UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA CIVIL E AMBIENTAL

A METHODOLOGY FOR FLUID-SOLID INTERACTION BASED ON LBM-DEM-IMB COUPLING

JOAQUIM ARAÚJO COSTA NETO

ORIENTADOR: MÁRCIO MUNIZ DE FARIAS

COORIENTADOR: LEANDRO LIMA RASMUSSEN

TESE DE DOUTORADO EM GEOTECNIA

PUBLICAÇÃO: G.TD-213/2026

BRASÍLIA/DF: FEVEREIRO DE 2026

UNIVERSIDADE DE BRASÍLIA

FACULDADE DE TECNOLOGIA

DEPARTAMENTO DE ENGENHARIA CIVIL E AMBIENTAL

A METHODOLOGY FOR FLUID-SOLID INTERACTION BASED ON LBM-DEM-IMB COUPLING

JOAQUIM ARAÚJO COSTA NETO

TESE DE DOUTORADO SUBMETIDA AO DEPARTAMENTO DE ENGENHARIA CIVIL DA UNIVERSIDADE DE BRASÍLIA COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA OBTENÇÃO DO GRAU DE DOUTOR.

APROVADA POR:

_____

MÁRCIO MUNIZ DE FARIAS, PhD (ENC/UnB)

(ORIENTADOR)

_____

LEANDRO LIMA RASMUSSEM, PhD (UnB)

(COORIENTADOR)

_____

MANOEL PORFÍRIO CORDÃO NETO, PhD (UnB)

(EXAMINADOR EXTERNO)

_____

ROBINSON ANDRÉS GIRALDO ZULUAGA, PhD (UFG)

(EXAMINADOR EXTERNO)

_____

MAURÍCIO MARTINES SALES, PhD (UFG)

(EXAMINADOR EXTERNO)

DATA: BRASÍLIA/DF, 05 DE FEVEREIRO DE 2020

FICHA CATALOGRÁFICA

COSTA NETO, JOAQUIM ARAÚJO

A Methodology for Fluid-Solid Interaction Based on LBM-DEM-IMB Coupling . [Distrito Federal] 2026

xviii, 129 p., 210x297 mm (ENC/FT/UnB, Doutor, Geotecnia, 2026)

Tese de Doutorado – Universidade de Brasília, Faculdade de Tecnologia

Departamento de Engenharia Civil e Ambiental

Palavras chaves:

1. Lattice Boltzmann Method
2. Discrete Element Method
3. Immersed Moving Boundary Method
4. Fluid in Porous Media

I. ENC/FT/UnB II. Doutor

REFERÊNCIA BIBLIOGRÁFICA

COSTA NETO, J. A. (2026). A Methodology for Fluid-Solid Interaction Based on LBM-DEM-IMB Coupling. Tese de Doutorado, Publicação G.TD-213/2026, Departamento de Engenharia Civil e Ambiental, Universidade de Brasília, Brasília, DF, 129 p.

CESSÃO DE CRÉDITOS

NOME DO AUTOR: Joaquim Araújo Costa Neto
TÍTULO DA TESE DE DOUTORADO: A Methodology for Fluid-Solid Interaction Based on LBM-DEM-IMB Coupling
GRAU/ANO: Doutor / 2026

_____

Joaquim Araújo Costa Neto

Universidade de Brasília, Secretaria da Coordenação de Pós-Graduação em Geotecnia
Campus Darcy Ribeiro, Departamento de Engenharia Civil e Ambiental. Prédio SG-12, Universidade de Brasília. CEP 70910-900 – Brasília, DF – Brasil

engcjoaquim@gmail.com

DEDICATÓRIA

Dedico esta dissertação

ao meu pai Evaristo e minha mãe Maria.

AGRADECIMENTOS

A Deus, agradeço primeiramente por ter me concedido saúde, força e sabedoria ao longo de toda esta jornada. Sua presença foi essencial nos momentos de dificuldade, guiando minhas escolhas e sustentando minha perseverança até a conclusão deste trabalho.

Aos meus pais, expresso minha mais profunda gratidão pelo amor incondicional, apoio constante e pelos valores que sempre me ensinaram. Tudo o que sou e conquistei é reflexo dos ensinamentos, do incentivo e da confiança que sempre depositaram em mim.

À minha noiva e futura esposa, Lorena Souza Lago, agradeço pelo amor, apoio, paciência e incentivo ao longo de toda esta trajetória. Sua presença constante e parceria foram essenciais para superar os desafios e alcançar a conclusão deste trabalho, sendo esta conquista também fruto do seu apoio.

Aos meus orientadores, agradeço pela orientação técnica, disponibilidade e pelos valiosos ensinamentos compartilhados durante o desenvolvimento desta pesquisa, fundamentais para o amadurecimento científico deste trabalho.

À Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) e à Furnas Centrais Elétricas, por meio do projeto de Pesquisa e Desenvolvimento (P&D Sedimentos), agradeço pelo apoio financeiro, indispensável para a realização desta pesquisa.

Aos colegas da Universidade de Brasília (UnB), agradeço pelo convívio, pelas discussões acadêmicas, pela troca de conhecimentos e pelo apoio ao longo desta trajetória. O ambiente colaborativo e o companheirismo foram essenciais para tornar essa caminhada mais leve e enriquecedora.

Dedico um agradecimento especial à minha avó e ao meu pai, que infelizmente faleceram durante o desenvolvimento desta pesquisa. Mesmo ausentes fisicamente, suas presenças foram constantes em minha memória, em meus pensamentos e na força que encontrei para seguir em frente. Este trabalho também é dedicado a eles, como forma de honra, gratidão e eterno reconhecimento por tudo o que representaram em minha vida.

Por fim, agradeço a todos que, direta ou indiretamente, contribuíram para a realização deste trabalho e para minha formação pessoal e profissional.

Uma Metodologia de Interação Fluido-Sólido Baseada no Acoplamento LBM-DEM-IMB.

RESUMO

Esta pesquisa apresenta o desenvolvimento e a validação do FSIT (Fluid–Solid Interaction Toolkit), um arcabouço computacional projetado para simulações bidimensionais de problemas de interação fluido–sólido. O FSIT foi implementado em linguagem C++ e combina o Método Lattice Boltzmann (LBM) para a simulação do escoamento do fluido, o Método dos Elementos Discretos (DEM) para a mecânica dos sólidos e o acoplamento hidrodinâmico por meio da técnica de Fronteira Móvel Imersa (Immersed Moving Boundary – IMB). O framework oferece suporte aos operadores de colisão BGK e MRT, permitindo simulações em uma ampla faixa de regimes de escoamento, com maior estabilidade numérica e acurácia. A fase fluida é modelada por meio do LBM utilizando uma malha do tipo D2Q9, enquanto as partículas sólidas são representadas pelo DEM com leis de contato elástico lineares e integração temporal do movimento das partículas realizada pelo método Leapfrog. O acoplamento hidrodinâmico entre as fases fluida e sólida é obtido por meio da formulação IMB, possibilitando uma troca de momento consistente, bem como o cálculo das forças e torques hidrodinâmicos atuantes sobre os elementos discretos. As capacidades do FSIT são validadas por meio de uma série de casos clássicos de referência, incluindo o fluxo de Poiseuille, o escoamento em cavidade com tampa móvel (lid-driven cavity flow), o decaimento do vórtice de Taylor–Green, o escoamento ao redor de partículas circulares e reais, o escoamento em meios porosos formados por empacotamentos de elementos discretos e a sedimentação de partículas governada pela Lei de Stokes. Os resultados numéricos apresentam forte concordância com soluções analíticas, observações experimentais e correlações empíricas reportadas na literatura. As simulações em meios porosos demonstram a capacidade do FSIT em capturar caminhos preferenciais de escoamento e calcular com precisão valores de permeabilidade, os quais apresentam boa concordância com as equações clássicas de Kozeny–Carman e Ergun. As simulações envolvendo geometrias realistas de partículas reforçam ainda mais a robustez do framework na modelagem de cenários complexos de interação fluido–sólido. De forma geral, o FSIT mostra-se uma ferramenta confiável e versátil para a investigação de fenômenos de interação fluido–sólido em mesoescala, com elevado potencial de aplicação em problemas geotécnicos e de engenharia, incluindo análises em escala de laboratório e estudos orientados a processos.

A Methodology for Fluid-Solid Interaction Based on LBM-DEM-IMB Coupling

ABSTRACT

This research presents the development and validation of FSIT (Fluid–Solid Interaction Toolkit), a computational framework designed for two-dimensional simulations of fluid–solid interaction problems. FSIT is implemented in C++ and combines the Lattice Boltzmann Method (LBM) for fluid flow, the Discrete Element Method (DEM) for solid mechanics, and hydrodynamic coupling through the Immersed Moving Boundary (IMB) technique. The framework supports both the BGK and MRT collision operators, allowing simulations over a wide range of flow regimes with improved numerical stability and accuracy. The fluid phase is modeled using the LBM with a D2Q9 lattice, while solid particles are represented using DEM with linear elastic contact laws and Leapfrog time integration for particle motion. Hydrodynamic coupling between fluid and solid phases is achieved through the IMB formulation, enabling consistent momentum exchange and the computation of hydrodynamic forces and torques acting on discrete elements. The capabilities of FSIT are validated through a series of classical benchmark problems, including Poiseuille flow, lid-driven cavity flow, Taylor–Green vortex decay, flow around circular and realistic particles, porous media flow through discrete element packings, and particle settling governed by Stokes' law. Numerical results show strong agreement with analytical solutions, experimental observations, and empirical correlations reported in the literature. Porous media simulations demonstrate FSIT's ability to capture preferential flow paths and accurately compute permeability values, which compare well with the classical Kozeny–Carman and Ergun equations. Simulations involving realistic particle geometries further highlight the robustness of the framework in handling complex fluid–solid interaction scenarios. Overall, FSIT proves to be a reliable and versatile tool for investigating fluid–solid interaction phenomena at the mesoscale, with strong potential for geotechnical and engineering applications, including laboratory-scale analyses and process-oriented studies.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

Roman symbols

B	Buoyancy force
B_n	Solid fraction weighting function
C	Empirical constant (Kozeny–Carman / Ergun)
c	Lattice speed
c_i	Discrete velocity vector
c_s	Speed of sound (LBM)
D	Particle diameter
dt	Time step
dx	Lattice spacing
d_{un}	Normal overlap
d_{us}	Tangential displacement
F	Force vector
F_{hydro}	Hydrodynamic force
f_i	Particle distribution function
f_i^{eq}	Equilibrium distribution function
g	Gravitational acceleration
I	Moment of inertia
k	Permeability
k_n	Normal stiffness
k_s	Tangential stiffness
M_a	Mach number
m	Particle mass
n_{sub}	Number of DEM subcycles
p	Pressure
R	Particle radius
Re	Reynolds number
T	Torque
u	Fluid velocity vector
u_x, u_γ	Velocity components
u_{max}	Maximum velocity
V	Particle volume (unit depth in 2D)
W	Particle weight
w_i	Lattice weight

Greek symbols

ε	Porosity
μ	Dynamic viscosity
ν	Kinematic viscosity
ρ	Fluid density
ρ_f	Fluid density
ρ_s	Solid particle density
τ	Relaxation time
ϕ	Friction angle
Ω	Collision operator
Ω_s	Solid collision operator (IMB)

LIST OF ABBREVIATIONS

BGK	Bhatnagar–Gross–Krook collision operator
CFD	Computational Fluid Dynamics
D2Q9	Two-dimensional, nine-velocity lattice
DEM	Discrete Element Method
DSS	Direct Simple Shear
FDM	Finite Difference Method
FEM	Finite Element Method
FSIT	Fluid–Solid Interaction Toolkit
FVM	Finite Volume Method
IMB	Immersed Moving Boundary
KC	Kozeny–Carman
LBM	Lattice Boltzmann Method
LU	Lattice Units
MRT	Multiple Relaxation Time
P&D	Research and Development
RMS	Root Mean Square error

CHAPTER I

1 INTRODUCTION

Fluid-solid interaction (FSI) phenomena play a fundamental role in a wide range of natural processes and engineering applications, including particulate flows and sediment transport. These problems are characterized by the strong coupling between fluid dynamics and solid mechanics, often involving large numbers of interacting particles and complex geometries. Capturing such interactions remains a significant challenge in computational mechanics due to the multiphysics nature of the problem and the wide range of spatial and temporal scales involved.

The modeling of FSI systems requires approaches capable of representing both the fluid phase and the discrete nature of solid particles, as well as the mutual exchange of momentum and forces at the fluid-solid interface. Traditional computational fluid dynamics (CFD) methods based on continuum mechanics (finite element method, finite volume method, and finite difference method) have been successfully applied to many flow problems. However, their extension to fluid-solid scenarios involving many moving solids is often associated with high computation cost and complex mesh handling.

On the other hand, fully microscopic approaches that attempt to describe fluid behavior at the molecular level are computationally prohibitive for engineering-scale problems. As a result, mesoscopic modeling strategies have gained increasing attention as a compromise between physical fidelity and computation efficiency. In this context, the Lattice Boltzmann Method (LBM) has emerged as a robust alternative for fluid simulation, particularly well suited for problems involving complex geometries and moving boundaries. By evolving particle distribution functions on a discrete lattice, the LBM can recover the incompressible Navier–Stokes equations while naturally accommodating local interactions and boundary conditions (Gardner & Sitar, 2019).

The accurate representation of the solid phase in FSI problems further requires numerical methods capable of modeling the mechanical behavior and interaction of discrete particles. The Discrete Element Method (Cundall, 1971; Cundall & Strack, 1979) has become a widely adopted technique for this purpose, as it explicitly tracks the motion of individual particles and

represents particle-particle and particle-boundary interactions through contact force models. This particle-based description enables the simulation of granular assemblies and particulate systems with a high level of physical detail.

To enable a consistent interaction between the fluid and solid phases, coupling strategies must be employed to exchange forces and kinematic information across the fluid–solid interface. Among the available approaches, Immersed Boundary Methods (IMB) provide a flexible and efficient framework for representing moving solid boundaries within a fixed fluid grid, avoiding the need for mesh regeneration. The combination of LBM, DEM, and IMB techniques has therefore emerged as a promising strategy for the simulation of complex FSI systems involving particulate media.

In this context, this research presents the development of FSIT (Fluid–Solid Interaction Toolkit), a computational framework designed for the numerical simulation of fluid–solid interaction problems based on the coupling of the Lattice Boltzmann Method, the Discrete Element Method, and Immersed Boundary techniques. The proposed toolkit aims to provide a flexible and physically consistent platform for investigating multiphase and particulate systems, serving both as a research tool and as a foundation for applied engineering analyses.

1.1 PROBLEM STATEMENT

The scientific problem addressed in this research may be summarized by the following question: Is it possible to develop a robust and efficient computational framework capable of simulating fluid-solid interaction phenomena involving large numbers of solid particles through the coupled use of the Lattice Boltzmann Method, the Discrete Element Method, and Immersed Boundary techniques?

1.2 MOTIVATION

Fluid–solid interaction phenomena play a fundamental role in a wide range of geotechnical and engineering applications, including erosion processes, sediment transport, filtration, internal instability of soils, and particulate flows. The inherent complexity of these phenomena arises from the strong coupling between fluid dynamics and solid mechanics, as well as from the large number of parameters governing particle-scale interactions. Despite significant advances in experimental and numerical research, there remains a need for flexible

computational tools capable of capturing the fundamental mechanisms of fluid–solid interaction and supporting predictive analyses under different conditions (Zubeldia, 2017).

In this context, the motivation of this research lies in the development and application of numerical techniques capable of addressing fluid–solid interaction problems in a unified and physically consistent manner.

1.3 HYPOTHESIS

The main hypothesis of this research is that the hydrodynamic coupling of the Lattice Boltzmann Method, the Discrete Element Method, and Immersed Boundary techniques provides an effective and accurate framework for the numerical simulation of fluid–solid interaction phenomena involving particulate systems.

1.4 RESEARCH OBJECTIVES

The primary objective of this research is to develop, implement, and validate a computational toolkit for the numerical analysis of fluid–solid interaction phenomena, based on the coupling of the Lattice Boltzmann Method, the Discrete Element Method, and Immersed Boundary techniques, with particular emphasis on particle-scale interactions and multiphysics consistency.

1.5 THESIS OUTLINE

This thesis consists of 5 chapters each is briefly described below:

Chapter 2 provides the theoretical background of the numerical methods required for the development of the computational framework.

Chapter 3 describes the organization and architecture of the computational code and presents the simulation cases used for the validation of the proposed tool.

Chapter 4 presents and discusses the results obtained from the numerical simulations.

Chapter 5 summarizes the main conclusions of the research and highlights the key findings derived from the proposed methodology.

CHAPTER II

2 THEORETICAL BACKGROUND

This chapter presents the theoretical foundation for this research. The mathematical formulation of the Lattice Boltzmann Method (LBM), the Discrete Element Method (DEM) and coupling techniques are introduced.

All bold variables presented in this chapter denote vectors.

2.1 LATTICE BOLTZMANN METHOD

The Lattice Boltzmann Method (LBM) is based on statistical mechanics and can be derived from the Boltzmann equation through discretization in both time and space using a finite set of discrete velocities. The method is formulated at a mesoscopic scale, i.e. the particle or grain scale, and employs particle distribution functions to recover macroscopic fluid quantities such as density, velocity, and pressure. The governing LBM equation is written as:

$$\boldsymbol{f_i(x + c_i.\Delta t, t + \Delta t) - f_i(x, t) = \Omega_i(x, t)} \qquad \textit{(2.1)}$$

Equation (2.1) can be decomposed into two fundamental processes, represented by the left- and right-hand sides of the equation: streaming and collision. In this formulation, a probabilistic ensemble of particles is described by the particle distribution function f_i. During the collision step, particle population arriving at a lattice node interact locally and are relaxed toward equilibrium through the collision operator Ω_i. Following collision, the particle populations are streamed to neighboring lattice nodes along the discrete velocity directions $\boldsymbol{c_i}$. This sequence of collision and streaming steps is repeated throughout the simulation and is schematically illustrated in Figure 2-1.

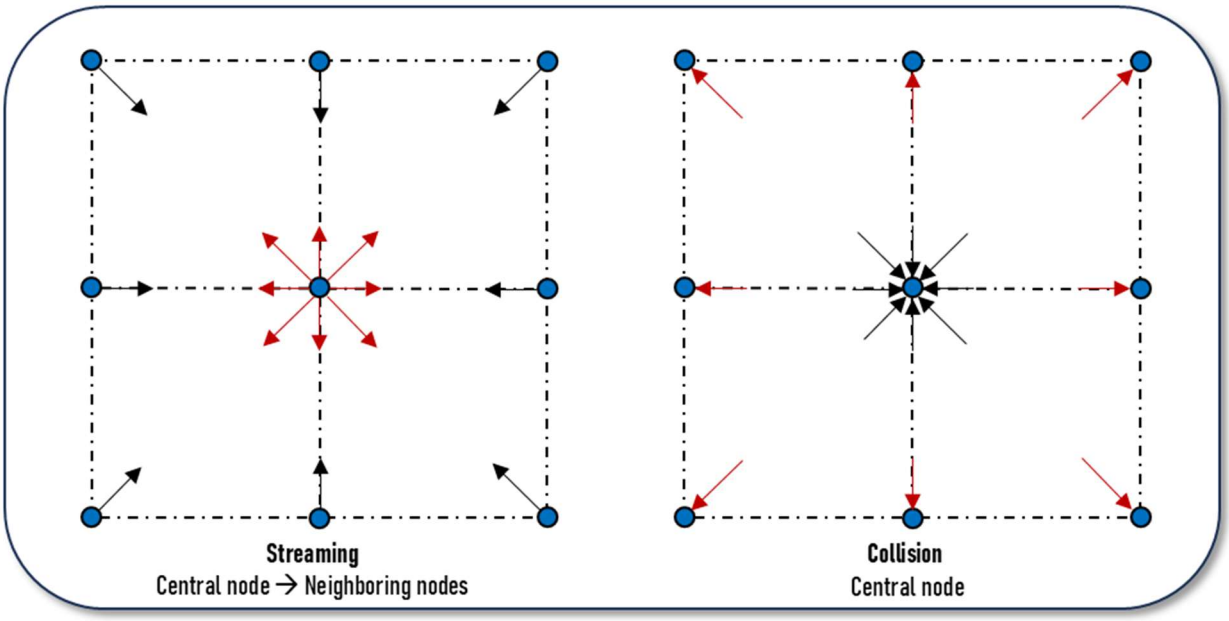

Figure 2-1: LBM streaming and collision scheme.

The particle distribution functions represent a collection of particles sharing the same position $\boldsymbol{x}$ and discrete velocity $\boldsymbol{c}_i$ at a given time step t. Macroscopic fluid properties, such as density ρ and velocity $\boldsymbol{u}$, are obtained as moments of these distribution functions and are computed as follows:

$$\boldsymbol{\rho} = \sum_{i=0}^{8} \boldsymbol{f}_i \tag{2.2}$$

$$\boldsymbol{u} = \frac{\mathbf{1}}{\boldsymbol{\rho}} . \sum_{i=0}^{8} \boldsymbol{c}_i . \boldsymbol{f}_i \tag{2.3}$$

2.1.1 LATTICE STRUCTURE

The discretization of the computational domain in the Lattice Boltzmann Method (LBM) is performed using a regular, orthogonal lattice. In two-dimensional problems, the lattice is typically composed of square cells, while in three-dimensional formulations it is composed of cubic cells, with uniform lattice spacing in all spatial directions. This regularity is a fundamental assumption of the standard LBM formulation, ensuring isotropy and consistency of the discrete velocity sets.

In general, the lattice configuration depends on the dimensionality of the problem being simulated. According to Qian et al. (1992), lattice arrangements are commonly denoted using the notation DdQn, where *d* represents the number of spatial dimensions (2D or 3D) and *n* denotes the number of discrete velocity directions in the local lattice neighborhood, including

the rest direction. For two-dimensional fluid simulations, the $D2Q9$ lattice is most employed, as illustrated in Figure 2-2.

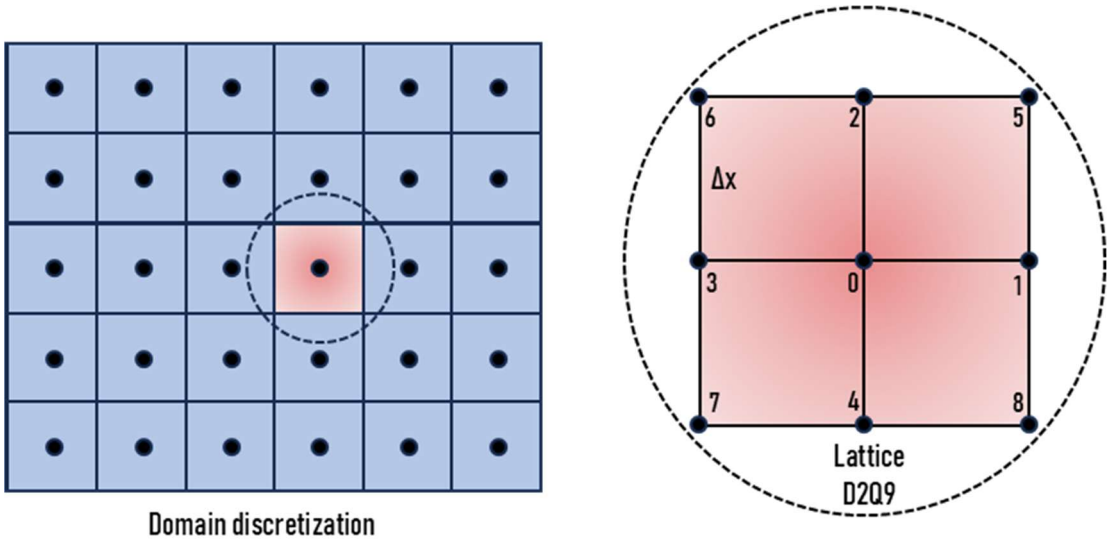

Figure 2-2: Discretization of the domain using $D2Q9$ lattice.

Each lattice arrangement is characterized by a specific set of discrete velocity vectors, lattice weight and a lattice speed of sound. For isothermal flows, the discrete velocity vectors point toward the immediate neighboring nodes and are associated with three distinct groups of weight: the weight corresponding to the zero-velocity direction (central node, $i = 0$) weights associated with low-speed axial directions (nearest neighbors, $i = 1 \dots 4$) and weights associated with higher-speed diagonal directions (next-nearest neighbors, $5 \dots 8$) (Ocampo-Gómez, 2013).

For the $D2Q9$ lattice, the discrete velocity vectors $\boldsymbol{c_i}$ and the lattice speed of sound c_s are defined as follows:

$$\boldsymbol{c_i} = \begin{cases} (\mathbf{0},\mathbf{0}), & \boldsymbol{i = 0} \\ \boldsymbol{c.}\left(\boldsymbol{cos}\left(\frac{\boldsymbol{\pi.(i-1)}}{\mathbf{2}}\right), \boldsymbol{sen}\left(\frac{\boldsymbol{\pi.(i-1)}}{\mathbf{2}}\right)\right), & \boldsymbol{i = 1 \dots 4} \\ \boldsymbol{\sqrt{2}c.}\left(\boldsymbol{cos}\left(\frac{\boldsymbol{\pi.(i-5)}}{\mathbf{2}} + \frac{\boldsymbol{\pi}}{\mathbf{4}}\right), \boldsymbol{sen}\left(\frac{\boldsymbol{\pi.(i-1)}}{\mathbf{2}} + \frac{\boldsymbol{\pi}}{\mathbf{4}}\right)\right), & \boldsymbol{i = 5 \dots 8} \end{cases} \tag{2.4}$$

$$\boldsymbol{c_s} = \frac{\boldsymbol{c}}{\sqrt{\mathbf{3}}} \tag{2.5}$$

where c is the lattice speed, defined as the ratio between the lattice spacing Δx and the simulation time step Δt.

The lattice weights, which define the contribution of each discrete velocity direction to the simulation, are given by:

$$\omega_i = \begin{cases} 4/9, & i = 0 \\ 1/9, & i = 1 \dots 4 \\ 1/36, & i = 5 \dots 8 \end{cases} \quad (2.6)$$

The selection of lattice parameters must not be arbitrary. For the LBM to accurately recover the incompressible Navier-Stokes equations, the Mach number M_a (ratio between the maximum fluid velocity and the lattice speed c) must be sufficiently small. According to Abdelhamid & El Shamy (2014), to ensure numerical stability and maintain incompressible (or weakly compressible) flow conditions, the Mach number should typically be limited to values bellow 0,1.

2.1.2 COLLISION OPERATOR

The modeling of particle collisions at a lattice node is described through a collision operator. The most used collision operator in the Lattice Boltzmann Method is the Bhatnagar–Gross–Krook (BGK) model (Bhatnagar et al., 1954), owing to its simplicity and computational efficiency, as it employs a single relaxation time τ. The BGK collision operator is expressed as:

$$\Omega_{BGK} = -\frac{\Delta t}{\tau}.\left[f_i(x,t) - f_i^{eq}(x,t)\right] \quad (2.7)$$

where f_i^{eq} is the equilibrium distribution function, defined as:

$$f_i^{eq} = \omega_i.\rho.\left[1 + 3\frac{c_i\,.\,u}{c^2} + 4,5\frac{(c_i\,.\,u)^2}{c^4} - 1,5\frac{u\,.\,u}{c^2}\right] \quad (2.8)$$

The BGK operator may exhibit numerical instabilities when simulations are performed under turbulent or high Reynolds number flow conditions. This behavior is primarily attributed to the use of a single relaxation time, which limits the method's ability to independently control the relaxation of different kinetic moments. Consequently, more advanced collision models are required for simulations involving complex flow regimes.

An alternative to the BGK model is the Multiple Relaxation Time (MRT) collision operator. Unlike the BGK formulation, the MRT model employs a relaxation matrix $\boldsymbol{S}$, allowing different moments of the distribution function to relax at distinct rates. This additional flexibility enhances numerical stability and enables the simulation of flows at higher Reynolds numbers. The MRT collision operator is defined as:

$$\Omega_{MRT} = -M^{-1}SM\left[f_i(x,t) - f_i^{eq}(x,t)\right].\Delta t \quad (2.9)$$

where $\boldsymbol{M}$ is the transformation matrix that maps the particle distribution function from velocity space to moment space, and $\boldsymbol{S}$ is the diagonal relaxation matrix.

The key difference between the BGK and MRT formulations lies in the space in which the collision process is performed. In the MRT approach, collisions are carried out in moment space rather than directly in velocity space. Accordingly, the distribution functions are first transformed into moments, relaxed independently according to the relaxation matrix, and subsequently transformed back to velocity space, as schematically illustrated in Figure 2-3.

The use of moment space in the MRT approach enables selective control over the relaxation of different physical modes. While conserved moments, such as mass and momentum, are left unchanged by the collision process, non-conserved moments associated with viscous stresses and higher-order kinetic effects are relaxed at prescribed rates. This additional degree of freedom enhances numerical stability and reduces spurious numerical artifacts, particularly in simulations involving high Reynolds number flows.

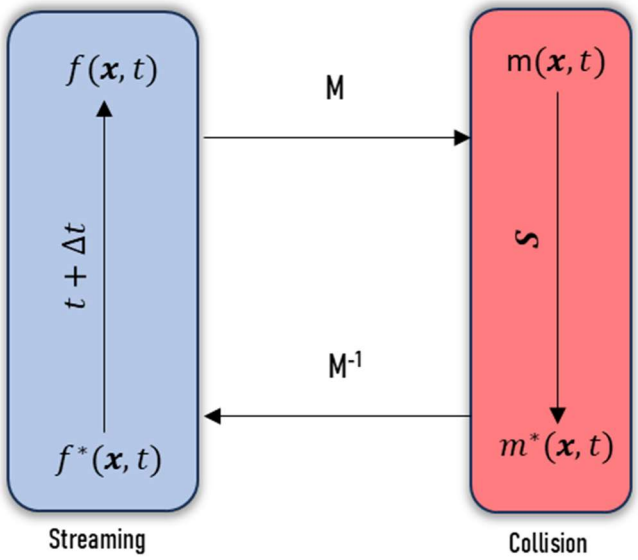

Figure 2-3: Schematic representation of the MRT collision operator.

As indicated in Eq. (2.9), the transformation from velocity space to moment space is performed using the transformation matrix $\boldsymbol{M}$. The structure and ordering of this matrix, as well as the elements of the relaxation matrix $\boldsymbol{S}$, depend on the lattice model adopted. In this research, only the $D2Q9$ lattice is employed; therefore, both matrices are of order nine. The transformation matrix is given by:

$$\boldsymbol{M} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ -4 & -1 & -1 & -1 & -1 & 2 & 2 & 2 & 2 \\ 4 & -2 & -2 & -2 & -2 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & -1 & 0 & 1 & -1 & -1 & 1 \\ 0 & -2 & 0 & 2 & 0 & 1 & -1 & -1 & 1 \\ 0 & 0 & 1 & 0 & -1 & 1 & 1 & -1 & -1 \\ 0 & 0 & -2 & 0 & 2 & 1 & 1 & -1 & -1 \\ 0 & 1 & -1 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 \end{bmatrix} \quad \textit{(2.10)}$$

It should be noted that the transformation matrix in Eq. (2.10) corresponds to the standard $D2Q9$ formulation under the assumption of unit lattice spacing ($\Delta x = 1$), as commonly employed in lattice units

Following the transformation, the individual relaxation of each kinetic moment is governed by the diagonal relaxation matrix given by:

$$\boldsymbol{S^{DIAG}} = \begin{bmatrix} s_1 & s_2 & s_3 & s_4 & s_5 & s_6 & s_7 & s_8 & s_9 \end{bmatrix} \quad \textit{(2.11)}$$

Although the MRT collision operator provides improved numerical stability and allows for the simulation of more complex flow conditions, it presents certain drawbacks, including increased computational cost and the need for calibration of the relaxation parameters.

2.1.3 BODY FORCES

The incorporation of body forces in the LBM can be achieved by introducing an additional source term into the evolution equation (streaming + collision), leading to:

$$f_i(\boldsymbol{x} + \boldsymbol{c_i}.\Delta t, t + \Delta t) - f_i(\boldsymbol{x}, t) = \Omega_i(\boldsymbol{x}, t) + S_i(\boldsymbol{x}, t) \quad (2.12)$$

where S_i denotes the source term accounting for external forces acting on the fluid. Guo *et al*. (2002) demonstrated that, to correctly recover Navier-Stokes equations, this forcing term must be formulated as follows:

$$S_i = \left(1 - \frac{1}{2\tau}\right).\omega_i.\left[\frac{\boldsymbol{c_i} - \boldsymbol{u}}{c_s^2} + \frac{(\boldsymbol{c_i} \bullet \boldsymbol{u})}{c_s^4}\boldsymbol{c_i}\right] \bullet \boldsymbol{F} \quad (2.13)$$

where $\boldsymbol{F}$ is the force density vector. The inclusion of the forcing term also requires a modification to the macroscopic fluid velocity to account for the momentum contribution of the applied force. The macroscopic velocity is therefore computed as:

$$\boldsymbol{u} = \frac{1}{\rho}\left(\sum_{i=0}^{8} f_i . \boldsymbol{c_i} + \frac{\boldsymbol{F}.\Delta t}{2}\right) \tag{2.14}$$

2.1.4 INITIAL AND BOUNDARY CONDITIONS

In contrast to classical numerical methods – such as the Finite Element Method – in which boundary conditions are directly prescribed on macroscopic fluid variables, the Lattice Boltzmann Method requires boundary and initial conditions to be formulated in terms of particle distribution functions. Consequently, appropriate strategies must be adopted to define both the initial state of the distribution functions and their behavior at the boundaries of the computational domain.

The specification of initial conditions for the particle distribution functions in an LBM simulation depends on the nature of the flow being analyzed, whether steady or transient. In cases where a steady-state solution is expected, a common and effective approach consists of initializing the distribution functions using their equilibrium values, as follows:

$$f_i(\boldsymbol{x}, t = 0) = f_i^{eq}(\rho_0, \boldsymbol{u_0}) \tag{2.15}$$

where ρ_0 and $\mathbf{u}_0$ denote the prescribed initial fluid density and velocity vector, respectively. For steady flows, the final solution is independent of the initial condition, making this initialization strategy a practical and robust choice for defining the starting state of the simulation.

The boundary conditions most employed in LBM-based fluid simulations include periodic boundaries, the bounce-back scheme, and the velocity and density boundary conditions proposed by Zou and He (1997).

The periodic boundary condition is the simplest among these techniques. Under this condition, particle distribution functions exiting the computational domain through one boundary re-enter the domain through the opposite boundary, effectively treating the domain as a closed system.

The bounce-back boundary condition is applied at solid boundaries, such as walls or discrete solid elements. In this approach, solid boundaries are represented by solid lattice nodes, and particle distribution functions that stream toward these nodes are reflected along the opposite

discrete velocity direction. This mechanism enforces a no-slip condition at the fluid–solid interface and is widely used due to its simplicity and robustness.

The velocity and density boundary conditions developed by Zou & He (1997) enable the prescription of macroscopic velocity or density values at the domain boundaries. In this formulation, the unknown particle distribution functions are reconstructed such that the imposed macroscopic quantities are satisfied, ensuring consistency between the boundary conditions and the underlying LBM formulation.

2.2 DISCRETE ELEMENT METHOD

The Discrete Element Method (DEM) was originally developed by researchers (Cundall, 1971; Cundall & Strack, 1979), with initial applications in the field of rock mechanics. In this method, solids are represented as a collection of discrete elements whose motion is governed by a set of physical laws and parameters that define their position and kinematics as a function of the forces upon them. These forces may arise from body forces, such as gravity, or from interactions with other discrete elements or surrounding media, including hydrodynamic.

According to Zuluaga (2016), DEM is particularly well suited for the study of particulate and granular systems, as it explicitly accounts for the discrete nature of the material. This approach enables the investigation of the physical and mechanical behavior of granular media by liking macroscopic responses to microscopic properties of the particles and their interaction.

