} \left[B_k(\vec{r_j}) \sum_i \Omega^s_{k,i}(\vec{r_j}) \vec{c_i} \right]$
    solidTorqueR[solidIndex] += $-\frac{1}{R_k} \frac{\Delta x}{\Delta t} \left[B_k(\vec{r_j}) \cdot \left((\vec{r_j} - \vec{r_k}) \times \sum_i \Omega^s_{k,i}(\vec{r_j}) \vec{c_i} \right) \right]$

Algorithm 3.12: Coupling - Step 3

Data
solidPosition: Positions of the solids in the simulation.
solidRadius: radii of the solids in the simulation.
solidVelocity: velocities of the solids in the simulation.
solidAngularVelocity: angular velocities of the solids in the simulation.
B_tot: total solid fraction coefficient in each cell.
B: solid fraction weights Bk for each solid in each cell.
BsolidIds: Ids of the solids in each fluid cell
nbSolidPerCell: number of solid in each fluid cell
nbCells: number of fluid cells

▷ *Iterate over the fluid cells in parallel* ◁
for cellIndex = 0, . . . , nbCells-1 **do**
  ▷ *we only iterate on the cells where there is 0 or more than 1 solids* ◁
  if nbSolidPerCell[cellIndex] == 0 **then**
    ▷ *if there is no solid in the cell, the fluid is updated* ◁
    $\overrightarrow{f^{out}} = \overrightarrow{f^{in}} + \overrightarrow{\Omega^c}$
  else if nbSolidPerCell[cellIndex] > 1 **then**
    $\overrightarrow{r_j}$ = getCellCenter(cellIndex)
    ▷ *if there is more than one solid, we iterate over them to compute the sum of Omega Bk* ◁
    sumOmegaBk = $\overrightarrow{0}$
    solidsInCell = getSolidsInCell(cellIndex)
    for all solidIndex in solidsInCell **do**
      R_k = solidRadius[solidIndex]
      $\overrightarrow{r_k}$ = solidPosition[solidIndex]
      $\overrightarrow{v_k}$ = solidVelocity[solidIndex]
      $\overrightarrow{\omega_k}$ = solidAngularVelocity[solidIndex]
      B_k = getBk(cellIndex, solidIndex)
      ρ, $\overrightarrow{u}$ = getMacro(cellIndex)
      $\overrightarrow{u_k} = \overrightarrow{v_k} - \overrightarrow{\omega_k} \times (\overrightarrow{r_j} - \overrightarrow{r_k})$
      $\overrightarrow{\Omega_k^s}$= computeCollisionSolidTerm(cellIndex, ρ, $\overrightarrow{u}$, $\overrightarrow{u_k}$)
      ▷ *Add Bk Omega to the sum* ◁
      sumOmegaBk += $B_k \cdot \overrightarrow{\Omega_k^s}$
    ▷ *Update the fluid* ◁
    $\overrightarrow{f^{out}} = \overrightarrow{f^{in}} + (1 - B_{tot}) \cdot \overrightarrow{\Omega^c}$ + sumOmegaBk

In these pseudocodes, we did not describe how to obtain the list of cells located inside a sphere. In fact, it is sufficient to use the uniform grid that is used to detect sphere collisions. If there is a 1-to-1 ratio between the size of this grid and the LBM lattice, the result is trivial. In practice, the LBM lattice usually has more cells than our uniform grid. Therefore, we propose to ensure that the ratio of the size of a cell in the uniform grid to the size of an LBM cell is an integer number. In this case, we can directly obtain the coordinates of the LBM cells that potentially contain a solid by using this ratio and the minimum and maximum coordinates of the solid in the uniform grid (by multiplying these coordinates by the ratio, we obtain the minimum and maximum coordinates of the sphere in the LBM lattice).

3.4.3 Going back to the unit conversion

In section 3.2.4, we saw how to recover the physical units from units normalized by the time step Δt and the grid spacing Δx. However, we said that we want to ensure that the ratio of the size of a cell in the uniform grid to the size of an LBM cell is an integer number. Thus, we need to redefine the way we choose the size of the LBM lattice. Let D be the length of one side of the domain in simulation units and N_{DEM} the size of the uniform grid used for collisions detection for the same side of the domain. We want $N_{LB} = \frac{D \cdot L_{LB}}{L}$ (the number of LBM cells for the same side of the domain than D and N_{LB}) such that $\frac{N_{LB}}{N_{DEM}}$ is an integer number greater or equal to one. Therefore we need to change the value of characteristic length in lattice units L_{LB} in order to satisfy this constraint:

$$N_{LB} = \left\lceil \frac{D \cdot L_{LB}}{L \cdot N_{DEM}} \right\rceil \cdot N_{DEM}. \tag{91}$$

Therefore, the new characteristic length in lattice units, the grid spacing and the time step are given by:

$$L'_{LB} = \frac{L \cdot N_{LB}}{D}, \tag{92}$$

$$\Delta x = \frac{L}{L'_{LB}}, \tag{93}$$

$$\Delta t = \frac{\Delta x \cdot u_{LB}}{u}. \tag{94}$$

3.5 The C++17 parallel algorithms

The goal of this section is not to provide a full introduction to the basics of C++, but we will briefly address some important concepts to better explain how parallel algorithms can be implemented in practice. C++17 introduces the concept of execution policies, which determines how an algorithm can be executed in parallel. There are three available execution policies accessible through the following instances[17] which can be passed as parameters to certain algorithms:

- `std::execution::seq`

  This policy indicates that the algorithm should be executed sequentially.

- `std::execution::par`

  This policy indicates that the algorithm should be executed in parallel.

- `std::execution::par_unseq`

  This policy indicates that the algorithm should be executed in parallel and that vectorization is allowed.

[17] Since C++20, there is a forth execution policy available through the instance `std::execution::unseq`.

Where these execution policies become powerful is that they can be used with certain algorithms from the Standard Template Library (STL). This makes it possible to parallelize code in a nearly transparent manner while remaining at a high level. However, it should be noted that in practice, the standard does not guarantee that the code will necessarily be parallelized, nor does it define the implementation of the standard library. Therefore, the results will strongly depend on the compiler used. The STL algorithms work on various data structures called "containers" such as `std::vector` or `std::array`. The elements inside these containers are made accessible via iterators, which are passed as arguments to STL functions. Here is a non-exhaustive list of some of the STL algorithms that can be parallelized and that we have used to implement this project.

- `std::transform`

  The `std::transform` function applies a given unary function to a given input range of data and write the result in the given output container. This function is very useful when we want to apply the same function to a list of element in parallel.

  Example 3.3: `std::transform`

  Squaring the elements of a vector in parallel with `std::transform`.

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <execution>

int main()
{
    std::vector<int> v = {3, 1, 4, 1, 5, 9, 2, 6, 5};
    // square the elements of v in parallel. In-place modification
    std::transform(std::execution::par_unseq, v.begin(), v.end(), v.begin(),
                   [](auto x) {return x*x;}
    );
    std::cout << v[0] << std::endl; // output 3*3 = 9
    return 0;
}
```

- `std::for_each`

  The `std::for_each` function takes as arguments an input range and a unary function that will be executed to perform a specific action for each element in the input range. Unlike `std::transform`, it does not return a value and the eventual modifications are perform through side-effect operations. It can be used as a parallel version of the for loop.

Example 3.4: `std::for_each`

Squaring the elements of a vector in parallel with `std::for_each`.

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <execution>

int main()
{
    std::vector<int> v = {3, 1, 4, 1, 5, 9, 2, 6, 5};
    // square the elements of v in parallel. In-place modification
    std::for_each(std::execution::par_unseq, v.begin(), v.end(),
                  [](auto &x) {
                     x = x*x;
                  }
    );
    std::cout << v[0] << std::endl; // output 3*3 = 9
    return 0;
}
```

- `std::reduce`

  The `std::reduce` function performs a reduction operation given an input range, an initial value and a binary operation[18] and then returns the result. The result is non-deterministic if the binary operation is neither associative nor commutative[19]. A simple use case is when we want to calculate the sum of the elements in an array.

[18] By default, the binary operation is the addition (`std::plus<>()`).

[19] If the binary operation is not associative or not commutative, `std::accumulate` can be used instead but cannot be parallelized.

Example 3.5: `std::reduce`

Summing the element of a vector in parallel with `std::reduce`.

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <execution>

int main()
{
    std::vector<int> v = {3, 1, 4, 1, 5, 9, 2, 6, 5};
    // Sum of the elements of v in parallel.
    auto sum = std::reduce(std::execution::par_unseq, v.begin(), v.end(),
                           0, std::plus<>());
    std::cout << sum << std::endl; // sum = 36
    return 0;
}
```

- `std::sort`

  The `std::sort` function allows sorting comparable elements within the input range (the operation is done in-place). A custom comparison function can also be provided. When used with a parallel execution policy, it becomes a very powerful function because many parallel algorithms are based on sorting.

Example 3.6: `std::sort`

Sort a vector in parallel with `std::sort`.

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <execution>

int main()
{
    std::vector<int> v = {3, 1, 4, 1, 5, 9, 2, 6, 5};
    // Sort the elements of v in parallel.
    std::sort(std::execution::par_unseq, v.begin(), v.end());
    // after sorting, v = {1, 1, 2, 3, 4, 5, 5, 6, 9}
    return 0;
}
```

- `std::inclusive_scan`/`std::exclusive_scan`

The `std::inclusive_scan` and `std::exclusive_scan` both perform a cumulative sum on an input range given an associative binary operation (if the binary operation is not associative the result is not deterministic) and an initial value and then store the result in the output range. The difference between the two is that `std::inclusive_scan` starts with the sum of the initial value and the first element and includes the last element whereas `std::exclusive_scan` starts with the initial value and excludes the last element. In our case, inclusive and exclusive scans are useful when we have multiple contiguous lists in memory and we want to compute an index list indicating the starting positions of these lists, given the number of elements in each of them.

Example 3.7: `std::exclusive_scan` and `std::inclusive_scan`

Cumulative sum of a vector in parallel with `std::exclusive_scan` and `std::inclusive_scan`.

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <execution>

int main()
{
    std::vector<int> v_excl(9);
    std::vector<int> v_incl(9);
    std::vector<int> v = {3, 1, 4, 1, 5, 9, 2, 6, 5};
    // exclusive scan
    std::exclusive_scan(std::execution::par_unseq, v.begin(), v.end(),
                        v_excl.begin(), 0, std::plus<>());
    // v_excl = {0, 3, 4, 8, 9, 14, 23, 25, 31}

    // inclusive scan
    std::inclusive_scan(std::execution::par_unseq, v.begin(), v.end(),
                        v_incl.begin(), std::plus<>());
    // v_incl = {3, 4, 8, 9, 14, 23, 25, 31, 36}
    return 0;
}
```

3.5.1 nvc++

As mentioned before, due to the permissive nature of the C++ standard, the choice of compiler is crucial. In our case, we will use the nvc++ compiler from Nvidia (Nvidia Corporation, USA) included in the NVIDIA HPC SDK [27]. This compiler allows for transparent code parallelization on various CPUs and Nvidia GPUs using the parallel algorithms introduced in the C++17 standard. However, when deploying code on GPUs, certain constraints must be observed: it is not possible to pass stack pointers between the CPU and GPU (only heap pointers) and the same applies to function pointers.

Abstract classes cannot either be passed between the CPU and the GPU [20]. Parallel algorithms can be enabled with the option "`-stdpar=TARGET`" where "`TARGET`" can be either "`cpu`" or "`gpu`" depending on the execution that we want.

Example 3.8: nvc++ usage

Here is a simple example on how to use the nvc++ compiler to use parallel algorithms with GPU acceleration. We compile a source file called "`MyProgram.cpp`" using GPU acceleration with the flag "`-stdpar=gpu`". "`MyProgram`" is the output program.