The main assumption underlying the DEM formulation can be summarized as follows:

- Discrete elements are considered rigid bodies with analytically defined geometric shapes;
- Discrete elements are allowed to translate and rotate freely relative to one another during each step;
- Elements may exhibit simple geometries (e.g. disks, spheres, polyhedral, or clusters of spheres) or more complex shapes representing real particles;
- Contact interactions occur exclusively between pairs of elements, over an infinitesimal area or volume, depending on whether the simulation domain is two- or three-dimensional;

- A small overlap between contacting elements is permitted. This overlap is interpreted as a numerical representation of the local deformation and is governed by a constitutive contact law, which may be elastic, plastic, viscous, or a combination thereof, depending on the selected contact model.

2.2.1 FORMULATION

In general, a basic DEM simulation cycle is founded on two fundamental steps: the evaluation of forces acting on the discrete elements and the subsequent kinematic update of their motion at each time step. From a simplified perspective, the forces acting on a discrete element may be classified into body forces, contact interaction forces between discrete elements, and interaction forces with the surrounding medium (e.g. Wall).

Body forces are straightforward to define during the simulation and are typically computed from the mass, volume, and specific weight of each discrete element, together with the gravity. In contrast, the determination of interaction forces may vary significantly in complexity and computational cost, depending on the dimensionality of the problem (two- or three-dimensional) and on the geometry of the discrete elements.

For clarity, the DEM formulation presented herein is illustrated for the case of two disk-shaped elements interacting with each other or with the surrounding medium. Furthermore, a linear contact interaction law (analogous to the response of elastic springs) is adopted. Figure 2-4 illustrates the disk–disk and disk–medium interaction mechanisms.

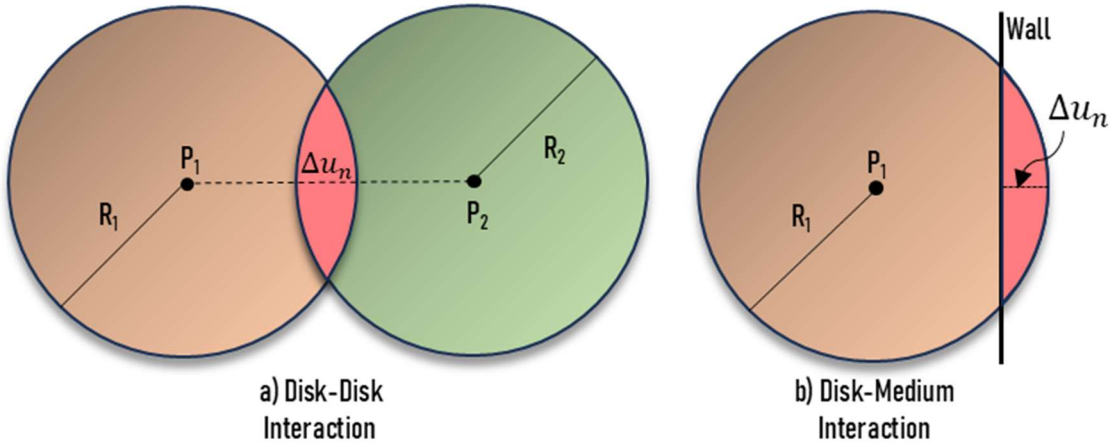

Figure 2-4: Interaction mechanisms between disk-shaped discrete elements.

Based on the interaction between a discrete element and another element or the surrounding medium, increments of normal and tangential contact forces are computed according to the selected interaction law. For a linear contact model, the force increments are given by:

$$\boldsymbol{\Delta F_n = k_n . \Delta u_n} \quad \textit{(2.16)}$$

$$\boldsymbol{\Delta F_s = k_s . \Delta u_s} \quad \textit{(2.17)}$$

where k_n and k_s denote the normal and tangential contact stiffnesses, respectively, and Δu_n and Δu_s represent the normal and tangential displacement increments, determined from the overlap between contacting elements or between an element and the surrounding medium.

The contact forces at the current time step are then obtained by accumulating the force increments as:

$$\boldsymbol{F_n^t = F_n^{t-1} + \Delta F_n} \quad \textit{(2.18)}$$

$$\boldsymbol{F_s^t = F_s^{t-1} + \Delta F_s} \quad \textit{(2.19)}$$

It is important to note that tangential contact forces must satisfy a frictional limit defined by the material properties of the interacting bodies. In this research, discrete elements represent soil particles; therefore, the tangential force is limited using the Mohr–Coulomb friction criterion, which is widely employed in geotechnical engineering. This constraint is enforced as:

$$\boldsymbol{F_s^t = F_s^t . \frac{F_s^{Max}}{|F_s^t|}} \quad \textit{(2.20)}$$

Once the contact forces and torques acting on a discrete element have been determined, the kinematic update of the particle motion is performed. This update is carried out using Newton's second law to compute the translational and rotational accelerations, which govern the evolution of the particle position and orientation. Accordingly, the equations of motion are given by:

$$\boldsymbol{m . \frac{dv}{dt} = \sum_i F_i} \quad \textit{(2.21)}$$

$$\boldsymbol{I . \frac{d\omega}{dt} = \sum_i T_i} \quad \textit{(2.22)}$$

where $\mathbf{v}$ and $\boldsymbol{\omega}$ are the translational and rotational velocity vectors of the discrete element, respectively; m is its mass and I its moment of inertia; and $\mathbf{F}_i$ and $\mathbf{T}_i$ denote the forces and torques acting on the element.

2.2.2 INTEGRATION METHOD

The equations governing the motion of discrete elements, Eqs. (2.23) and (2.24), are first-order differential equations and require an appropriate time integration scheme for their numerical solution. In DEM simulations, these equations are typically solved using explicit integration methods based on centered finite-difference schemes. Several integration techniques have been proposed in the literature, including the Euler method, the Leapfrog method, the Velocity-Verlet method, and the Beeman method (Beeman, 1976), among others.

The explicit Leapfrog integration scheme is often adopted due to its second-order accuracy, exact conservation of angular momentum, global numerical stability, and straightforward computational implementation. In the Leapfrog method, the translational and rotational velocities of a discrete element are defined at half time steps, staggered with respect to the particle positions, and the calculations are interleaved throughout the simulation.

The translational velocity $\mathbf{v}_p$ and angular velocity $\boldsymbol{\omega}_p$ of a discrete element are updated according to:

$$\boldsymbol{v}]_{t+\Delta t/2} = \boldsymbol{v}]_{t-\Delta t/2} + \frac{\boldsymbol{F}]_t}{\boldsymbol{m}} . \boldsymbol{\Delta t_{DEM}} \tag{2.25}$$

$$\boldsymbol{\omega}]_{t+\Delta t/2} = \boldsymbol{\omega}]_{t-\Delta t/2} + \frac{\boldsymbol{T}]_t}{\boldsymbol{I}} . \boldsymbol{\Delta t_{DEM}} \tag{2.26}$$

where $\mathbf{F}^t$ and $\mathbf{T}^t$ denote the resultant force and torque acting on the discrete element at time t, m is the particle mass, and I is its moment of inertia.

Once the velocities have been updated, the particle position and orientation are advanced using:

$$\boldsymbol{x}]_{t+\Delta t} = \boldsymbol{x}]_t + \boldsymbol{v}]_{t+\Delta t/2} . \boldsymbol{\Delta t_{DEM}} \tag{2.27}$$

$$\boldsymbol{\theta}]_{t+\Delta t} = \boldsymbol{\theta}]_t + \boldsymbol{\omega}]_{t+\Delta t/2} . \boldsymbol{\Delta t_{DEM}} \tag{2.28}$$

The Leapfrog scheme requires an initial half-time-step initialization to compute the translational and rotational velocities at $t = \Delta t/2$. This initialization is performed as:

$$\boldsymbol{v}]_{\Delta t/2} = \boldsymbol{v}]_{t=0} + \frac{\mathbf{1}}{\mathbf{2}} . \frac{\boldsymbol{F}]_{t=0}}{\boldsymbol{m}} . \boldsymbol{\Delta t_{DEM}} \tag{2.29}$$

$$\boldsymbol{\omega}]_{\Delta t/2} = \boldsymbol{\omega}]_{t=0} + \frac{\mathbf{1}}{\mathbf{2}} . \frac{\boldsymbol{T}]_{t=0}}{\boldsymbol{I}} . \boldsymbol{\Delta t_{DEM}} \tag{2.30}$$

In all the equations presented in this section, a key parameter controlling numerical stability is the DEM time step Δt_{DEM}. The time step must be selected such that disturbances generated during particle contacts do not propagate beyond neighboring elements within a single time step, which would lead to numerical instability.

To satisfy this condition, the DEM time step is required to be smaller than a critical value estimated as:

$$\boldsymbol{\Delta t_{ref} = 2\lambda.\sqrt{\frac{m_{Min}}{k}}} \qquad \textit{(2.31)}$$

where $m_{\min}$is the minimum particle mass in the system, k is the contact stiffness, and λ is a safety factor ranging between 0 and 1. According to Feng et al. (2007), values of λ close to 0.1 are recommended to ensure both numerical stability and solution accuracy in DEM simulations.

2.3 LBM-DEM COUPLING TECHNICS

The coupling between the Lattice Boltzmann Method (LBM) and the Discrete Element Method (DEM) is primarily achieved through the exchange of linear and angular momentum between the fluid and solid phases. Consequently, the selected coupling strategy must be capable of accurately quantifying the hydrodynamic forces and torques acting on discrete elements, while consistently incorporating their effects into the fluid particle distribution functions.

Several coupling techniques have been proposed in the literature, differing in implementation complexity and computational cost (Ladd, 1994; Noble & Torczynski, 1998; Aidun et al., 200 Wang et al., 2021). The most used technique to couple LBM-DEM is the one proposed by Noble and Torczynski (1998) known as the Immersed Moving Boundary (IMB) or partially-solid scheme. According to Owen et al. (2011), this method offers significant advantages, such as preserving the locality of collision operations and introducing a simple linear collision operator Ω_i^s. These features make the IMB approach highly suitable for parallel computing environments and for simulations involving complex particle motion and large numbers of discrete elements.

Within the IMB framework, the standard LBM evolution equation (Eq. 2.32) is modified to account for the presence of moving solid boundaries, resulting in:

$$\boldsymbol{f_i(x + c_i\Delta t, t + \Delta t) = f_i(x,t) - (1 - B_n).\left[\Omega_i.\left(f_i(x,t) - f_i^{eq}(x,t)\right)\right] + B_n.\Omega_i^s} \qquad \textit{(2.33)}$$

where B_n is a weighting function that represents the fraction of the lattice cell occupied by the discrete element. This function is defined as:

$$B_n = \begin{cases} \dfrac{\epsilon.(\tau - 1/2)}{(1-\epsilon)+(\tau-1/2)}, & \Omega_i: BGK \\ \dfrac{\epsilon.(1/s_7 - 1/2)}{(1-\epsilon)+(1/s_7-1/2)}, & \Omega_i: MRT \end{cases} \tag{2.34}$$

where ϵ denotes the fraction of the lattice cell area occupied by the discrete element. The accurate determination of B_n is essential to ensure momentum conservation between the numerical methods coupled. This is achieved by classifying lattice cells into fluid, partially occupied, and solid nodes, and computing the corresponding solid fraction ϵ, as illustrated in Figure 2-5.

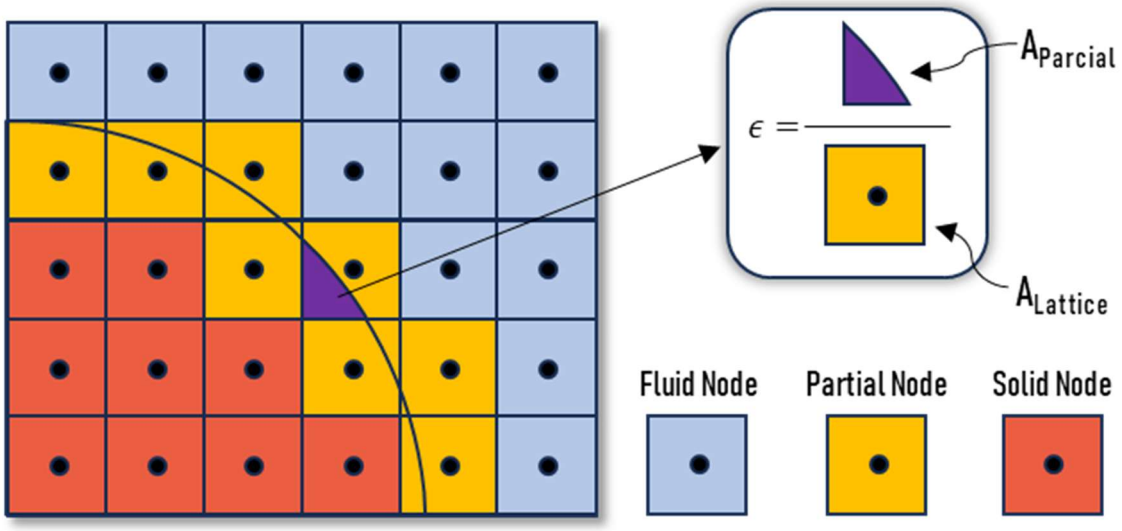

Figure 2-5: Lattice mesh division on fluid, partial and solid nodes to determine B_n.

The additional collision operator introduced in the IMB method is based on the bounce-back concept applied to the non-equilibrium part of the distribution function and is expressed as:

$$\Omega_i^s = f_{i'}(x,t) - f_i(x,t) + f_i^{eq}(\rho, v_p) - f_i^{eq}(\rho, u) \tag{2.35}$$

where $\mathbf{v}_p$ is the velocity of the discrete element and i'denotes the lattice direction opposite to i.

Finally, the hydrodynamic force and torque exerted by the fluid on the discrete elements are obtained by summing the momentum exchange contributions over all intersected lattice cells, according to:

$$F_{fluid} = \frac{\Delta x^2}{\Delta t} . \sum_n B_n . \sum_i \Omega_i^s . c_i \tag{2.36}$$

$$\boldsymbol{M_{fluid}} = \frac{\boldsymbol{\Delta x^2}}{\boldsymbol{\Delta t}} \cdot \sum_{\boldsymbol{n}} (\boldsymbol{x_c} - \boldsymbol{x_P}) \times \left(\boldsymbol{B_n} \cdot \sum_{\boldsymbol{i}} \boldsymbol{\Omega_i^s} \cdot \boldsymbol{c_i} \right) \qquad \textit{(2.37)}$$

where $\mathbf{x}_c$ and $\mathbf{x}_p$ are the position vectors of the lattice cell center and the discrete element centroid, respectively, and × denotes the vector cross product.

2.3.1 TIME-STEP COMPATIBILITY

Since the LBM and DEM employ different time-step sizes, their coupling requires a time-step compatibility strategy to avoid numerical instabilities or inaccuracies during the simulation. Owen et al. (2011) proposed two criteria to ensure a stable and consistent coupling between the two methods.

The first criterion applies to simulations in which the DEM time step is greater than or equal to the LBM time step, i.e., $\Delta t_{DEM} \geq \Delta t_{LBM}$. In this case, the global time step of the coupled simulation is set equal to Δt_{LBM}. This conservative approach guarantees numerical stability by preventing the propagation of contact-induced disturbances beyond neighboring discrete elements.

Conversely, when the LBM time step is larger than the DEM time step ($\Delta t_{LBM} > \Delta t_{DEM}$), a subcycling strategy must be employed. In this approach, a number of DEM subcycles, denoted by n_{sub}, are executed within a single LBM time step. During the subcycles, the hydrodynamic forces and torques acting on the discrete elements are held constant, while the kinematic quantities (particle positions and velocities) are updated. The required number of DEM subcycles is defined as:

$$\boldsymbol{n_{sub}} = \frac{\boldsymbol{\Delta t_{LBM}}}{\boldsymbol{\Delta t_{DEM}}} + \mathbf{1} \qquad \textit{(2.38)}$$

The selection of an appropriate number of subcycles is nontrivial and generally requires preliminary numerical testing. The value of n_{sub} must be chosen such that discrete elements do not traverse multiple lattice cells within a single LBM time step, which would compromise the accurate evaluation of hydrodynamic interactions at the fluid–solid interface.

CHAPTER III

3 FLUID-SOLID INTERACTION TOOLKIT

This chapter presents the guiding principles that led the development of the computational code Fluid–Solid Interaction Toolkit (FSIT), as well as the overall architecture of the framework. The design philosophy and structural organization adopted in FSIT are discussed with the objective of providing a clear understanding of how the numerical methods are implemented and integrated within a unified computational environment.

In addition, this chapter introduces the simulation cases considered for the validation of the numerical methodologies employed in this research, namely the Lattice Boltzmann Method (LBM), the Discrete Element Method (DEM), the Immersed Boundary Method (IMB), and their hydromechanical coupling. These benchmark problems are used to assess the accuracy, stability, and consistency of the individual solvers and of the coupled framework.

Finally, more complex simulation scenarios are presented in order to demonstrate the capability of FSIT to model representative geotechnical fluid–solid interaction problems, highlighting its potential for simulating particle-scale mechanisms in engineering applications.

3.1 COMPUTATIONAL CODE

In order to achieve the objectives proposed in this research, a computational code was developed to perform two-dimensional analyses of fluid–solid interaction involving a fluid phase, represented by water, and solid particles representing soil grains. The code is based on the Lattice Boltzmann Method (LBM) for fluid simulation and the Discrete Element Method (DEM) for particle dynamics, with hydromechanical coupling performed through the Immersed Moving Boundary (IMB) method.

FSIT was developed using the Cursor integrated development environment (IDE), with compilation performed using the GNU Compiler Collection (GCC) in a Linux environment through the Windows Subsystem for Linux (WSL-2). The implementation language chosen for the framework is C++, which enables object-oriented programming, efficient code reuse, and high computational performance for solving large systems of algebraic equations, thereby accelerating numerical simulations. In addition, CMake was employed to manage the build

system and facilitate the integration of multiple source files and header modules within the codebase. Post-processing and visualization of the simulation results generated by FSIT are performed using ParaView, selected for its open-source nature and its extensive capabilities for analyzing and visualizing complex numerical datasets.

Within FSIT, the following numerical methodologies and features have been implemented:

1. The LBM solver implemented in FSIT supports both the single-relaxation-time BGK and the Multiple Relaxation Time (MRT) collision operators, allowing the user to select the most appropriate formulation according to the characteristics of the simulated flow. Domain discretization is performed using a lattice-based grid structure;
2. A $D2Q9$ lattice configuration is adopted, which is widely used for two-dimensional fluid flow simulations.
3. The DEM module models mechanical interactions associated with the motion and contact between solid particles, which are currently limited to circular bodies (disks). These interactions include normal and tangential contact forces arising during particle–particle collisions and sustained contacts. While the framework allows the use of realistic particle geometries for the evaluation of fluid–particle interaction forces, particle–particle interactions involving arbitrary particle shapes have not yet been implemented and are therefore restricted to circular particles in the present formulation.
4. The IMB coupling strategy is employed to compute the transfer of linear and angular momentum arising from fluid–solid interaction. This approach also enables the consistent coupling and synchronization of the LBM and DEM time-stepping schemes.

All header and source (.cpp) files associated with the FSIT framework are provided in Appendix A for reference.

3.1.1 CODE ARCHITECTURE

The Fluid–Solid Interaction Toolkit (FSIT) was developed following an object-oriented design, in which the computational framework is organized into a set of interacting classes that represent the physical entities, numerical methods, and data management structures required for fluid–solid interaction simulations. Program execution is initialized through a singleton instance of the Scene class, which serves as the central controller of the simulation.

The Scene class stores pointers to objects of the following main types: Cell, which contains the $D2Q9$ lattice nodes used for the LBM discretization; Body, which represents the discrete solid elements; Interaction, which stores the interactions between discrete elements; and Engine, which encapsulates the numerical solvers associated with the LBM, DEM, and IMB methodologies. In addition, the Cell and Body classes maintain pointers to auxiliary objects that store information related to position, material properties, and geometric representation, which are defined through the State, Material, and Shape classes, respectively. Figure 3-1 illustrates the overall organization of the FSIT code architecture.

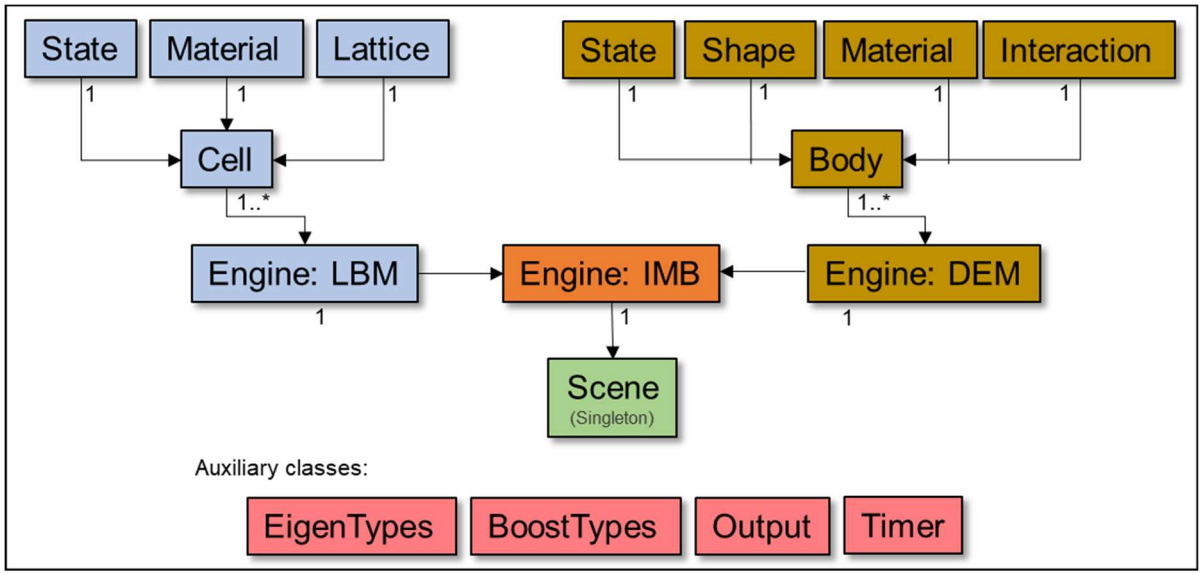

Figure 3-1: Software architecture and class organization of the FSIT framework.

The FSIT code is structured using a class-based architecture. Program execution is initialized through a singleton instance of the Scene class, which manages the simulation domain and global parameters. The Scene class stores pointers to objects of the following main types:

- Cell, which represents the lattice cells containing the D2Q9 discretization used in the LBM;
- Body, which represents the discrete solid elements modeled by the DEM;
- Interaction, which stores information related to particle–particle and particle–boundary interactions;
- Engine, which encapsulates the numerical solvers associated with the LBM, DEM, and IMB formulations.

Within the Cell and Body classes, additional pointers are used to store information related to state variables, material properties, and geometrical representation, which are defined by the

State, Material, and Shape classes, respectively. Figure 4.1 illustrates the overall organization and hierarchy of the FSIT code architecture.

Each class was developed with the following purposes:

- The Lattice class is responsible for storing lattice-related information, such as particle distribution functions, nodal weights, and discrete velocity sets;
- The Material class defines the constitutive behavior associated with each material used in the simulation;
- The State class stores kinematic and dynamic information, including position, velocity, forces, and moments acting on lattice cells or discrete bodies;
- The Shape class is responsible for generating the geometrical representation of discrete elements using point clouds and for performing the geometric integration required for coupling the numerical methods;
- The Cell and Body classes aggregate all information related to lattice cells and discrete solid elements within the simulated domain;
- The Engine classes act as numerical solvers, implementing the algorithms associated with the LBM, DEM, and IMB methodologies;
- The Scene class, implemented as a singleton, manages the simulation setup, including domain definition and numerical parameters associated with each simulation scenario.

In addition to the main classes, several **auxiliary classes** were developed to support numerical calculations and post-processing:

- The EigenTypes and BoostTypes classes are based on well-established C++ libraries and are used to perform vectorial, tensorial, and geometric operations required by the numerical methods;
- The Output class is responsible for generating output files in .vtk and .csv formats for visualization and analysis of the simulation results;
- The Timer class was implemented to identify computational bottlenecks and to monitor performance, contributing to the optimization of simulation time and computational cost.

Taken together, these features make FSIT a flexible, modular, and extensible computational framework, with strong potential for application in simulations involving fluid–solid interaction and particulate systems with complex geometries and material properties.

3.1.2 GENERAL SIMULATION WORKFLOW

A simulation scenario in the Fluid–Solid Interaction Toolkit (FSIT) is defined through a structured workflow that integrates domain configuration, numerical parameterization, initialization procedures, and time integration of the coupled fluid–solid system. This workflow was designed to ensure numerical consistency, modularity, and flexibility, allowing different physical problems to be simulated within a unified computational framework.

The simulation process begins with the definition of the computational domain, which includes the spatial discretization of the fluid phase using a lattice-based grid and the geometric description of the solid phase through discrete elements. At this stage, the domain size, lattice resolution, boundary conditions, and material properties for both fluid and solid phases are specified. The numerical solvers associated with the LBM and DEM, as well as the collision operators and coupling strategies, are selected according to the characteristics of the problem under investigation.

Once the domain and numerical parameters are defined, the initial conditions of the system are prescribed. For the fluid phase, this involves initializing the particle distribution functions in accordance with the desired macroscopic fields, such as density and velocity. For the solid phase, the initial positions, velocities, and orientations of the discrete elements are defined, together with their material and contact properties.

The simulation then proceeds through a time-marching loop, in which the fluid and solid solvers are advanced in a synchronized manner. Within each global time step, the LBM solver updates the fluid distribution functions through collision and streaming processes, while the DEM solver computes contact forces, updates particle kinematics, and integrates the equations of motion for the discrete elements. The interaction between the fluid and solid phases is handled through the Immersed Moving Boundary approach, which enables the consistent exchange of linear and angular momentum between the two phases.

When different time scales are required for the fluid and solid solvers, time-step compatibility strategies are applied, such as the use of sub-cycling in the DEM solver. During this process,

hydrodynamic forces and torques acting on the solid elements are evaluated and applied, ensuring that the effects of fluid–solid coupling are consistently reflected in both solvers.

Throughout the simulation, relevant physical quantities are monitored and stored for analysis. At user-defined intervals, the simulation state is exported to output files in formats suitable for visualization and post-processing. This allows the evolution of flow fields, particle motion, and interaction forces to be analyzed in detail.

The simulation is terminated once the prescribed final time or convergence criteria are reached. The modular structure of FSIT allows simulation scenarios to be easily extended or modified, enabling the investigation of different physical conditions, numerical parameters, and coupling strategies within the same computational framework.

3.2 REFERENCE CASES

This section presents the reference cases used to assess the validity and robustness of the Fluid–Solid Interaction Toolkit (FSIT) in the numerical simulation of fluid–solid interaction phenomena.

Initially, classical benchmark problems are employed to validate the Lattice Boltzmann Method (LBM) implementation. These cases correspond to flow configurations for which analytical solutions are available or that have been extensively investigated in the literature. The selected benchmarks include Poiseuille flow, lid-driven cavity flow, the Taylor–Green vortex, and flow around a circular cylinder. These problems allow for the evaluation of the accuracy of the fluid solver, as well as its numerical stability and its ability to correctly recover the incompressible Navier–Stokes equations.

Subsequently, the LBM–DEM coupling is validated through the simulation of a single solid particle suspended in a fluid, with the objective of evaluating the hydrodynamic forces acting on the particle and verifying the correct transfer of linear and angular momentum between the fluid and solid phases.

Following the validation of the numerical methods and their coupling strategy, a set of geotechnical-oriented simulation scenarios is investigated in order to demonstrate the capability of FSIT to represent more complex and physically realistic systems. These scenarios include:

1. Evaluation of the flow around a real particle geometry, accounting for geometric complexity;

2. Evaluation of flow through particle packings, considering distributions of circular particles and real particle shapes;

3. Evaluation of sediment transport processes, with emphasis on the interaction between the fluid flow and the particulate medium.

For all aforementioned cases, a detailed description is provided, including the theoretical background of the problem, the procedure adopted to construct the numerical model using FSIT, and the simulation input data, such as geometric configuration, material properties, boundary conditions, and numerical parameters. This detailed presentation aims to ensure clarity, reproducibility, and transparency of the numerical analyses.

For cases admitting an analytical solution, the numerical error is quantified using the root mean square (RMS) error, defined as:

$$\boldsymbol{RMS} = \sqrt{\frac{\mathbf{1}}{\boldsymbol{N}}\sum_{\boldsymbol{i=1}}^{\boldsymbol{N}}(\boldsymbol{\phi_i^{num}} - \boldsymbol{\phi_i^{ana}})^{\mathbf{2}}} \qquad \textit{(3.1)}$$

where ϕ_i^{num} is the numerical solution obtained from FSIT simulation at the i-th lattice node, ϕ_i^{ana} is the analytical solution evaluated at the same location, and N is the total number of lattice nodes considered in the calculation of the error.

In all numerical simulations performed here, the governing equations are solved using dimensionless quantities expressed in lattice units (LU), which is a standard practice in the LBM. In this formulation, the lattice spacing and the time step are set to unity ($\Delta x = \Delta t = 1$), and all physical variables are nondimensionalized accordingly. The mapping between physical and lattice scales is performed by prescribing the target Reynolds number and the characteristic flow velocity, while ensuring numerical stability and low compressibility effects. As a result, parameters such as kinematic viscosity, relaxation time, body force magnitude, and sound speed are consistently derived within the lattice framework, allowing the simulated flow to accurately reproduce the corresponding analytical solution while preserving the inherent assumptions of the LBM formulation.

All source (.cpp) files associated with the simulation cases are provided in Appendix B for reference.

3.2.1 POISEUILLE FLOW

Poiseuille flow is a laminar flow that occurs inside pipes or in the space between two parallel plates. The velocity profile associated with this type of flow is parabolic, exhibiting zero velocity at the walls due to the no-slip boundary condition and a maximum velocity at the channel centerline. Figure 3-2 illustrates the geometry and the typical velocity profile observed for this flow configuration.

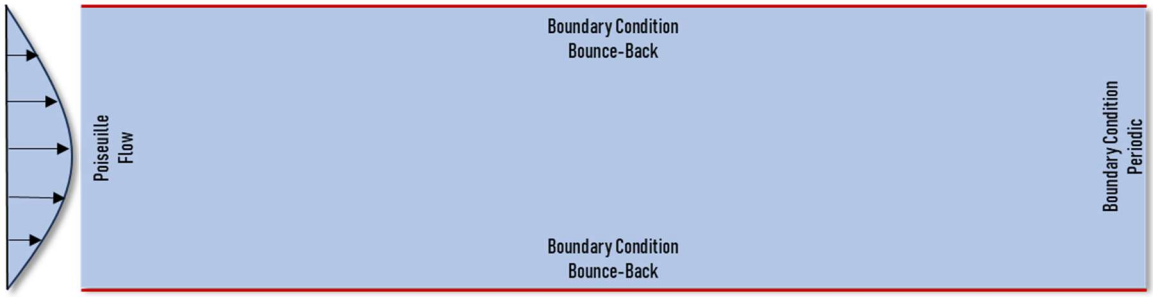

Figure 3-2: Geometry and velocity profile for Poiseuille flow scenario.

This scenario was selected due to its simplicity of implementation and its suitability for validation purposes, as an analytical solution is available. Accordingly, the Poiseuille flow case was constructed within the FSIT framework using two different approaches. In the first approach, the parabolic velocity profile was directly prescribed at the inlet boundary. In the second approach, a body force was applied to the fluid domain in order to generate the velocity field.

In the body-force-driven Poiseuille flow case, the fluid motion is generated by a uniform volumetric force applied in the streamwise direction, rather than by prescribing a velocity profile at the boundaries. The magnitude of this force is derived directly from the analytical solution of the fully developed laminar flow between two parallel plates given by:

$$fx = \frac{8.\nu.u_{max}}{H^2} \qquad (3.2)$$

Beyond validating the Lattice Boltzmann Method implementation, the objective of this scenario was to assess whether the boundary condition engines and the force-application mechanisms are correctly implemented within the FSIT code. The simulation scripts corresponding to both approaches are presented in Appendix B: Scenario 01 and Scenario 02, respectively.

Table 3-1 shows the numerical parameters adopted for the Poiseuille flow simulations using velocity boundary conditions and Guo forcing scheme.

Table 3-1: Numerical parameters adopted for the Poiseuille flow simulations using velocity boundary conditions (Zou & He) and body-force-driven flow (Guo forcing), expressed in lattice units.

Parameter	**Zou & He Velocity BC**	**Guo Forcing Scheme**
Maximum channel velocity, $\boldsymbol{u_{max}}$	0.10	0.10
Channel geometry, $\boldsymbol{N_x \times N_y}$	$\boldsymbol{400 \times 102}$	$\boldsymbol{400 \times 102}$
Total number of lattice cells	40,800	40,800
Initial fluid density, $\boldsymbol{\rho_0}$	1.00	1.00
Initial velocity field	Prescribed parabolic profile	(0.0, 0.0)
Reynolds number, Re	5	5
Kinematic viscosity, $\boldsymbol{\nu}$	2.00	2.00
Relaxation time, $\boldsymbol{\tau}$	6.50	6.50
Lattice spacing, $\boldsymbol{\Delta x}$	1.00	1.00
Time step, $\boldsymbol{\Delta t}$	1.00	1.00
Lattice velocity, $\boldsymbol{c = \Delta x/\Delta t}$	1.00	1.00
Speed of sound, $\boldsymbol{c_s = c/\sqrt{3}}$	$\boldsymbol{1/\sqrt{3}}$	$\boldsymbol{1/\sqrt{3}}$
Collision operator	BGK	BGK
Body force applied	No	Yes
Forcing scheme	–	Guo et al. (2002)
Forcing term, $\boldsymbol{F_x}$(LU)	–	$\boldsymbol{8\nu u_{max}/H^2}$
Bounce-back boundary condition	Yes	Yes
Periodic boundary condition	Yes	Yes
Velocity (Zou & He) BC	Yes	No

3.2.2 LID-DRIVEN CAVITY FLOW

The lid-driven cavity flow is a classical benchmark problem widely used for the validation of computational fluid dynamics codes. Owing to its simple geometry and well-documented reference solutions, this problem has become a standard test case for assessing the accuracy and stability of numerical methods for incompressible flows.

The problem consists of a square cavity filled with fluid, where the flow is induced by imposing a constant velocity on the top lid, while the remaining walls are kept stationary and subject to no-slip boundary conditions. The motion of the lid generates a primary vortex within the cavity, as well as secondary vortical structures near the corners, depending on the Reynolds number.

Figure 3-3 illustrates the geometry of the lid-driven cavity scenario simulated using FSIT.

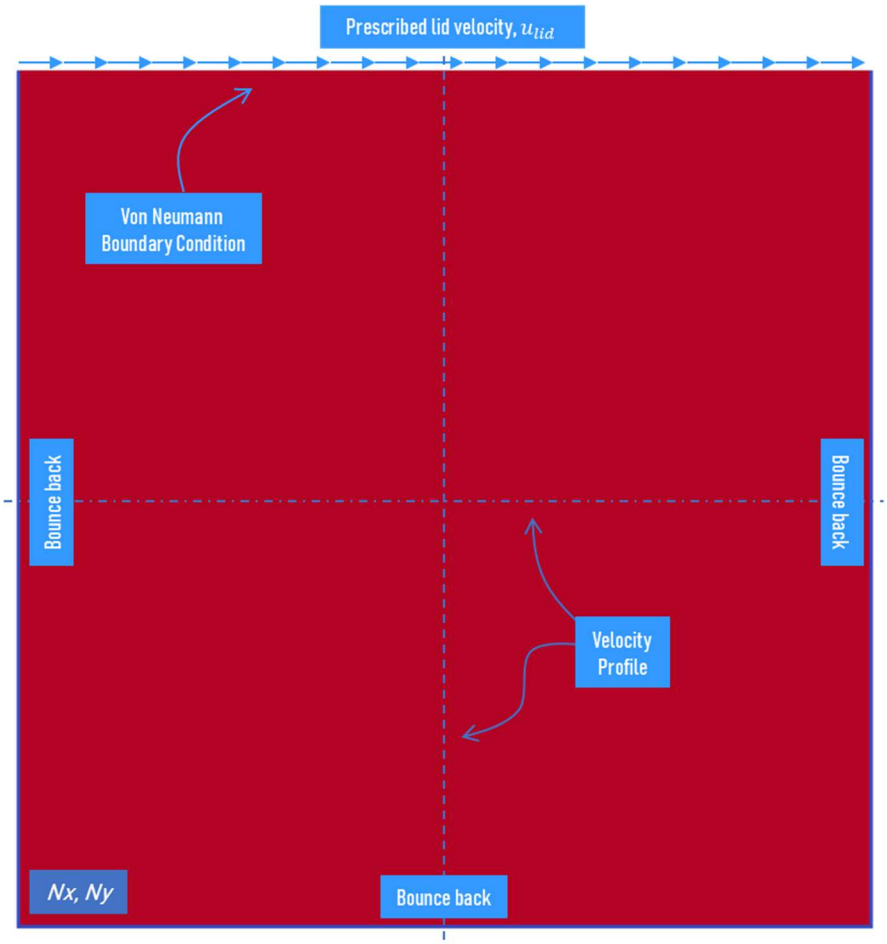

Figure 3-3: Geometry of the lid-driven cavity scenario.

Validation is performed by comparing the velocity profiles developed inside the cavity (typically along the vertical and horizontal centerlines) with benchmark solutions available in the literature. This comparison allows for a quantitative assessment of the numerical accuracy of the implemented method and its ability to correctly capture flow recirculation and boundary layer effects.

Table 3-2 shows a summary of the parameters used to simulate the lid-driven cavity flow.

Table 3-2: Summary of simulation parameters for the lid-driven cavity flow, expressed in lattice units.

Parameter	**BGK**	**MRT**
Cavity geometry, $N_x \times N_y$	256 x 256	256 x 256
Total number of lattice cells	65,536	65,536
Lid velocity, u_{lid}	0.05	0.05
Reynolds number, Re	100	100
Kinematic viscosity, ν	0.128	0.128
Relaxation time, τ	0,884	DiagonalMatrix(0, 1.4, 1.4, 0.75, 1.2, 1.2, 1.13, 1.13)
Initial fluid density, ρ_0	1.0	1.0
Initial velocity field	(0.0, 0.0)	(0.0, 0.0)
Lattice spacing, Δx	1.0	1.0

Parameter	BGK	MRT
Time step, Δt	1.0	1.0
Lattice velocity, $c = \Delta x/\Delta t$	1.0	1.0
Speed of sound, $c_s = c/\sqrt{3}$	$1/\sqrt{3}$	$1/\sqrt{3}$
Collision operator	BGK	MRT
Boundary condition (walls)	No-slip (bounce-back)	No-slip (bounce-back)
Boundary condition (lid)	Prescribed velocity (Zou & He)	Prescribed velocity (Zou & He)

3.2.3 TAYLOR-GREEN VORTEX

The Taylor–Green vortex simulation is one of the classical benchmark cases widely used for the validation of newly developed computational fluid dynamics codes, as it admits a closed-form analytical solution of the incompressible Navier–Stokes equations.

In this scenario, the formation and temporal evolution of vortical structures are examined within the quadrants of a square domain, where an initially prescribed velocity field leads to the development of symmetric vortices that progressively decay due to viscous dissipation. The analytical solution provides an exact reference for both the velocity and pressure fields, making this case particularly suitable for assessing the accuracy of numerical schemes.

The development of this scenario was carried out in a square computational domain divided into four equal quadrants, in which the initial velocity field (u_x, u_y) was prescribed to generate a symmetric vortex pattern. The flow was initialized using the following analytical formulation:

$$\boldsymbol{u_x = +u_0 . sin\left(\frac{2\pi}{L} . x\right) . cos\left(\frac{2\pi}{L} . y\right)} \quad \textit{(3.3)}$$

$$\boldsymbol{u_y = -u_0 . cos\left(\frac{2\pi}{L} . x\right) . sin\left(\frac{2\pi}{L} . y\right)} \quad \textit{(3.4)}$$

Considering absolute values of $x$and y, the initial distribution of the velocity field across the quadrants of the domain is illustrated in Figure 3-7.

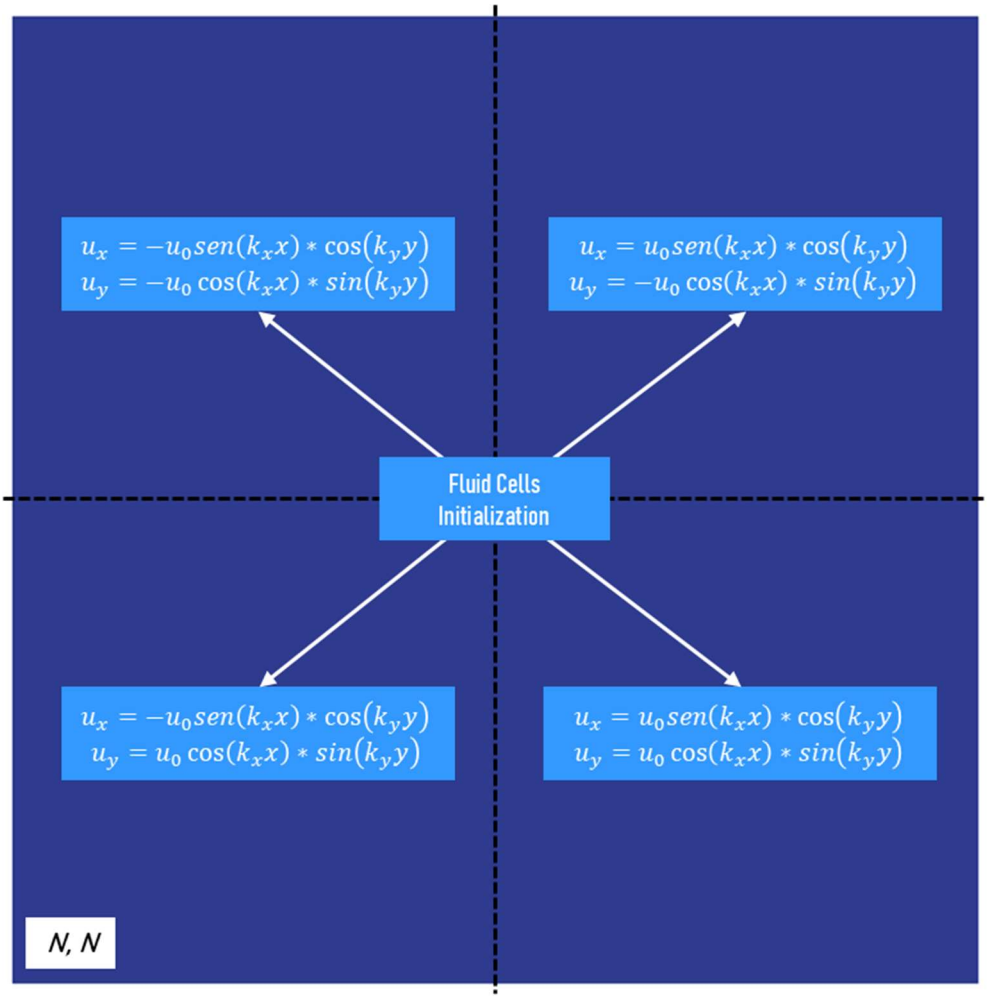

Figure 3-4: Geometry and velocity profile for the Taylor-Green vortex scenario.

Table 3-3 shows a summary of the parameters used to simulate the Taylor-Green vortex scenario.

Table 3-3: Summary parameters for the Taylor-Green vortex scenario, expressed in lattice units.

Parameter	***Value***
Domain size, $N_x \times N_y$	128×128
Total number of lattice cells	***16,384***
Initial velocity amplitude, u_0	***0.05***
Reynolds number, Re	***5***
Kinematic viscosity, ν	***1.28***
Relaxation time, τ	***4.34***
Initial fluid density, ρ_0	***1.0***
Lattice spacing, Δx	***1.0***
Time step, Δt	***1.0***
Lattice speed, $c = \Delta x/\Delta t$	***1.0***
Speed of sound, $c_s = c/\sqrt{3}$***(LU)***	$1/\sqrt{3}$
Collision operator	***BGK***
Total number of time steps	***10,000***

3.2.4 FLUID FLOW AROUND A CYLINDER AND REAL SOIL PARTICLE

Fluid flow around a cylinder is a classical benchmark problem widely used to validate numerical methodologies in computational fluid dynamics. This case has been extensively investigated in the literature (Sato & Kobayashi, 2012; Galindo-Torres, 2013) and presents well-established reference solutions for different flow regimes. Depending on the Reynolds number (Re), both laminar and transitional flow characteristics can be observed, including flow separation, wake formation, and vortex shedding.

In the present analysis, the simulation domain consists of a rectangular channel containing a fixed solid obstacle with a circular shape representing the cylinder. A Poiseuille velocity profile is prescribed at the inlet of the channel, while bounce-back boundary conditions are applied at the solid boundaries to enforce the no-slip condition. Periodic boundary conditions are employed where appropriate to ensure numerical stability and consistency of the flow field.

Figure 3-5 illustrates the geometry and velocity profile of the flow around a cylinder scenario simulated using FSIT

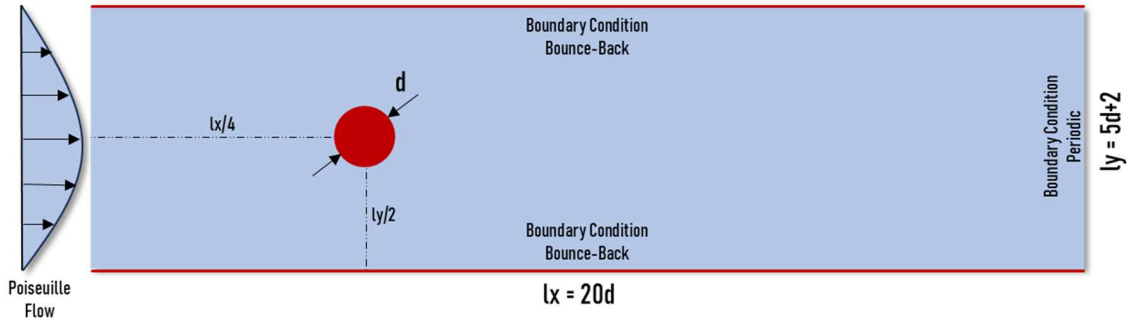

Figure 3-5: Flow around a cylinder geometry and velocity profile implemented on FSIT.

In this simulation, Reynolds numbers of 5, 25, 45, and 100 were adopted to assess the ability of the FSIT framework to simulate laminar and transitional flow regimes around a fixed cylinder. The numerical results were compared with the experimental observations reported by Taneda (1956).

As previously described, the BGK collision operator may become numerically unstable at Reynolds numbers exceeding approximately 100. To address this limitation, the FSIT framework provides the option of using the Multiple Relaxation Time (MRT) collision operator. Accordingly, additional simulations of the flow around a cylinder were performed for Reynolds numbers of 250 and 500. In both cases, the relaxation matrix proposed by Razzaghian et al. (2012) was adopted.

In addition to simulations involving idealized disk-shaped particles, FSIT also allows the integration of realistic solid particles. The generation of real soil particles, however, is not a trivial task. Researchers (Morfa et al., 2017; Recarey et al., 2019) proposed a methodology capable of representing soil grains based on high-resolution scanning techniques, followed by the computational reconstruction of particle geometry using Fourier descriptors. All realistic particles presented in this section, as well as those employed in other simulation scenarios throughout this research, were generated using this methodology.

The integration of realistic particles into FSIT was performed by generating planar cross-sections through the three-dimensional particle geometry, due to the current limitation of the framework to two-dimensional analyses. These cross-sectional planes were subsequently converted into point clouds, which were then used by the code to reconstruct the particle geometry as discrete elements within the numerical domain. Following this methodology, an initial simulation was conducted involving a single realistic soil particle in order to evaluate how FSIT handles such geometries within a fluid–solid interaction scenario.

Figure 3-6 illustrates a realistic soil particle computationally generated using the Fourier descriptor–based methodology adopted in this study.

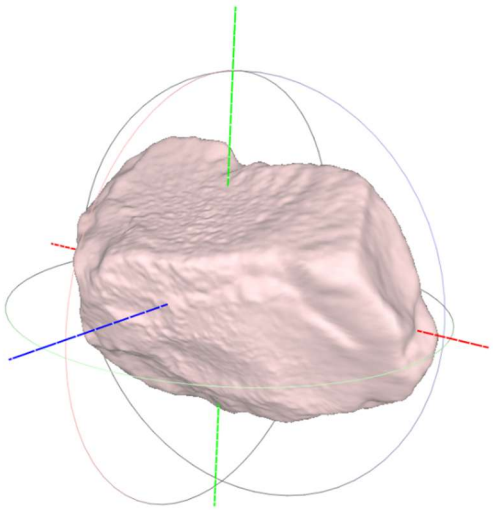

Figure 3-6: Realistic soil particle computationally generated using the Fourier descriptor–based methodology.

After the computational generation of the realistic soil particle, the geometry was sectioned using three orthogonal planes (XY, YZ, and ZX), as illustrated in Figure 3-7.

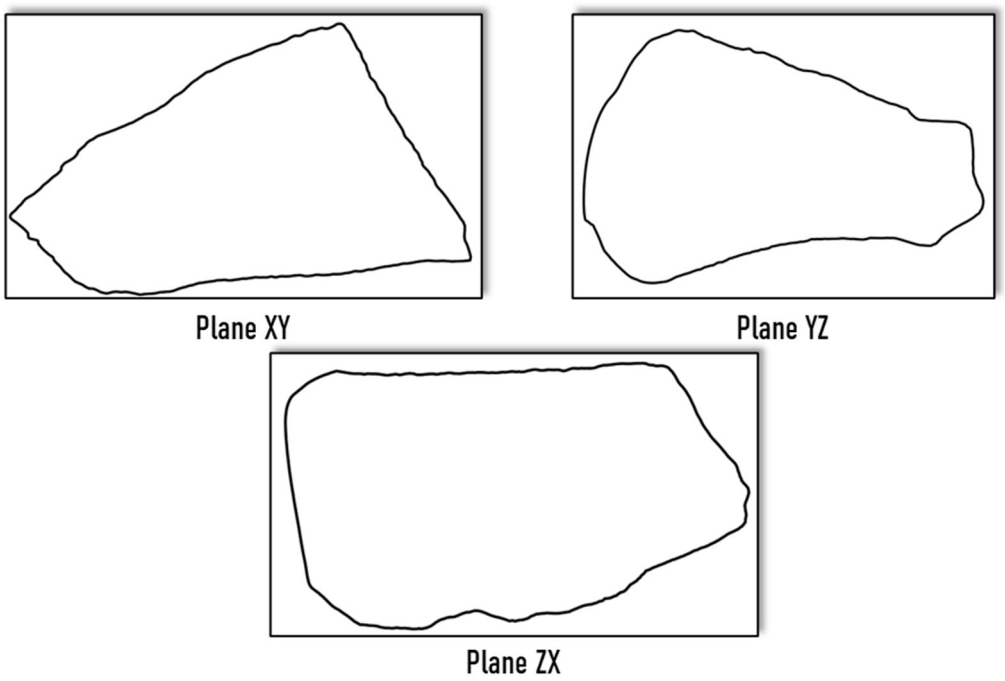

Figure 3-7: Generation of discrete elements from the XY, YZ, and ZX planes of the realistic soil particle.

Each sectional plane was subsequently converted into a cloud of points, which constitutes the geometric representation adopted by FSIT for reconstructing realistic particle shapes under the current two-dimensional formulation. Based on these point clouds, the code was able to reassemble the sectional geometries and transform them into discrete elements suitable for numerical simulation.

To evaluate the capability of the framework to correctly reconstruct the particle geometry from the point clouds and to represent the associated fluid–solid interaction, a numerical simulation was performed using a Poiseuille flow imposed at the inlet of a rectangular channel. In this analysis, a Reynolds number of 100 was adopted, corresponding to a transitional flow regime.

3.2.5 DISCRETE ELEMENT PACKING

In addition to the simulation of a single discrete element, it is necessary to evaluate the behavior of a packing of particles. In this scenario, the objective is to investigate the formation of preferential flow paths arising from the spatial distribution and shape of the discrete elements.

The geometry adopted for this analysis consists of a rectangular channel in which the particles are positioned within a central region of the domain. Buffer zones were intentionally introduced

at both the inlet and outlet of the channel to ensure that the observed flow phenomena are not influenced by boundary-condition effects.

Figure 3-8 illustrates a schematic representation of the computational domain, highlighting the geometry and the positioning of the discrete elements used in this scenario.

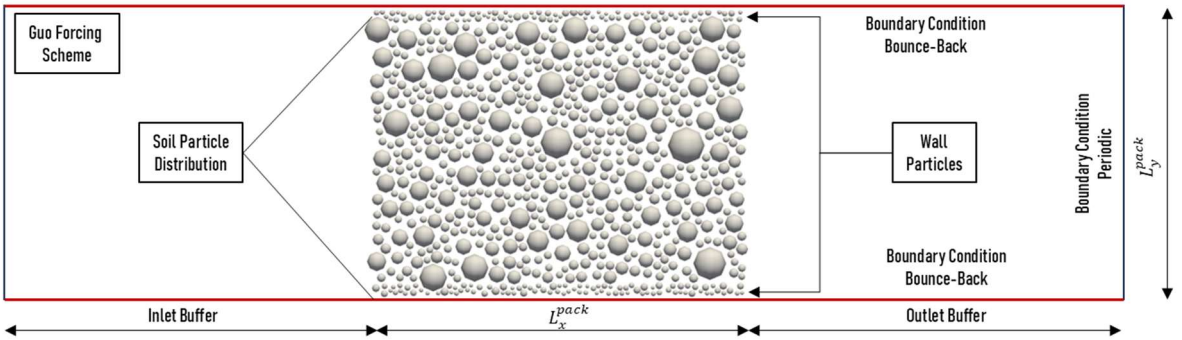

Figure 3-8: Geometry for the discrete element packing scenario.

In this scenario, the forcing scheme proposed by Guo is employed to generate a constant pressure gradient along the channel. In practical terms, a uniform body force is applied to the fluid nodes, which induces fluid acceleration and drives the flow through the particulate medium. The resulting pressure gradient is defined according to the following formulation:

$$\frac{dp}{dx} = -\rho . g_x \tag{3.5}$$

Based on the imposed pressure gradient, the permeability of the porous medium can be assessed through the evaluation of the mean flow velocity ($\bar{u}$). This quantity is directly proportional to the applied pressure gradient and inversely proportional to the dynamic viscosity of the fluid (μ), as shown below:

$$\bar{u} = -\frac{k}{\mu}.\frac{dp}{dx} \rightarrow \bar{u} = \frac{k}{\mu}.(\rho . g_x) \tag{3.6}$$

However, the evaluation of the mean velocity across the transverse section of the particle packing requires consideration of the porosity (n) of the medium. The relationship between the cross-sectional (superficial) average velocity of the porous matrix ($\bar{u}_{pore}$) and the average pore velocity is expressed as follows:

$$\bar{u}_{pore} = \frac{\bar{u}}{n} \tag{3.7}$$

Therefore, by combining Eqs. (3.6) and (3.7), the permeability of the porous medium can be computed, as expressed below:

$$k = \frac{\mu . n . \bar{u}_{pore}}{\rho . g_x} \tag{3.8}$$

Based on the numerical evaluation of permeability, the obtained values were compared with the classical formulations proposed by Kozeny–Carman (1927) and Ergun (1952). These equations are empirical models that estimate the permeability of a porous medium as a function of its porosity and particle geometry, typically represented by an equivalent particle diameter (d). It is important to note that these formulations were originally derived for three-dimensional analyses and are employed in this research primarily to provide an order-of-magnitude reference for the numerically obtained permeability values.

The Kozeny–Carman (1927) and Ergun (1952) equations are structurally similar, differing mainly in the empirical coefficient C, which takes a value of 180 for the Kozeny–Carman model and 150 for the Ergun model. The general form of these equations is presented below.

$$\boldsymbol{k = \frac{n^3 d^2}{C(1-n)^2}} \qquad \textit{(3.9)}$$

Finally, to investigate preferential flow paths and evaluate the numerical permeability, three simulation scenarios were developed, each characterized by different particle shapes and spatial distributions. The first configuration consisted of disk-shaped particles arranged in a regular pattern, for which the Ergun and Kozeny–Carman equations are more suitable. Subsequently, a packing with randomly distributed particle positions was analyzed to assess the consistency of the numerical results with the empirical correlations proposed by these authors. Lastly, packings composed of realistic particle geometries were considered, with particle shapes reconstructed using the methodology proposed by Morfa et al. (2017) and Recarey et al. (2019).

3.2.6 PARTICLE SETTLING: STOKES' LAW

Up to this point, all simulations were performed under conditions in which the discrete elements were not allowed to move freely within the computational domain. However, for fluid–solid interaction problems of practical interest, allowing particle motion is essential in order to properly capture the exchange of linear and angular momentum between the fluid and solid phases.

To assess this hydrodynamic coupling, the particle settling problem governed by Stokes' law was selected as a reference scenario. This classical benchmark provides a well-established analytical solution and is widely used to validate numerical approaches for fluid–particle interaction.

It should be noted, however, that Stokes' law was originally derived for three-dimensional conditions. Consequently, appropriate adaptations of the classical formulation are required to ensure consistency with the two-dimensional framework adopted in the present simulations. These modifications are discussed in this topic.

The Stokes' law establishes a mathematical relationship governing the motion of small spheres moving through a viscous fluid. Its derivation is based on analytical solutions of the Navier–Stokes equations under specific assumptions, namely: incompressible flow, laminar regime, very low Reynolds number (Re $\ll$ 1), and perfectly spherical particles, corresponding to the three-dimensional conditions previously mentioned.

The analysis of particle settling represents one of the most important applications of Stokes' law. In this scenario, a particle immersed in a fluid is released and allowed to fall freely until it reaches a constant settling velocity, commonly referred to as the terminal velocity. When the particle attains terminal velocity, the forces acting on it are in kinematic equilibrium.

Figure 3-9 illustrates the force balance acting on the particle, namely the particle weight (W), the buoyant force (B), and the drag force (D).

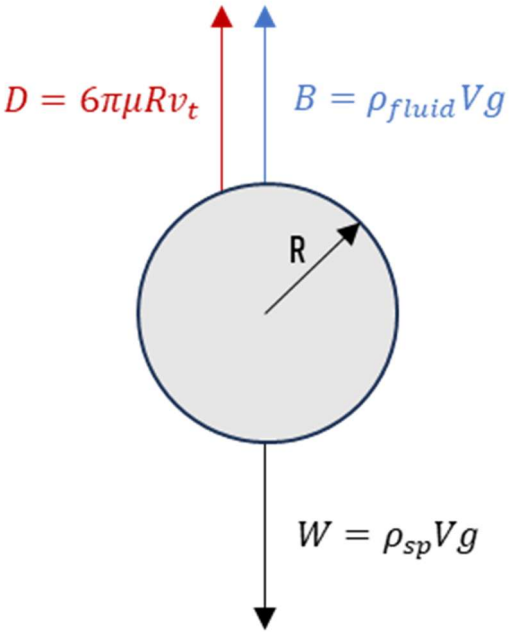

Figure 3-9: Force balance acting on the sphere.

Based on the force balance described above, which is valid under the three-dimensional conditions for which Stokes' law applies, the terminal settling velocity (v_t) can be derived analytically. From the balance of forces, the terminal velocity of the particle may be expressed as follows:

$$v_t = \frac{2gR^2(\rho_{sp} - \rho_{fluid})}{9\mu} \quad (3.10)$$

However, since FSIT is a tool designed for two-dimensional simulations, it is necessary to reassess these formulations in order to properly account for the dimensionality of the problem. To this end, the particle is modeled as a cylinder with unit height, and its volume V is defined accordingly. Under this assumption, the terminal settling velocity for the two-dimensional case can be expressed as follows.

$$v_t = \frac{Rg(\rho_{sp} - \rho_{fluid})}{6\mu} \quad (3.11)$$

Finally, in order to validate this scenario, the drag force acting on the discrete element was evaluated. In the FSIT implementation, the resultant force arising from the fluid–solid interaction is computed using the Immersed Moving Boundary (IMB) method. This force therefore represents the hydrodynamic drag experienced by the particle as it moves through the fluid.

For validation purposes, it was assumed that the numerically computed hydrodynamic force must be in equilibrium with the difference between the particle weight and the buoyant force. This assumption is consistent with the terminal settling condition, under which the net force acting on the particle is zero and all forces are in balance, as discussed previously.

$$F_{hydro} = W - B$$

$$F_{hydro} = (\rho_{sp} - \rho_{fluid}).V.g \quad (3.12)$$

$$F_{hydro} = (\rho_{sp} - \rho_{fluid}).\pi R^2.1.g \therefore F_{hydro} = (\rho_{sp} - \rho_{fluid}).g\pi R^2$$

CHAPTER IV

4 RESULTS AND DISCUSSION

This chapter presents the data processing procedures and the visualization of the results obtained from the simulation scenarios introduced in the previous sections. For all scenarios, the discretization of the problem's geometry into fluid, solid, and partial nodes is presented, along with relevant information such as velocity profiles, preferential flow paths, and other quantities required for a comprehensive understanding of the analyzed problem.

4.1 POISEUILLE FLOW

The Poiseuille flow scenario was developed using two different approaches. In the first approach, the analytical parabolic velocity profile characteristic of Poiseuille flow was directly imposed at the channel inlet. In the second approach, the flow was generated by accelerating the fluid particles through the application of a constant pressure gradient, implemented via the forcing scheme proposed by Guo. For both approaches, the problem geometry was identical, as illustrated below.

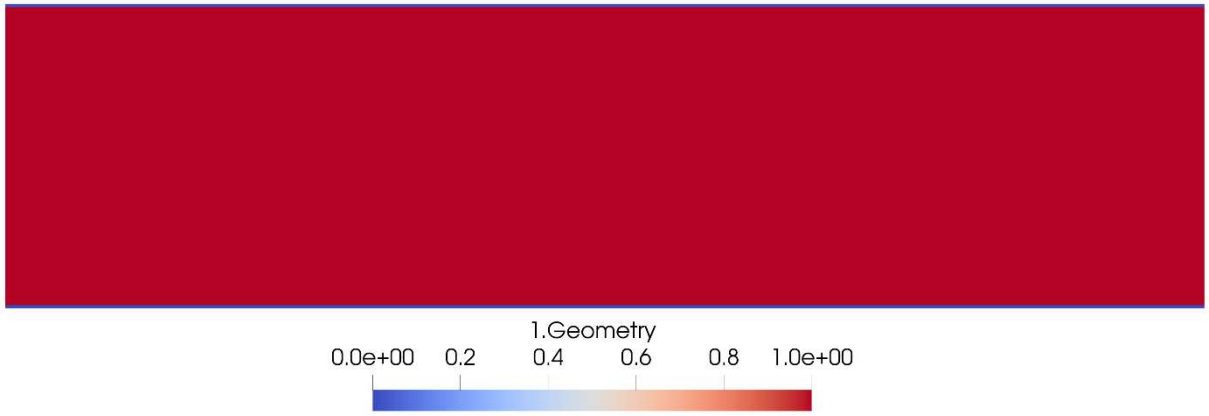

Figure 4-1: Geometry for the Poiseuille flow scenario.

The geometry presented above shows that lattice nodes were classified into fluid nodes (value 1: red color) and wall nodes (value 0: blue color).

Once the geometry was defined, the simulation scenarios were performed. For the first case, a parabolic velocity profile was imposed at the channel inlet, and the resulting velocity distribution was analyzed at three locations along the channel: the inlet, the mid-section, and the outlet. This procedure was adopted to verify the preservation of the velocity profile

throughout the channel. Figure 4-2 illustrates the velocity profiles obtained during the simulation of the Zou and He Poiseuille flow scenario.

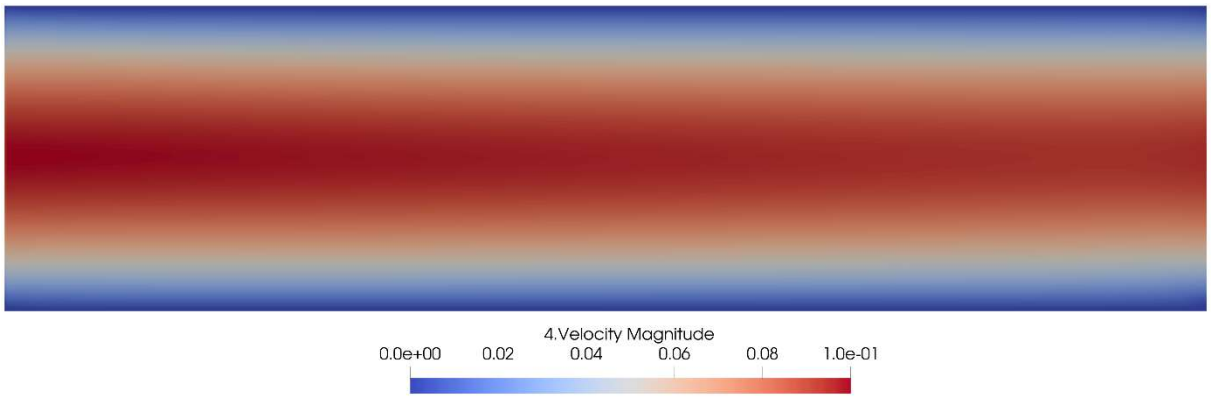

Figure 4-2: Velocity profile for the Zou and He Poiseuille flow scenario.

As shown in the figure, the velocity field along the entire channel displays a parabolic distribution, with the maximum velocity located at the channel center. Subsequently, a comparison between the analytical solution and the numerical velocity profiles obtained at the analyzed cross-sections is presented.

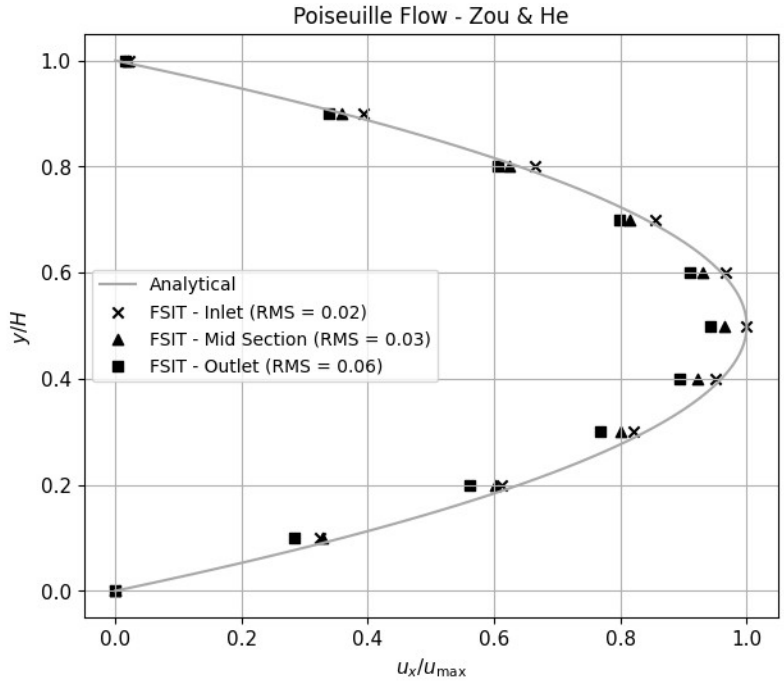

Figure 4-3: Comparison between the analytical solution and the numerical velocity profiles obtained at the analyzed cross-sections – Zou and He Scenario.

The Poiseuille flow scenario employing the Zou and He boundary conditions shows that, at the channel inlet, the analytical solution and the numerical results are in near-perfect agreement, with a very low error (RMS = 0.02). However, as the analysis sections move away from the inlet, a slight loss of accuracy can be observed. At the center and outlet of the channel, the error remains low (RMS values of 0.03 and 0.06, respectively), although a gradual increase is

noticeable. Nevertheless, this variation does not invalidate the implemented methodology, since the numerical results remain within acceptable error limits. It is also worth noting that this scenario was simulated using the BGK collision operator; for more demanding applications, the MRT collision operator may be employed to further improve numerical accuracy.

A similar analysis was conducted for the scenario in which a pressure gradient was imposed using the forcing scheme proposed by Guo. As in the previous case, velocity profiles were evaluated at the same cross-sectional locations along the channel. The velocity profile obtained for the Guo forcing scheme is illustrated below.

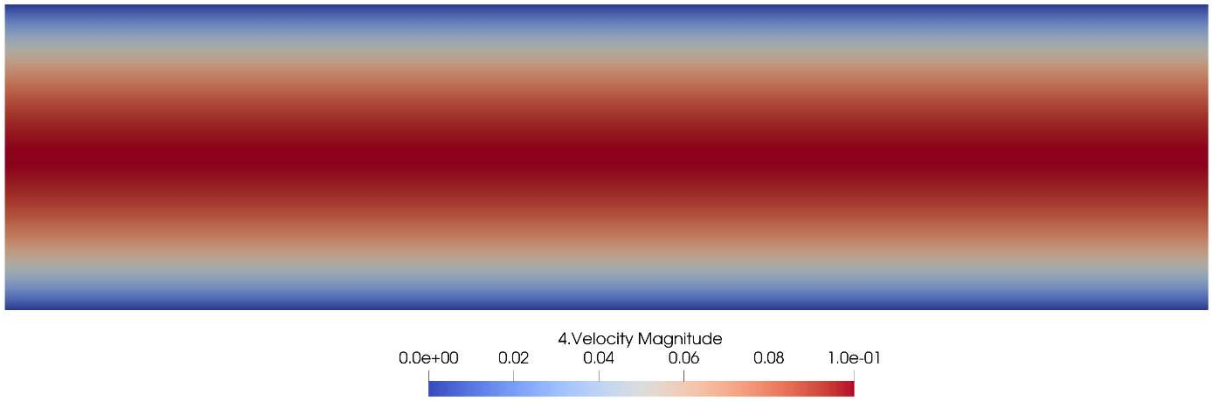

Figure 4-4: Velocity profile for the Guo forcing Poiseuille flow scenario.

The velocity field obtained in this scenario clearly shows a continuous parabolic distribution along the channel. To further evaluate this behavior, a comparison between the numerical results and the analytical solution is presented below.

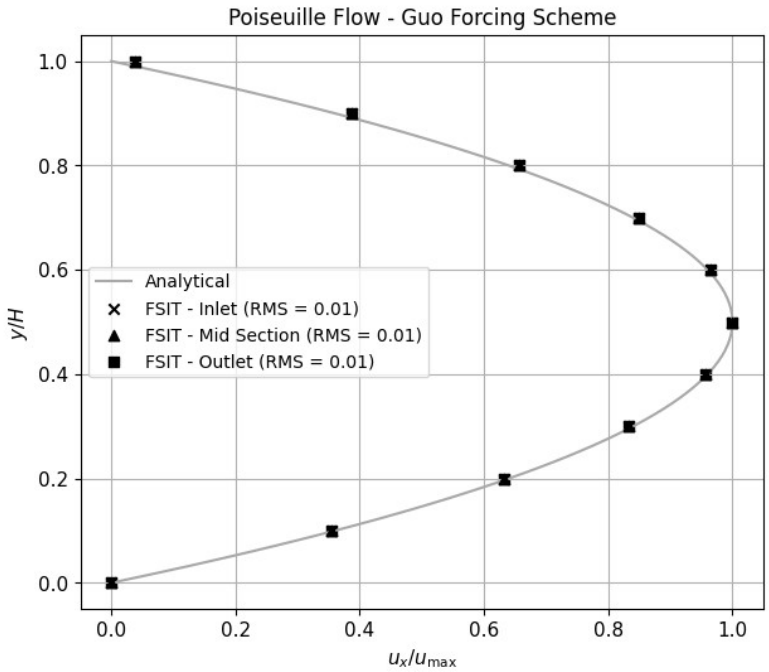

Figure 4-5: Comparison between the analytical solution and the numerical velocity profiles obtained at the analyzed cross-sections – Guo Forcing Scheme scenario.