```
$ nvc++ -stdpar=gpu MyProgram.cpp -o MyProgram
```

[20] Further information on the constraints can be found in the documentation [27].

4 Results and discussion

4.1 DEM validation

4.1.1 Energy and momentum conservation

The first thing we can do to validate the model is to check that the momentum is preserved before and after a perfectly elastic collision ($\epsilon = 1$) between two spheres without friction ($\mu = 0$). We can also verify that the energy of the system, which is the sum of the kinetic energy and the elastic potential energy is preserved during the entire duration of the collision. Our setup consists of two identical spheres i and j moving towards each other. The material properties of the spheres are the following (in physical units):

- $R_i = R_j = 10$
- $m_i = m_j = 1$
- $\upsilon_i = \upsilon_j = 0.4$
- $E_i = E_j = 100$

The initial velocities of the spheres i and j along the x axis are $v_i(0) = 1$ and $v_j(0) = -1$ and their initial positions along the x axis are $r_i(0) = 10.1$ and $r_j(0) = 30.1$. The time step of the simulation is $\Delta t = 5 \times 10^{-4}$ and the number of iterations is 1000. At time $t = 0$, the momentum of the system is null ($p(0) = m_i(0)v_i(0) + m_j(0)v_j(0) = 0$).

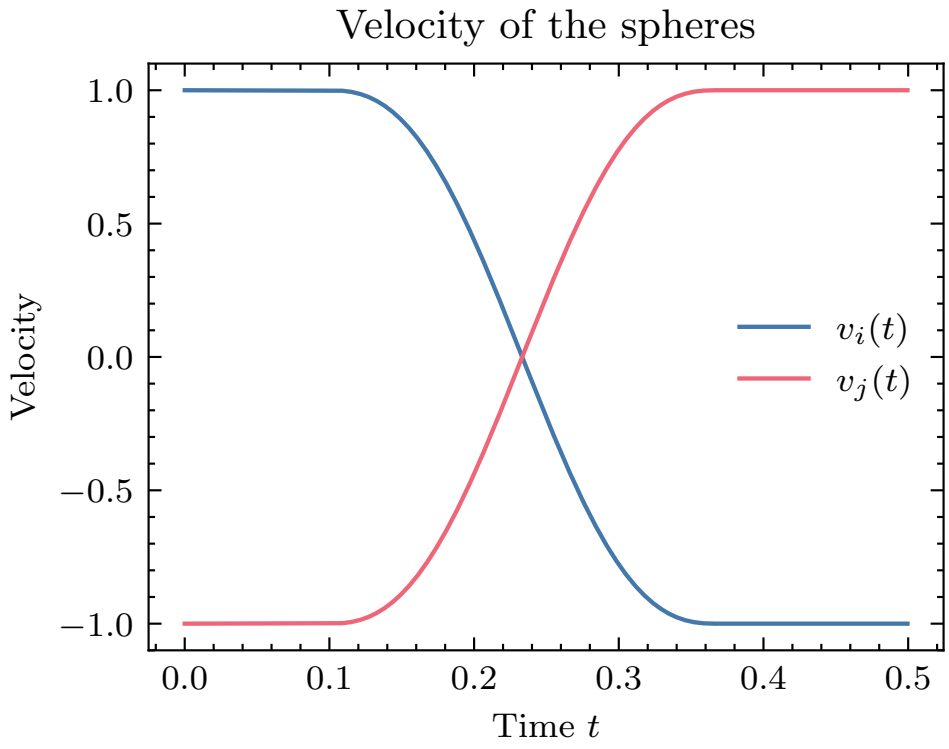


Figure 17: Velocities of the the particles i and j as a function of the time.

As we can observe in Figure 17, the particle velocities decrease in absolute value as the collision progresses until they reach zero. Then, the sign of the velocities changes and they start to increase in absolute value until both particles move in opposite directions with opposite velocities. Therefore, we see that the momentum after the collision remains unchanged ($p(T) = m_i v_i(T) + m_j v_j(T) = 0, T = 1000 \cdot \Delta t$). We can also verify if the total energy $E_t = E_{n_{el}} + E_K$ of the system is preserved during the duration of the collision where $E_K(t)$ is the kinetic energy of the spheres and $E_{n_{el}}$ is the normal elastic potential energy. $E_K(t)$ is given by:

$$E_K(t) = \frac{1}{2}\left(m_i v_i(t)^2 + m_j v_j(t)^2\right). \tag{95}$$

To compute the normal elastic potential energy $E_{n_{el}}$, we can use the fact that the normal elastic force is given by $f_{n_{el}} = k_n \delta_n$ (42). Thus:

$$f_{n_{el}} = k_n \delta_n = \frac{4}{3} Y^* \sqrt{R^* \delta_n} \delta_n = \frac{4}{3} Y^* \sqrt{R^*} \delta_n^{3/2}. \tag{96}$$

Therefore, we can obtain the normal elastic potential energy $E_{n_{el}}$ by integrating $f_{n_{el}}$ over δ_n (an example of the evolution of the elastic potential energy as a function of the overlap can be seen in Figure 18):

$$E_{n_{el}}(\delta_n) = \int_0^{\delta_n} f_{n_{el}}(x)\,\mathrm{d}x = \frac{4}{3} Y^* \sqrt{R^*} \int_0^{\delta_n} x^{3/2}\,\mathrm{d}x = \frac{8}{15} Y^* \sqrt{R^*} \delta_n^{5/2}. \tag{97}$$

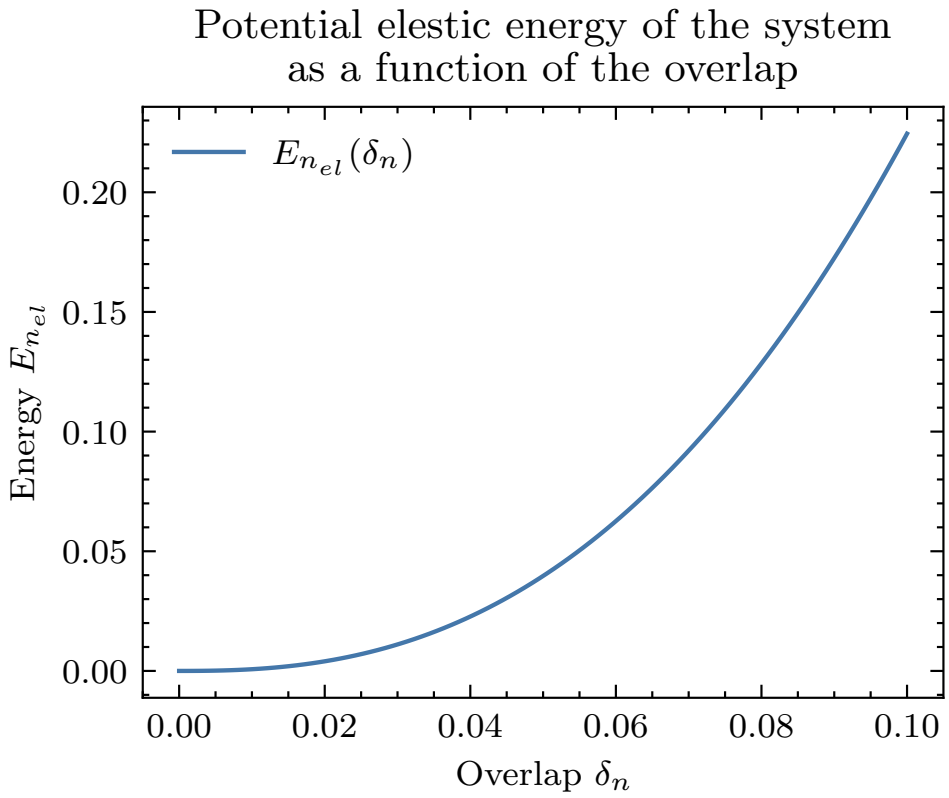


Figure 18: Elastic potential energy as a function of the overlap with the material properties used for the experiment.

The evolution of the system's energy is as follows:

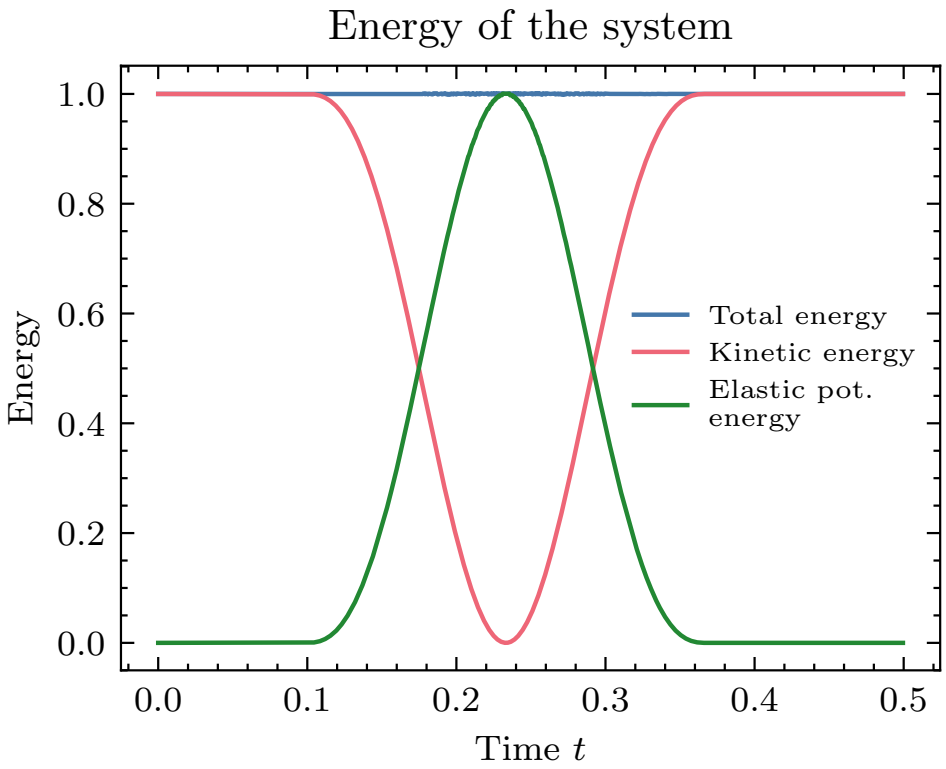


Figure 19: Energy of the system as a function of time.

As we can see in Figure 19, the total energy of the system remains constant throughout the collision. Before the collision, all the energy of the system is in the form of kinetic energy. Then, during the first part of the collision (when the spheres approach each other), the kinetic energy is gradually converted into elastic potential energy until the particle velocities become zero and the entire energy of the system is in the form of elastic potential energy. Afterwards, the spheres will start to separate due to the elastic normal force, and the energy contained in the system as elastic potential energy will gradually be transformed into kinetic energy until the spheres, as they move away from each other, are no longer in contact.

4.1.2 Collision angle

Another valuable validation test for the DEM model is to compute the angle before and after the collision of a sphere approaching a wall with an incident angle and no initial angular velocity.

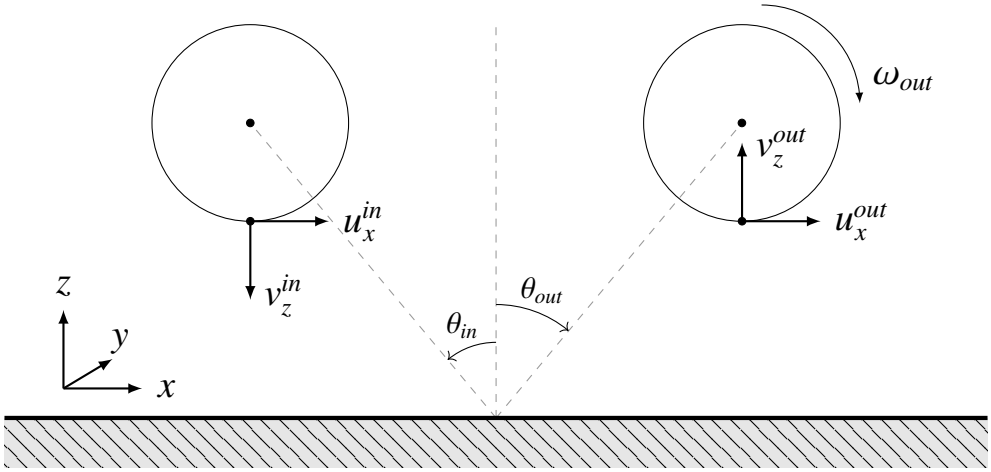


Figure 20: Setup of the simulation for the test. Before the collision the sphere is not rotating. The angles θ_{in} and θ_{out} are taken relatively to the normal direction of the wall.

If there is no friction during the collision between the wall and the sphere ($\mu = 0$), the reflected angle after the collision should be equal to the incident angle and no energy is transferred to the rotation of the sphere.

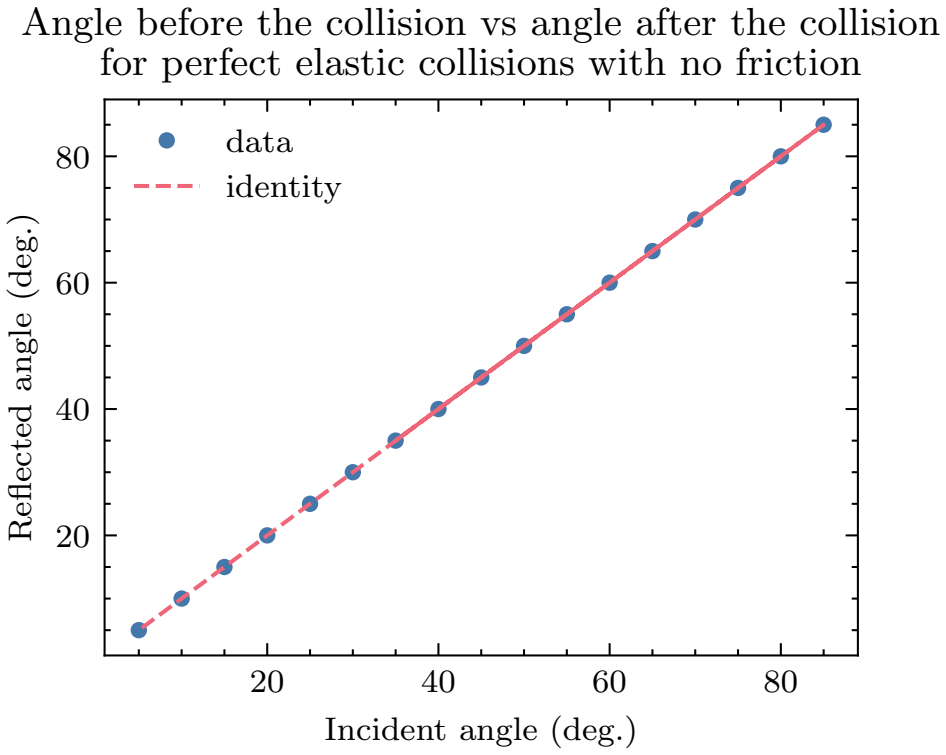


Figure 21: Reflected angle as a function of the incident angle.

As we can see in Figure 21, the results from the simulations are the one we expect as the incident angle is equal to the reflected angle. If the friction coefficient is not zero, the analysis of the problem becomes more difficult since it reduces the horizontal velocity of the sphere and induces a rotational

motion. Instead of comparing the angles, we can use the ratio of the surface tangential velocity $u_x = v_x - \omega R$ (taken on the closest point on sphere to the wall) to the vertical velocity v_z where ω is the scalar angular velocity of the sphere around the y axis and R is the radius of the sphere. Let Ψ_{in} be the velocity ratio before the collision and Ψ_{out} the velocity ratio after the collision:

$$\Psi_{in} = -\frac{v_x^{in}}{v_z^{in}}, \tag{98}$$

$$\Psi_{out} = -\frac{u_x^{out}}{v_z^{in}}. \tag{99}$$

where $u_x^{out} = v_x^{out} - \omega_{out} R$. Note that both ratios are given relatively to the incident vertical velocity v_z^{in} (in our case, $v_z^{in} = -v_z^{out}$ since $\epsilon = 1$). When we have a sliding regime (when the tangential force is limited by the normal force (37)) and a restitution coefficient of one ($\epsilon = 1$), Ψ_{out} can be computed as a function of Ψ_{in} and μ with the following formula [28, 4, 29]:

$$\Psi_{out} = \Psi_{in} - 7\mu. \tag{100}$$

We can also determine the value of Ψ_{in} at which the regime becomes sliding [29]:

$$\Psi_{in} = 5\mu. \tag{101}$$

Before this value, the collision is a mix of sticking and sliding regime and is more difficult to analyze. More information on these different regimes can be found in Ref. [29] and experimental measurements have been studied in Ref. [30]. We performed our simulation with different values of different friction coefficients : $\mu \in [0, 0.01, 0.1, 0.2, 0.3, 0.4]$ and with incident angles ranging from 5° to 85°. Note that $\tan^{-1}(\Psi_{in})$ corresponds to the incident angle in radian.

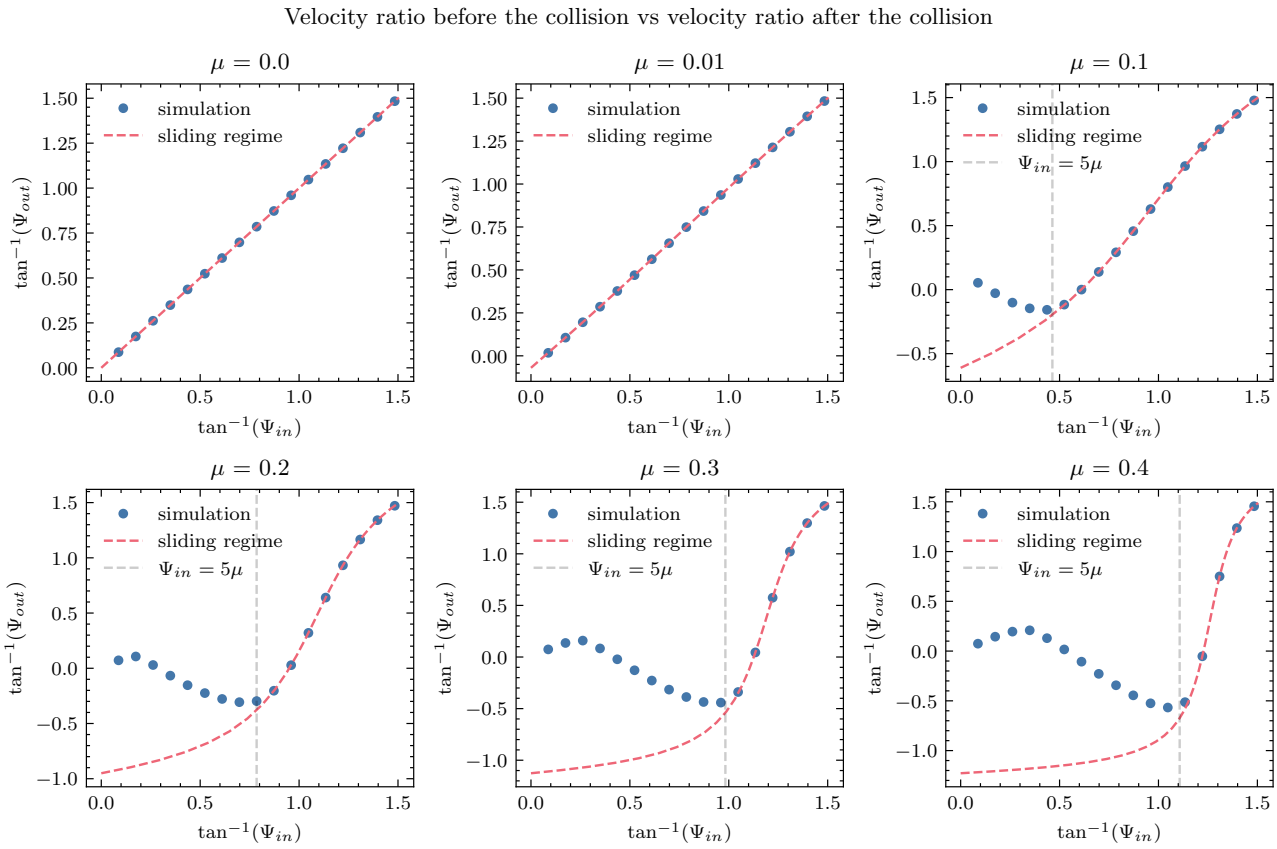


Figure 22: Velocity ratio between before and after the collision.

As we can see in Figure 22, the results obtained from our simulation are consistent with the theory when the sliding regime is reached (101). Even though we have not conducted an in-depth analysis of the regime before it becomes sliding, we still observe this "S-shaped" behavior similar to the experimental results reported in [30].

4.1.3 Velocity distribution

Another test we can do to validate the DEM model is to observe the evolution of the velocity distribution of the particles. In this test, a box with four walls is created and all particles within the box are assigned the same velocity $\vec{v} = (1, 0, 0)$. When the number of particles is high enough, this test boils down to the simulation of a gas in an enclosed box. For the latter, the velocity, the velocity distribution is expected to converge to a Gaussian distribution centered at zero as the test progresses towards the thermodynamic equilibrium. This distribution is given by the Maxwell–Boltzmann distribution. If we only consider the velocity along the x axis v_x, the probability density function is given by:

$$f(v_x) = \sqrt{\frac{m}{2\pi k_b T}} \exp\left(\frac{-mv_x^2}{2k_b T}\right), \tag{102}$$

where T is the absolute temperature of the system, m the mass of each particle and k_b is the Boltzmann constant. Therefore, this distribution is a Gaussian centered at zero with a variance of $\sigma^2 = \frac{k_b T}{m}$.

If collisions are perfectly elastic, which is the case in this simulation, no energy is lost during the simulation. Therefore, the theory of ideal gases can be used to determine the parameters of this Gaussian distribution. According to the ideal gas theory, the relationship between the mean kinetic energy of the particles $\bar{E}$ and temperature T is given by:

$$\frac{3}{2}k_bT = \bar{E} = \frac{1}{2}m < v^2 >, \tag{103}$$

where $< v^2 >$ is the mean square speed. Thus:

$$\sigma^2 = \frac{k_bT}{m} = \frac{2}{3}\frac{\bar{E}}{m} = \frac{1}{3} < v^2 > . \tag{104}$$

Since all particles have an initial velocity of $\vec{v} = (1, 0, 0)$, we have that $< v^2 >= 1$. Therefore, the standard deviation should be $\sigma = \frac{1}{\sqrt{3}} \approx 0.577$. The simulation is performed with 10'000 spheres over 80'000 iterations. The spheres have a radius of a 1 and a mass of 500.

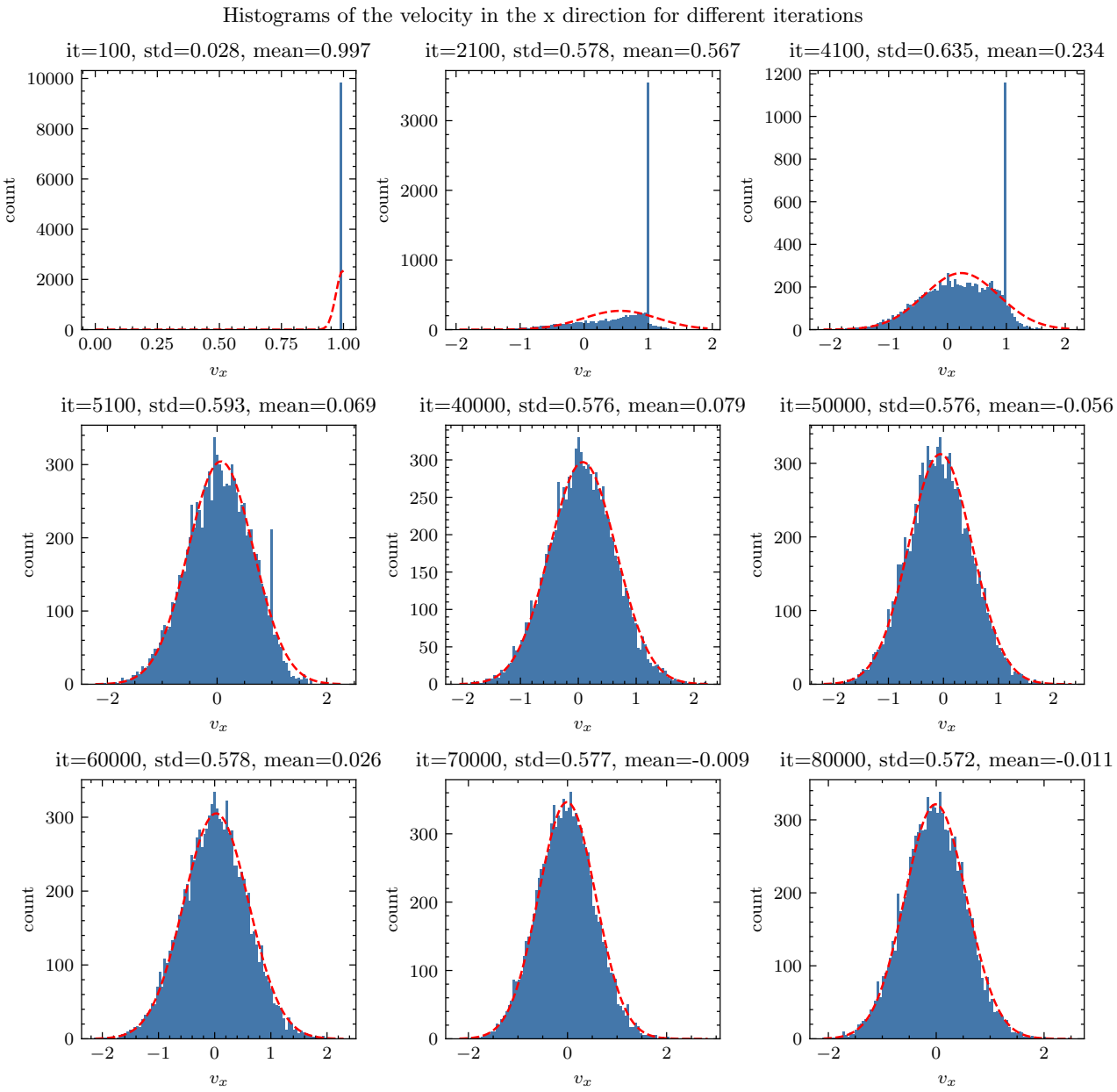


Figure 23: Histogram of the velocity profile in the x direction.

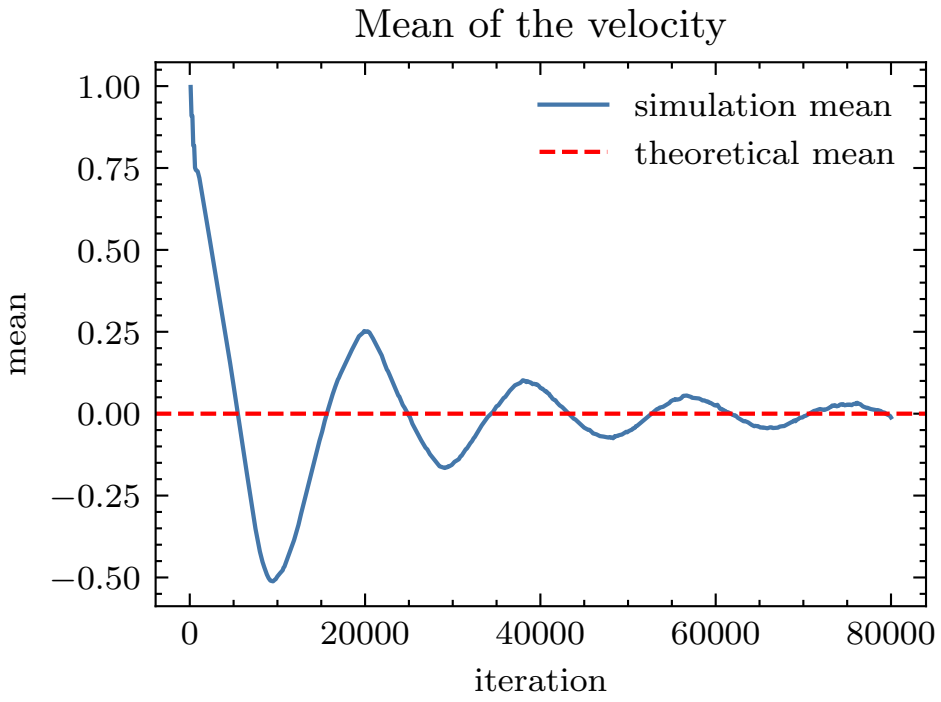


Figure 24: Evolution of the mean of the velocities.

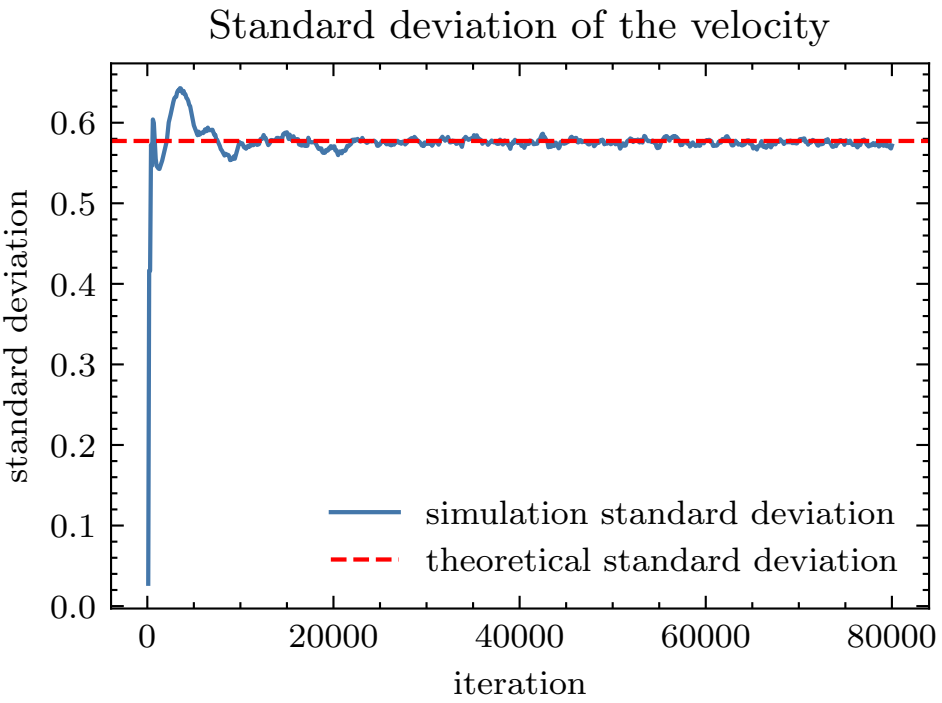


Figure 25: Evolution of the standard deviation of the velocities.

As we can see in Figure 23, the distribution of velocities converges nicely towards a Gaussian shape. we can also see in Figure 24 that the mean velocity converges toward zero and in Figure 25 that the standard deviation converges towards $\sigma = \frac{1}{\sqrt{3}}$, which is what we expected.

4.2 Performance analysis

In order to measure the performance of our code, we will consider 5 different levels of modeling:

- The first one is the case where we only consider the movement of the particles without any interaction between them (only the integration of the equations of motion).
- The second one is based on the first one but with the addition of the movement of the particles in the uniform grid structure.
- The third one adds the computation of the potential collisions between the particles (without the computation of the forces).
- The fourth one is the case where we perform the full DEM simulation (with the computation of the forces).
- The last one is the case where we perform the coupling between the DEM and the LBM simulation.

The advantage of this approach is that with the first case (the case where we only consider the movement of the particles), we can easily measure performances because we know exactly the number of operations that are performed. For the other cases, we will compare the results obtained with those of the first case in order to estimate the performance. The results obtained for the cases where we consider the interaction between the particles are very dependent on the number of collisions at each step. This is why we will consider a system with a volume fraction of 0.4 which corresponds roughly to the volume fraction of red blood cells in blood. Our performance tests will be done using the Nvidia's A100 GPU which is based on the Ampere architecture. There are two available A100 GPUs, the A100 40GB and the A100 80GB. Both of them have a computational power of 19.5 TFlops for the 32-bit case and 9.7 TFlops for the 64-bit case. The memory bandwidth is 1935 GB/s for the 80GB model and 1555 GB/s for the 40GB model. In our case, we will consider the 40GB model.

For the simulations on the GPU, we use uniform parameters for the solid spheres. Each sphere has a radius of 1 and a mass of 500. The number of iterations and spheres represented will differ for each of the different levels of modeling, ensuring that memory is used optimally and that computation time remains reasonable. In the simulation where we only integrate the equations of motion, we consider a total of 1×10^8 particles and run the simulation for 5000 steps. For the simulation where we move the particles using the uniform grid, we have 1×10^7 particles and perform the simulation over 200 steps. In the simulation where we compute the potential collisions, we work with 3×10^5 particles and run the simulation for 1000 steps. For the full DEM simulation, we utilize 3×10^5 particles and perform the simulation over 1000 steps. In the coupling of the DEM and the LBM simulation, we have 33,000 solid particles interacting with a substantial number of fluid cells, amounting to 21,952,000. The simulation is carried out for 10,000 steps. All the different modeling levels are tested in both 32-bit and 64-bit configurations.

Let us begin by examining the simplest case, where we only consider the motion of the spheres without fluid and without collisions. The first thing to consider is whether we are facing a memory bound problem. This can be eventually checked by computing the arithmetic intensity of our code, which is defined as the ratio of the number of floating point operations to the number of bytes read or written. For the case where we only consider the integration of the equations of motion, we obtain an arithmetic intensity of 0.17 Flops/bytes for the 32-bit case and 0.08 for the 64-bit case. Such small arithmetic intensities are usually the signatures of a memory bound problem. This is confirmed by the roofline model, which describes the upper limit of the computational performances of an application with a given arithmetic intensity, based on the memory bandwidth and the computational power of the system.

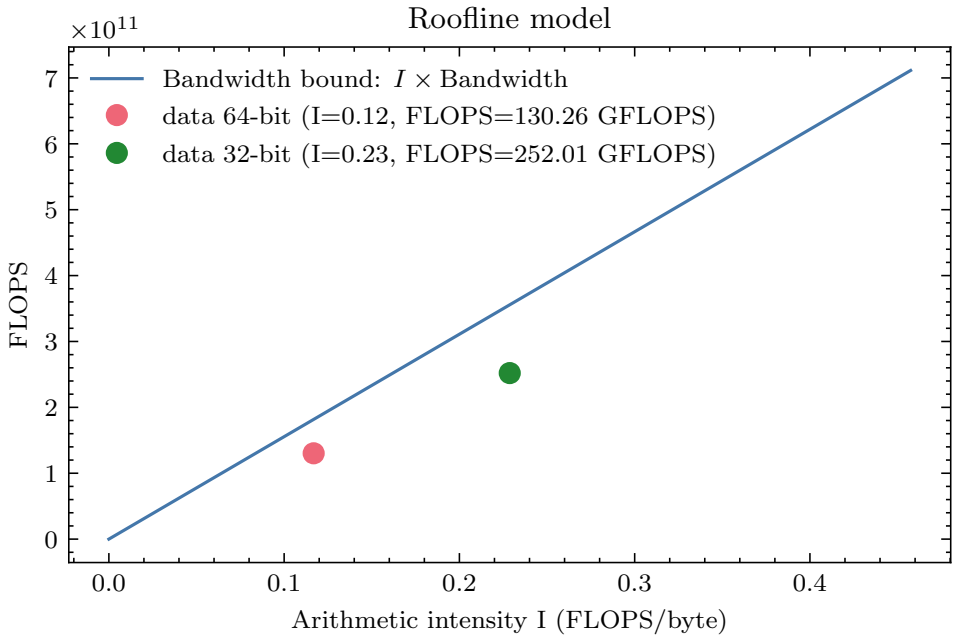


Figure 26: Roofline model (log scale). The bandwidth bound is computed as the product between the arithmetic intensity and the memory bandwith of the GPU.

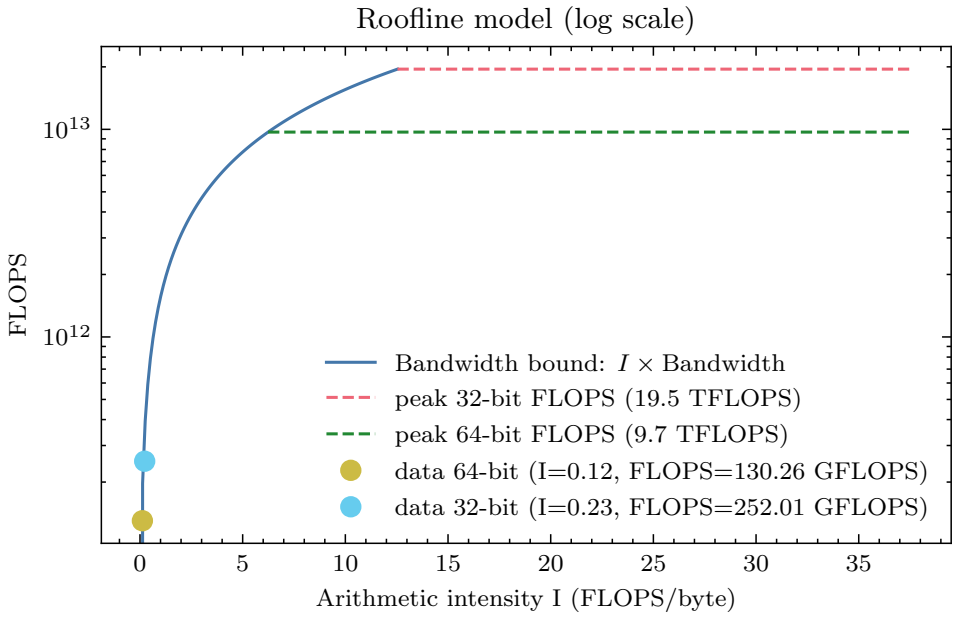


Figure 27: Roofline model.

As we can see in Figure 26 and Figure 27, we are clearly in the memory bound regime, which means that it is the memory bandwidth and not the computational power that is the limiting factor. Another thing we can do is to measure the peak performance ratio. This is the ratio between the theoretical peak performance and the performance obtained with our code. Since the problem is memory bound, the theoretical peak performance is given by the memory bandwidth divided by the number of read and written bytes for one particle. This theoretical result is then compared to the performance obtained with our code (the number of particles times the number of iterations divided by the time needed to perform the simulation).

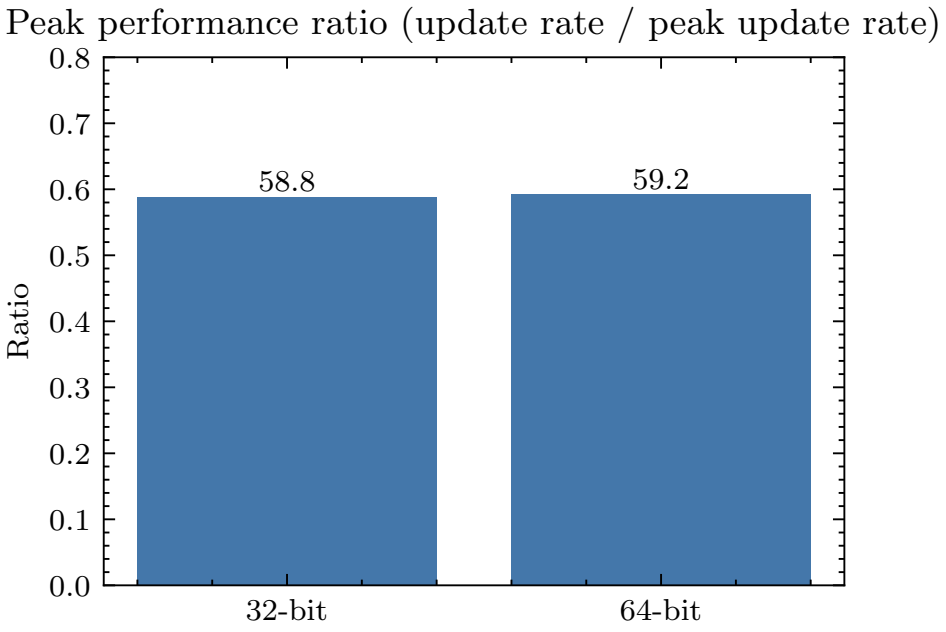


Figure 28: Peak performance ratio.

The results given in Figure 28 are slightly disappointing since we would have expected a ratio closer to 80-90%. However, we have to keep in mind that the peak performance ratio is only valid when we use a large portion of the GPU memory. In our case, we only use 1/4 of the memory because of limitations of the system memory. This means that we are not able to use the full memory bandwidth of the GPU. With a Nvidia RTX 3090 GPU, we were able to obtain 75% of the peak performance ratio by using 1/2 of the memory. It is also possible that the code is simply not optimized enough. Nevertheless, the obtained results are within the order of magnitude one would expect from such a simulation on a GPU.

The next step is to compare the different levels of modeling. We will compare the results we obtained for the simplest level (without any interaction between the particles) to the other cases. The results are shown in the following logscale bar plot.

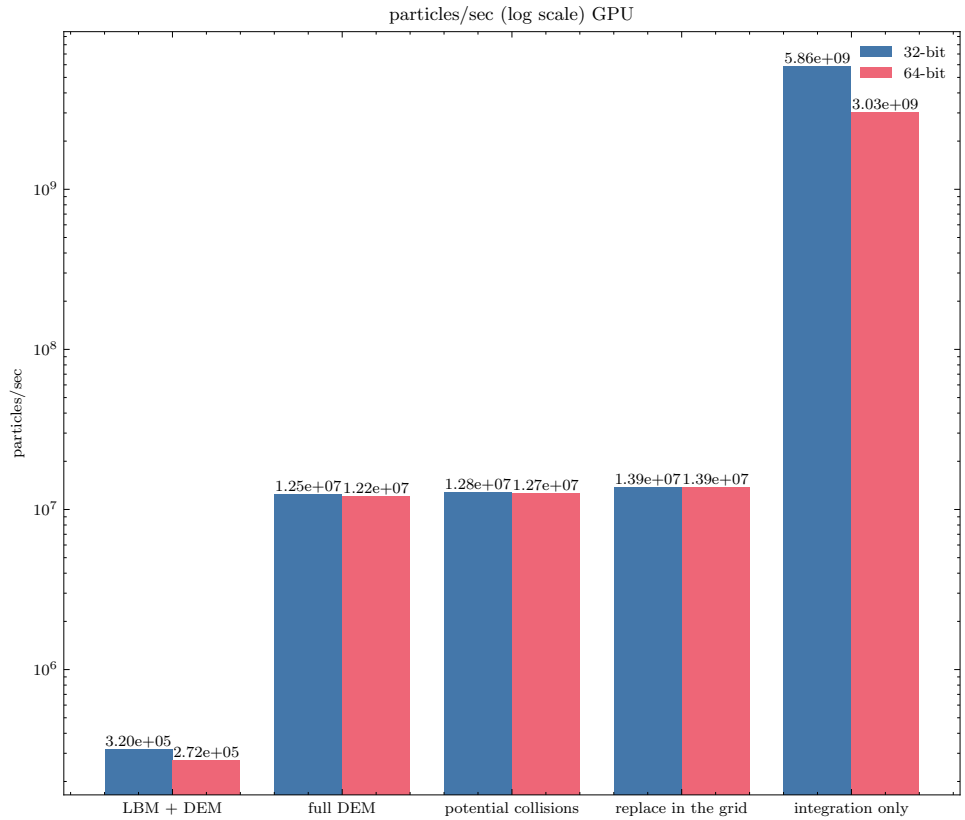


Figure 29: Comparing the different levels of modeling

Comparing the different levels of modeling, we observe in Figure 29 various performance results for different simulation cases. In the simple scenario where only particle movements are considered, the 32-bit case performs approximately two times faster than the 64-bit case. This outcome is expected as the problem is memory-bound and particles in 32-bit representation are represented by approximately 2 times fewer bits (approximately because some integer values remain unchanged). In cases where the modeling becomes more complex, such as moving the particles in the uniform grid, computing the potential collisions and with the full DEM simulations, the three methods take almost the same amount of time. However, these simulations are significantly slower than the first scenario,

being around 400 times slower than the integration-only level of modeling for the 32-bit case. As we can see, the difference in performance between the 32-bit and 64-bit cases is not very significant in the full simulation case. This is due to the fact that most of the computation is done on integer values and not on floating point values (the list of the potential collisions that needs to be sorted, the uniform grid structure, etc.). However, the results we obtain for the DEM simulation are also dependent on the number of collisions at each step and thus of the volume fraction of the system. In fact, if we consider a system with a very low volume fraction, the number of collisions will be very low and the performance will be much better. With our volume fraction of 0.4, we have an average of 6 potential collisions per particle at each step for the full DEM simulation. This number is not necessarily representative of the real number of collisions since it also depends on the granularity of the uniform grid structure. Incorporating coupling with the fluid simulation significantly impacts performance, making it approximately 40 times slower than the DEM-only simulation. This is reasonable given that there are far more fluid cells than solid particles (21'952'000 fuild cells for 33'000 spheres for the full DEM simulation), resulting in a more computationally intensive process. In this latter case, what influences the performance of the simulation is the number of fluid cells. This number is directly related to the number of cells needed to obtain the diameter of a sphere. In our case, the diameter of a sphere corresponds to 8 fluid cells, which is just sufficient to have a resolved simulation.