Based on the analysis of the Figure 4-5, it can be observed that the forcing scheme proposed by Guo is able to develop a parabolic velocity profile without any noticeable loss along the channel. Moreover, in all analyzed cross-sections, the observed error remained very low (RMS = 0.01), indicating that the methodology was correctly implemented in FSIT.

Although differences exist between the two approaches, both analyses successfully captured the development of Poiseuille flow within the channel and can therefore be used to evaluate other simulation scenarios. With respect to the model employing the Zou and He boundary conditions, the MRT collision operator may be adopted if increased accuracy is required for the scenario under investigation.

4.2 LID-DRIVEN CAVITY FLOW

The lid-driven cavity flow scenario consists of a two-dimensional square cavity discretized using a uniform Cartesian lattice composed of 256 × 256 nodes, resulting in a total of 65,536 lattice cells. The computational domain is fully enclosed by rigid walls on the left, right, and bottom boundaries, while the top boundary acts as a moving lid with a prescribed constant horizontal velocity. The geometry considered for this analysis is illustrated below.

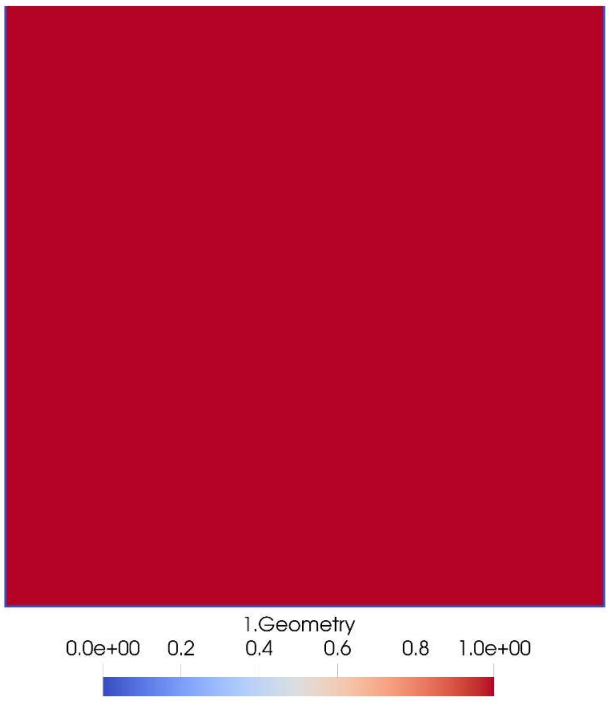

Figure 4-6: Geometry for the Lid-driven cavity flow scenario.

The geometry presented above shows that lattice nodes were classified into fluid nodes (value 1: red color) and wall nodes (value 0: blue color).

Based on Figure 4-6, simulations were performed to evaluate fluid behavior. In this scenario, the Reynolds number was set to 100. The velocity field developed in this case is illustrated below.

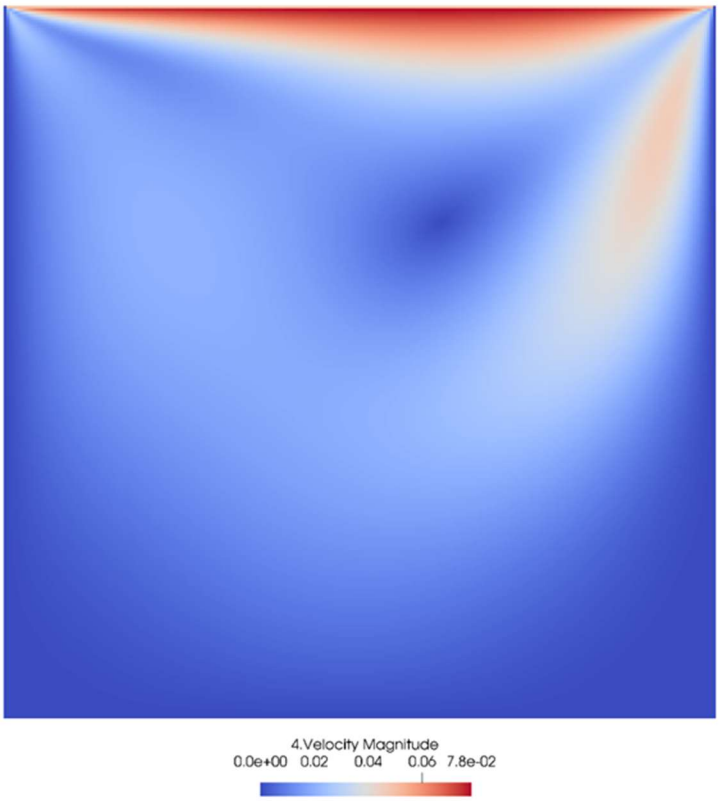

Figure 4-7: Velocity profile for the Lid-driven cavity flow scenario.

To provide further insight into the fluid behavior observed in this scenario, the velocity vector field is presented below.

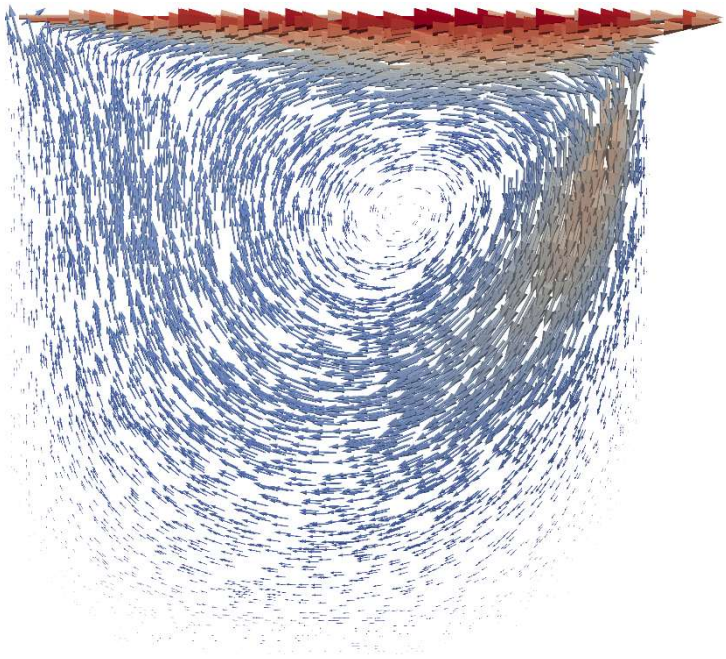

Figure 4-8: Velocity vector field for the Lid-driven cavity flow scenario.

It is noteworthy from the velocity vector field that, when a constant velocity is applied at the top boundary of the cavity for a system with a Reynolds number of 100, vortical structures develop within the fluid domain. To validate this scenario, velocity profiles were extracted along vertical and horizontal centerlines, as illustrated in Figure 3-3, and subsequently compared with the benchmark results reported by Ghia et al. (1982).

Since the flow regime analyzed corresponds to a relatively high Reynolds number and Zou and He boundary conditions were employed, simulations were performed using both the BGK and MRT collision operators. This comparison aimed to assess the influence of the collision model on the accuracy of the velocity profiles captured along the selected sections.

Figure 4-9 illustrates the comparison between the results obtained using FSIT framework with BGK and MRT collision operators and the data reported by Ghia *et al.* (1982).

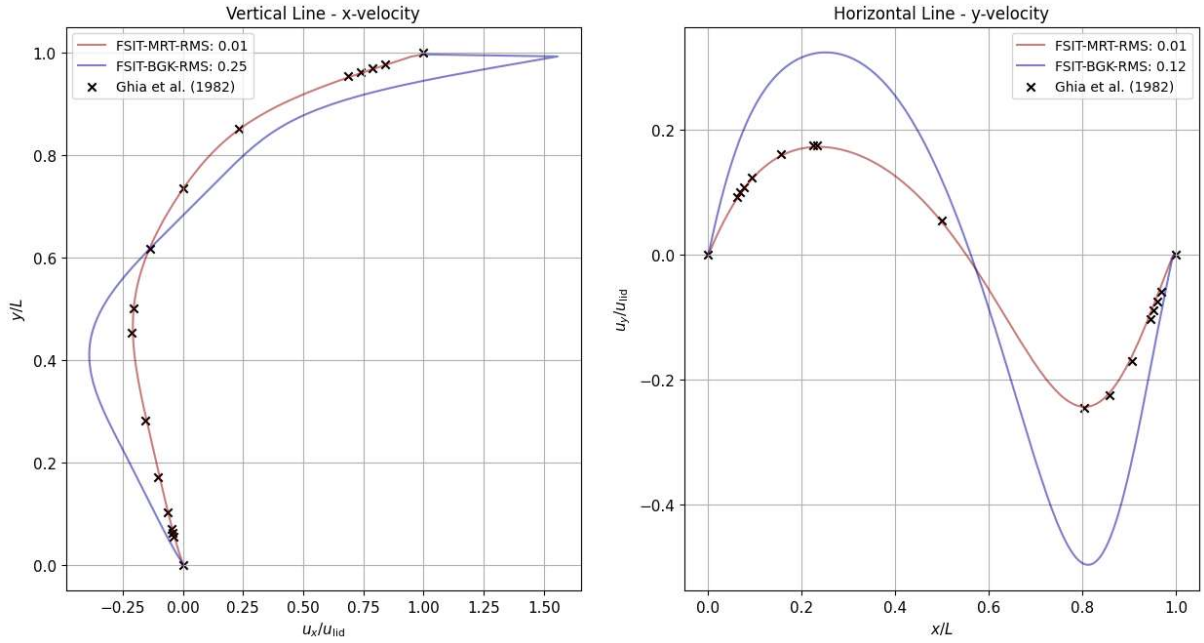

Figure 4-9: Comparison between results obtained using FSIT and data reported by Ghia *et al*. (1982).

It is important to note that, although the results obtained using the BGK collision operator are of the expected order of magnitude for this problem, the associated error is significant for this analysis (RMS = 0.25). As discussed previously, since the BGK operator relaxes the distribution functions using a single relaxation parameter, its accuracy deteriorates as the Reynolds number increases.

In contrast, the MRT collision operator provides greater flexibility by allowing different relaxation rates for distinct moments. As a result, the outcomes obtained for this scenario using the MRT operator show excellent agreement with the benchmark results reported by Ghia et al.

(1982), yielding a much lower error (RMS = 0.01). It should be emphasized, however, that the use of the MRT operator requires a more careful calibration of the relaxation parameters.

Although both collision models are capable of capturing the main features of the flow, the BGK operator exhibits reduced accuracy when applied to simulations at higher Reynolds numbers. Therefore, for such flow regimes, the MRT collision operator is preferred.

4.3 TAYLOR-GREEN VORTEX

For the development of this scenario, a square domain (N × N) was constructed without the use of solid boundaries, such that all cells represent fluid nodes. The domain is then divided into four quadrants, and a distinct initial velocity field is prescribed in each quadrant to represent the formation of vortices. The geometry adopted for this scenario is illustrated below.

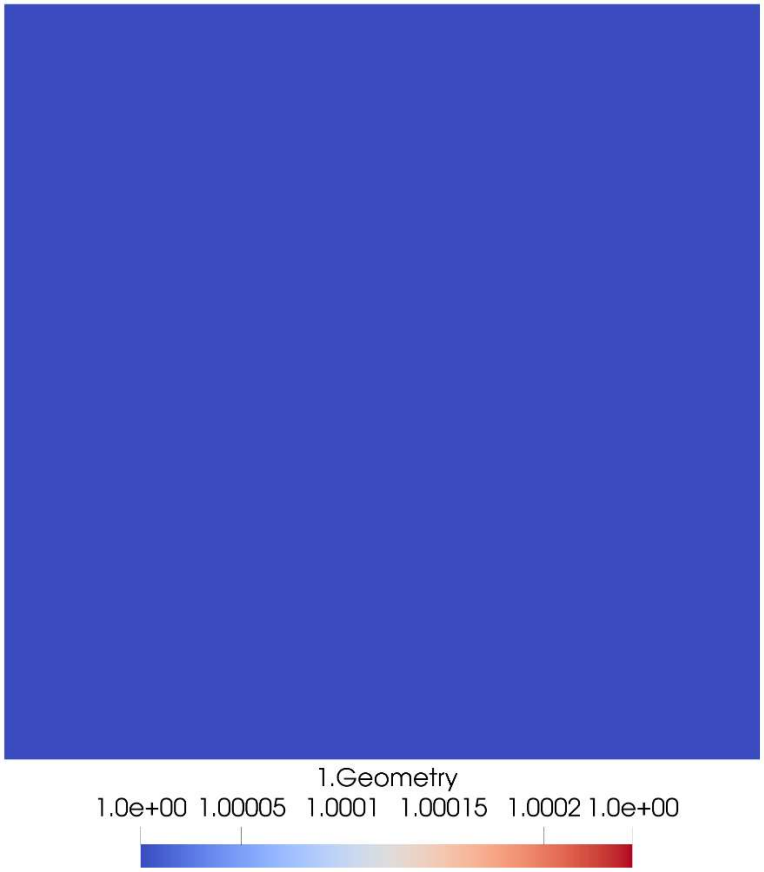

Figure 4-10: Geometry for the Taylor-Green Vortex scenario.

As illustrated in Figure 4-10, the lattice nodes were classified as fluid nodes (value 1, blue color).

Using this geometry, velocity fields were initialized in each quadrant, and the resulting velocity field were examined, as shown below.

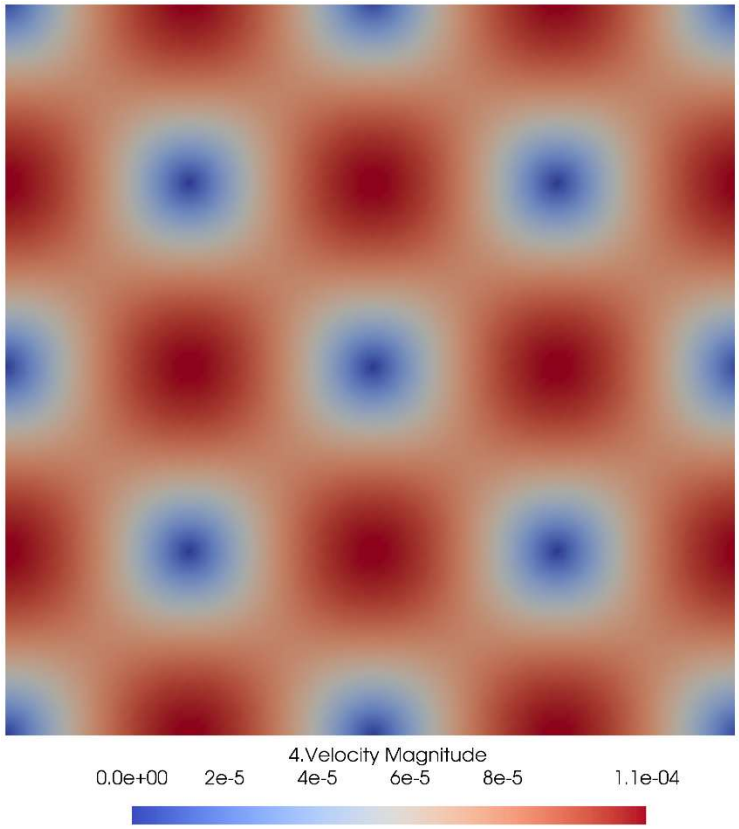

Figure 4-11: Velocity profile for the Taylor-Green Vortex scenario.

For improved visualization of the vortical structures developing in each quadrant, the velocity vector field is shown below.

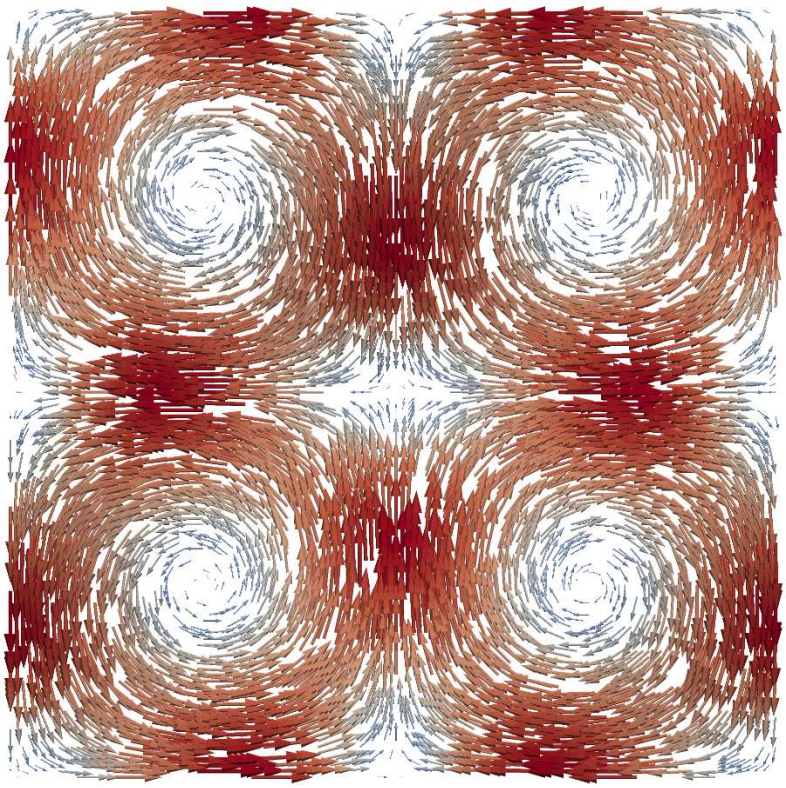

Figure 4-12: Velocity vector field for the Taylor-Green Vortex scenario.

It is noteworthy that the LBM implemented in FSIT is capable of accurately capturing and reproducing classical scenarios widely studied in the field of computational fluid dynamics. Up to this point, the focus has been on describing fluid behavior across a range of benchmark problems, considering only the fluid phase. The following simulation cases were therefore developed to demonstrate the capability of FSIT in modeling fluid–solid interaction problems.

4.4 FLOW AROUND A CYLINDER AND REAL SOIL PARTICLE

For the development of this scenario, a rectangular channel was constructed, and a disk was positioned to represent a soil particle. In this initial analysis, the soil particle geometry was simplified as a circular element with radius R (i.e., a disk in a two-dimensional domain). The geometry constructed in FSIT for this simulation scenario is illustrated below.

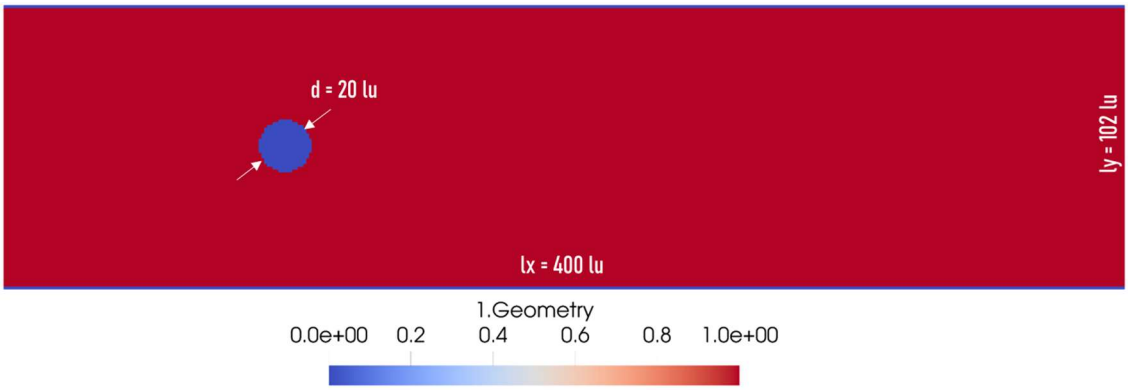

Figure 4-13: Geometry for the flow around a cylinder scenario.

The geometry presented above shows that lattice nodes were classified into fluid nodes (value 1: red color), wall and solid nodes (value 0: blue color).

In addition to the geometry, it is important to evaluate the distribution of the solid fraction, since LBM–DEM coupling is required in this simulation. Accordingly, Figure 4-14 illustrates the solid fraction distribution for this scenario.

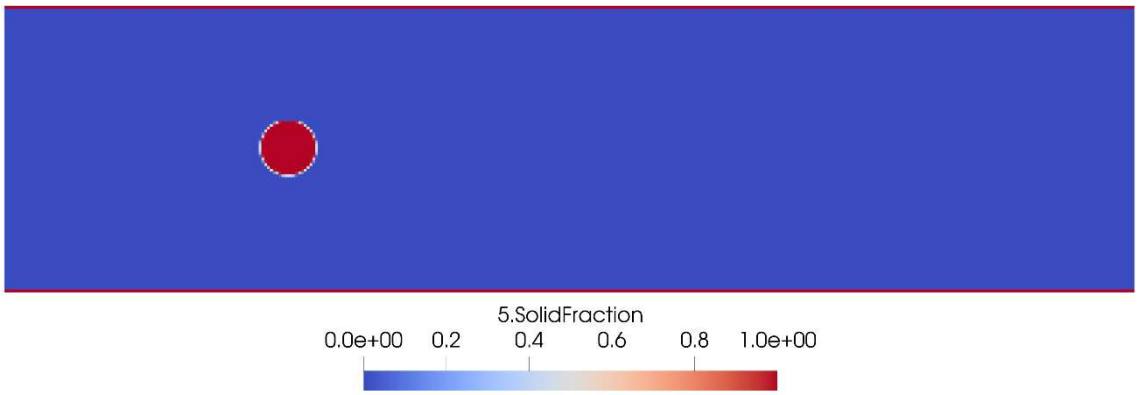

Figure 4-14: Solid fraction for the flow around a cylinder scenario.

It is noteworthy to observe the distribution of the solid fraction along the particle boundary, confirming that FSIT correctly identifies lattice cells that are fully or partially occupied by the discrete element.

Once the analysis conditions were defined, simulations were conducted for Reynolds numbers of 5, 25, 45, and 100. The objective of these simulations was to evaluate fluid–solid interaction and to investigate the transition from laminar to turbulent flow regimes as a function of the Reynolds number. The simulation results are illustrated below and compared with the flow patterns reported by Taneda (1956).

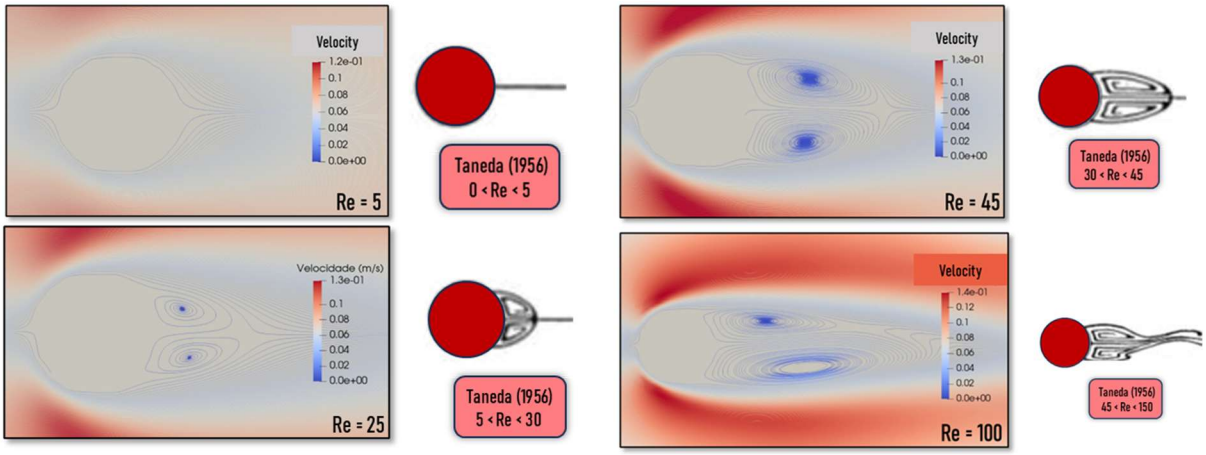

Figure 4-15: Comparison between results obtained using FSIT and Taneda (1956).

As the Reynolds number increases from 5 to 100, the flow around the cylinder undergoes a clear transition from a steady and symmetric regime to an unsteady, vortex-dominated wake. At low Reynolds numbers (Re = 5), the flow remains attached and exhibits a symmetric velocity field downstream of the cylinder. As the Reynolds number increases to intermediate values (Re = 25 and 45), flow separation becomes evident, leading to the formation of recirculation regions and the emergence of counter-rotating vortices in the wake. For Re = 100, the wake is characterized by pronounced vortex structures and increased unsteadiness, indicating the onset of more complex flow dynamics.

The numerical results obtained with FSIT successfully capture the key features of this transition, including flow separation, vortex growth, and wake development. The observed flow patterns show strong qualitative agreement with the experimental flow regime classifications reported by Taneda (1956), demonstrating the capability of the proposed LBM–DEM–IMB framework to accurately reproduce classical fluid–structure interaction phenomena across a range of Reynolds numbers.

For a Reynolds number of 100, the flow exhibits the formation of a Von Kármán vortex street in the wake of the cylinder. To provide a clearer representation of the velocity field associated with this regime, the corresponding velocity vector field illustrated in Figure 4-16.

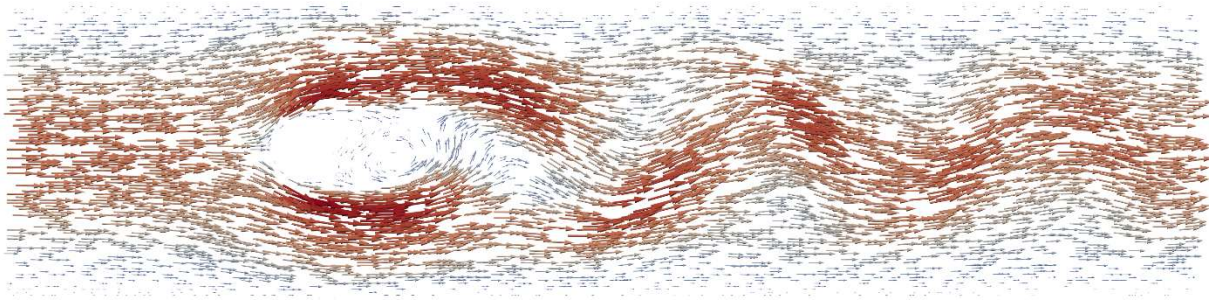

Figure 4-16: Velocity vector field for the flow around a cylinder scenario with Re = 100 using BGK collision operator.

As the Reynolds number in the previous simulations reached values up to 100, the BGK collision operator begins to exhibit numerical limitations when applied to flows with higher Reynolds numbers. Therefore, to assess flow behavior under more turbulent regimes, it becomes necessary to employ the MRT collision operator. The results of a simulation performed using the MRT formulation for a Reynolds number of 500 is illustrated in Figure 4-17.

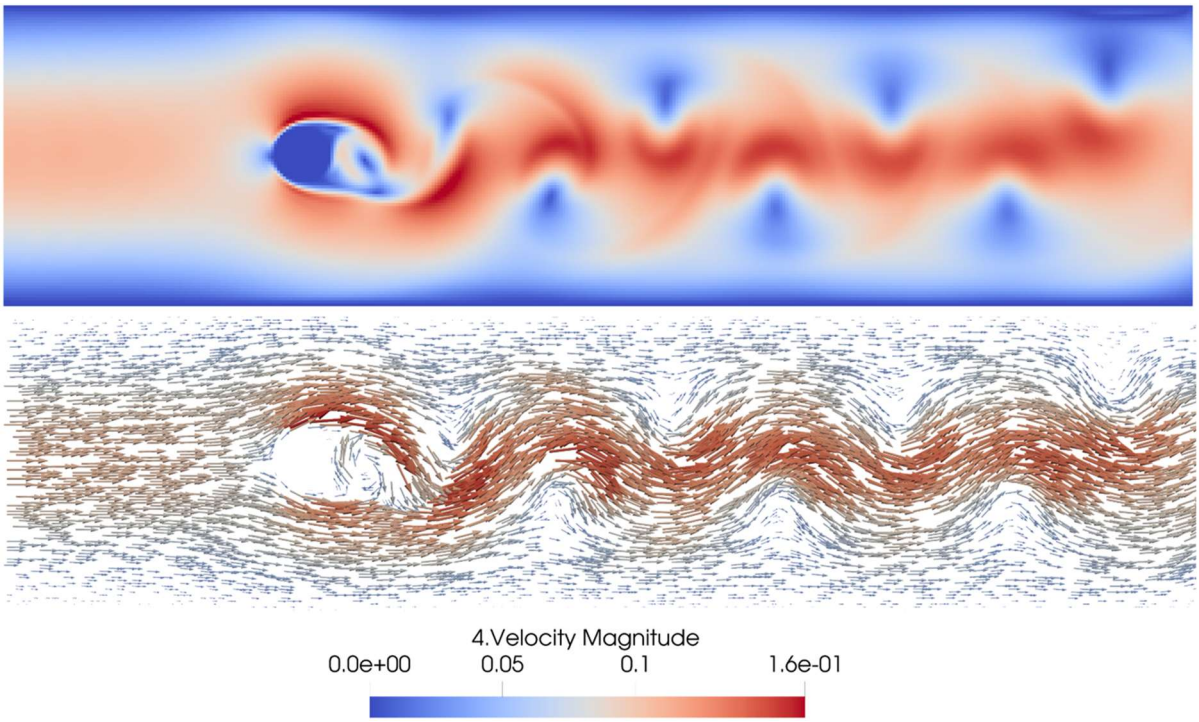

Figure 4-17: Velocity profile and vector field for the flow around a cylinder scenario with Re = 500 using MRT collision operator.

As expected, increasing the Reynolds number leads to a significant intensification of vortex formation in the wake region downstream of the cylinder. For $Re = 500$, the flow becomes strongly unsteady, with the emergence of a well-defined von Kármán vortex street characterized

by alternating vortices shed from the cylinder surface. The velocity magnitude field highlights the pronounced oscillatory behavior of the wake, while the velocity vector field reveals complex flow structures and strong recirculation zones. These results indicate a transition to a highly dynamic and chaotic flow regime, demonstrating the capability of the FSIT framework combined with the MRT collision operator to capture turbulent wake dynamics and complex fluid–solid interaction phenomena at high Reynolds numbers.

In addition to simulations involving discrete elements with idealized circular shapes, FSIT allows the reconstruction of realistic particle geometries based on point cloud representations. To demonstrate this capability, a real soil particle was numerically reconstructed and subsequently sectioned along three orthogonal planes. Using this reconstructed geometry, the same domain configuration and boundary conditions adopted in the previous circular particle simulations were applied. The resulting velocity field around the particle was then evaluated for a Reynolds number of 100. The velocity profile obtained for this scenario is illustrated in Figure 4-18.

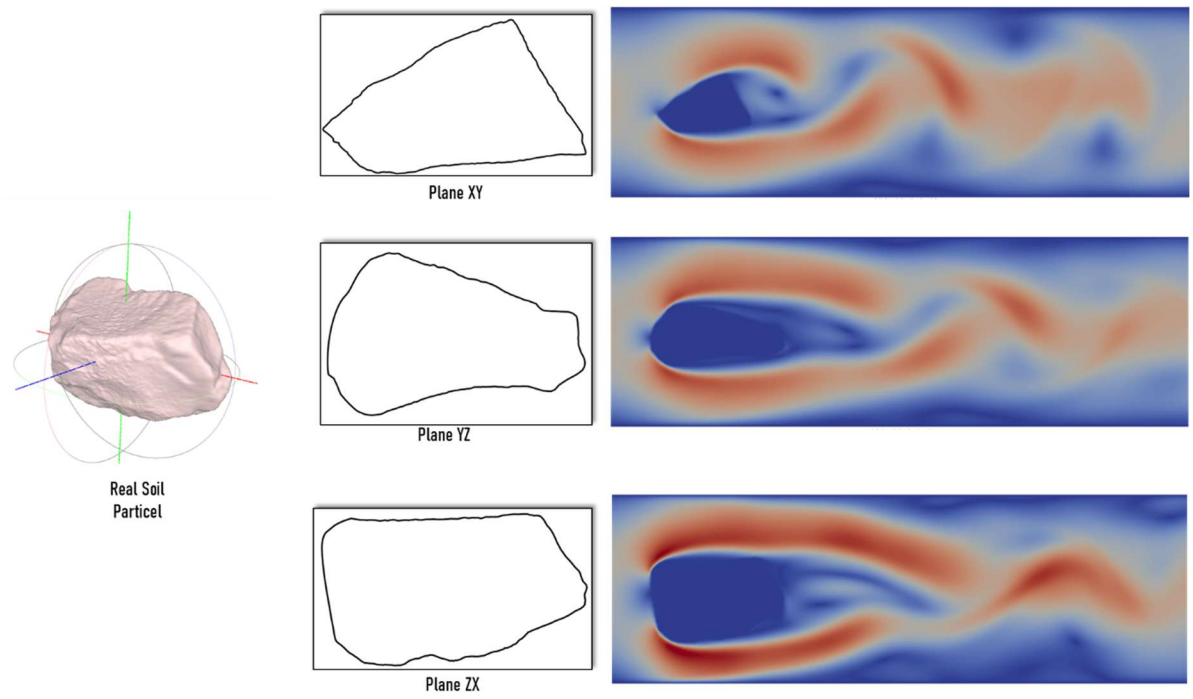

Figure 4-18: Velocity profile for the flow around a real soil particle scenario with Re = 100 using BGK collision operator.

All these features demonstrate the capability of FSIT to simulate both idealized and realistic scenarios, highlighting its potential for application in the analysis of real engineering problems and geotechnical processes, such as laboratory-scale experiments and practical engineering studies.

4.5 DISCRETE ELEMENT PACKING

This scenario consists of three distinct analyses. The first two analyses focus on the evaluation of fluid flow through porous media composed of discrete elements with circular shapes (disks). For these analyses, a rectangular channel was constructed and filled with particle packings generated according to two different configurations: a uniform disk packing and a randomly distributed disk packing.

In the first configuration, disks with identical radii were arranged in a regular pattern, resulting in a uniform porous structure. In the second configuration, disks with varying radii were placed at random positions to form a packing that more closely resembles the heterogeneity typically observed in soil samples. In both cases, buffer regions were intentionally introduced at the inlet and outlet of the channel to minimize the influence of boundary conditions on the flow response within the porous medium.

The geometrical configurations adopted for these analyses are illustrated below.

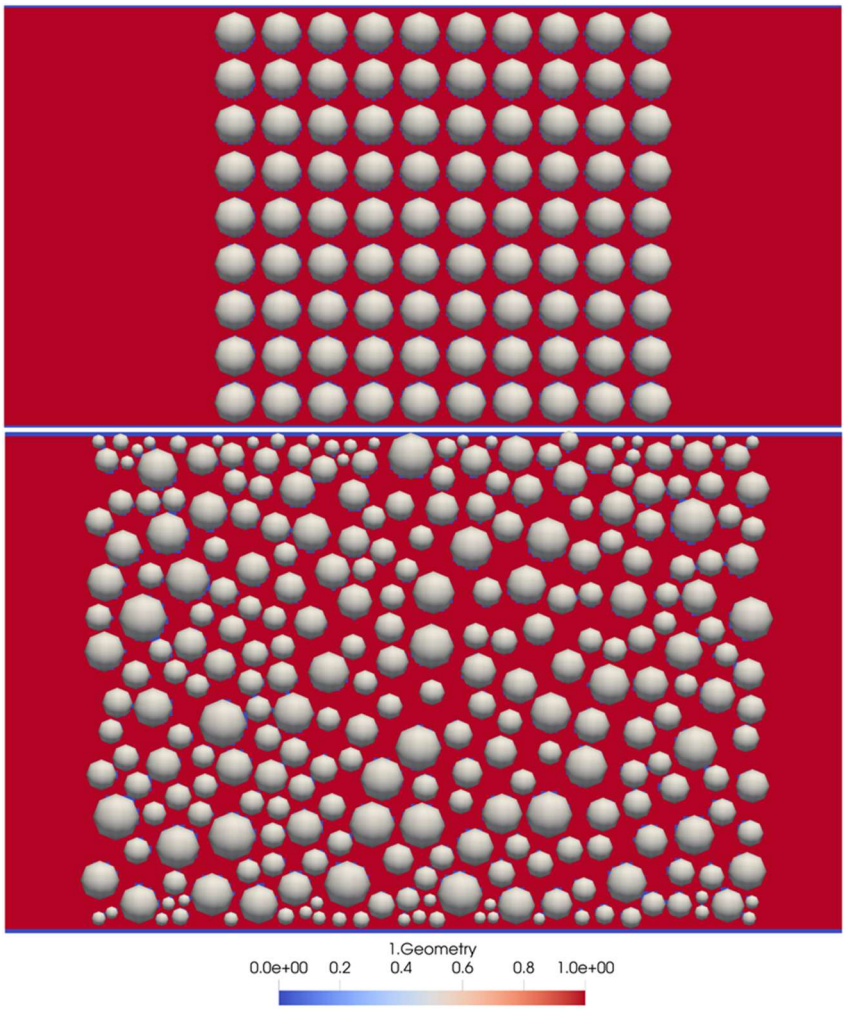

Figure 4-19: Geometry for the discrete element packing – disks scenario.