4.2.1 CPU/GPU speedup

It is also interesting to observe the performance comparison between a high-end CPU and the GPU. The CPU used is a double socket Intel Xeon Gold 6240R CPU, operating at 2.40GHz and featuring a total of 48 cores. We can observe the performances of the different modeling levels on the CPU in Figure 30.

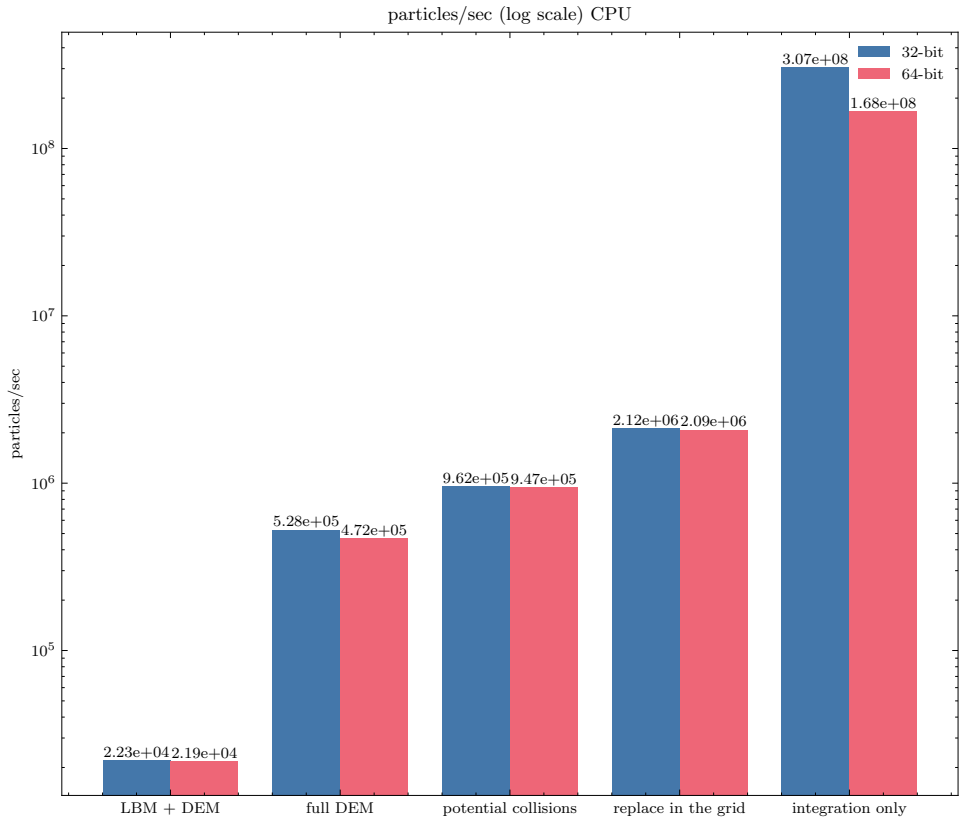


Figure 30: CPU performances

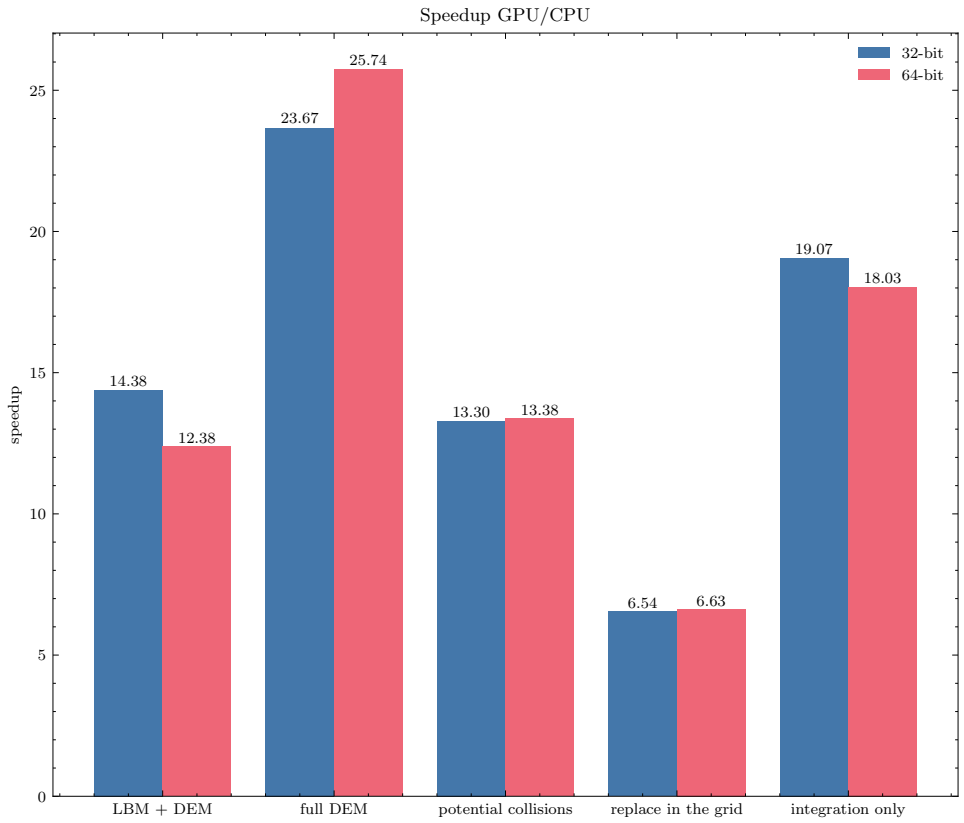


Figure 31: GPU/CPU speedup

As we can see in Figure 31, for the integration-only scenario, the GPU demonstrates an impressive speedup of almost 20 percent compared to the CPU. In the case of replacement in the grid, there is only a very small speedup, less than 7, when using the GPU instead of the CPU. However, in the full DEM simulation, the GPU exhibits a much more significant speedup, approximately 25 times faster than the CPU. This improvement can be attributed to sorting algorithms that are better suited for a larger number of cores, allowing the GPU to effectively compute forces. In the combined LBM and DEM simulation, both the 32-bit and 64-bit cases show substantial speedups. The 32-bit case achieves a speedup of approximately 14, while the 64-bit case reaches a speedup of around 12. These results are not surprising since simulations of fluid are well-suited for GPUs, making them more efficient for these types of calculations.

It should be noted that the change in volume fraction does not affect execution time in the same way on the GPU and the CPU. In Figure 32, it can be observed that on the GPU, there is a factor of 4 between the execution time without suspended particles and the execution time for a volume fraction of 0.45. On the CPU, increasing the volume fraction seems to have only a minor impact on performance. Several hypotheses could explain this difference in behavior. One hypothesis could be that on the GPU, the coupling involves the use of relatively complex kernels (involving loops and conditions), and that GPUs might struggle more to handle such cases, whereas on the CPU, parallelization being managed differently, it would pose fewer problems. Another hypothesis could be that the compiler on the GPU performs a number of optimizations that handle cases with very few particles in the system exceptionally well. In any case, it should be noted that these results are preliminary, and it would be necessary to conduct a larger number of tests, using for example a profiler, to really understand and explain these differences in behavior between the CPU and the GPU.

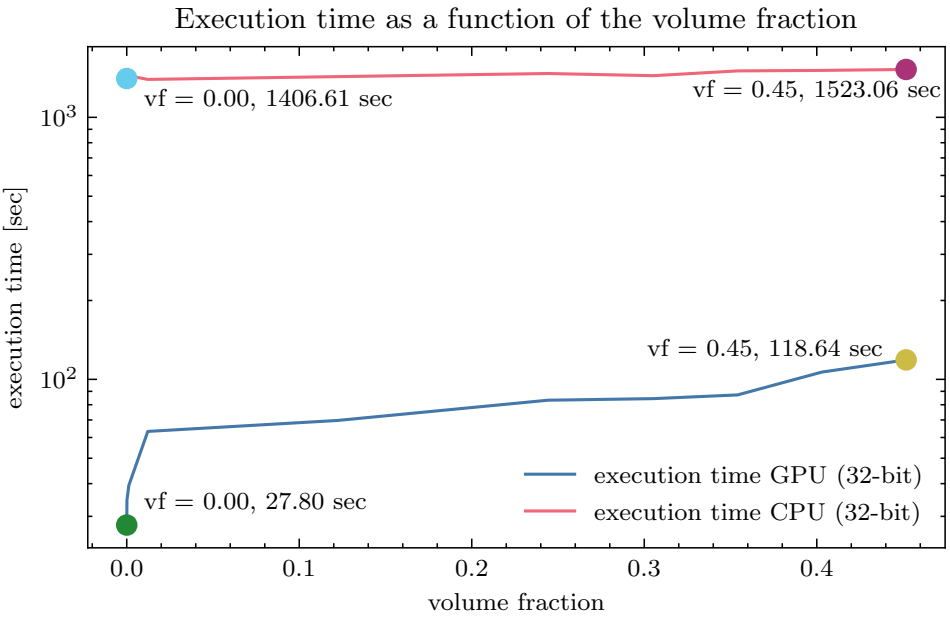


Figure 32: Execution time for different volume fractions.

4.3 Apparent viscosity in a Couette flow

A Couette flow is an experiment where a fluid is contained between two moving parallel plates. The two plates are moving with an opposite and constant relative velocity and a shearing effect is thus exerted by the moving plates on the fluid. With a Newtonian fluid, the velocity profile of the fluid along the axis perpendicular to the plates is linear. However, when particles are suspended in the fluid, the behavior of the flow changes and the fluid behaves like a non Newtonian fluid characterized by a "S-shaped" velocity profile (the change in velocity in greater when we are near the plates). Let us assume that the plates are moving along the y axis and that they are positioned in the yz-plane at $z = 0$ for the lower plate and at $z = h$ for the upper plate. The wall shear stress is defined as the total force applied on one of the plates in the y-direction, divided by the surface area of that plate. We can decompose the shear stress into two components: one component is due to the forces exerted by the fluid on the wall, and the other component is due to the forces exerted by the solid particles on the wall. Taking the lower plate as an example (similar results apply to the upper plate), we have the following formula for the wall shear stress τ_w (its SI unit is pascal) [31]:

$$\tau_w = \frac{F_{tot}}{S} = \underbrace{\nu_0 \left(\frac{\partial u_y}{\partial x}\right)_{x=0}}_{\text{fluid contribution}} + \underbrace{\frac{F_{solids}}{S}}_{\text{solids contribution}}, \tag{105}$$

where S is the surface of one of the plates, u_y is the fluid velocity in the y direction, ν_0 is the fluid viscosity and F_{solids} is the forces exerted by the solids on the bottom plate. The velocity gradient at the wall is computed using finite differences:

$$\left(\frac{\partial u_y}{\partial x}\right)_{x=0} \approx \frac{u_w^{\text{bottom}} - u_y^{\text{bottom}}}{0.5 \cdot \Delta x} \tag{106}$$

where u_w^{bottom} is the velocity of the bottom plate and u_y^{bottom} is the average fluid velocity in the y-direction for the fluid cells adjacent to the bottom wall (the calculation is similar, taking the upper wall as the reference). The factor of 0.5 in front of Δx is due to the fact that the physical position of the wall is at the midpoint between the fluid cell and the wall cell (see section 3.2.5). The shear rate $\dot{\gamma}$ is defined as the difference of velocity between the 2 plates divided by the distance between them (its SI unit is reciprocal seconds):

$$\dot{\gamma} = \frac{\Delta u_y}{h} = \frac{u_w^{\text{top}} - u_w^{\text{bottom}}}{h}, \tag{107}$$

where Δu_y is the difference of velocity between the two plates, u_w^{top} is the velocity of the top plate and h is the distance between the two plates. The apparent viscosity is then given by the ratio of the shear

stress to the shear rate:

$$\nu_{app} = \frac{\tau_w}{\dot{\gamma}}. \tag{108}$$

Since the apparent viscosity depends on the value of the fluid viscosity ν_0, it is practical to normalize the apparent viscosity by ν_0:

$$\tilde{\nu}_{app} = \frac{\nu_{app}}{\nu_0} = \frac{\tau_w}{\dot{\gamma}\nu_0}. \tag{109}$$

Einstein discovered in 1906 the relationship between the apparent viscosity and the volume fraction for a liquid containing suspended spheres in a shear flow [32]:

$$\nu_{app} = \nu_0(1 + \frac{5}{2}\phi). \tag{110}$$

However, the results he found are only valid for a low volume fraction ($\phi < 0.02$ [33]), where the contribution of particles to the shear stress remains small. A more recent law, known as the Krieger-Dougherty law [34], allows for a relatively accurate representation of the evolution of the apparent viscosity with higher volume fractions. This law is given by the following formula:

$$\nu_{app} = \nu_0(1 - \frac{\phi}{\phi_{sat}})^{-\frac{5}{2}\phi_{sat}}, \tag{111}$$

where ϕ_{sat} is the volume fraction at which the fluid is completely saturated by the solids, which, according to this model, is the value at which the fluid flow is completely jammed. In theory, the close-packing of equal spheres implies a maximum volume fraction of $\phi_{sat} \approx 0.74$. however, in practice the spheres are not arranged and we use the volume fraction of a random close pack of spheres, which is about $\phi_{sat} \approx 0.64$ [33].

In the following simulations, we will compute the apparent viscosity of the fluid for different volume fractions of spheres, ranging from 0.09 to 0.52 and a number of spheres from 1000 to 6000 depending on the volume fraction. The spacing between the lattice cells is $\Delta x = 0.25$ and the simulation time step we chose is $\Delta t = 0.00625$ (see formula (23)) The domain size is specified as 20x50x50, representing the size of the domain along the x, y and z axis. The size of the LBM lattice is 80x200x200, meaning that we have a total of 3'200'000 lattice cells. All the spheres in the simulation have a radius of 1, meaning that diameter of a sphere is represented by 8 fluid cells. The viscosity of the fluid is specified as $\nu_0 = 3.2$ and the velocities of the walls in the y direction are 0.8 for the top wall and -0.8 for the bottom wall (in physical units), meaning that we have an absolute shear rate of 0.08. The simulation runs for a total of 100,000 steps, except for the last case with a volume fraction of 0.52 where we perform 400'000 iterations. The number of iterations is typically given in conjunction with the convective time t_c, which in our case corresponds to the time it takes for a particle to traverse the channel while assuming it has a velocity equal to that of the wall. In our case, we have a convective time (in physical units) of $t_c = \frac{50}{0.8} = 62.5$, which corresponds to 10,000 iterations with our time step

of $\Delta t = 0.00625$.

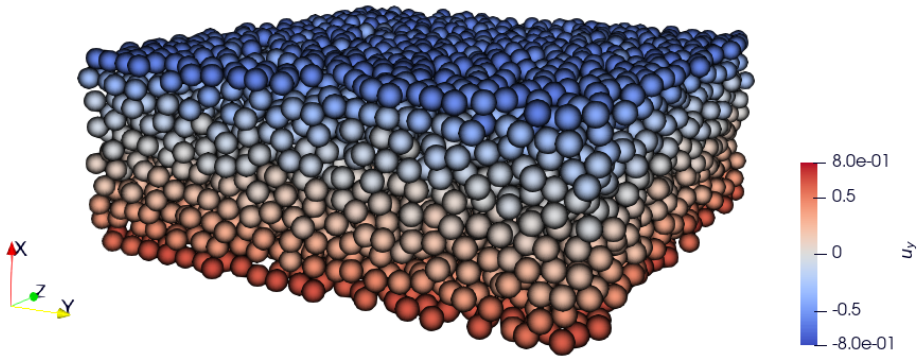


Figure 33: Particles in a Couette flow for a volume fraction of 0.48.

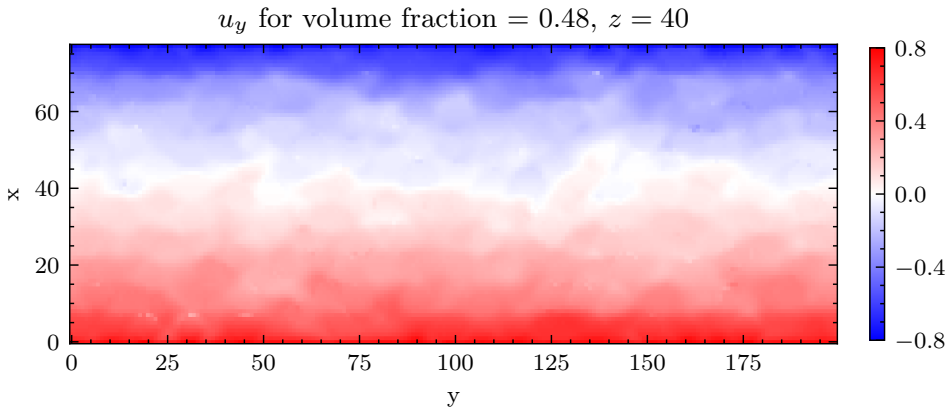


Figure 34: u_y velocity of the fluid at $z = 25$ for a volume fraction of 0.48.

To obtain accurate results, it is essential to wait for the simulation to stabilize. One simple way to assess this is by monitoring the evolution of the force exerted by the solids on the walls. For instance, with a volume fraction of 0.44, the obtained results are as follows (see Figure 35):

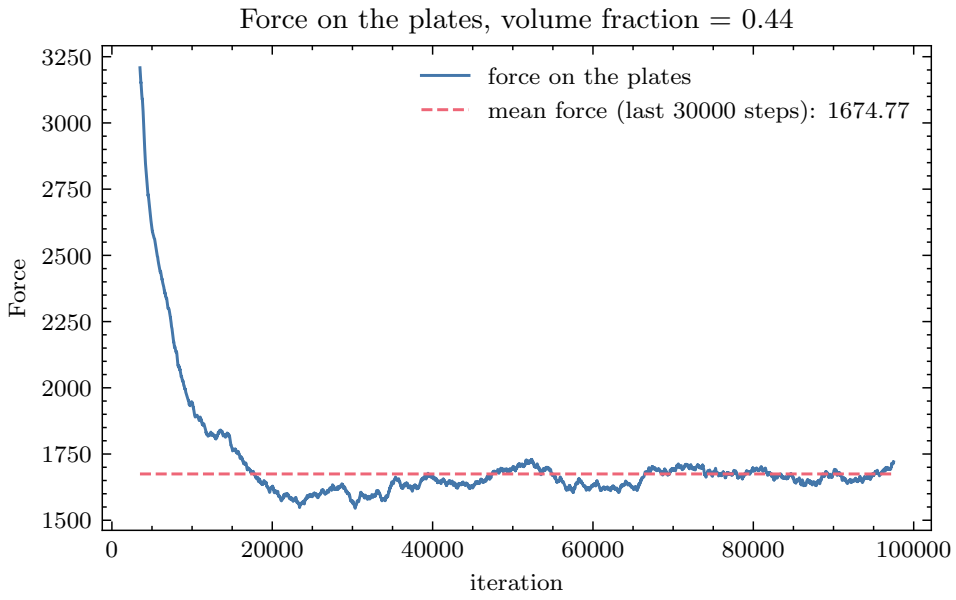


Figure 35: Force on the plates for a volume fraction of 0.44

It is suggested [33] to wait for at least a total time T such that $\dot{\gamma}T \approx 100$, to ensure that the system stabilizes. In our case, we observed that a value of $\dot{\gamma}T = 50$ was more than sufficient to achieve stabilization for almost all volume fractions, except for the case with a volume fraction of 0.52 where a minimum value of $\dot{\gamma}T = 200$ seems to be required. Therefore, it is more straightforward to monitor the evolution of the force on the walls to assess the stabilization of the system.

If we observe the evolution of the velocity profile as a function of the volume fraction (see Figure 36), we notice that when the volume fraction is relatively small, the particles have minimal influence on the fluid, and the velocity profile is close to that of a straight line (like a Newtonian fluid). As the volume fraction increases, the simulation becomes increasingly dominated by the interactions between particles, and we observe a more pronounced "S-shaped" profile.

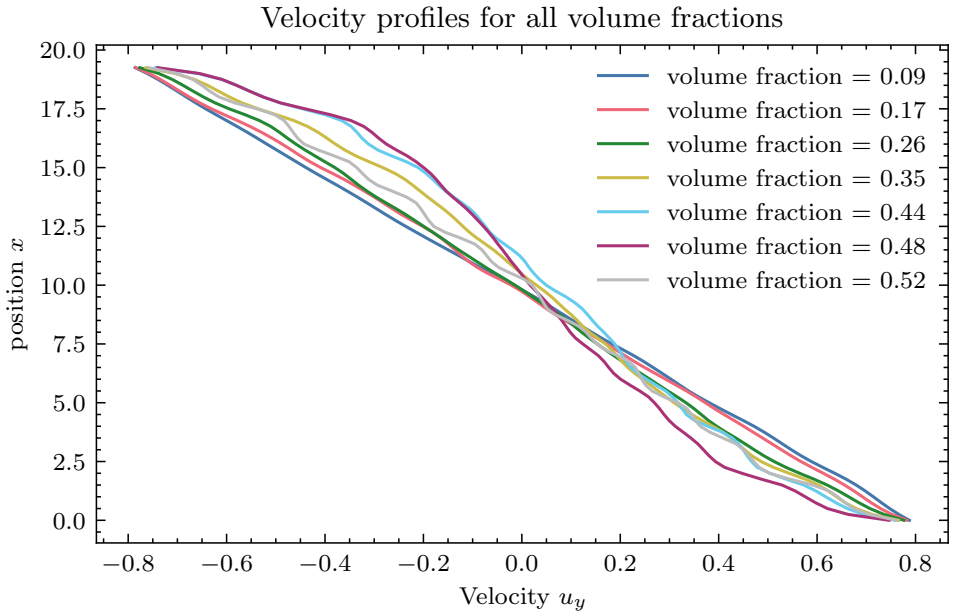


Figure 36: Velocity profile of the particles u_y velocity as a function of their x position.

By computing the normalized apparent viscosity for each volume fraction (see equation (109)), we can study the evolution of this value and compare it with theoretical models.

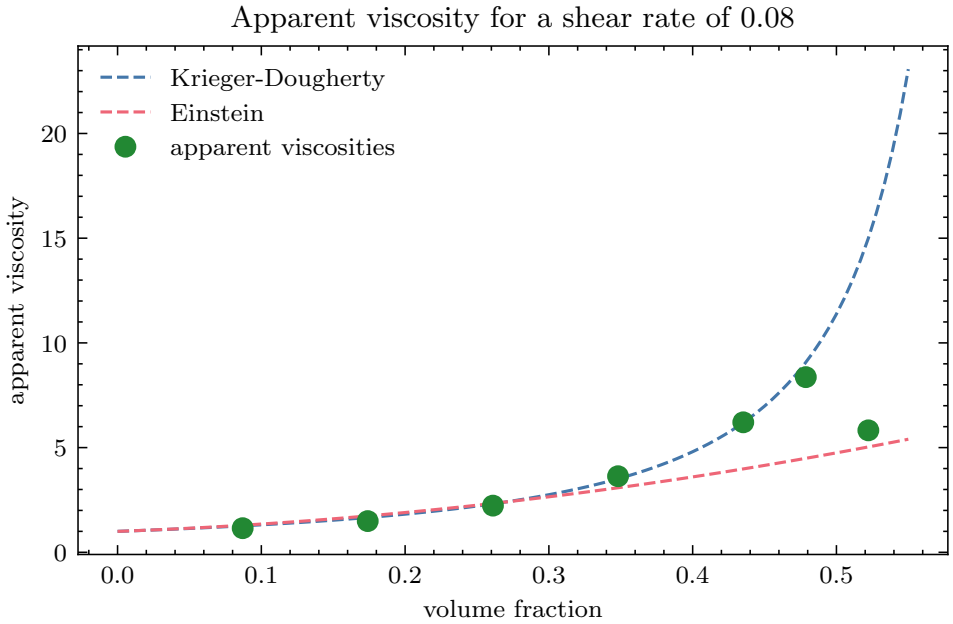


Figure 37: Normalized apparent viscosity of the fluid for different volume fractions with a constant shear rate of 0.08

The Einstein model, being valid for highly diluted solutions, is not expected to accurately predict the evolution of the apparent viscosity in our simulation. However, the Krieger-Dougherty model seems to correspond well to the results we obtain in our simulation up to a volume fraction of approximately 0.48 (see Figure 37). For the case where the volume fraction is 0.52, the normalized apparent

viscosity suddenly decreases and no longer corresponds to the model, despite the system "converging". One possible interpretation of this result could be related to the system's initialization. Initially, the particles are uniformly distributed in space. For volume fractions up to 0.48, the arrangement of the spheres quickly becomes random after a certain number of iterations. However, with a volume fraction of 0.52, a structure in the particle arrangement similar to the initial distribution persists, even after the system's "convergence". A random initial distribution of particles could potentially yield a different final result, but due to the forces between particles, it is not trivial to randomly arrange the particles when the volume fraction is very high. Another possibility is that the Krieger-Dougherty model is no longer suitable when the volume fraction exceeds a certain limit. When the volume fraction approaches the saturation volume fraction, other phenomena such as force chains [35] can form between particles, drastically changing the fluid's behavior. In any case, our model appears to be consistent with the Krieger-Dougherty model for volume fraction values similar to those of red blood cells in the blood, which is our primary interest.