The geometry, illustrated in Figure 4-19, shows that lattice nodes were classified into fluid nodes (value 1: red color), wall and solid nodes (value 0: blue color). Also, discrete elements are shown in white.

After defining the geometrical configurations, a constant pressure gradient was imposed along the channel to drive the flow and to assess the development of preferential flow paths within the porous structures. The resulting velocity fields for each scenario are illustrated below.

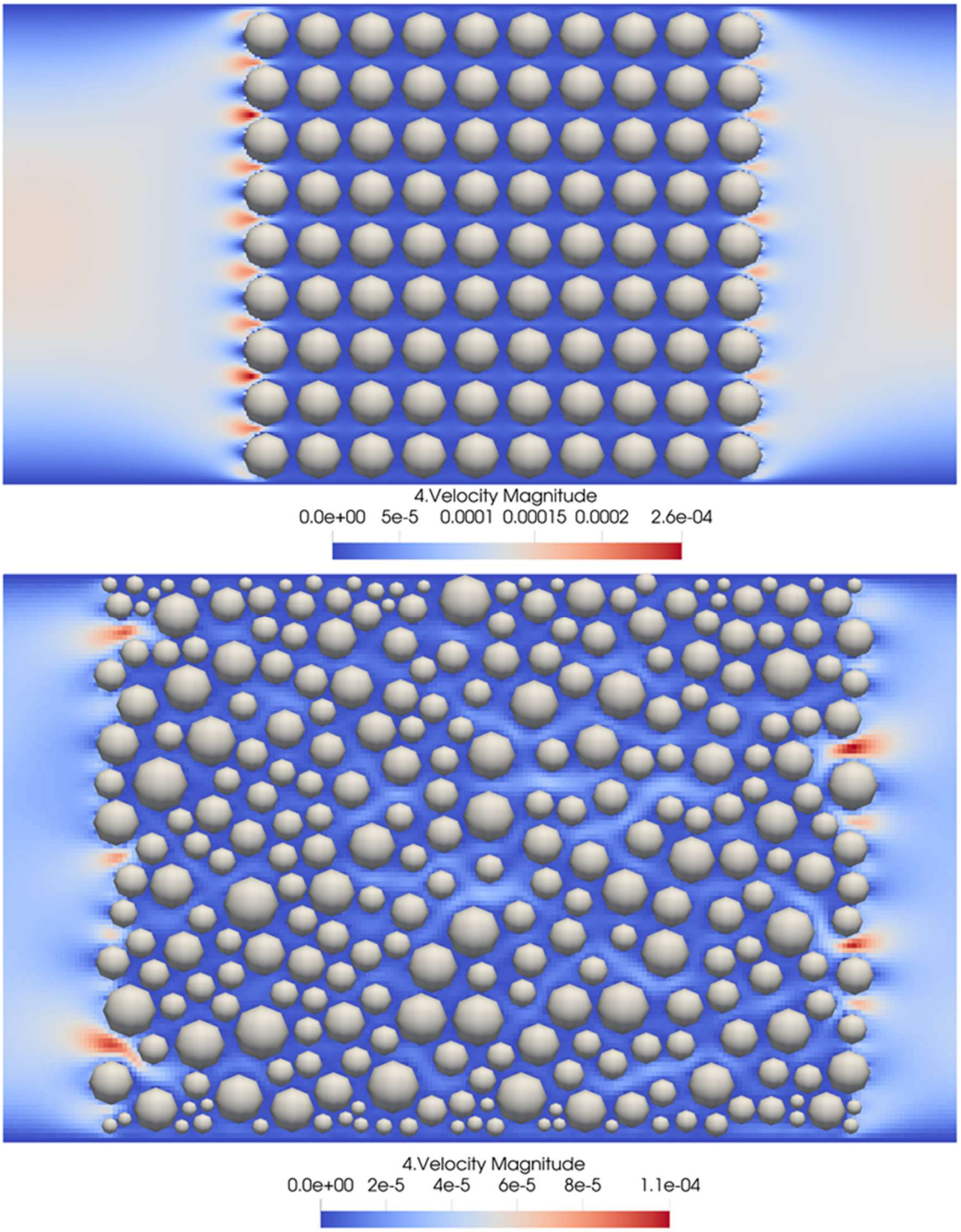

Figure 4-20: Velocity field for the discrete element packing – disks scenario.

For the uniform particle distribution, the porous structure constrains the flow evenly, inhibiting the development of preferential pathways. As a result, flow disturbances are attenuated and the velocity field within the packed region exhibits predominantly laminar behavior. The velocity vector field illustrating this behavior is illustrated in Figure 4-21.

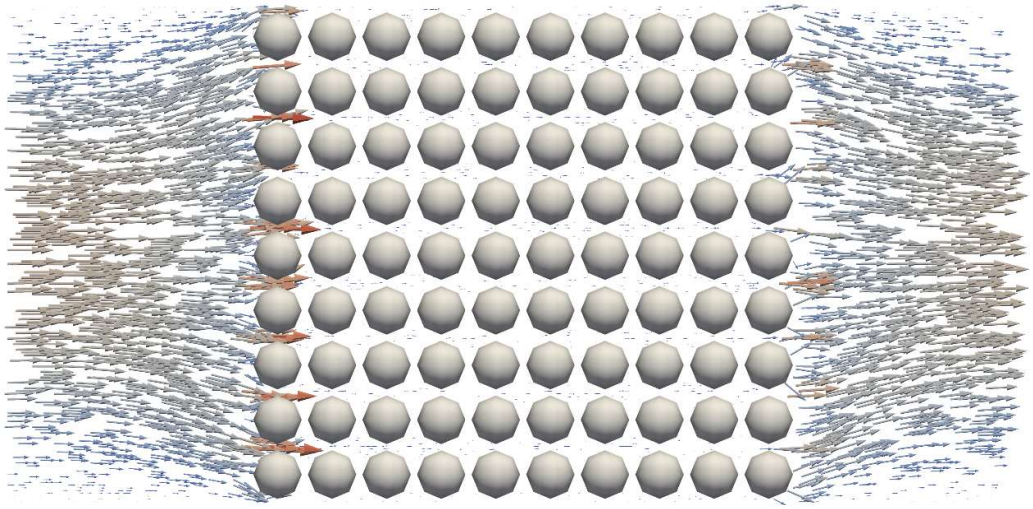

Figure 4-21: Velocity vector field for the discrete element packing – uniform disks scenario using MRT collision operator.

In contrast, the randomly distributed particle packing clearly exhibits the formation of preferential flow paths, with up to three dominant channels developing primarily through the central region of the sample. It is noteworthy that no preferential flow paths are formed near the channel walls, as smaller particles were placed in these regions to restrict flow and minimize boundary effects. The velocity vector field corresponding to the random packing scenario is illustrated in Figure 4-21.

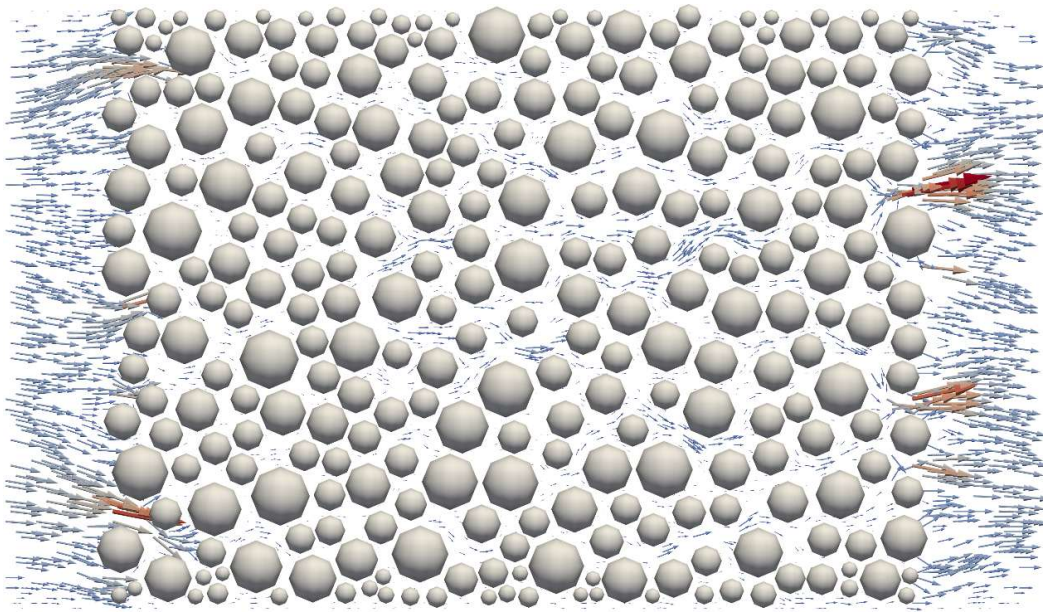

Figure 4-22: Velocity vector field for the discrete element packing – random disks scenario using MRT collision operator.

In addition to the assessment of preferential flow path formation, the agreement between the numerically computed permeability and the classical correlations proposed by Kozeny–Carman (1927) and Ergun (1952) were also evaluated. The comparison between the numerical permeability results and these analytical formulations is illustrated in Figure 4-23.

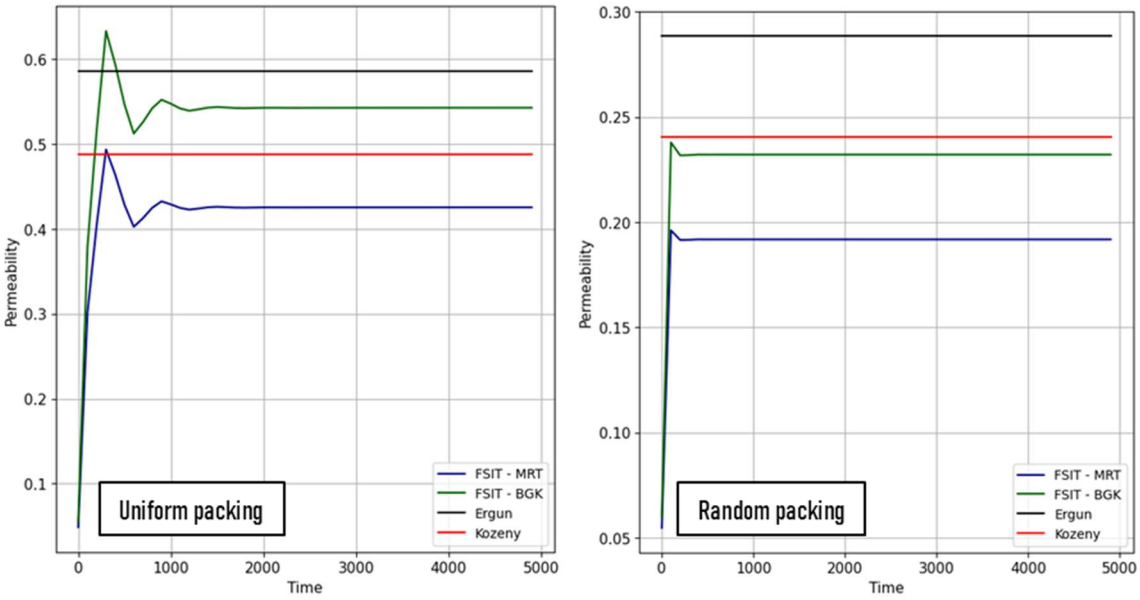

Figure 4-23: Comparison of permeability for the uniform and random disk packing with Ergun and Kozeny-Carman formulation.

For the uniform packing configuration, the permeability computed using FSIT converges to values bounded by the Ergun and Kozeny–Carman correlations. This outcome is expected, since the regular particle arrangement closely approximates the idealized pore geometry assumed in the derivation of these empirical models. Small discrepancies observed at the initial stages of the simulation are primarily associated with flow initialization effects and progressively decrease as the numerical solution reaches convergence.

In the case of random packing, the permeability predicted by FSIT is slightly lower than that obtained for the uniform arrangement, reflecting the higher tortuosity and increased pore-scale heterogeneity inherent to disordered particle distributions. Although the Ergun and Kozeny–Carman formulations were originally developed for three-dimensional porous media, they still provide a meaningful reference for evaluating the magnitude and overall trends of the numerical permeability results obtained in this study.

In addition, it is important to note that simulations were performed using two different collision operators for the fluid phase. Among them, the BGK operator yielded results that were closer to the permeability values predicted by the empirical correlations. The differences observed when using the MRT operator can be attributed to the relaxation parameters adopted,

which, in this case, were the same as those employed in the lid-driven cavity scenario. With appropriate calibration of the MRT relaxation parameters for porous flow conditions, higher accuracy in the numerical results can be achieved.

Finally, a third packing configuration composed of discrete elements was developed. In this case, realistic particle geometries were numerically reconstructed and arranged within the same domain geometry described previously. For this scenario, an accurate determination of the solid fraction is particularly important, as the particles exhibit irregular shapes. Figure 4-24 illustrates the solid fraction distribution of the realistic particles as computed by FSIT.

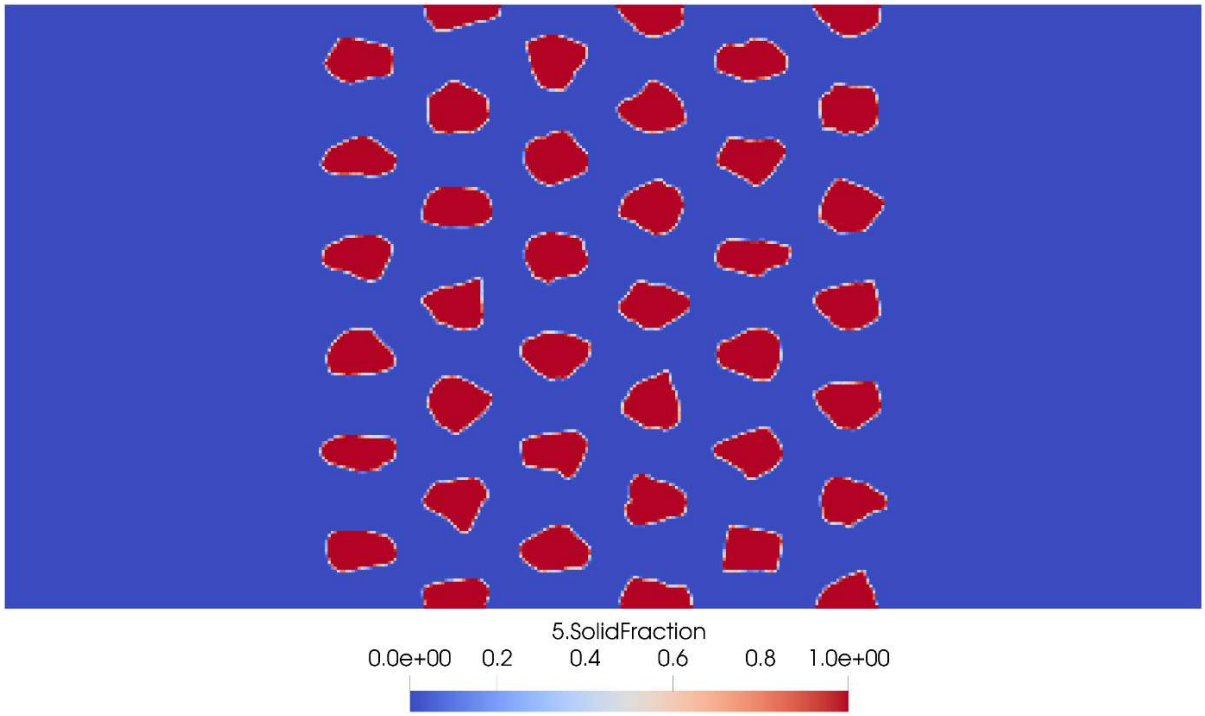

Figure 4-24: Solid fraction for the real particle packing scenario.

As illustrated, FSIT accurately identifies the lattice cells occupied by the discrete element, regardless of particle shape, and assigns solid fraction values according to the degree of interaction between the particle and the surrounding lattice cells.

Based on this solid fraction definition, the analysis was then conducted using the same boundary conditions described previously. The resulting velocity field observed for this scenario is illustrated in Figure 4-25.

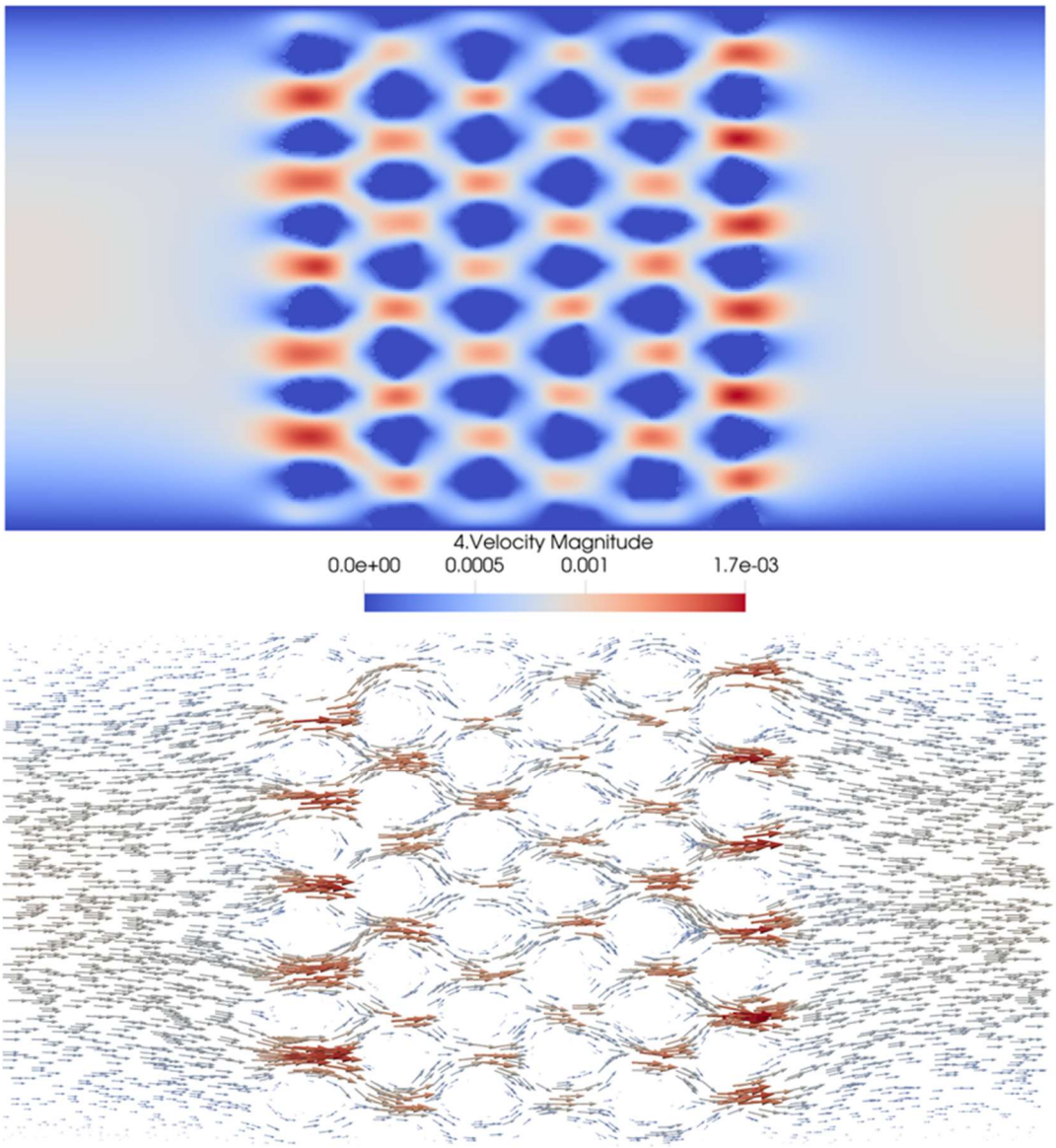

Figure 4-25: Velocity magnitude and vector field for the real particle packing scenario using MRT collision operator.

Due to high porosity of the packing, a well-defined separation of velocity magnitudes within the porous medium can be observed. Nevertheless, this scenario demonstrates the capability of FSIT to accurately capture fluid–solid interaction behavior for discrete elements with complex geometries.

In addition to the analysis of preferential flow paths, the permeability was also evaluated for this case using the same procedure adopted previously. It is important to emphasize that the empirical formulations proposed by Kozeny–Carman and Ergun were originally derived for spherical particles. Therefore, their application in this context serves primarily to provide an order-of-magnitude reference for the permeability values associated with the porosity of the medium. The comparison between the numerically computed permeability and the empirical predictions is illustrated in Figure 4-26.

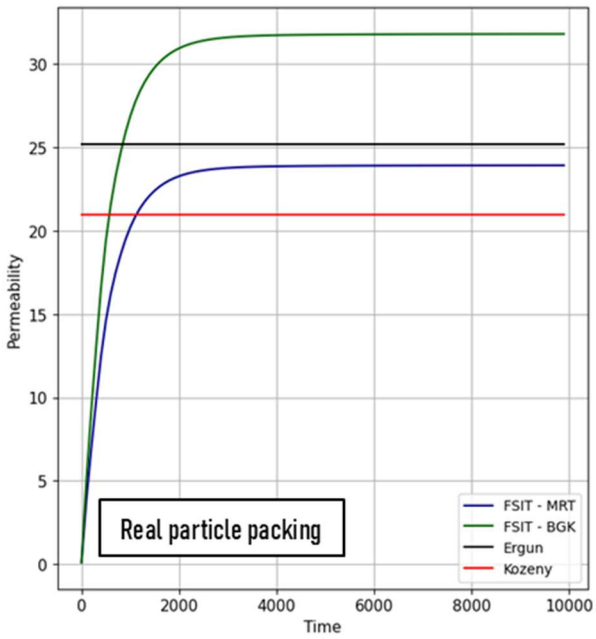

Figure 4-26: Velocity magnitude and vector field for the real particle packing scenario.

Figure 4-26 illustrates the temporal evolution of permeability for the real particle packing configuration, comparing the numerical results obtained with the FSIT framework using the BGK and MRT collision operators with the analytical estimates provided by the Ergun and Kozeny–Carman equations. The numerical permeability rapidly converges toward a steady value after an initial transient, indicating the establishment of a fully developed flow regime.

The FSIT results indicate that the MRT formulation produces permeability estimates closer to analytical predictions than the BGK model, due to improved numerical stability and reduced dissipation. Despite their limitations in two-dimensional and irregular configurations, the analytical correlations provide useful reference bounds for the numerical results.

4.6 PARTICLE SETTLING: STOKES' LAW

For the development of this scenario, a disk-shaped discrete element was positioned at the center of a vertical channel, located at approximately 75% of the channel height. To avoid interference in the numerical response caused by boundary conditions, such as the influence of lateral walls, the horizontal length of the channel (L_x) was extended. This configuration reduces confinement effects and allows a more realistic development of the flow around the particle. The geometry constructed in FSIT for this simulation scenario is illustrated in Figure 4-27.

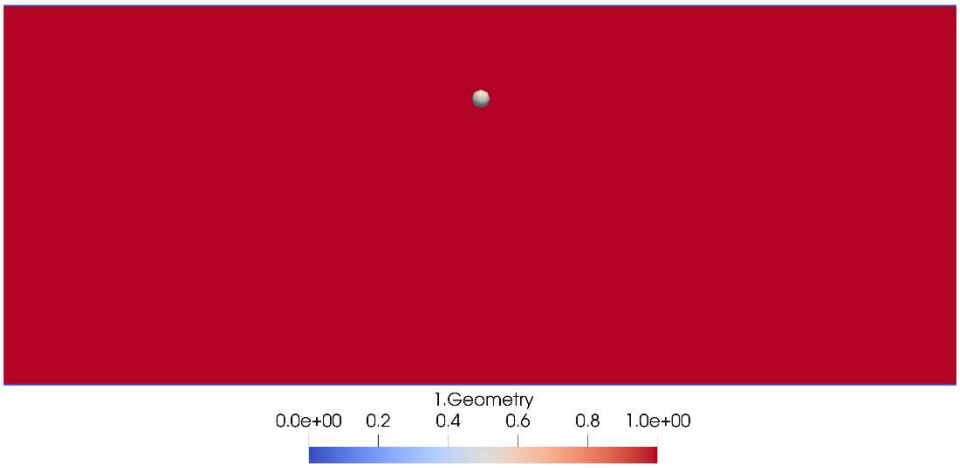

Figure 4-27: Geometry for the particle settling scenario.

The geometry presented above shows that lattice nodes were classified into fluid nodes (value 1: red color) and wall nodes (value 0: blue color). Also, discrete element is shown in white.

Once the geometry is defined, the particle is released and allowed to fall freely until kinematic equilibrium is reached. At this stage, the forces acting on the particle are balanced, and it settles with a constant terminal velocity. As discussed previously, the classical definition of terminal velocity is derived for a three-dimensional spherical particle. However, since the present simulations are performed in a two-dimensional framework, an alternative formulation based on the volume of a cylinder with unit depth was adopted. The temporal evolution of the terminal velocity obtained with FSIT is presented below and compared with both the two-dimensional and three-dimensional analytical formulations.

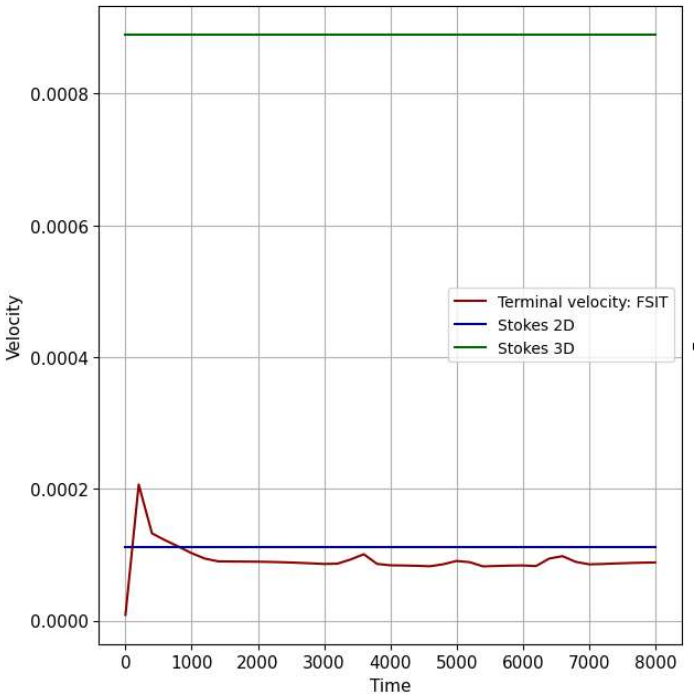

Figure 4-28: Temporal evolution of the terminal velocity.

It can be observed that the disk rapidly reaches terminal velocity, converging closely to the value predicted by the theoretical formulation for the two-dimensional case. In contrast, when the three-dimensional formulation is considered, a larger discrepancy is observed. This difference arises mainly from the assumptions under which the three-dimensional Stokes formulation was derived, particularly its dependence on spherical geometry and fully three-dimensional flow conditions. Nevertheless, this discrepancy does not invalidate the two-dimensional simulation framework adopted in this study, as will be further demonstrated through the subsequent analysis of the forces acting on the particle.

The most appropriate validation of this scenario is achieved by evaluating the force balance acting on the discrete element once terminal velocity is reached. At this condition, the hydrodynamic drag force computed by FSIT should balance the net driving force acting on the particle, defined as the difference between its weight and the buoyancy force. Accordingly, the drag force obtained numerically is compared with the theoretical force equilibrium. The temporal evolution of both the numerical drag force and the expected theoretical force, based on the adopted parameters, is illustrated below.

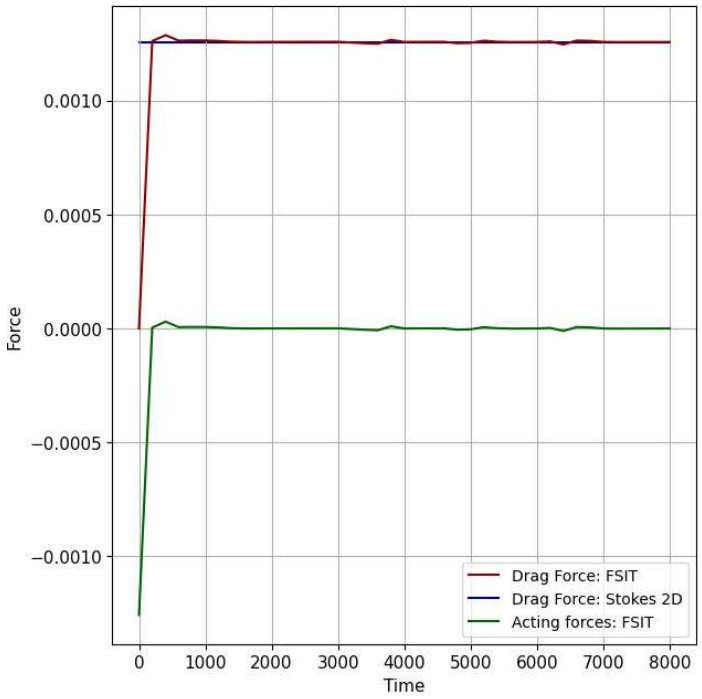

Figure 4-29: Temporal evolution of the drag force acting on the disk.

Initially, the only force acting on the particle is its own weight. However, as the particle rapidly reaches terminal velocity, the forces acting on the discrete element enter a state of kinematic equilibrium. As observed, the drag force computed by FSIT shows excellent agreement with the theoretical drag force once terminal velocity is attained. In addition, it is

noteworthy that the net force acting on the particle approaches zero, as expected under equilibrium conditions. These results demonstrate that the fluid–solid momentum exchange is being correctly captured by FSIT, highlighting the robustness of the coupling strategy and enabling a wide range of fluid–solid interaction simulations.

To ensure the validity of Stokes' law, the Reynolds number of the system must remain very low ($Re \ll 1$). Under this condition, the flow around the disk is laminar and characterized by smooth, continuous streamlines, without the formation of vortices in the wake of the particle. The streamlines around the discrete element at the instant when terminal velocity is reached are illustrated below, together with the temporal evolution of the Reynolds number throughout the simulation.

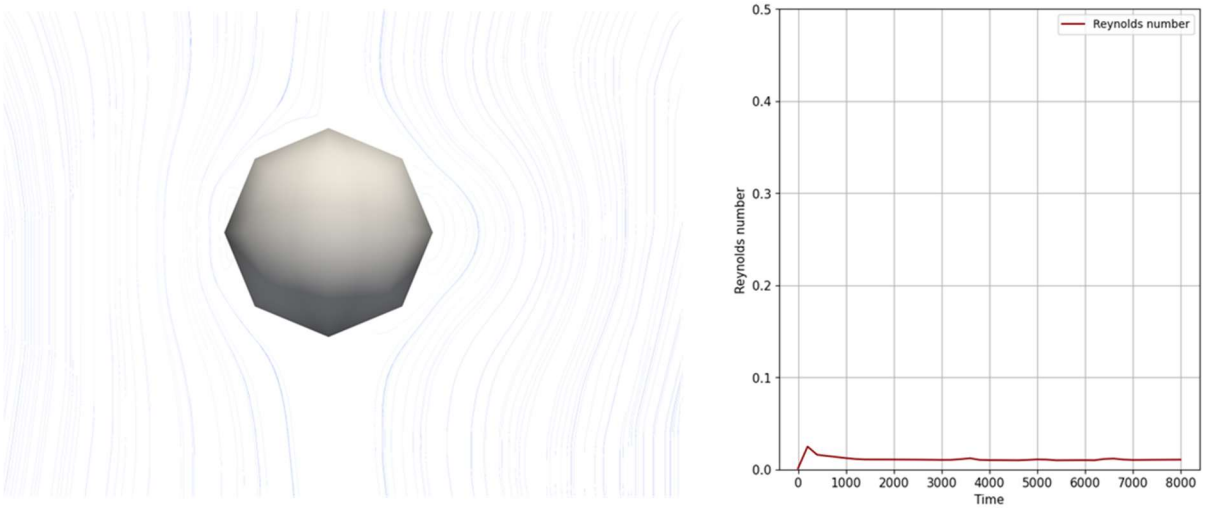

Figure 4-30: Streamlines observed when the body reaches terminal velocity and temporal evolution of Reynolds number.

It can be observed that the Reynolds number remains extremely low throughout the entire simulation, which ensures that the conditions required for the applicability of Stokes' law are satisfied. In addition, the streamlines around the discrete element are smooth and continuous, with no evidence of vortex formation in the wake of the particle, as expected for this flow regime. These observations further confirm the capability of FSIT to accurately capture fluid–solid interaction phenomena under low-Reynolds-number conditions.

CHAPTER V

5 CONCLUSIONS

In this research, a computational framework was developed with the objective of performing two-dimensional simulations of fluid–solid interaction problems. The code was implemented in C++ and combines the Lattice Boltzmann Method (LBM) for fluid flow, the Discrete Element Method (DEM) for solid mechanics, and hydrodynamic coupling through the Immersed Moving Boundary (IMB) technique. The resulting tool was named FSIT: Fluid–Solid Interaction Toolkit.

Fluid flow simulations within FSIT are carried out using the Lattice Boltzmann Method. The framework supports two collision operators: the single-relaxation-time BGK operator, commonly adopted due to its simplicity and computational efficiency, and the multiple-relaxation-time (MRT) operator, which is employed to improve numerical accuracy and stability, particularly in simulations involving higher Reynolds numbers where the BGK formulation reaches its limitations. In FSIT, the LBM is implemented for two-dimensional domains discretized using a D2Q9 lattice arrangement.

The solid phase is modeled using the Discrete Element Method (DEM), which enables the simulation of particle–particle and particle–boundary interactions through a linear elastic constitutive contact law. Particle kinematics, including translational and rotational motion, are updated using the Leapfrog time integration scheme, which provides second-order accuracy and good stability properties for DEM simulations.

Hydrodynamic coupling between the LBM and DEM formulations is achieved using the Immersed Moving Boundary (IMB) method proposed by Noble and Torczynski (1998). This approach was selected due to its relative simplicity, conservation properties, and suitability for parallel implementation. Through the IMB formulation, hydrodynamic forces and torques acting on discrete elements are computed, enabling consistent momentum exchange between the fluid and solid phases. This coupling strategy significantly expands the applicability of FSIT to a wide range of geotechnical and multiphase flow problems.

The validation and assessment of FSIT were conducted through a series of classical benchmark problems with well-established analytical or empirical solutions. These included

Poiseuille flow, lid-driven cavity flow, Taylor–Green vortex decay, flow around circular and realistic particles, porous media flow through discrete element packings, and particle settling governed by Stokes' law.

The Poiseuille flow scenario was investigated using two distinct approaches: velocity boundary conditions based on the formulation of Zou and He (1997) and pressure-driven flow generated via the forcing scheme proposed by Guo et al. (2002). In both cases, the expected parabolic velocity profile was successfully reproduced at multiple sections along the channel. While minor deviations were observed when using velocity boundary conditions, the forcing-based approach produced highly accurate results across the entire domain. Nevertheless, both methodologies were shown to be capable of capturing the essential physics of Poiseuille flow.

The lid-driven cavity flow problem was simulated using both BGK and MRT collision operators, and the results were compared against reference data reported by Ghia et al. (1982). As expected for simulations conducted at a Reynolds number of 100, the BGK operator exhibited reduced accuracy due to its single relaxation parameter. In contrast, the MRT formulation demonstrated excellent agreement with the reference solution, highlighting its superior performance for moderate-to-high Reynolds number flows. The Taylor–Green vortex case further confirmed the ability of FSIT to accurately reproduce vortex formation and decay in incompressible flows.

Fluid–solid interaction capabilities were evaluated through simulations of flow around circular and realistic particles. For circular particles, simulations were performed over a range of Reynolds numbers from 5 to 100, and the numerical flow patterns showed strong qualitative agreement with experimental observations reported by Taneda (1956). Additional simulations at Reynolds numbers up to 500 were conducted using the MRT collision operator, demonstrating the robustness of FSIT in capturing highly unsteady and turbulent wake dynamics. The integration of numerically reconstructed realistic particles further expanded the range of applications supported by FSIT.

Porous media simulations were carried out to investigate preferential flow paths and to compute numerical permeability values. Three packing configurations were considered: uniform disk packing, random disk packing, and random packing of realistic particle geometries. The numerical permeabilities obtained using FSIT showed strong agreement with classical empirical correlations proposed by Kozeny–Carman (1927) and Ergun (1952), providing confidence in the framework's ability to capture pore-scale flow behavior.

Differences between BGK- and MRT-based simulations were attributed to relaxation parameter selection, indicating that further calibration can improve accuracy in porous flow applications.

Finally, particle settling simulations were performed to assess the hydrodynamic forces acting on a single circular particle immersed in a fluid. This scenario was designed to evaluate the numerical representation of Stokes' law under two-dimensional conditions. Validation was achieved through force balance at terminal velocity, demonstrating that FSIT is capable of accurately reproducing drag forces acting on submerged particles.

Overall, FSIT proved to be a robust and versatile computational tool for simulating fluid–solid interaction phenomena in two-dimensional domains. Its ability to handle idealized and realistic particle geometries, reproduce classical benchmark solutions, and capture complex flow regimes highlights its strong potential for geotechnical applications at the mesoscale. As such, FSIT represents a valuable numerical platform for engineers and researchers seeking to investigate fluid–solid interaction problems relevant to laboratory testing and engineering practice.

5.1 SUGGESTIONS FOR FUTURE RESEARCH

Based on the results obtained in this research, the following directions are suggested for future studies aimed at extending and enhancing the capabilities and applications of the FSIT framework:

Extend the current two-dimensional FSIT formulation to three-dimensional simulations, enabling the representation of realistic particle interactions and flow patterns, and broadening the applicability of the framework to practical geotechnical and hydraulic engineering problems;

Apply FSIT to the simulation of additional laboratory tests of interest in geotechnical engineering, such as drained and undrained triaxial compression tests, direct shear tests, and direct simple shear (DSS) tests;

Investigate the parameters that influence soil liquefaction phenomena, particularly in loose sands and silts, by means of hydromechanical coupling;

Evaluate the parameters governing sediment generation, transport, and deposition, allowing FSIT to be applied to studies related to erosive processes, submerged slope stability, and particulate flow dynamics.

REFERENCES

AIDUN, C. K., LU, Y., DING, E-J. (2000). Direct Analysis of Particulate Suspensions with Inertia Using the Discrete Boltzmann Equation. Journal of Fluid Mechanics, 373, 287-311

BEEMAN, D. (1976). Some Multistep Methods for Use in Molecular Dynamics Calculations. Journal of Computational Physics, 20, 130-139.

BHATNAGAR, P. L., GROSS, E. P. & KROOK, M. (1954). A Model for Collision Processes in Gases. I. Small Amplitude Processes in Charged and Neutral One-Component Systems. Physical Review, 94, 511-525.

CUNDALL, P. A. (1971). A Computer Model for Simulating Progressive Large Scale Movements in Blocky Rock Systems. Proceedings of the Symposium of the International Society for Rock Mechanics, Society for Rock Mechanics (ISRM), France, II-8.

CUNDALL, P. A. & STRACK, O. D. L. (1979). A Discrete Numerical Model for Granular Assemblies, Géotechnique, 29, 47-65.

ERGUN, S. (1952). Fluid Flow Through Packed Columns. Chemical Engineering Progress, 48, 89-94.

EL SHAMY, U. & ABDELHAMID, Y. (2014). Pore-scale Modeling of Surface Erosion in a Particle Bed. International Journal for Numerical and Analytical Methods in Geomechanics, 38, 142-166.

GALINDO-TORRES, S. A. (2013). A Coupled Discrete Element Lattice Boltzmann Method for the Simulation of Fluid-Solid Interaction with Particles of General Shapes. Computer Methods in Applied Mathematics and Engineering, 265, 107-119.

GARDNER, M. & SITAR, N. (2019). Modelling of Dynamic Rock-Fluid Interaction Using Coupled 3-D Discrete Element and Lattice Boltzmann Methods. Rock Mechanics and Rock Engineering, 1-10.

GHIA, U., GHIA, N, SHIN, C. T. (1982). High-Re Solutions for Incompressible Flow Using the Navier-Stokes Equations and a Multigrid Method. Journal of Computational Physics, 48, 387-411.

GUO, Z., ZHENG, C. & SHI, B. (2002). Discrete Lattice Effects on the Forcing Term in the Lattice Boltzmann Method. Physical Review E, 65.

KOZENY, J. (1927). Uber Kapilare Leitung der Wasser in Boden. Sitzungzber, 136, 271-306.

LADD, A. J. C. (1994). Numerical Simulations of Particulate Suspensions via a Discretized Boltzmann Equation. Part 1. Theoretical Foundation. Journal of Fluid Mechanics, 271, 284-309.

LADD, A. J. C. (1994). Numerical Simulations of Particulate Suspensions via a Discretized Boltzmann Equation. Part 2. Numerical Results. Journal of Fluid Mechanics, 271, 311-339.

MORFA, C. R., CORTÉS, L. A., FARIAS, M. M., MORALES, I. P. P., VALERA, R. R., OÑATE, E. (2017). Systemic characterization and evaluation of particle packings as initial sets for discrete element simulations. Computational Particle Me-chanics 5:319–334.

NOBLE, D. R. & TORCZYNSKI, J. R. (1998). A Lattice-Boltzmann Method for Partially Saturated Computational Cells. International Journal of Modern Physics, 8, 1189-1201.

OCAMPO-GÓMES, D. A. (2013). Modelagem de Problemas de Fluxo na Escala Granular Usando o Método Lattice Boltzmann. Dissertação de Mestrado, Departamento de Engenharia Civil, Universidade de Brasília, 118 p.

OWEN, D. R. J., LEONARDI, C. R. & FENG, Y. T. (2011). An Efficient Framework for Fluid-Structure Interaction Using the Lattice Boltzmann Method and Immersed Moving Boundaries. International Journal for Numerical Methods in Engineering, 87, 66-95.

QIAN, Y. H., D'HUMIÈRES, D. & LALLEMAND, P. (1992). Lattice BGK Models for the Navier-Stokes Equation. Europhysics Letter, 17, 479-484.

RECAREY, C., PÉREZ, I., ROSELLÓ, R., MUNIZ, M., HERNÁNDEZ, E., GIRALDO, R., OÑATE, E. (2019). Advances in particle packing algorithms for generating the medium in the Discrete Element Method. Computer methods in applied mechanics and engineering. 345: 336-362.

SATO, M. & KOBAYASHI, T. (2012). A Fundamental Study of the Flow Past a Circular Cylinder Using Abaqus/CFD. Simulia Community Conference, 15 p.

TANEDA, S. (1956). Experimental Investigations of the Wakes Behind Cylinders and Plates at Low Reynolds Numbers. J. Phys. Soc. Japan 11.

ZOU, Q. & HE, X. (1997). On Pressure and Velocity Boundary Conditions for the Lattice Boltzmann BGK Model. Physics of Fluids, 9, 1591-1598.

ZUBELDIA, E. H. (2017). Aplicação do Método *Smoothed Particle Hydrodynamics* ao Estudo de Erosão Superficial de Solos. Tese de Doutorado, Departamento de Engenharia Civil, Universidade de Brasília, 106 p.

ZULUAGA, R. A. G. (2016). Relação Entre Características Microestruturais e o Comportamento Macroscópico de Solos Granulares. Tese de Doutorado, Departamento de Engenharia Civil, Universidade de Brasília, 171 p.

Appendix

APPENDIX A: FSIT SOURCE CODE

This appendix contains the complete set of headers (.h) and source (.cpp) files that compose the Fluid–Solid Interaction Toolkit (FSIT). The files are presented to ensure transparency, reproducibility, and to allow full inspection of the numerical implementation.

LBM FILES: Lattice.h

```
#ifndef LATTICE_H
#define LATTICE_H

#include "maths/Math.h"

class Lattice {
    public:
        EIGEN_MAKE_ALIGNED_OPERATOR_NEW
        std::vector<int> neighbor_node;
        double sound_speed         = 0;
        double density_bc          = 0;
        Vector3r velocity_bc       = Vector3r::Zero();
        Matrix9r m                 = Matrix9r::Zero();
        Matrix9r m_inv             = Matrix9r::Zero();
        Matrix9r relaxation_matrix = Matrix9r::Zero();
        Vector9r f_tmp             = Vector9r::Zero();
        Vector9r f                 = Vector9r::Zero();
        const int number_of_nodes  = 9;

        const std::vector<double> node_weight = { 4.0/9.0, 1.0/9.0, 1.0/9.0,
1.0/9.0, 1.0/9.0, 1.0/36.0, 1.0/36.0, 1.0/36.0, 1.0/36.0 };

        const std::vector<int> opposite_node  = {0, 3, 4, 1, 2, 7, 8, 5, 6};

        const std::vector<Vector3r, Eigen::aligned_allocator<Vector3r>>
discrete_velocity = {{0,0,0}, {1, 0,0}, {0,1,0}, {-1,0,0}, {0,-1,0}, {1, 1,0},
{-1,1,0}, {-1,-1,0}, {1,-1,0}};
};

#endif
```

LBM FILES: Cell.h

```
#ifndef CELL_H
#define CELL_H

#include "Lattice.h"
#include "property/State.h"
#include "property/Material.h"

class Scene;

class Cell {
    public:
    Cell(int _id, std::shared_ptr<Lattice> _lattice, std::shared_ptr<Material>
_material, std::shared_ptr<State> _state) : id(_id), lattice(_lattice),
material(_material), state(_state), is_solid(false), is_wall(false){}
    ~Cell(){}

    int  id;
    bool is_solid;
    bool is_wall;
    double solid_fraction = 0.0;
    polygon grid;

    std::shared_ptr<Lattice> lattice;
    std::shared_ptr<Material> material;
    std::shared_ptr<State> state;

    double CalculateEqFunction(double _density, Vector3r _velocity, int k);
    void CalculateDensityAndVelocity();
    void set_neighbor_node();
    void set_initial_condition();
    void set_mrt_parameters();
};
#endif
```

LBM FILES: Cell.cpp

```
#include "Cell.h"
#include "scene/Scene.h"

double Cell::CalculateEqFunction(double rho, Vector3r u, int k) {
    const Scene&  S  = Scene::get_Scene();
    const double  c  = S.latticeSpeed;                // usually 1.0
    const double  cs2 = (c*c)/3.0;                    // D2Q9
    const double  inv_cs2 = 1.0 / cs2;
    const double  inv_cs4 = inv_cs2 * inv_cs2;
```

```
    const double cu = lattice->discrete_velocity[k].dot(u);
    const double u2 = u.squaredNorm();

    return lattice->node_weight[k] * rho *
           (1.0 + inv_cs2*cu + 0.5*inv_cs4*cu*cu - 0.5*inv_cs2*u2);
}

void Cell::CalculateDensityAndVelocity() {
    const Scene&  S  = Scene::get_Scene();
    if (is_solid || is_wall || solid_fraction >= 0.99) {
        material->density = 0.0;
        state->vel.setZero();
        return;
    }

    double   rho = 0.0;
    Vector3r mom = Vector3r::Zero();

    for (int k = 0; k < lattice->number_of_nodes; ++k) {
        const double fk = lattice->f[k];

        if (!std::isfinite(fk)) {
            std::cout << "NaN/Inf in f at cell " << id << " pos = " << state-
>pos.transpose() << " k = " << k << " fk = " << fk << "\n";
            std::abort();
        }

        rho += fk;
        mom += lattice->discrete_velocity[k] * fk;
    }

    material->density = rho;

    if (rho > 1e-14) {
        Vector3r mom_eff = mom + 0.5 * S.GUO_fluid_forcing * S.dt_lbm;
        state->vel = mom_eff / rho;
        state->vel[2] = 0.0;
    } else {
        state->vel.setZero();
    }

    ASSERT_MSG(std::isfinite(material->density), "NaN density");
    ASSERT_MSG(std::isfinite(state->vel[0]) && std::isfinite(state->vel[1]),
"NaN velocity");
}


void Cell::set_neighbor_node(){
```

```
    const Scene& S = Scene::get_Scene();

    lattice->neighbor_node.clear();
    lattice->neighbor_node.reserve(lattice->number_of_nodes);

    const int Nx = static_cast<int>(S.model_max_corner[0]);
    const int Ny = static_cast<int>(S.model_max_corner[1]);

    for (int k = 0; k < lattice->number_of_nodes; ++k){
        Vector3r aux = state->pos + lattice->discrete_velocity[k];

        int ax = static_cast<int>(aux[0]);
        int ay = static_cast<int>(aux[1]);

        if (ax < 0) ax = Nx - 1;
        if (ay < 0) ay = Ny - 1;
        if (ax >= Nx) ax = 0;
        if (ay >= Ny) ay = 0;

        lattice->neighbor_node.push_back(ax + ay * Nx);
    }
}

void Cell::set_initial_condition(){
    const Scene& S = Scene::get_Scene();
    for (int k = 0; k < lattice->number_of_nodes; ++k)
        lattice->f[k] = CalculateEqFunction(S.initial_density,
S.initial_velocity, k);
    CalculateDensityAndVelocity();
}

void Cell::set_mrt_parameters(){
    Scene& S = Scene::get_Scene();

    lattice->m << 1,1,1,1,1,1,1,1,1,
                 -4,-1,-1,-1,-1,2,2,2,2,
                  4,-2,-2,-2,-2,1,1,1,1,
                  0,1,0,-1,0,1,-1,-1,1,
                  0,-2,0,2,0,1,-1,-1,1,
                  0,0,1,0,-1,1,1,-1,-1,
                  0,0,-2,0,2,1,1,-1,-1,
                  0,1,-1,1,-1,0,0,0,0,
                  0,0,0,0,0,1,-1,1,-1;
    lattice->m_inv = lattice->m.inverse();
    lattice->relaxation_matrix = S.relaxation_parameters.asDiagonal();
}
```

DEM FILES: Body.h

```
#ifndef BODY_H
#define BODY_H

#include "maths/Math.h"
#include "property/Material.h"
#include "property/State.h"
#include "property/Shape.h"
#include "dem/Interaction.h"
#include "lbm/Cell.h"

class Interaction;

class Body {
public:
    EIGEN_MAKE_ALIGNED_OPERATOR_NEW
    Body(int _id, std::shared_ptr<Shape> _shape, std::shared_ptr<State>
_state, std::shared_ptr<Material> _material) : id(_id), shape(_shape),
state(_state), material(_material){}

    bool CheckInteraction(int _body_id);

    //Body variables:
    int    id;

    //Smart pointers:
    std::vector<std::weak_ptr<Interaction>> inter;
    std::vector<std::shared_ptr<Cell>> fluid_interactions;
    std::shared_ptr<State> state;
    std::shared_ptr<Shape> shape;
    std::shared_ptr<Material> material;

    Vector3r blockedDOFs = { 1, 1, 1 };
    double blockedMDOFs  = 1;

    ~Body() {};
};

#endif //BODY_H
```

DEM FILES: Body.cpp

```
#include "Body.h"

bool Body::CheckInteraction(int _body_id) {
    for (auto& I : inter) {
```

```
        auto Il = I.lock();

        auto Ilb1 = Il->body1.lock();
        auto Ilb2 = Il->body2.lock();

        if ((Ilb1->id == id && Ilb2->id == _body_id) || (Ilb2->id == id &&
Ilb1->id == _body_id)) {
            return true;
        }
    }
    return false;
}
```

DEM FILES: Interaction.h

```
#ifndef INTERACTION_H
#define INTERACTION_H

#define _USE_MATH_DEFINES
#include "maths/Math.h"
#include "dem/Body.h"
#include "scene/Scene.h"

class Body;

class Interaction {
public:
    EIGEN_MAKE_ALIGNED_OPERATOR_NEW
    Interaction(std::weak_ptr<Body> _body1, std::weak_ptr<Body> _body2) :
body1(_body1), body2(_body2) {};

    //Methods:
    bool CheckContact();
    void CalculateUnitVectorandContact();
    void CalculateForceAndShearIncrements();
    void ApplyFrictionLaw();

    //Smart pointers:
    std::weak_ptr<Body> body1;
    std::weak_ptr<Body> body2;

    //Variables:
    Vector3r  unit_normal_vector  = Vector3r::Zero();
    Vector3r  unit_shear_vector   = Vector3r::Zero();
    Vector3r  contact     = Vector3r::Zero();
    double normal_force = 0.0;
    double shear_force  = 0.0;
```

```
};

#endif
```

DEM FILES: Interaction.cpp

```
#include "Interaction.h"

bool Interaction::CheckContact() {
    auto b1 = body1.lock();
    auto b2 = body2.lock();
    if ((b2->state->pos - b1->state->pos).norm() < b1->shape->radius + b2-
>shape->radius) {
        return true;
    }
    return false;
}

void Interaction::CalculateUnitVectorandContact() {
    auto b1 = body1.lock();
    auto b2 = body2.lock();

    unit_normal_vector = (b2->state->pos - b1->state->pos) / (b2->state->pos -
b1->state->pos).norm();
    unit_shear_vector = { unit_normal_vector[1], -unit_normal_vector[0], 0.0
};

    // Contact point on surface of body1
    contact = b1->state->pos + unit_normal_vector * b1->shape->radius;
}

void Interaction::CalculateForceAndShearIncrements() {
    const Scene& S = S.get_Scene();
    auto b1 = body1.lock();
    auto b2 = body2.lock();

    // Calculate overlap
    double distance = (b2->state->pos - b1->state->pos).norm();
    double overlap = b1->shape->radius + b2->shape->radius - distance;

    // Relative velocity (linear + rotational contributions)
    Vector3r relVel = (b1->state->vel - b2->state->vel)
                    - (b1->state->rotVel * b1->shape->radius - b2->state-
>rotVel * b2->shape->radius) * unit_shear_vector;

    double shear_increment = (relVel.dot(unit_shear_vector)) * S.dt_dem;
```

```
    // Normal force from current overlap
    normal_force = S.normal_stiffness * overlap;
    // Shear force accumulated incrementally
    shear_force  += S.shear_stiffness  * shear_increment;
    ASSERT(S.normal_stiffness > 0 && S.shear_stiffness > 0);
}

void Interaction::ApplyFrictionLaw() {
    const Scene& S = S.get_Scene();
    double maxShearForce = normal_force * tan(S.friction_angle * M_PI /
180.0);
    if (abs(shear_force) > maxShearForce) {
        if (shear_force > 0)  shear_force =  maxShearForce;
        if (shear_force < 0)  shear_force = -maxShearForce;
    }
}
```

PROPERTIES FILES: State.h

```
#ifndef STATE_H
#define STATE_H

#include "maths/Math.h"

class State {
    public:
    EIGEN_MAKE_ALIGNED_OPERATOR_NEW
    Vector3r prevPos   = Vector3r::Zero();
    Vector3r pos       = Vector3r::Zero();
    Vector3r vel       = Vector3r::Zero();
    Vector3r force     = Vector3r::Zero();
    Vector3r permForce = Vector3r::Zero();
    Vector3r hydro_force = Vector3r::Zero();  // Hydrodynamic force from fluid

    double moment = 0.0;
    double rotVel = 0.0;
    double rot    = 0.0;
    double hydro_torque = 0.0;  // Hydrodynamic torque from fluid

};

#endif
```

PROPERTIES FILES: Material.h

```
#ifndef MATERIAL_H
#define MATERIAL_H

class Material {
    public:

    int id = -1;
    double density = 0;
    double mass = 0;       // Computed from density and shape
    double inertia = 0;
};

class Fluid : public Material {
    public:

    double kinViscosity = 0;
};

#endif
```

PROPERTIES FILES: Shape.h

```
#ifndef SHAPE_H
#define SHAPE_H

#include "maths/Math.h"

class Shape {
    public:
    void generate_particle(std::string _file_name);

    std::vector<Vector3r, Eigen::aligned_allocator<Vector3r>> coordinates;
    std::vector<Vector3r, Eigen::aligned_allocator<Vector3r>> conectivities;
    double radius;
    double inertia_moment;
    polygon cloud;
};

#endif
```

PROPERTIES FILES: Shape.cpp

```
#include "Shape.h"

void Shape::generate_particle(std::string _file_name){
    //Atribuição dos pontos das coordenadas
    std::ifstream file(_file_name);
    double x,y,z;
    while(file >> x >> y >> z){
        coordinates.emplace_back(Vector3r(x,y,z));
    }

    for(int i = 0; i < coordinates.size(); ++i){
        cloud.outer().push_back(point(coordinates[i][0],coordinates[i][1]));
        if(i == coordinates.size()-1){
            cloud.outer().push_back(point(coordinates[0][0],coordinates[0][1])
);
        }
    }

    //Geração das conectividades
    for(int i = 0; i < coordinates.size(); ++i){
        if(i == coordinates.size()-1){
            conectivities.emplace_back(Vector3r(coordinates.size()-1,0,0));
        } else {
            conectivities.emplace_back(Vector3r(i,i+1,0));
        }
    }
    boost::geometry::correct(cloud);
    file.close();

}
```

ENGINE FILES: Engine.h

```
#ifndef ENGINE_H
#define ENGINE_H

#include "maths/Math.h"
class Scene;

class Engine {
    public:
        virtual void action() = 0;
};

class ZouAndHeBC : public Engine {
```

```
    public:
        virtual void action() override;
};

class FluidCollision : public Engine {
    public:
        virtual void action() override;
};

class FluidStreaming : public Engine {
    public:
        virtual void action() override;
};

class ApplyFluidForcing : public Engine {
    public:
        virtual void action() override;
};

class FluidParticleForce : public Engine {
    public:
        virtual void action() override;
};

class ImbBoundary : public Engine {
    public:
        virtual void action() override;
};

class LatticeSearch : public Engine {
    public:
        virtual void action() override;
};

class NeighborSearch : public Engine {
    public:
    virtual void action() override;
};

class ContactResolution : public Engine {
    public:
    virtual void action() override;
};

class InteractionLoop : public Engine {
    public:
    virtual void action() override;
};
```

```
class BodyLoop : public Engine {
    public:
    virtual void action() override;
};

class Integrator : public Engine {
    public:
    virtual void action() override;
};

class UpdateContact : public Engine {
    public:
    virtual void action() override;
};

#endif
```