5 Summary and outlook

In this master's thesis, we successfully developed a DEM-LBM coupling simulation framework that can be parallelized using standard C++. Additionally, with the Nvidia nvc++ compiler, we were able to port the code onto the GPU, resulting in significant speedup compared to a high-end CPU for both the DEM simulation and the DEM-LBM coupling simulation. The LBM model we used is based on the TRT model and the interactions with wall are performed using the bounce-back method with velocities. On the other hand, the DEM model handles collisions of solid spheres using a spring-dashpot model and detects collisions through a uniform grid structure to reduce the initial $O(n^2)$ complexity. The coupling between the two simulations is achieved using the partially saturated bounce-back method, requiring the computation of the solid fraction. We implemented a simple linear approximation based only on the distance to determine the solid fraction. The DEM collision model accurately respected the conservation of energy for perfectly elastic collisions, and the behavior of collisions with obstacles, as seen in the tests with different collision angles, was also accurately represented. Moreover, the coupling simulation performed well even for high volume fractions, with apparent viscosity in a shear flow consistent with theory up to 48%, which closely matches the hematocrit levels we have in blood. However, at higher volume fractions, some divergence from theory was observed. For future improvements, one natural step would be to implement ellipsoids or superquadrics in the DEM to better represent the shape of RBCs. For the coupling, it would also be interesting to implement a more precise calculation of the solid fraction, for example, by using the linear approximation developed by Jones and Williams [6]. Following the work of Melchionna [2], introducing a different fluid viscosity inside and outside the RBCs, along with considering the tank threading motion in an ad hoc way, could lead to further enhancements of the model, allowing us to realistically simulate blood flows. However, more ambitious improvements would involve introducing fully deformable RBCs, requiring an overhaul of the entire DEM model. With these advancements, extensive tests on rheology would be necessary to determine the simulation's accuracy in representing blood flow dynamics.

Bibliography

[1] Christoph Kloss, Christoph Goniva, Alice König, Stefan Amberger, and Stefan Pirker. Models, algorithms and validation for opensource DEM and CFD-DEM. *Progress in Computational Fluid Dynamics*, 12:140–152, June 2012. doi: 10.1504/PCFD.2012.047457.

[2] Simone Melchionna. A Model for Red Blood Cells in Simulations of Large-scale Blood Flows. *Macromolecular Theory and Simulations*, 20(7):548–561, 2011. ISSN 1521-3919. doi: 10.1002/mats.201100012. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/mats.201100012. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/mats.201100012.

[3] Timm Krüger. *Computer Simulation Study of Collective Phenomena in Dense Suspensions of Red Blood Cells under Shear*. Vieweg+Teubner Verlag, Wiesbaden, 2012. ISBN 978-3-8348-2375-5 978-3-8348-2376-2. doi: 10.1007/978-3-8348-2376-2. URL http://link.springer.com/10.1007/978-3-8348-2376-2.

[4] Christoph Rettinger and Ulrich Rüde. An efficient four-way coupled lattice Boltzmann - discrete element method for fully resolved simulations of particle-laden flows. Technical Report arXiv:2003.01490, arXiv, March 2020. URL http://arxiv.org/abs/2003.01490. arXiv:2003.01490 [physics] type: article.

[5] Christoph Rettinger, Sebastian Eibl, Ulrich Rüde, and Bernhard Vowinckel. Rheology of mobile sediment beds in laminar shear flow: effects of creep and polydispersity. *Journal of Fluid Mechanics*, 932:A1, February 2022. ISSN 0022-1120, 1469-7645. doi: 10.1017/jfm.2021.870. URL https://www.cambridge.org/core/product/identifier/S0022112021008703/type/journal_article.

[6] Bruce D. Jones and John R. Williams. Fast computation of accurate sphere-cube intersection volume. *Engineering Computations*, 34(4):1204–1216, January 2017. ISSN 0264-4401. doi: 10.1108/EC-02-2016-0052. URL https://doi.org/10.1108/EC-02-2016-0052. Publisher: Emerald Publishing Limited.

[7] Timm Krüger, Halim Kusumaatmaja, Alexandr Kuzmin, Orest Shardt, Goncalo Silva, and Erlend Magnus Viggen. *The Lattice Boltzmann Method: Principles and Practice*. Graduate Texts in Physics. Springer International Publishing, Cham, 2017. ISBN 978-3-319-44647-9 978-3-319-44649-3. doi: 10.1007/978-3-319-44649-3. URL http://link.springer.com/10.1007/978-3-319-44649-3.

[8] Sauro Succi. *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond*. Numerical Mathematics and Scientific Computation. Oxford University Press, Oxford, New York, August 2001. ISBN 978-0-19-850398-9.

[9] P. L. Bhatnagar, E. P. Gross, and M. Krook. A Model for Collision Processes in Gases. I. Small Amplitude Processes in Charged and Neutral One-Component Systems. *Physical Review*, 94 (3):511–525, May 1954. doi: 10.1103/PhysRev.94.511. URL https://link.aps.org/doi/10.1103/PhysRev.94.511. Publisher: American Physical Society.

[10] Irina Ginzburg. Equilibrium-type and link-type lattice Boltzmann models for generic advection and anisotropic-dispersion equation. *Advances in Water Resources*, 28(11):1171–1195, November 2005. ISSN 03091708. doi: 10.1016/j.advwatres.2005.03.004. URL https://linkinghub.elsevier.com/retrieve/pii/S0309170805000874.