ENGINE FILES: Engine.cpp

```
#include "Engine.h"
#include "scene/Scene.h"

void ZouAndHeBC::action() {
    Scene& S = Scene::get_Scene();

    const double div = 1.0 / 6.0;
    const double aux = 2.0 / 3.0;

    const int Nx = static_cast<int>(S.model_max_corner[0]);
    const int Ny = static_cast<int>(S.model_max_corner[1]);

    // ---------------- Velocity BCs ----------------
    switch (S.apply_velocity_bc) {
        // case 1: top (j = Ny-1)
        case 1: {
            for (int i = 0; i < S.model_max_corner[0]; ++i) {
                int id = S.CalculateCellId(i, S.model_max_corner[1] - 1);
                auto& C = S.cells[id];
                if (C->is_solid || C->is_wall) continue;

                const double ux = C->lattice->velocity_bc[0];
                const double uy = 0.0;

                // Reconstruct rho using known populations (Zou & He)
                double rho = (C->lattice->f[0] + C->lattice->f[1] + C-
>lattice->f[3]
```

```
                            + 2.0 * (C->lattice->f[2] + C->lattice->f[5] + C-
>lattice->f[6]))
                            / (1.0 - uy);

                // Unknown distributions at north boundary: f4,f7,f8
                C->lattice->f[4] = C->lattice->f[2];
                C->lattice->f[8] = C->lattice->f[6]
                                 + 0.5 * (C->lattice->f[3] - C->lattice->f[1])
                                 + (1.0 / 6.0) * rho * ux;
                C->lattice->f[7] = C->lattice->f[5]
                                 + 0.5 * (C->lattice->f[1] - C->lattice->f[3])
                                 - (1.0 / 6.0) * rho * ux;

                C->material->density = rho;
                C->state->vel = Vector3r(ux, uy, 0.0);
            }
        } break;

        // case 2: right (i = Nx-1)
        case 2: {
            int i = Nx - 1;
            for (int j = 1; j <= Ny - 2; ++j) {
                int id = S.CalculateCellId(i, j);
                auto& C = S.cells[id];
                if (C->is_solid || C->is_wall) continue;

                double rho = (C->lattice->f[0] + C->lattice->f[2] + C-
>lattice->f[4]
                            + 2.0*(C->lattice->f[1] + C->lattice->f[5] + C-
>lattice->f[8]))
                            / (1.0 + C->lattice->velocity_bc[0]);

                C->lattice->f[3] = C->lattice->f[1] - aux * rho * C->lattice-
>velocity_bc[0];
                C->lattice->f[7] = C->lattice->f[5] - div * rho * C->lattice-
>velocity_bc[0]
                                + 0.5 * (C->lattice->f[2] - C->lattice-
>f[4]);
                C->lattice->f[6] = C->lattice->f[8] - div * rho * C->lattice-
>velocity_bc[0]
                                + 0.5 * (C->lattice->f[4] - C->lattice-
>f[2]);
                C->CalculateDensityAndVelocity();
            }
        } break;

        // case 3: bottom (j = 0)
        case 3: {
            int j = 0;
```

```
            for (int i = 1; i <= Nx - 2; ++i) {
                int id = S.CalculateCellId(i, j);
                auto& C = S.cells[id];
                if (C->is_solid || C->is_wall) continue;

                double rho = (C->lattice->f[0] + C->lattice->f[1] + C-
>lattice->f[3]
                              + 2.0*(C->lattice->f[4] + C->lattice->f[7] + C-
>lattice->f[8]))
                              / (1.0 - C->lattice->velocity_bc[1]);

                C->lattice->f[2] = C->lattice->f[4] + aux * rho * C->lattice-
>velocity_bc[1];
                C->lattice->f[5] = C->lattice->f[7] + div * rho * C->lattice-
>velocity_bc[1]
                                   - 0.5 * (C->lattice->f[1] - C->lattice-
>f[3]);
                C->lattice->f[6] = C->lattice->f[8] + div * rho * C->lattice-
>velocity_bc[1]
                                   - 0.5 * (C->lattice->f[3] - C->lattice-
>f[1]);
                C->CalculateDensityAndVelocity();
            }
        } break;

        // case 4: left (i = 0)  <-- Poiseuille inlet (ux)
        case 4: {
            int i = 0;
            for (int j = 1; j <= Ny - 2; ++j) {
                int id = S.CalculateCellId(i, j);
                auto& C = S.cells[id];
                if (C->is_solid || C->is_wall) continue;

                double rho = (C->lattice->f[0] + C->lattice->f[2] + C-
>lattice->f[4]
                              + 2.0*(C->lattice->f[3] + C->lattice->f[6] + C-
>lattice->f[7]))
                              / (1.0 - C->lattice->velocity_bc[0]);

                C->lattice->f[1] = C->lattice->f[3] + aux * rho * C->lattice-
>velocity_bc[0];
                C->lattice->f[5] = C->lattice->f[7] + div * rho * C->lattice-
>velocity_bc[0]
                                   - 0.5 * (C->lattice->f[2] - C->lattice-
>f[4]);
                C->lattice->f[8] = C->lattice->f[6] + div * rho * C->lattice-
>velocity_bc[0]
                                   - 0.5 * (C->lattice->f[4] - C->lattice-
>f[2]);
```

```
                C->CalculateDensityAndVelocity();
            }
        } break;
    }

    // ---------------- Density BCs ----------------
    switch (S.apply_density_bc) {
        // case 1: top density
        case 1: {
            int j = Ny - 1;
            for (int i = 1; i <= Nx - 2; ++i) {
                int id = S.CalculateCellId(i, j);
                auto& C = S.cells[id];
                if (C->is_solid || C->is_wall) continue;

                double vy = -1.0 + (C->lattice->f[0] + C->lattice->f[1] + C-
>lattice->f[3]
                                    + 2.0*(C->lattice->f[2] + C->lattice->f[5]
+ C->lattice->f[6]))
                                    / C->lattice->density_bc;

                C->lattice->f[4] = C->lattice->f[2] - aux * C->lattice-
>density_bc * vy;
                C->lattice->f[7] = C->lattice->f[5] - div * C->lattice-
>density_bc * vy
                                    + 0.5 * (C->lattice->f[1] - C->lattice-
>f[3]);
                C->lattice->f[8] = C->lattice->f[6] - div * C->lattice-
>density_bc * vy
                                    + 0.5 * (C->lattice->f[3] - C->lattice-
>f[1]);
                C->CalculateDensityAndVelocity();
            }
        } break;

        // case 2: right density  <-- Poiseuille outlet (rho)
        case 2: {
            int i = Nx - 1;
            for (int j = 1; j <= Ny - 2; ++j) {
                int id = S.CalculateCellId(i, j);
                auto& C = S.cells[id];
                if (C->is_solid || C->is_wall) continue;

                double vx = -1.0 + (C->lattice->f[0] + C->lattice->f[2] + C-
>lattice->f[4]
                                    + 2.0*(C->lattice->f[1] + C->lattice->f[5]
+ C->lattice->f[8]))
                                    / C->lattice->density_bc;
```

```
                C->lattice->f[3] = C->lattice->f[1] - aux * C->lattice-
>density_bc * vx;
                C->lattice->f[7] = C->lattice->f[5] - div * C->lattice-
>density_bc * vx
                                   + 0.5 * (C->lattice->f[2] - C->lattice-
>f[4]);
                C->lattice->f[6] = C->lattice->f[8] - div * C->lattice-
>density_bc * vx
                                   + 0.5 * (C->lattice->f[4] - C->lattice-
>f[2]);
                C->CalculateDensityAndVelocity();
            }
        } break;

        // case 3: bottom density
        case 3: {
            int j = 0;
            for (int i = 1; i <= Nx - 2; ++i) {
                int id = S.CalculateCellId(i, j);
                auto& C = S.cells[id];
                if (C->is_solid || C->is_wall) continue;

                double vy = -1.0 + (C->lattice->f[0] + C->lattice->f[1] + C-
>lattice->f[3]
                                   + 2.0*(C->lattice->f[4] + C->lattice->f[7]
+ C->lattice->f[8]))
                                   / C->lattice->density_bc;

                C->lattice->f[2] = C->lattice->f[4] - aux * C->lattice-
>density_bc * vy;
                C->lattice->f[5] = C->lattice->f[7] - div * C->lattice-
>density_bc * vy
                                   + 0.5 * (C->lattice->f[3] - C->lattice-
>f[1]);
                C->lattice->f[6] = C->lattice->f[8] - div * C->lattice-
>density_bc * vy
                                   + 0.5 * (C->lattice->f[1] - C->lattice-
>f[3]);
                C->CalculateDensityAndVelocity();
            }
        } break;

        // case 4: left density
        case 4: {
            int i = 0;
            for (int j = 1; j <= Ny - 2; ++j) {
                int id = S.CalculateCellId(i, j);
                auto& C = S.cells[id];
                if (C->is_solid || C->is_wall) continue;
```

```
                double vx = -1.0 + (C->lattice->f[0] + C->lattice->f[2] + C-
>lattice->f[4]
                                    + 2.0*(C->lattice->f[3] + C->lattice->f[6]
+ C->lattice->f[7]))
                                    / C->lattice->density_bc;

                C->lattice->f[1] = C->lattice->f[3] + aux * C->lattice-
>density_bc * vx;
                C->lattice->f[5] = C->lattice->f[7] + div * C->lattice-
>density_bc * vx
                                    + 0.5 * (C->lattice->f[4] - C->lattice-
>f[2]);
                C->lattice->f[8] = C->lattice->f[6] + div * C->lattice-
>density_bc * vx
                                    + 0.5 * (C->lattice->f[2] - C->lattice-
>f[4]);
                C->CalculateDensityAndVelocity();
            }
        } break;
    }
}

void FluidCollision::action() {
    Scene& S = Scene::get_Scene();

    for (auto& C : S.cells) {
        if (C->is_solid || C->is_wall || C->solid_fraction > 0.0) {
            continue;
        }

        const double rho = C->material->density;
        const Vector3r u = C->state->vel;
        Vector9r fEq = Vector9r::Zero();
        for (int k = 0; k < C->lattice->number_of_nodes; ++k) {
            fEq[k] = C->CalculateEqFunction(rho, u, k);
        }

        if (S.collision_operator == "MRT"){
            C->lattice->f -= C->lattice->m_inv *
                             C->lattice->relaxation_matrix *
                             C->lattice->m *
                             (C->lattice->f - fEq);
        }
        else {
            C->lattice->f -= (1.0 / S.relaxation_time) * (C->lattice->f -
fEq);
        }
    }
```

```
}

void FluidStreaming::action() {
    Scene& S = Scene::get_Scene();

    for (auto& Cptr : S.cells) {
        Cell& C = *Cptr;
        C.lattice->f_tmp.setZero();
    }

    for (size_t idx = 0; idx < S.cells.size(); ++idx) {
        Cell& C = *S.cells[idx];
        if (C.is_solid || C.is_wall) continue;

        for (int k = 0; k < C.lattice->number_of_nodes; ++k) {
            const int nb = C.lattice->neighbor_node[k];
            const int ko = C.lattice->opposite_node[k];

            Cell& N = *S.cells[nb];

            if (N.is_solid || N.is_wall) {
                C.lattice->f_tmp[ko] += C.lattice->f[k];
            } else {
                N.lattice->f_tmp[k] += C.lattice->f[k];
            }
        }
    }

    for (auto& Cptr : S.cells) {
        Cell& C = *Cptr;

        std::swap(C.lattice->f, C.lattice->f_tmp);

        if (C.is_solid || C.is_wall) {
            C.material->density = 0.0;
            C.state->vel = Vector3r::Zero();
            continue;
        }

        C.CalculateDensityAndVelocity();
    }
}

void ApplyFluidForcing::action(){
    Scene& S = Scene::get_Scene();
    const double cs2 = 1.0 / 3.0;
    for (auto& Cptr : S.cells) {
            auto& C = *Cptr;
            if (C.is_solid || C.is_wall) continue;
```

```
            Vector3r F = S.GUO_fluid_forcing;
            for (int k = 0; k < C.lattice->number_of_nodes; ++k) {
                const auto& ci = C.lattice->discrete_velocity[k];
                const double ciF = ci.dot(F);
                const double ciu = ci.dot(C.state->vel);
                const double uF  = C.state->vel.dot(F);
                const double term = (ciF/cs2) + (ciu*ciF)/(cs2*cs2) -
(uF/cs2);
                C.lattice->f[k] += (1.0 - 0.5 / S.relaxation_time)
                                * C.lattice->node_weight[k] * term;
            }
        }
}

void LatticeSearch::action() {
    Scene& S = Scene::get_Scene();
    if (S.bodies.empty()) return;

    const std::size_t nCells = S.cells.size();
    std::vector<double> prev_chi(nCells, 0.0);
    std::vector<bool>   prev_is_solid(nCells, false);

    for (std::size_t id = 0; id < nCells; ++id) {
        auto& C = S.cells[id];
        prev_chi[id]      = C->solid_fraction;
        prev_is_solid[id] = C->is_solid;
    }

    for (auto& C : S.cells) {
        C->solid_fraction = 0.0;

        if (!C->is_wall) {
            C->is_solid = false;
        }
    }

    aabb box;

    for (auto& B : S.bodies) {
        B->fluid_interactions.clear();

        boost::geometry::envelope(B->shape->cloud, box);

        int aabb_x_min = static_cast<int>(box.min_corner().get<0>());
        int aabb_x_max = static_cast<int>(box.max_corner().get<0>() + 1);
        int aabb_y_min = static_cast<int>(box.min_corner().get<1>());
        int aabb_y_max = static_cast<int>(box.max_corner().get<1>() + 1);
```

```
        aabb_x_min = std::max(aabb_x_min,
static_cast<int>(S.model_min_corner[0]));
        aabb_y_min = std::max(aabb_y_min,
static_cast<int>(S.model_min_corner[1]));
        aabb_x_max = std::min(aabb_x_max,
static_cast<int>(S.model_max_corner[0]));
        aabb_y_max = std::min(aabb_y_max,
static_cast<int>(S.model_max_corner[1]));

        for (int i = aabb_x_min; i < aabb_x_max; ++i) {
            for (int j = aabb_y_min; j < aabb_y_max; ++j) {

                int id = S.CalculateCellId(i, j);
                auto& C = S.cells[id];

                if (C->is_wall) {
                    continue;
                }

                if (!boost::geometry::intersects(C->grid, B->shape->cloud)) {
                    continue;
                }

                B->fluid_interactions.push_back(C);

                std::vector<polygon> output;
                boost::geometry::intersection(C->grid, B->shape->cloud,
output);

                double overlap_area = 0.0;
                for (const auto& poly : output) {
                    overlap_area += boost::geometry::area(poly);
                }

                const double cell_area = boost::geometry::area(C->grid);
                double phi = 0.0;

                if (overlap_area > 0.0) {
                    phi = overlap_area / cell_area;
                } else {
                    point cell_center(C->state->pos[0], C->state->pos[1]);
                    if (boost::geometry::within(cell_center, B->shape->cloud))
{
                        phi = 1.0;
                    }
                }

                phi = std::min(1.0, std::max(0.0, phi));
                C->solid_fraction = std::min(1.0, C->solid_fraction + phi);
```

```
                if (C->solid_fraction >= 0.999) {
                    C->solid_fraction = 1.0;
                    C->is_solid = true;
                }
            }
        }
    }

    for (std::size_t id = 0; id < nCells; ++id) {
        auto& C = S.cells[id];

        if (C->is_wall) continue;

        const double chi_old = prev_chi[id];
        const bool   solid_old = prev_is_solid[id];
        const double chi_new = C->solid_fraction;

        if ((chi_old > 0.0 || solid_old) && chi_new == 0.0) {
            const double alpha = 0.3; // 0<alpha<=1
            Vector9r fEq = Vector9r::Zero();
            for (int k = 0; k < C->lattice->number_of_nodes; ++k) {
                fEq[k] = C->CalculateEqFunction(S.initial_density,
S.initial_velocity, k);
            }
            C->lattice->f = (1.0 - alpha) * C->lattice->f + alpha * fEq;
            C->material->density = S.initial_density;
            C->state->vel = S.initial_velocity;
        }
    }
}



void ImbBoundary::action()
{
    Scene& S = Scene::get_Scene();
    if (S.bodies.empty()) return;

    const double tau = S.relaxation_time;
    const int Q = 9;

    const double chi_eps    = 1e-12;
    const double chi_full_0 = 0.98;
    const double chi_full_1 = 0.999;
    const double f_clamp_lo = -10.0;
    const double f_clamp_hi =  10.0;
    const bool   use_chi_smoothing = false;
    const double beta_chi = 0.3
```

```
    auto smoothstep = [](double x) {
        x = std::clamp(x, 0.0, 1.0);
        return x*x*(3.0 - 2.0*x);
    };

    for (auto& B : S.bodies) {
        B->state->hydro_force  = Vector3r::Zero();
        B->state->hydro_torque = 0.0;
    }

    // Loop bodies and their interacting cells
    for (auto& B : S.bodies)
    {
        const Vector3r ub = B->state->vel;

        for (auto& C : B->fluid_interactions)
        {
            if (C->is_wall) continue;

            // --- chi (solid fraction) ---
            double chi = C->solid_fraction;
            chi = std::clamp(chi, 0.0, 1.0);
            if (use_chi_smoothing) {
            if (chi <= chi_eps) continue;
            double   rho = C->material->density;
            Vector3r uf  = C->state->vel;

            if (!std::isfinite(rho) || rho < 0.1 || !std::isfinite(uf[0]) ||
!std::isfinite(uf[1])) {
                rho = S.initial_density;
                uf  = ub;
                for (int i = 0; i < Q; ++i) {
                    C->lattice->f[i] = C->CalculateEqFunction(rho, uf, i);
                }
                C->material->density = rho;
                C->state->vel        = uf;
            }

            // --- compute Bn (Noble & Torczynski weighting) ---
            const double denom = (1.0 - chi) + (tau - 0.5);
            const double Bn = (denom > 0.0) ? (chi * (tau - 0.5) / denom) :
0.0;

            if (chi >= chi_full_0) {
                double x = (chi - chi_full_0) / (chi_full_1 - chi_full_0);
                const double gamma = smoothstep(x); // 0..1
                for (int i = 0; i < Q; ++i) {
                    const double feq_bi = C->CalculateEqFunction(rho, ub, i);
```

```
                    C->lattice->f[i] = (1.0 - gamma) * C->lattice->f[i] +
gamma * feq_bi;
                }
                C->state->vel = ub;
            }

            if (C->is_solid) continue;

            Vector9r feq_f   = Vector9r::Zero();
            Vector9r omega_s = Vector9r::Zero();

            for (int i = 0; i < Q; ++i)
            {
                const int opp = C->lattice->opposite_node[i];

                const double feq_f_i     = C->CalculateEqFunction(rho, uf, i);
                const double feq_b_opp   = C->CalculateEqFunction(rho, ub,
opp);

                feq_f[i] = feq_f_i;

                omega_s[i] = (C->lattice->f[opp] - feq_b_opp)
                           - (C->lattice->f[i]   - feq_f_i);
            }

            Vector9r df_coll = Vector9r::Zero();
            const bool isBGK = (S.collision_operator == "BGK");

            if (!isBGK) {
                df_coll = C->lattice->m_inv
                        * C->lattice->relaxation_matrix
                        * C->lattice->m
                        * (C->lattice->f - feq_f);
            }

            for (int i = 0; i < Q; ++i)
            {
                const double fi = C->lattice->f[i];
                double fi_new   = fi;

                if (isBGK) {
                    const double coll_i = (1.0 / tau) * (fi -
feq_f[i]);     // BGK
                    fi_new = fi - (1.0 - Bn) * coll_i + Bn * omega_s[i];
                } else {
                    // MRT
                    fi_new = fi - (1.0 - Bn) * df_coll[i] + Bn * omega_s[i];
                }
```

```
                if (!std::isfinite(fi_new) || fi_new < f_clamp_lo || fi_new >
f_clamp_hi) {
                    fi_new = feq_f[i];
                }

                C->lattice->f[i] = fi_new;
            }

            // --- hydrodynamic force/torque (Noble & Torczynski)
            Vector3r cell_force = Vector3r::Zero();
            for (int i = 0; i < Q; ++i) {
                cell_force += omega_s[i] * C->lattice->discrete_velocity[i];
            }
            cell_force *= Bn;

            B->state->hydro_force -= cell_force;

            // 2D torque about body center (z-component)
            const Vector3r r = C->state->pos - B->state->pos;
            B->state->hydro_torque -= (r[0] * cell_force[1] - r[1] *
cell_force[0]);
        }
    }
}


void ContactResolution::action(){
    Scene& S = Scene::get_Scene();

    int bodySize = (int)S.bodies.size();
    ASSERT(bodySize > 0);
    //Brute force method:
    for (int i = 0; i < bodySize - 1; ++i) {
        for (int j = i + 1; j < bodySize; ++j) {
            if (S.bodies[i]->CheckInteraction(S.bodies[j]->id)) continue;
            if ((S.bodies[i]->state->pos - S.bodies[j]->state->pos).norm() <
S.bodies[i]->shape->radius + S.bodies[j]->shape->radius) {
                S.interactions.push_back(std::make_shared<Interaction>(S.bodie
s[i], S.bodies[j]));
                S.bodies[i]-
>inter.push_back(S.interactions[S.interactions.size() - 1]);
                S.bodies[j]-
>inter.push_back(S.interactions[S.interactions.size() - 1]);
            }
        }
    }
}

void InteractionLoop::action(){
```

```
    Scene& S = Scene::get_Scene();

    for (auto& I : S.interactions) {
        I->CalculateUnitVectorandContact();
        I->CalculateForceAndShearIncrements();
        I->ApplyFrictionLaw();
    }
}

void BodyLoop::action(){
    Scene& S = Scene::get_Scene();

    //Force calculation:
    for (auto& B : S.bodies) {
        //Force Resetter:
        B->state->force = Vector3r::Zero();
        B->state->moment = 0.0;

        //Contact force
        for (auto& I : B->inter) {
            auto Il = I.lock();
            auto iBody1 = Il->body1.lock();
            auto iBody2 = Il->body2.lock();
            if (B->id == iBody1->id) {
                B->state->force += -Il->normal_force * Il->unit_normal_vector;
                B->state->force += -Il->shear_force * Il->unit_shear_vector;
                B->state->moment += -Il->shear_force * iBody1->shape->radius;
            }
            if (B->id == iBody2->id) {
                B->state->force += Il->normal_force * Il->unit_normal_vector;
                B->state->force += Il->shear_force * Il->unit_shear_vector;
                B->state->moment += -Il->shear_force * iBody2->shape->radius;
            }
        }

        Vector3r unitUp = { 0, 1, 0 };
        Vector3r unitDown = { 0, -1, 0 };
        Vector3r unitRight = { 1, 0, 0 };
        Vector3r unitLeft = { -1, 0, 0 };
        Vector3r force = Vector3r::Zero();
        if ((B->state->pos[0] - B->shape->radius) <
S.model_min_corner[0])  force += S.border_stiffness * abs(B->shape->radius -
B->state->pos[0]) * unitRight;
        if ((B->state->pos[0] + B->shape->radius) >
S.model_max_corner[0])  force += S.border_stiffness * abs(B->shape->radius -
(S.model_max_corner[0] - B->state->pos[0])) * unitLeft;
        if ((B->state->pos[1] - B->shape->radius) <
S.model_min_corner[1])  force += S.border_stiffness * abs(B->shape->radius -
B->state->pos[1]) * unitUp;
```

```
        if ((B->state->pos[1] + B->shape->radius) >
S.model_max_corner[1])  force += S.border_stiffness * abs(B->shape->radius -
(S.model_max_corner[1] - B->state->pos[1])) * unitDown;

        //Body, fluid and border force:
        B->state->force += force;
        B->state->force += B->material->mass * S.gravity;

        // For IMB coupling
        if (S.dem_coupling == "IMB") {

            // Buoyancy force
            double V_particle = M_PI * B->shape->radius * B->shape->radius;
            B->state->force -= S.initial_density * V_particle * S.gravity;

            // Hydrodynamic force and torque
            B->state->force += B->state->hydro_force;
            B->state->moment += B->state->hydro_torque;
        }
    }
}

void Integrator::action(){
    Scene& S = Scene::get_Scene();

    for (auto& B : S.bodies) {

        //Calculate accelaration from forces:
        Vector3r  linAccel = Vector3r::Zero();
        Vector3r  f = B->state->force;
        double    m = B->state->moment;
        double    rotAccel = 0.0;
        int       signV = 0;
        int       signM = 0;

        for (int i = 0; i < linAccel.size(); ++i) {
            B->state->vel[i] > 0 ? signV = 1 : signV = -1;
            linAccel[i] = ((f[i] - S.local_damping * abs(f[i]) * signV) / B-
>material->mass) * B->blockedDOFs[i];
        }

        B->state->rotVel > 0 ? signM = 1 : signM = -1;
        rotAccel = ((m - S.local_damping * abs(m) * signM) / B->shape-
>inertia_moment) * B->blockedMDOFs;

        //Update velocity and position (LeapFrog method):
        if (S.nIter == 0) {
            B->state->vel += linAccel * S.dt_dem * 0.5;
            B->state->rotVel += rotAccel * S.dt_dem * 0.5;
```

```
        }
        else {
            B->state->vel += linAccel * S.dt_dem;
            B->state->rotVel += rotAccel * S.dt_dem;
        }
        B->state->pos += B->state->vel * S.dt_dem;
        B->state->rot += B->state->rotVel * S.dt_dem;
    }
    ++S.nIter;
    ASSERT(S.nIter > 0);

}

void UpdateContact::action(){
    Scene& S = Scene::get_Scene();

    S.interactions.erase(std::remove_if(std::begin(S.interactions),
std::end(S.interactions), [](std::shared_ptr<Interaction> I) {return !I-
>CheckContact(); }), std::end(S.interactions));
    for (auto& B : S.bodies) {
        B->inter.erase(std::remove_if(std::begin(B->inter), std::end(B-
>inter), [](std::weak_ptr<Interaction> I) {return I.expired(); }), std::end(B-
>inter));
    }
}
```

SCENE FILES: Output.h

```
#ifndef OUTPUT_H
#define OUTPUT_H

#include "maths/Math.h"


class Scene;

class Output {
    public:
    void DisplaySimulationInfo();

    void ExportFluidVtk(std::string _file_name);
    void ExportParticleVtk(std::string _file_name);

    void ExportFluidCsv(Vector3r _pos, std::string _file_name);
    void ExportVelocityProfileCsv(const std::string& file_name);
    void ExportParticleCsv(std::string _file_name);
};
```

```
#endif
```

SCENE FILES: Output.cpp

```
#include <fstream>
#include "scene/Scene.h"
#include "dem/Body.h"
#include "property/State.h"
#include "property/Shape.h"
#include "scene/Output.h"

void Output::DisplaySimulationInfo(){
    Scene& S = Scene::get_Scene();
    int ignore = std::system("clear");
    std::cout << "----------------------- 2D LBM/DEM Simulation  -------------
--------------------" << "\n";
    std::cout << "Iteration Number: " <<
S.iter                                                  << "\n";
    std::cout << "Number of Cells : " <<
S.cells.size()                                          << "\n";
    std::cout << "Number of Bodies: " <<
S.bodies.size()                                         << "\n";
    std::cout << "----------------------- LBM Parameters ---------------------
-------------------"  << "\n";
    std::cout << "Time Step       : " <<
S.dt_lbm                                                << "\n";
    std::cout << "Lattice Spacing : " <<
S.lattice_spacing                                       << "\n";
    std::cout << "Relaxation Time : " <<
S.relaxation_time                                       << "\n";
    std::cout << "Guo Forcing     : " << S.GUO_fluid_forcing[0] << ", " <<
S.GUO_fluid_forcing[1]   << "\n";
    std::cout << "------------------------- DEM Parameters ------------------
--------------------" << "\n";
    std::cout << "Number of bodies: " <<
S.bodies.size()                                         << "\n";
    std::cout << "Time Step       : " <<
S.dt_dem                                                << "\n";
    std::cout << "Friction Angle  : " <<
S.friction_angle                                        << "\n";
    std::cout << "Shear Stiffness : " <<
S.shear_stiffness                                       << "\n";
    std::cout << "Normal Stiffness: " <<
S.normal_stiffness                                      << "\n";
}
```

```cpp
void Output::ExportFluidVtk(std::string file_prefix) {
    Scene& S = Scene::get_Scene();

    const int Nx = static_cast<int>(S.model_max_corner[0]);
    const int Ny = static_cast<int>(S.model_max_corner[1]);
    const int Nz = static_cast<int>(S.model_max_corner[2]);
    const int Npts = Nx * Ny * Nz;

    const double cs2 = 1.0 / 3.0;

    std::ofstream out(file_prefix + std::to_string(S.iter) + ".vtk");
    out << std::fixed << std::setprecision(6);

    // --- Header ---
    out << "# vtk DataFile Version 3.0\n";
    out << "Fluid state\n";
    out << "ASCII\n";
    out << "DATASET STRUCTURED_POINTS\n";
    out << "DIMENSIONS " << Nx << " " << Ny << " " << Nz << "\n";
    out << "ORIGIN 0 0 0\n";
    out << "SPACING 1 1 1\n";
    out << "POINT_DATA " << Npts << "\n";

    // --- Geometry mask: 1=fluid, 0=solid/wall ---
    out << "SCALARS 1.Geometry float 1\n";
    out << "LOOKUP_TABLE default\n";
    for (auto& Cptr : S.cells) {
        const auto& C = *Cptr;
        float geometry = (C.is_solid || C.is_wall) ? 0.0f : 1.0f;
        out << geometry << "\n";
    }

    // --- Density ---
    out << "SCALARS 2.Density float 1\n";
    out << "LOOKUP_TABLE default\n";
    for (auto& Cptr : S.cells) {
        const auto& C = *Cptr;
        out << static_cast<float>(C.material->density) << "\n";
    }

    // --- Pressure = cs^2 * rho ---
    out << "SCALARS 3.Pressure float 1\n";
    out << "LOOKUP_TABLE default\n";
    for (auto& Cptr : S.cells) {
        const auto& C = *Cptr;
        const float p = static_cast<float>(cs2 * C.material->density);
        out << p << "\n";
    }
```

```
    // --- Velocity vector ---
    out << "VECTORS 4.Velocity float\n";

    for (auto& Cptr : S.cells) {
        const auto& C = *Cptr;
        if (C.is_solid || C.is_wall || C.solid_fraction == 1.0) {
            out << "0 0 0\n";
        } else {
            out << static_cast<float>(C.state->vel[0]) << " "
                << static_cast<float>(C.state->vel[1]) << " "
                << static_cast<float>(C.state->vel[2]) << "\n";
        }
    }

    // --- Geometry mask: 1=fluid, 0=solid/wall ---
    out << "SCALARS 5.SolidFraction float 1\n";
    out << "LOOKUP_TABLE default\n";
    for (auto& Cptr : S.cells) {
        const auto& C = *Cptr;
        out << C.solid_fraction << "\n";
    }
    out.close();
}

void Output::ExportParticleVtk(std::string _file_name) {
    Scene& S = Scene::get_Scene();

    // create output file name: e.g., dem_0001.vtk
    std::ostringstream fname;
    fname << _file_name << std::setfill('0') << std::setw(4) << S.iter <<
".vtk";

    std::ofstream out(fname.str());
    if (!out.is_open()) {
        std::cerr << "Error: cannot open " << fname.str() << " for writing\n";
        return;
    }

    out << "# vtk DataFile Version 3.0\n";
    out << "DEM particle output\n";
    out << "ASCII\n";
    out << "DATASET POLYDATA\n";

    const size_t N = S.bodies.size();
    out << "POINTS " << N << " double\n";
    for (const auto& B : S.bodies) {
        const auto& p = B->state->pos;
        out << p.x() << " " << p.y() << " " << p.z() << "\n";
    }
```

```
    out << "\nPOINT_DATA " << N << "\n";

    // --- Radius ---
    out << "SCALARS Radius double 1\n";
    out << "LOOKUP_TABLE default\n";
    for (const auto& B : S.bodies)
        out << B->shape->radius << "\n";

    // --- Velocity ---
    out << "\nVECTORS Velocity double\n";
    for (const auto& B : S.bodies) {
        const auto& v = B->state->vel;
        out << v.x() << " " << v.y() << " " << v.z() << "\n";
    }

    // --- Force ---
    out << "\nVECTORS Force double\n";
    for (const auto& B : S.bodies) {
        const auto& f = B->state->force;
        out << f.x() << " " << f.y() << " " << f.z() << "\n";
    }

    out.close();
}

void Output::ExportFluidCsv(Vector3r _pos, std::string _file_name){
    Scene& S = Scene::get_Scene();
    int id = S.CalculateCellId(_pos[0], _pos[1]);
    std::ofstream out;
    out.open(_file_name + std::to_string(S.iter) + ".csv",
std::ios_base::app);
    out << S.cells[id]->state->pos[0]     << ", " << S.cells[id]->state-
>pos[1]     << ", " << S.cells[id]->state->pos[2] << ", "
        << S.cells[id]->state->vel[0]     << ", " << S.cells[id]->state-
>vel[1]     << ", " << S.cells[id]->state->vel[2] << ", "
        << S.cells[id]->state->vel.norm() << ", " << S.cells[id]->material-
>density << "\n";
    out.close();
}

void Output::ExportVelocityProfileCsv(const std::string& file_name) {
    Scene& S = Scene::get_Scene();

    int Nx = static_cast<int>(S.model_max_corner[0]);
    int Ny = static_cast<int>(S.model_max_corner[1]);

    std::ofstream out(file_name + "_" + std::to_string(S.iter) + ".csv");
    out << "y, ux_avg\n";
```

```
    for (int j = 0; j < Ny; ++j) {
        double ux_sum = 0.0;
        int count = 0;
        for (int i = 0; i < Nx; ++i) {
            auto& C = *S.cells[S.CalculateCellId(i, j)];
            if (C.is_solid || C.is_wall) continue;
            ux_sum += C.state->vel[0];
            ++count;
        }
        if (count > 0) {
            double ux_avg = ux_sum / count;
            out << j << "," << ux_avg << "\n";
        }
    }
    out.close();
}


void Output::ExportParticleCsv(std::string _file_name){
    Scene& S = Scene::get_Scene();
    std::ofstream out;
    out.open(_file_name, std::ios_base::app);
    if (S.bodies.size() == 1){
        out << S.bodies[0]->state->pos[1]<< ", " << S.bodies[0]->material-
>density * S.bodies[0]->state->pos[1] * (9.81) << ", " << S.bodies[0]-
>material->density * (S.bodies[0]->state->vel[1] * S.bodies[0]->state->vel[1])
/ 2 << "\n";
    } else {
        out << S.bodies[0]->state->pos[0] << ", " << S.bodies[0]->material-
>density * (S.bodies[0]->state->vel[0] * S.bodies[0]->state->vel[0]) / 2 << ",
"
            << S.bodies[1]->state->pos[0] << ", " << S.bodies[1]->material-
>density * (S.bodies[1]->state->vel[0] * S.bodies[1]->state->vel[0]) / 2 <<
"\n";
    }
    out.close();
}
```

SCENE FILES: Scene.h

```
#ifndef SCENE_H
#define SCENE_H

// #include "maths/Math.h"
#include "lbm/Cell.h"
#include "dem/Body.h"
```

```cpp
#include "dem/Interaction.h"
#include "engine/Engine.h"
#include "maths/Math.h"

class Body;
class Interaction;

class Scene {
    public:
    EIGEN_MAKE_ALIGNED_OPERATOR_NEW
    static Scene& get_Scene();
    int  CalculateCellId(int i, int j) {return i + j * model_max_corner[0];}
    void AddRectangularCanal();
    void addParticle(std::string _fileName, double _density, Vector3r
_velocity);
    void AddDisk(Vector3r _center, double _radius, double _density);
    void MoveToNextTimeStep();
    void SetRelaxationParamters(double _s1, double _s2, double _s3, double
_s4, double _s5, double _s6, double _s7, double _s8, double _s9);
    // Generates a random disk packing; optional params control
count/densities
    void addDiskPacking(double _Rmin, double _Rmax, double _wall_gap, double
_inlet_buf, double _outlet_buf, double _non_overlap,
                        int _max_particles = 0, int _wall_particles = 0,
double _rho_p = 1.0,  double _Rw_min = 0.0, double _Rw_max = 0.0);

    std::vector<std::shared_ptr<Cell>> cells;
    std::vector<std::shared_ptr<Body>> bodies;
    std::vector<std::shared_ptr<Interaction>> interactions;
    std::vector<std::shared_ptr<Engine>> engines;

    //Simulation Parameters:
    int      iter           = 0;
    double   time           = 0.0;
    Vector3r model_min_corner = Vector3r::Zero();
    Vector3r model_max_corner = Vector3r::Zero();
    Vector3r gravity        = {0.0, -9.81, 0.0};

    //LBM Parameters:
    int apply_velocity_bc = -1;
    int apply_density_bc = -1;
    double lattice_spacing      = 1.0;
    double dt_lbm   = 1.0;
    double relaxation_time     = 1.0;
    double initial_density = 1.0;
    double latticeSpeed = 1.0;
    std::string collision_operator = "MRT";
    std::string dem_coupling = "None";
    Vector3r initial_velocity = Vector3r::Zero();
```

```
    Vector3r GUO_fluid_forcing = Vector3r::Zero();
    Vector9r relaxation_parameters = Vector9r::Zero();

    //DEM Parameters:
    double dt_dem            = 1.0;
    double friction_angle    = 30;
    double local_damping     = 0.0;
    double factor_of_safety  = 0.3;
    double normal_stiffness  = 1e6;
    double shear_stiffness   = 0.5e6;
    double border_stiffness  = 1e6;
    int nIter = 0;

};

#endif
```

SCENE FILES: Scene.cpp

```
#include "scene/Scene.h"


static Scene scene;

Scene& Scene::get_Scene(){ return scene;}

void Scene::AddRectangularCanal(){
    Scene& S = Scene::get_Scene();
    int id;

     //Add cells:
    for (int j = 0; j < model_max_corner[1]; ++j)
    for (int i = 0; i < model_max_corner[0]; ++i){
        id = CalculateCellId(i, j);
        cells.push_back(std::make_shared<Cell>(id,
std::make_shared<Lattice>(), std::make_shared<Material>(),
std::make_shared<State>()));
        cells[id]->state->pos = Vector3r(i, j, 0);
        cells[id]->set_neighbor_node();
        double h = lattice_spacing * 0.5;

        //Setting boost geometry polygon
        Vector3r A = Vector3r(i,j,0) + h * Vector3r(-1,-1, 0);
        Vector3r B = Vector3r(i,j,0) + h * Vector3r(-1, 1, 0);
        Vector3r C = Vector3r(i,j,0) + h * Vector3r( 1, 1, 0);
        Vector3r D = Vector3r(i,j,0) + h * Vector3r( 1,-1, 0);
```

```
        cells[id]->grid.outer().push_back(point(A[0], A[1]));
        cells[id]->grid.outer().push_back(point(B[0], B[1]));
        cells[id]->grid.outer().push_back(point(C[0], C[1]));
        cells[id]->grid.outer().push_back(point(D[0], D[1]));
        cells[id]->grid.outer().push_back(point(A[0], A[1]));

        boost::geometry::correct(cells[id]->grid);
        }

        // Lattice search initialization if body present
        if (S.bodies.size() > 0 && S.dem_coupling == "IMB") {
            S.engines.push_back(std::make_shared<LatticeSearch>());
            for (auto& E : S.engines){
                E->action();
            }
        }

        //Define walls:
        //Top:
        for (int i = 0; i < model_max_corner[0]; ++i){
            id = CalculateCellId(i, model_max_corner[1]-1);
            cells[id]->is_wall = true;
            if (S.dem_coupling == "IMB") cells[id]->solid_fraction = 1.0;
        }
        //Bot:
        for (int i = 0; i < model_max_corner[0]; ++i){
            id = CalculateCellId(i, 0);
            cells[id]->is_wall = true;
            if (S.dem_coupling == "IMB") cells[id]->solid_fraction = 1.0;
        }

        for (auto& C : cells){
            C->set_initial_condition();
            if (S.collision_operator == "MRT")  C->set_mrt_parameters();
        }
}

void Scene::addParticle(std::string _fileName, double _density, Vector3r
_velocity) {
    int id = bodies.size();
    bodies.push_back(std::make_shared<Body>(id, std::make_shared<Shape>(),
std::make_shared<State>(), std::make_shared<Material>()));
    bodies[id]->shape->generate_particle(_fileName);
    Vector3r sum = Vector3r::Zero();
    for(auto c : bodies[id]->shape->coordinates){
        sum += c;
    }
    bodies[id]->state->pos = sum / bodies[id]->shape->coordinates.size();
```

```
    bodies[id]->shape->radius = (bodies[id]->state->pos - bodies[id]->shape-
>coordinates[0]).norm();
    bodies[id]->material->density = _density;
    bodies[id]->state->vel = _velocity;
    bodies[id]->material->mass = _density * M_PI * bodies[id]->shape->radius *
bodies[id]->shape->radius;
    bodies[id]->shape->inertia_moment = bodies[id]->material->density *
bodies[id]->shape->radius * bodies[id]->shape->radius * 0.5;
}

void Scene::AddDisk(Vector3r _center, double _radius, double _density){
    int id = bodies.size();
    bodies.push_back(std::make_shared<Body>(id, std::make_shared<Shape>(),
std::make_shared<State>(), std::make_shared<Material>()));

    bodies[id]->shape->cloud.clear();
    bodies[id]->shape->coordinates.clear();

    for (double theta = 0.0; theta < 2.0 * M_PI; theta += 0.1) {
        double x = _center[0] + _radius * std::cos(theta);
        double y = _center[1] + _radius * std::sin(theta);
        boost::geometry::append(bodies[id]->shape->cloud.outer(), point(x,
y));
        bodies[id]->shape->coordinates.emplace_back(Vector3r(x, y, 0.0));
    }
    boost::geometry::correct(bodies[id]->shape->cloud);

    bodies[id]->state->pos = _center;
    bodies[id]->shape->radius = _radius;

    bodies[id]->material->density = _density;
    bodies[id]->material->mass = _density * M_PI * _radius * _radius;
    bodies[id]->shape->inertia_moment = 0.5 * bodies[id]->material->mass *
_radius * _radius;
}

void Scene::MoveToNextTimeStep(){
    for(auto E : engines){
        E->action();
    }
    time += dt_lbm;
    ++iter;
}

void Scene::SetRelaxationParamters(double _s1, double _s2, double _s3, double
_s4, double _s5, double _s6, double _s7, double _s8, double _s9){
    relaxation_parameters  << _s1, _s2, _s3, _s4, _s5, _s6, _s7, _s8, _s9;
}
```

```
void Scene::addDiskPacking(double _Rmin, double _Rmax, double _wall_gap,
double _inlet_buf, double _outlet_buf, double _non_overlap,
                           int _max_particles, int _wall_particles, double
_rho_p, double _Rw_min, double _Rw_max){

    // Get the number of cells in the x and y directions
    int Nx = static_cast<int>(model_max_corner[0]);
    int Ny = static_cast<int>(model_max_corner[1]);

     // Particle sizes and number of particles
     double Rmin = _Rmin;
     double Rmax = _Rmax;
     int max_particles = _max_particles;

     // Buffers and gaps
     double wall_gap   = _wall_gap;
     double inlet_buf  = _inlet_buf;
     double outlet_buf = _outlet_buf;
     double x_min_p = inlet_buf  + Rmax + wall_gap;
     double x_max_p = Nx - outlet_buf - Rmax - wall_gap;
     double y_min_p = Rmax + wall_gap;
     double y_max_p = Ny - Rmax - wall_gap;
     double non_overlap = _non_overlap;

     //Check overlap
     auto overlap_ok = [&](const Vector3r& pos, double Rnew){
        for (auto& B : bodies){
            const double dist = (pos - B->state->pos).head<2>().norm();
            if (dist < non_overlap * (Rnew + B->shape->radius))
                return false;
        }
        return true;
    };

    // Random packing
    std::mt19937 gen(123);
    std::uniform_real_distribution<double> urx(x_min_p, x_max_p);
    std::uniform_real_distribution<double> ury(y_min_p, y_max_p);
    std::uniform_real_distribution<double> urr(Rmin, Rmax);

    for (int n = 0; n < max_particles; ++n){
        bool placed = false;
        for (int trial = 0; trial < 800 && !placed; ++trial){
            const double R = urr(gen);
            Vector3r pos(urx(gen), ury(gen), 0.0);
            if (overlap_ok(pos, R)) {
                AddDisk(pos, R, _rho_p);
                placed = true;
            }
```

```
        }
    }

    // Extra small disks near top/bottom to reduce preferential channels
    double Rw_min = _Rw_min;
    double Rw_max = _Rw_max;

    std::uniform_real_distribution<double> urx_w(x_min_p, x_max_p);
    std::uniform_real_distribution<double> ury_bottom(Rw_max + 0.1, 3.0 *
Rw_max);
    std::uniform_real_distribution<double> ury_top(Ny - 3.0 * Rw_max, Ny -
(Rw_max + 0.1));
    std::uniform_real_distribution<double> urr_w(Rw_min, Rw_max);

    for (int n = 0; n < _wall_particles; ++n){
        bool placed = false;
        for (int trial = 0; trial < 800 && !placed; ++trial){
            const double R = urr_w(gen);
            const bool top = (n % 2 == 0);
            const double y = top ? ury_top(gen) : ury_bottom(gen);
            Vector3r pos(urx_w(gen), y, 0.0);
            if (overlap_ok(pos, R)) {
                AddDisk(pos, R, _rho_p);
                placed = true;
            }
        }
    }
}
```

APPENDIX B: SCENARIOS

This appendix presents a detailed description of all simulation scenarios shown in this research.