[11] Irina Ginzburg and Frederik Verhaeghe. Two-Relaxation-Time Lattice Boltzmann Scheme: About Parametrization, Velocity, Pressure and Mixed Boundary Conditions. *Commun. Comput. Phys.*, 2008.

[12] Jonas Latt. Choice of units in lattice boltzmann simulations. 2008.

[13] Jonas Latt, Christophe Coreixas, and Joël Beny. Cross-platform programming model for many-core lattice Boltzmann simulations. *PLOS ONE*, 16(4):1–29, 04 2021. doi: 10.1371/journal.pone.0250306. URL https://doi.org/10.1371/journal.pone.0250306.

[14] Thorsten Pöschel and Thomas Schwager. *Computational granular dynamics: models and algorithms*. Springer-Verlag, Berlin ; New York, 2005. ISBN 978-3-540-21485-4. OCLC: ocm60613128.

[15] R. D. Mindlin. Compliance of Elastic Bodies in Contact. *Journal of Applied Mechanics*, 16(3):259–268, April 2021. ISSN 0021-8936. doi: 10.1115/1.4009973. URL https://doi.org/10.1115/1.4009973. _eprint: https://asmedigitalcollection.asme.org/appliedmechanics/article-pdf/16/3/259/6746333/259_1.pdf.

[16] LIGGGHTS. gran model hertz model — LIGGGHTS v3.X documentation. URL https://www.cfdem.com/media/DEM/docu/gran_model_hertz.html.

[17] Peter Haupt. *Continuum Mechanics and Theory of Materials*. Springer Science & Business Media, March 2002. ISBN 978-3-540-43111-4.

[18] Heinrich Hertz. Ueber die berührung fester elastischer körper. *Journal für die reine und angewandte Mathematik*, 92:156–171, 1882. URL http://eudml.org/doc/148490.

[19] Hamid Reza Norouzi, Reza Zarghami, Rahmat Sotudeh-Gharebagh, and Navid Mostoufi. *Coupled CFD-DEM modeling: formulation, implementation and application to multiphase flows*. Wiley, Chichester, West Sussex, United Kingdom, 2016. ISBN 978-1-119-00513-1.

[20] Nikolai V. Brilliantov, Frank Spahn, Jan-Martin Hertzsch, and Thorsten Pöschel. Model for collisions in granular gases. *Physical Review E*, 53(5):5382–5392, may 1996. doi: 10.1103/physreve.53.5382. URL https://doi.org/10.1103%2Fphysreve.53.5382.

[21] Donald E. Knuth. *The Art of Computer Programming: Volume 3: Sorting and Searching*. Addison-Wesley Professional, April 1998. ISBN 978-0-321-63578-5. Google-Books-ID: cYULBAAAQBAJ.

[22] Loup Verlet. Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules. *Physical Review*, 159(1):98–103, July 1967. ISSN 0031-899X. doi: 10.1103/PhysRev.159.98. URL https://link.aps.org/doi/10.1103/PhysRev.159.98.

[23] Giang T. Nguyen, Ei L. Chan, Takuya Tsuji, Toshitsugu Tanaka, and Kimiaki Washino. Resolved cfd–dem coupling simulation using volume penalisation method. *Advanced Powder Technology*, 32(1):225–236, 2021. ISSN 0921-8831. doi: https://doi.org/10.1016/j.apt.2020.12.004. URL https://www.sciencedirect.com/science/article/pii/S0921883120305410.

[24] D. R. Noble and J. R. Torczynski. A Lattice-Boltzmann Method for Partially Saturated Computational Cells. *International Journal of Modern Physics C*, 09(08):1189–1201, December 1998. ISSN 0129-1831, 1793-6586. doi: 10.1142/S0129183198001084. URL https://www.worldscientific.com/doi/abs/10.1142/S0129183198001084.

[25] Chrysovalantis Tsigginos, Jianping Meng, Xiao-Jun Gu, and David R. Emerson. Coupled LBM-DEM simulations using the partially saturated method: Theoretical and computational aspects. *Powder Technology*, 405:117556, June 2022. ISSN 00325910. doi: 10.1016/j.powtec.2022.117556. URL https://linkinghub.elsevier.com/retrieve/pii/S0032591022004508.

[26] Philippe Seil. LBDEMcoupling: implementation, validation, and applications of a coupled open-source solver for fluid-particle systems / eingereicht von Dipl.-Ing. Philippe Seil. 2016. URL https://www.semanticscholar.org/paper/LBDEMcoupling%3A-implementation%2C-validation%2C-and-of-a-Seil/552b6157d2b61ff912d87a978eb3427558e3523b.

[27] Nvidia Corporation. NVIDIA HPC Compilers Reference Guide, 2023. URL https://docs.nvidia.com/hpc-sdk/compilers/hpc-compilers-user-guide/index.html. Place: Santa Clara, California, United States.

[28] Y. C. Chung and J. Y. Ooi. Benchmark tests for verifying discrete element modelling codes at particle impact level. *Granular Matter*, 13(5):643–656, October 2011. ISSN 1434-5021, 1434-7636. doi: 10.1007/s10035-011-0277-0. URL http://link.springer.com/10.1007/s10035-011-0277-0.

[29] Antonio Doménech-Carbó. Analysis of rolling friction effects on oblique rebound by redefining tangential restitution and friction. *Physics of Fluids*, 31(4):043302, April 2019. ISSN 1070-6631, 1089-7666. doi: 10.1063/1.5091733. URL http://aip.scitation.org/doi/10.1063/1.5091733.

[30] Samuel F. Foerster, Michel Y. Louge, Hongder Chang, and Khédidja Allia. Measurements of the collision properties of small spheres. *Physics of Fluids*, 6(3):1108–1115, March 1994. ISSN 1070-6631, 1089-7666. doi: 10.1063/1.868282. URL http://aip.scitation.org/doi/10.1063/1.868282.

[31] Bijay Sultanian. *Fluid Mechanics: An Intermediate Approach*. 07 2015. ISBN 9780429168406. doi: 10.1201/b18762.

[32] A. Einstein. Eine neue Bestimmung der Moleküldimensionen. *Annalen der Physik*, 324(2):289–306, 1906. ISSN 1521-3889. doi: 10.1002/andp.19063240204. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/andp.19063240204. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/andp.19063240204.

[33] Y. Thorimbert. Lattice boltzmann simulations of complex flows. (1), 2019. doi: 10.13097/archive-ouverte/unige:129125. URL https://https://archive-ouverte.unige.ch/unige:129125.

[34] Irvin M. Krieger and Thomas J. Dougherty. A Mechanism for Non-Newtonian Flow in Suspensions of Rigid Spheres. *Transactions of The Society of Rheology*, 3(1):137–152, March 1959. ISSN 0038-0032. doi: 10.1122/1.548848. URL https://doi.org/10.1122/1.548848.

[35] John Peters, Maya Muthuswamy, Johannes Wibowo, and Antoinette Tordesillas. Characterization of force chains in granular material. *Physical review. E, Statistical, nonlinear, and soft matter physics*, 72:041307, November 2005. doi: 10.1103/PhysRevE.72.041307.