SCENARIO 01: POISEUILLE FLOW WITH ZOU AND HE BOUNDARY CONDITIONS

```
#include "scene/Scene.h"
#include "scene/Output.h"

int main(){

    Timer T;
    Scene& S = Scene::get_Scene();
    Output O;
```

```
    // ---------- Geometry ----------
    double D = 20;
    S.model_max_corner = { 20*D, 5*D + 2 , 1.0 };
    const int H = static_cast<int>(S.model_max_corner[1]) - 2;

    // ---------- Parameters ----------
    double uMax = 0.1;
    double Re = 5;
    double kinViscosity = uMax * H / Re;
    S.dt_lbm = 1.0;
    S.lattice_spacing = 1.0;
    S.relaxation_time = 3.0 * kinViscosity + 0.5;
    S.initial_density = 1.0;
    S.collision_operator = "BGK";

    // ---------- Grid ----------
    S.AddRectangularCanal();

    // ------ Zou And He BC  -------
    S.apply_velocity_bc = 4;
    for (int j = 1; j <= H; ++j) {
        double y  = j - 1.0;
        double ux = 4.0*uMax*(y/H - (y*y)/(H*H));
        int id_in  = S.CalculateCellId(0, j);
        S.cells[id_in]->lattice->velocity_bc = Vector3r(ux, 0.0, 0.0);
    }

    // Engines (Do not change order!)
    S.engines.clear();
    S.engines.push_back(std::make_shared<ZouAndHeBC>());
    S.engines.push_back(std::make_shared<FluidCollision>());
    S.engines.push_back(std::make_shared<FluidStreaming>());

    // -------- Solver ---------
    double nSteps = 50000;
    std::string file_name = "Poiseuille_ZOU: ";
    while (S.time < nSteps){
        if (S.iter % 100 == 0) {
            O.DisplaySimulationInfo();
            O.ExportFluidVtk(file_name);
        }
        S.MoveToNextTimeStep();
    }
    return 0;
}
```

SCENARIO 02: POISEUILLE FLOW WITH GUO FORCING SCHEME

```
#include "scene/Scene.h"
#include "scene/Output.h"

int main(){

    Timer T;
    Scene& S = Scene::get_Scene();
    Output O;

    // ---------- Geometry ----------
    double D = 20;
    S.model_max_corner = { 20*D, 5*D + 2 , 1.0 };
    const int H = static_cast<int>(S.model_max_corner[1]) - 2;

    // ---------- Parameters ----------
    double uMax = 0.1;
    double Re = 5;
    double kinViscosity = uMax * H / Re;
    S.dt_lbm = 1.0;
    S.lattice_spacing = 1.0;
    S.relaxation_time = 3.0 * kinViscosity + 0.5;
    S.initial_density = 1.0;
    S.collision_operator = "BGK";

    // ------ Guo Forcing ------
    const double F_x = 8.0 * kinViscosity * uMax / (H * H);
    S.GUO_fluid_forcing[0] = F_x;

    // ---------- Grid ----------
    S.AddRectangularCanal();

    // Engines (Do not change order!)
    S.engines.clear();
    S.engines.push_back(std::make_shared<FluidStreaming>());
    S.engines.push_back(std::make_shared<ApplyFluidForcing>());
    S.engines.push_back(std::make_shared<FluidCollision>());

    // -------- Solver ---------
    double nSteps = 50000;
    std::string file_name = "Poiseuille_GUO: ";
    while (S.time < nSteps){
        if (S.iter % 100 == 0) {
            O.DisplaySimulationInfo();
            O.ExportFluidVtk(file_name);
        }
        S.MoveToNextTimeStep();
    }
```

```
    return 0;
}
```

SCENARIO 03: LID-DRIVEN CAVITY FLOW

```
#include "scene/Scene.h"
#include "scene/Output.h"

int main(){
    Timer T;
    Scene& S = Scene::get_Scene();
    Output O;

    // ---------- Geometry ----------
    const int N = 256;
    S.model_max_corner = { double(N), double(N), 1.0 };

    // ---------- Parameters ----------
    const double u_lid = 0.05;
    const double Re = 100.0;
    const double kinViscosity = u_lid * N / Re;
    S.relaxation_time = 3.0 * kinViscosity + 0.5;
    S.initial_density = 1.0;
    S.collision_operator = "MRT";

    if (S.collision_operator == "MRT"){
        double s8 = 2/(1 + 6*kinViscosity);
        S.SetRelaxationParamters(0, 1.4, 1.4, 0.75, 1.2, 1, 1.2, s8, s8);
    }

    // ---------- Grid ----------
    S.AddRectangularCanal();

    // Adjusting walls of cavity
    for (auto& C : S.cells){
        C->is_wall  = false;
        C->is_solid = false;
    }
    for (int i=0;i<N;++i){
        S.cells[S.CalculateCellId(i,0)]->is_wall   = true; // bottom
    }
    for (int j=0;j<N;++j){
        S.cells[S.CalculateCellId(0,j)]->is_wall   = true; // left
        S.cells[S.CalculateCellId(N-1,j)]->is_wall = true; // right
    }

    // ---------- Engines ----------
```

```
    S.engines.clear();
    S.engines.push_back(std::make_shared<FluidCollision>());
    S.engines.push_back(std::make_shared<FluidStreaming>());
    S.engines.push_back(std::make_shared<ZouAndHeBC>());

    // ------ Zou And He BC  -------
    S.apply_velocity_bc = 1;
    for (int i = 1; i < N-1; ++i){
        int id = S.CalculateCellId(i, N-1);
        S.cells[id]->lattice->velocity_bc = Vector3r(u_lid, 0.0, 0.0);
    }

    // -------- Solver ---------
    double nSteps = 50000;
    std::string file_name = S.collision_operator + "_Lid-Driven: ";
    while (S.time < nSteps){
        if (S.iter % 500 == 0) {
            O.DisplaySimulationInfo();
            O.ExportFluidVtk(file_name);
        }
        S.MoveToNextTimeStep();
    }
}
```

SCENARIO 04: TAYLOR–GREEN VORTEX

```
#include "scene/Scene.h"
#include "scene/Output.h"

static inline double sq(double x){ return x*x; }

int main(){
    Timer T;
    Scene& S = S.get_Scene();
    Output O;

    // ---------- Geometry ----------
    S.model_max_corner = { 128, 128, 1.0 };
    const int H = static_cast<int>(S.model_max_corner[1]);

    // ---------- Parameters ----------
    const double u0 = 0.05;
    double Re = 5;
    double kinViscosity  = u0 * H / Re;
    S.dt_lbm  = 1.0;
    S.latticeSpeed = 1.0;
    S.relaxation_time = 3.0 * kinViscosity + 0.5;
```

```
    S.initial_density = 1.0;
    S.collision_operator = "BGK";

    // -------- Grid ---------
    S.AddRectangularCanal();
    for (auto& Cptr : S.cells) {
        Cptr->is_wall  = false;
        Cptr->is_solid = false;
    }

    // Engines (Do not change order!)
    S.engines.clear();
    S.engines.push_back(std::make_shared<FluidCollision>());
    S.engines.push_back(std::make_shared<FluidStreaming>());

    // -------- Taylor-Green Initialization --------
    const double Lx = S.model_max_corner[0]
    const double Ly = S.model_max_corner[1];
    const double kx = 2.0*M_PI / Lx;
    const double ky = 2.0*M_PI / Ly;

    for (auto& Cptr : S.cells){
        auto& C = *Cptr;
        const double x = C.state->pos[0];
        const double y = C.state->pos[1];

        const double ux0 = +u0 * std::sin(kx*x) * std::cos(ky*y);
        const double uy0 = -u0 * std::cos(kx*x) * std::sin(ky*y);
        S.initial_velocity[0] = ux0;
        S.initial_velocity[1] = uy0;

        for (int k = 0; k < C.lattice->number_of_nodes; ++k){
            C.lattice->f[k] = C.CalculateEqFunction(S.initial_density,
S.initial_velocity, k);
        }
        C.CalculateDensityAndVelocity();
    }

    // -------- Solver ---------
    double nSteps = 10000;
    std::string file_name = "Taylo-Green: ";
    while (S.time < nSteps){
        if (S.iter % 100 == 0) {
            O.DisplaySimulationInfo();
            O.ExportFluidVtk(file_name);
        }
        S.MoveToNextTimeStep();
    }
    return 0;
```

```
}
```

SCENARIO 05: FLUID-FLOW AROUND A CYLINDER

```
#include "scene/Scene.h"
#include "scene/Output.h"

int main(){

    Timer T;
    Scene& S = Scene::get_Scene();
    Output O;

    // --------- Grid ----------
    std::string file_name = "Cylinder: ";
    double D = 20;
    S.model_max_corner = { 20*D, 5*D + 2 , 1.0 };
    const int H = static_cast<int>(S.model_max_corner[1]) - 2;
    // --------- LBM -----------
    double uMax = 0.1;
    double Re = 500;
    double kinViscosity = uMax * D / Re;
    S.dt_lbm = 1.0;
    S.lattice_spacing = 1.0;
    S.relaxation_time = 3.0 * kinViscosity + 0.5;
    S.initial_density = 1.0;
    S.initial_velocity = Vector3r::Zero();
    S.collision_operator = "MRT";
    S.dem_coupling  = "IMB";

    double s8 = 2/(1 + 6*kinViscosity);
    S.SetRelaxationParamters(0, 1.4, 1.4, 0.75, 1.2, 1, 1.2, s8, s8);

    // -------- Bodies ---------
    S.AddDisk(Vector3r(S.model_max_corner[0]/4 ,S.model_max_corner[1]/2, 0),
D/2.0, 1.0);

    // -------- Walls ---------
    S.AddRectangularCanal();

    // ------ Zou and He BC + Poiseuille ------
    S.apply_velocity_bc = 4;
    for (int j = 1; j <= H; ++j) {
        double y  = j - 1.0;
        double ux = 4.0*uMax*(y/H - (y*y)/(H*H));
        int id_in  = S.CalculateCellId(0, j);
        S.cells[id_in ]->lattice->velocity_bc = Vector3r(ux, 0.0, 0.0);
```

```
    }

    // Engines (Do not change order!)
    S.engines.clear();
    S.engines.push_back(std::make_shared<FluidCollision>());
    S.engines.push_back(std::make_shared<LatticeSearch>());
    S.engines.push_back(std::make_shared<ImbBoundary>());
    S.engines.push_back(std::make_shared<FluidStreaming>());
    S.engines.push_back(std::make_shared<ZouAndHeBC>());

    // Solver
    while (S.time < 10000.0) {
        if (S.iter % 100 == 0) {
            O.DisplaySimulationInfo();
            O.ExportFluidVtk(file_name);
        }
        S.MoveToNextTimeStep();
    }
    return 0;
}
```

SCENARIO 06: DISCRETE ELEMENT PACKING – UNIFORM

```
#include "scene/Scene.h"
#include "scene/Output.h"

static inline double sqr(double x){ return x*x; }

int main() {
    Timer T;
    Scene& S = Scene::get_Scene();
    Output O;

    // ---------------- Domain ----------------
    double d = 18.0;
    double Rref = 0.5 * d;
    const int Nx = 20*d;
    const int Ny = 10*d + 2;
    S.model_min_corner = {0.0, 0.0, 0.0};
    S.model_max_corner = {double(Nx), double(Ny), 1.0};

    // ---------------- Fluid ----------------
    const double nu = 0.10;
    const double rho_f = 1.0;
    S.dt_lbm = 1.0;
    S.lattice_spacing = 1.0;
    S.initial_density = 1.0;
```

```
S.initial_velocity = Vector3r::Zero();
S.relaxation_time = 3.0 * nu + 0.5;   // tau
S.collision_operator = "BGK";
S.dem_coupling = "IMB";

// Guo forcing
const double g_x = 2e-6;
S.GUO_fluid_forcing = Vector3r(g_x, 0.0, 0.0);
S.gravity           = Vector3r::Zero();

// ---------------- DEM ----------------
S.friction_angle = 30.0;
S.local_damping = 0.2;
S.normal_stiffness = 1.0e3;
S.shear_stiffness = 5.0e2;
S.border_stiffness = 1.0e4;

// ---------------- Grid and walls ----------------
S.AddRectangularCanal();

// Ensure all cells start as fluid (except walls)
for (auto& C : S.cells) {
    C->is_wall  = false;
    C->is_solid = false;
}

// Top and bottom no-slip walls (LBM wall cells)
for (int i = 0; i < Nx; ++i) {
    S.cells[S.CalculateCellId(i, 0)]->is_wall = true;
    S.cells[S.CalculateCellId(i, Ny-1)]->is_wall = true;
}

//---------------- Particle Packing ----------------
double f = 0.1 * d;
double x0 = 5*d + Rref;
double y0 = Rref + f;

int nCols = (int)std::floor((Nx - 2*5*d) / (d + f)) + 1;
int nRows = (int)std::floor((Ny - y0 - Rref) / (d + f)) + 1;

for (int n = 0; n < nCols; ++n) {
    double x = x0 + n * (d + f);

    for (int k = 0; k < nRows; ++k) {
        double y = y0 + k * (d + f);
        S.AddDisk(Vector3r(x, y, 0.0), Rref, 1.0);
    }
}
```

```
    // Block motion (fixed geometry)
    for (auto& B : S.bodies) {
        B->blockedDOFs  = Vector3r::Zero();
        B->blockedMDOFs = 0.0;
    }

    // "Geometric" porosity from disk areas over total domain
    double area_solid = 0.0;
    for (auto& B : S.bodies) area_solid += M_PI * sqr(B->shape->radius);
    const double eps_geom = 1.0 - area_solid / ((Nx - 2*Rref) * (Ny -
2*Rref));

    // Carman–Kozeny-like estimate (approx)
    const double k_th_geom = (eps_geom*eps_geom*eps_geom * d*d) / (180.0 *
sqr(1.0 - eps_geom));

    // ---------------- Engines ----------------
    S.engines.clear();
    S.engines.push_back(std::make_shared<LatticeSearch>());
    S.engines.push_back(std::make_shared<FluidCollision>());
    S.engines.push_back(std::make_shared<ImbBoundary>());
    S.engines.push_back(std::make_shared<ApplyFluidForcing>());
    S.engines.push_back(std::make_shared<FluidStreaming>());

    std::vector<std::shared_ptr<Engine>> dem_engines;
    dem_engines.push_back(std::make_shared<ContactResolution>());
    dem_engines.push_back(std::make_shared<InteractionLoop>());
    dem_engines.push_back(std::make_shared<BodyLoop>());
    dem_engines.push_back(std::make_shared<Integrator>());
    dem_engines.push_back(std::make_shared<UpdateContact>());

    // DEM dt
    const double mass_ref = 2.5 * M_PI * sqr(Rref);
    const double dt_crit  = 0.1 * std::sqrt(mass_ref / S.normal_stiffness);
    const int dem_subcycles = std::max(1, int(S.dt_lbm / dt_crit) + 1);
    S.dt_dem = S.dt_lbm / dem_subcycles;

    // ---------------- Measurement window ----------------
    // Use same region where particles are placed (buffered zone)
    const int i0 = std::max(0,   int(std::floor(90)));
    const int i1 = std::min(Nx-1,int(std::floor(286)));
    const int j0 = std::max(0,   -int(std::floor(Rref)));
    const int j1 = std::min(Ny-1,int(std::floor(180)));

    auto compute_window_stats = [&]() {
        double u_sum = 0.0;
        int n_fluid = 0;
        int n_total = 0;
```

```
        for (int j = j0; j <= j1; ++j){
            for (int i = i0; i <= i1; ++i){
                auto& C = *S.cells[S.CalculateCellId(i,j)];
                if (C.is_wall) continue;
                ++n_total;
                if (C.is_solid) continue;
                u_sum += C.state->vel[0];
                ++n_fluid;
            }
        }

        const double U_pore = (n_fluid > 0) ? (u_sum / n_fluid) : 0.0;
        const double eps_region = (n_total > 0) ?
(double(n_fluid)/double(n_total)) : 0.0;

        return std::make_tuple(U_pore, eps_region, n_fluid, n_total);
    };

    // ---------------- Output ----------------
    std::string fileName = "Uniform_Disks.csv";
    std::ofstream csv(fileName);
    csv << "time,k,k_ergun,k_kozeny,C_eff,Porosity\n";

    const double max_time = 5000.0;
    const int output_interval = 100;
    const int vtk_interval    = 500;

    std::cout << "Permeability scenario (Darcy) with fixed disks\n";
    std::cout << "Window: i=[" << i0 << "," << i1 << "], j=[" << j0 << "," <<
j1 << "]\n";
    std::cout << "eps_geom=" << eps_geom << "  k_th_geom=" << k_th_geom <<
"\n";
    std::cout << "Guo forcing g_x=" << g_x << "  nu=" << nu << "  tau=" <<
S.relaxation_time << "\n";

    // ---------------- Time loop ----------------
    while (S.time < max_time) {

        // LBM step
        for (auto& E : S.engines) E->action();

        // DEM subcycles
        for (int sub = 0; sub < dem_subcycles; ++sub)
            for (auto& E : dem_engines) E->action();

        // Output
        if (S.iter % output_interval == 0) {
            const double mu = rho_f * nu;
```

```
            auto [U_pore, eps_region, n_fluid, n_total] =
compute_window_stats();

            //Kozeny-Carman equation
            double porosity = eps_region;
            const double k_kozeny = (porosity*porosity*porosity * d*d) /
(180.0 * sqr(1.0 - porosity));
            const double k_ergun = (porosity*porosity*porosity * d*d) / (150 *
sqr(1.0 - porosity));

            // Superficial (Darcy) velocity:
            const double U_super = eps_region * U_pore;

            // Darcy permeability
            const double k = (g_x != 0.0) ? (mu * U_super / (rho_f * g_x)) :
0.0;

            // Effective Kozeny constant inferred from the measured
permeability k
            const double C_eff = (porosity*porosity*porosity * d*d) / (k *
sqr(1.0 - porosity));

            // Kozeny-Carman coefficient calibrated against LBM results
            const double k_kozeny_cal = (porosity*porosity*porosity * d*d) /
(C_eff * sqr(1.0 - porosity));

            // Reynolds number
            const double Re_pore  = (nu != 0.0) ? (U_pore  * d / nu) : 0.0;
            const double Re_super = (nu != 0.0) ? (U_super * d / nu) : 0.0;

            csv << S.time << ","
            << k << ","
            << k_ergun << ","
            << k_kozeny << ","
            << C_eff << ","
            << eps_region << "\n";

            std::cout << "t=" << std::setw(7) << std::fixed <<
std::setprecision(0) << S.time
                      << "  U_pore="  << std::scientific <<
std::setprecision(3) << U_pore
                      << "  U_super=" << U_super
                      << "  k="       << k
                      << "  k_kozeny=" << k_kozeny
                      << "  k_ergun=" << k_ergun
                      << "  C_eff=" << C_eff
                      << "  eps_reg=" << std::fixed << std::setprecision(4) <<
eps_region
                      << "  porosity=" << porosity
```

```
                      << "  Re_pore=" << std::scientific <<
std::setprecision(3) << Re_pore
                      << std::endl;
        }

        if (S.iter % vtk_interval == 0) {
            O.ExportFluidVtk("(UD)Permeability_Fluid_");
            O.ExportParticleVtk("(UD)Permeability_Particles_");
        }

        S.time += S.dt_lbm;
        ++S.iter;
    }

    csv.close();
    return 0;
}
```

SCENARIO 06: DISCRETE ELEMENT PACKING – RANDOM

```
#include "scene/Scene.h"
#include "scene/Output.h"

static inline double sqr(double x){ return x*x; }
namespace fs = std::filesystem;

int main() {
    Timer T;
    Scene& S = Scene::get_Scene();
    Output O;

    // ---------------- Domain ----------------
    const int Nx = 200;
    const int Ny = 120;
    S.model_min_corner = {0.0, 0.0, 0.0};
    S.model_max_corner = {double(Nx), double(Ny), 1.0};

    // ---------------- Particle Packing ----------------


    // ---------------- Fluid ----------------
    const double nu = 0.10;
    const double rho_f = 1.0;
    S.dt_lbm = 1.0;
```

```
    S.lattice_spacing = 1.0;
    S.initial_density = 1.0;
    S.initial_velocity = Vector3r::Zero();
    S.relaxation_time = 3.0 * nu + 0.5;   // tau
    S.collision_operator = "BGK";
    S.dem_coupling = "IMB";

    // Guo forcing
    const double g_x = 2e-6;
    S.GUO_fluid_forcing = Vector3r(g_x, 0.0, 0.0);
    S.gravity           = Vector3r::Zero();

    // ---------------- DEM ----------------
    S.friction_angle   = 30.0;
    S.local_damping    = 0.2;
    S.normal_stiffness = 1.0e3;
    S.shear_stiffness  = 5.0e2;
    S.border_stiffness = 1.0e4;

    // ---------------- Grid and walls ----------------
    S.AddRectangularCanal();

    // Ensure all cells start as fluid (except walls you set)
    for (auto& C : S.cells) {
        C->is_wall  = false;
        C->is_solid = false;
    }

    // Top and bottom no-slip walls (LBM wall cells)
    for (int i = 0; i < Nx; ++i) {
        S.cells[S.CalculateCellId(i, 0    )]->is_wall = true;
        S.cells[S.CalculateCellId(i, Ny-1)]->is_wall = true;
    }

    //---------------- Particle Packing ----------------
    double Rmin = 3.0;
    double Rmax = 6.0;
    double wall_gap = 0.1;
    double inlet_buf = 0.08 * Nx;
    double outlet_buf = 0.08 * Nx;
    double non_overlap = 0.98;
    int max_particles = 800;
    int wall_particles = 160;
    double rho_p = 2.5;
    double Rw_min = 0.5 * Rmin;
    double Rw_max = 0.5 * Rmax;
    double x_min_p = inlet_buf  + Rmax + wall_gap;
    double x_max_p = Nx - outlet_buf - Rmax - wall_gap;
    double y_min_p = Rmax + wall_gap;
```

```
    double y_max_p = Ny - Rmax - wall_gap;
    S.addDiskPacking(Rmin, Rmax, wall_gap, inlet_buf, outlet_buf, non_overlap,
max_particles, wall_particles, rho_p, Rw_min, Rw_max);

    // Block motion (fixed geometry)
    for (auto& B : S.bodies) {
        B->blockedDOFs  = Vector3r::Zero();
        B->blockedMDOFs = 0.0;
    }

    // Calculate representative diameter
    double sum_r2 = 0.0;
    double sum_r  = 0.0;
    for (auto& Bp : S.bodies){
        double r = Bp->shape->radius;
        sum_r  += r;
        sum_r2 += r*r;
    }
    double d = 2.0 * (sum_r2 / sum_r);
    double Rref = 0.5 * d;

    // ---------------- Engines ----------------
    S.engines.clear();
    S.engines.push_back(std::make_shared<LatticeSearch>());
    S.engines.push_back(std::make_shared<FluidCollision>());
    S.engines.push_back(std::make_shared<ImbBoundary>());
    S.engines.push_back(std::make_shared<ApplyFluidForcing>());
    S.engines.push_back(std::make_shared<FluidStreaming>());

    std::vector<std::shared_ptr<Engine>> dem_engines;
    dem_engines.push_back(std::make_shared<ContactResolution>());
    dem_engines.push_back(std::make_shared<InteractionLoop>());
    dem_engines.push_back(std::make_shared<BodyLoop>());
    dem_engines.push_back(std::make_shared<Integrator>());
    dem_engines.push_back(std::make_shared<UpdateContact>());

    // DEM dt (bodies fixed, but keep stable)
    const double mass_ref = 2.5 * M_PI * sqr(Rref);
    const double dt_crit  = 0.1 * std::sqrt(mass_ref / S.normal_stiffness);
    const int dem_subcycles = std::max(1, int(S.dt_lbm / dt_crit) + 1);
    S.dt_dem = S.dt_lbm / dem_subcycles;

    // ---------------- Measurement window ----------------
    // Use same region where particles are placed (buffered zone)
    const int i0 = std::max(0,   int(std::floor(x_min_p)));
    const int i1 = std::min(Nx-1,int(std::floor(x_max_p)));
    const int j0 = std::max(0,   int(std::floor(y_min_p)));
    const int j1 = std::min(Ny-1,int(std::floor(y_max_p)));
```

```
    auto compute_window_stats = [&]() {
        double u_sum = 0.0;
        int n_fluid = 0;
        int n_total = 0;

        for (int j = j0; j <= j1; ++j){
            for (int i = i0; i <= i1; ++i){
                auto& C = *S.cells[S.CalculateCellId(i,j)];
                if (C.is_wall) continue;
                ++n_total;
                if (C.is_solid) continue;
                u_sum += C.state->vel[0];
                ++n_fluid;
            }
        }

        const double U_pore = (n_fluid > 0) ? (u_sum / n_fluid) : 0.0;
        const double eps_region = (n_total > 0) ?
(double(n_fluid)/double(n_total)) : 0.0;

        return std::make_tuple(U_pore, eps_region, n_fluid, n_total);
    };

    // ---------------- Output ----------------
    std::string fileName = "Random_Disks.csv";
    std::ofstream csv(fileName);
    csv << "time,k,k_ergun,k_kozeny,C_eff,Porosity\n";

    const double max_time = 5000.0;
    const int output_interval = 100;
    const int vtk_interval    = 500;

    std::cout << "Permeability scenario (Darcy) with fixed disks\n";
    std::cout << "Window: i=[" << i0 << "," << i1 << "], j=[" << j0 << "," <<
j1 << "]\n";
    std::cout << "Guo forcing g_x=" << g_x << "  nu=" << nu << "  tau=" <<
S.relaxation_time << "\n";

    // ---------------- Time loop ----------------
    while (S.time < max_time) {

        // LBM step
        for (auto& E : S.engines) E->action();

        // DEM subcycles
        for (int sub = 0; sub < dem_subcycles; ++sub)
            for (auto& E : dem_engines) E->action();

        // Output
```

```
        if (S.iter % output_interval == 0) {
            const double mu = rho_f * nu;

            auto [U_pore, eps_region, n_fluid, n_total] =
compute_window_stats();

            //Kozeny-Carman equation
            double porosity = eps_region;
            const double k_kozeny = (porosity*porosity*porosity * d*d) /
(180.0 * sqr(1.0 - porosity));
            const double k_ergun = (porosity*porosity*porosity * d*d) / (150 *
sqr(1.0 - porosity));

            // Superficial (Darcy) velocity:
            const double U_super = eps_region * U_pore;

            // dp/dx = -rho*g_x  =>  U_super = (k/mu)*(rho*g_x)
            const double k = (g_x != 0.0) ? (mu * U_super / (rho_f * g_x)) :
0.0;

            // Effective Kozeny constant inferred from the measured
permeability k
            const double C_eff = (porosity*porosity*porosity * d*d) / (k *
sqr(1.0 - porosity));

            // Reynolds number
            const double Re_pore  = (nu != 0.0) ? (U_pore  * d / nu) : 0.0;
            const double Re_super = (nu != 0.0) ? (U_super * d / nu) : 0.0;

            csv << S.time << ","
                << k << ","
                << k_ergun << ","
                << k_kozeny << ","
                << C_eff << ","
                << eps_region << "\n";


            std::cout << "t=" << std::setw(7) << std::fixed <<
std::setprecision(0) << S.time
                      << "  U_pore="  << std::scientific <<
std::setprecision(3) << U_pore
                      << "  U_super=" << U_super
                      << "  k="       << k
                      << "  k_kozeny=" << k_kozeny
                      << "  k_ergun=" << k_ergun
                      << "  C_eff=" << C_eff
                      << "  eps_reg=" << std::fixed << std::setprecision(4) <<
eps_region
                      << "  porosity=" << porosity
```

```
                      << "  Re_pore=" << std::scientific <<
std::setprecision(3) << Re_pore
                      << std::endl;
        }

        if (S.iter % vtk_interval == 0) {
            O.ExportFluidVtk("(RD)Permeability_Fluid_");
            O.ExportParticleVtk("(RD)Permeability_Particles_");
        }

        S.time += S.dt_lbm;
        ++S.iter;
    }

    csv.close();
    return 0;
}
```

SCENARIO 06: DISCRETE ELEMENT PACKING – REAL PARTICLE

```
#include "scene/Scene.h"
#include "scene/Output.h"

static inline double sqr(double x){ return x*x; }

int main() {
    Timer T;
    Scene& S = Scene::get_Scene();
    Output O;

    // ---------------- Domain ----------------
    double d = 20.0;
    double Rref = 0.5 * d;
    const int Nx = 20*d;
    const int Ny = 10*d + 2;
    S.model_min_corner = {0.0, 0.0, 0.0};
    S.model_max_corner = {double(Nx), double(Ny), 1.0};

    // ---------------- Fluid ----------------
    const double nu = 0.10;
    const double rho_f = 1.0;
    S.dt_lbm = 1.0;
    S.lattice_spacing = 1.0;
    S.initial_density = 1.0;
    S.initial_velocity = Vector3r::Zero();
```

```
    S.relaxation_time = 3.0 * nu + 0.5;   // tau
    S.collision_operator = "BGK";
    S.dem_coupling = "IMB";

    // Guo forcing (interpreted as acceleration g_x)
    const double g_x = 2e-6;
    S.GUO_fluid_forcing = Vector3r(g_x, 0.0, 0.0);
    S.gravity           = Vector3r::Zero();

    // ---------------- DEM (disks fixed) ----------------
    S.friction_angle   = 30.0;
    S.local_damping    = 0.2;
    S.normal_stiffness = 1.0e3;
    S.shear_stiffness  = 5.0e2;
    S.border_stiffness = 1.0e4;

    // ---------------- Grid and walls ----------------
    S.AddRectangularCanal();

    // Ensure all cells start as fluid (except walls you set)
    for (auto& C : S.cells) {
        C->is_wall  = false;
        C->is_solid = false;
    }

    // Top and bottom no-slip walls
    for (int i = 0; i < Nx; ++i) {
        S.cells[S.CalculateCellId(i, 0    )]->is_wall = true;
        S.cells[S.CalculateCellId(i, Ny-1)]->is_wall = true;
    }

    //---------------- Particle Packing ----------------
    std::string fileName;
    int numberOfParticles = 39;
    for(int i = 0; i < numberOfParticles; ++i){
        fileName = "particles/particles_displaced_d_20/" + std::to_string(i) +
".txt"; //Path para adicionar particulas
        S.addParticle(fileName, 1, Vector3r(0,0,0));
    }

    // Block motion (fixed geometry)
    for (auto& B : S.bodies) {
        B->blockedDOFs  = Vector3r::Zero();
        B->blockedMDOFs = 0.0;
    }

    // "Geometric" porosity from disk areas over total domain
    double area_solid = 0.0;
    for (auto& B : S.bodies) area_solid += M_PI * sqr(B->shape->radius);
```

```
    const double eps_geom = 1.0 - area_solid / ((Nx - 2*Rref) * (Ny -
2*Rref));

    // Carman–Kozeny-like estimate (approx)
    const double k_th_geom = (eps_geom*eps_geom*eps_geom * d*d) / (180.0 *
sqr(1.0 - eps_geom));

    // ---------------- Engines ----------------
    S.engines.clear();
    S.engines.push_back(std::make_shared<LatticeSearch>());
    S.engines.push_back(std::make_shared<FluidCollision>());
    S.engines.push_back(std::make_shared<ImbBoundary>());
    S.engines.push_back(std::make_shared<ApplyFluidForcing>());
    S.engines.push_back(std::make_shared<FluidStreaming>());

    std::vector<std::shared_ptr<Engine>> dem_engines;
    dem_engines.push_back(std::make_shared<ContactResolution>());
    dem_engines.push_back(std::make_shared<InteractionLoop>());
    dem_engines.push_back(std::make_shared<BodyLoop>());
    dem_engines.push_back(std::make_shared<Integrator>());
    dem_engines.push_back(std::make_shared<UpdateContact>());

    // DEM dt
    const double mass_ref = 2.5 * M_PI * sqr(Rref);
    const double dt_crit  = 0.1 * std::sqrt(mass_ref / S.normal_stiffness);
    const int dem_subcycles = std::max(1, int(S.dt_lbm / dt_crit) + 1);
    S.dt_dem = S.dt_lbm / dem_subcycles;

    // ---------------- Measurement window ----------------
    // Use same region where particles are placed (buffered zone)
    const int i0 = std::max(0,   int(std::floor(106)));
    const int i1 = std::min(Nx-1,int(std::floor(293)));
    const int j0 = std::max(0,   -int(std::floor(Rref)));
    const int j1 = std::min(Ny-1,int(std::floor(200)));

    auto compute_window_stats = [&]() {
        double u_sum = 0.0;
        int n_fluid = 0;
        int n_total = 0;

        for (int j = j0; j <= j1; ++j){
            for (int i = i0; i <= i1; ++i){
                auto& C = *S.cells[S.CalculateCellId(i,j)];
                if (C.is_wall) continue;
                ++n_total;
                if (C.is_solid) continue;
                u_sum += C.state->vel[0];
                ++n_fluid;
            }
```

```
        }

        const double U_pore = (n_fluid > 0) ? (u_sum / n_fluid) : 0.0;
        const double eps_region = (n_total > 0) ?
(double(n_fluid)/double(n_total)) : 0.0;

        return std::make_tuple(U_pore, eps_region, n_fluid, n_total);
    };

    // ---------------- Output ----------------
    std::string file_name = "Real_Particle_Packing.csv";
    std::ofstream csv(file_name);
    csv << "time,k,k_ergun,k_kozeny,C_eff,Porosity\n";

    const double max_time = 10000.0;
    const int output_interval = 100;
    const int vtk_interval    = 500;

    std::cout << "Permeability scenario (Darcy) with fixed disks\n";
    std::cout << "Window: i=[" << i0 << "," << i1 << "], j=[" << j0 << "," <<
j1 << "]\n";
    std::cout << "eps_geom=" << eps_geom << "  k_th_geom=" << k_th_geom <<
"\n";
    std::cout << "Guo forcing g_x=" << g_x << "  nu=" << nu << "  tau=" <<
S.relaxation_time << "\n";

    // ---------------- Time loop ----------------
    while (S.time < max_time) {

        // LBM step
        for (auto& E : S.engines) E->action();

        // DEM subcycles
        for (int sub = 0; sub < dem_subcycles; ++sub)
            for (auto& E : dem_engines) E->action();

        // Output
        if (S.iter % output_interval == 0) {
            const double mu = rho_f * nu;

            auto [U_pore, eps_region, n_fluid, n_total] =
compute_window_stats();

            //Kozeny-Carman equation
            double porosity = eps_region;
            const double k_kozeny = (porosity*porosity*porosity * d*d) /
(180.0 * sqr(1.0 - porosity));
            const double k_ergun = (porosity*porosity*porosity * d*d) / (150 *
sqr(1.0 - porosity));
```

```
            // Superficial (Darcy) velocity:
            const double U_super = eps_region * U_pore;


            // Darcy permeability (treat GUO forcing as acceleration g_x):
            // dp/dx = -rho*g_x  =>  U_super = (k/mu)*(rho*g_x)
            const double k = (g_x != 0.0) ? (mu * U_super / (rho_f * g_x)) :
0.0;

            // Effective Kozeny constant inferred from the measured
permeability k
            const double C_eff = (porosity*porosity*porosity * d*d) / (k *
sqr(1.0 - porosity));

            // Kozeny-Carman coefficient calibrated against LBM results
            const double k_kozeny_cal = (porosity*porosity*porosity * d*d) /
(C_eff * sqr(1.0 - porosity));

            // Reynolds number
            const double Re_pore  = (nu != 0.0) ? (U_pore  * d / nu) : 0.0;
            const double Re_super = (nu != 0.0) ? (U_super * d / nu) : 0.0;

            csv << S.time << ","
                << k << ","
                << k_ergun << ","
                << k_kozeny << ","
                << C_eff << ","
                << eps_region << "\n";

            std::cout << "t=" << std::setw(7) << std::fixed <<
std::setprecision(0) << S.time
                      << "  U_pore="  << std::scientific <<
std::setprecision(3) << U_pore
                      << "  U_super=" << U_super
                      << "  k="       << k
                      << "  k_kozeny=" << k_kozeny
                      << "  k_kozeny_cal=" << k_kozeny_cal
                      << "  C_eff=" << C_eff
                      << "  eps_reg=" << std::fixed << std::setprecision(4) <<
eps_region
                      << "  porosity=" << porosity
                      << "  Re_pore=" << std::scientific <<
std::setprecision(3) << Re_pore
                      << std::endl;
        }

        if (S.iter % vtk_interval == 0) {
            O.ExportFluidVtk("(RPP)Permeability_Fluid_");
```

```
            O.ExportParticleVtk("(RPP)Permeability_Particles_");
        }

        S.time += S.dt_lbm;
        ++S.iter;
    }

    csv.close();
    std::cout << "Finished. Results: permeability_results.csv\n";
    return 0;
}
```

SCENARIO 07: PARTICLE SETTLING – STOKES' LAW

```
#include "scene/Scene.h"
#include "scene/Output.h"

// ---------------- Helpers ----------------
static void BuildDiskPolygon(std::shared_ptr<Body>& B, double R, double dtheta
= 0.1) {
    // Safer than cloud.clear()
    B->shape->cloud.outer().clear();
    B->shape->cloud.inners().clear();

    for (double theta = 0.0; theta < 2.0 * M_PI; theta += dtheta) {
        double x = B->state->pos[0] + R * std::cos(theta);
        double y = B->state->pos[1] + R * std::sin(theta);
        boost::geometry::append(B->shape->cloud.outer(), point(x, y));
    }
    boost::geometry::correct(B->shape->cloud);
}

static double Mean(const std::vector<double>& v) {
    if (v.empty()) return 0.0;
    return std::accumulate(v.begin(), v.end(), 0.0) / (double)v.size();
}

static double StdDev(const std::vector<double>& v, double m) {
    if (v.size() < 2) return 0.0;
```

```
    double acc = 0.0;
    for (double x : v) {
        double d = x - m;
        acc += d * d;
    }
    return std::sqrt(acc / (double)(v.size() - 1));
}

// ---------------- Main ----------------
int main() {
    Timer T;
    Scene& S = Scene::get_Scene();
    Output O;

    // ---------- Geometry (set Ly manually for each run) ----------
    const double R = 6.0;
    const double D = 2.0 * R;

    const double Lx = 100.0 * R;
    const double Ly = 40.0 * R;   // <-- mude aqui (20R, 40R, 80R, 160R, ...)

    S.model_min_corner = {0.0, 0.0, 0.0};
    S.model_max_corner = {Lx, Ly, 1.0};

    // ---------- Fluid & Particle Parameters ----------
    const double rho_f = 1.0;
    const double nu = 0.10;
    const double mu = rho_f * nu;
    const double rho_p = 1.20;
    const double delta_rho = rho_p - rho_f;

    // Stokes Parameters
    const double g = 5.555555555555557e-05;
    const double F_net = delta_rho * M_PI * R * R * g;
    const double settling_velocity_3D = 2 * R * R * g * delta_rho / (9 * mu);
    const double settling_velocity_2D = R * g * delta_rho / (6 * mu);

    // ---------- LBM Parameters ----------
    S.dt_lbm = 1.0;
    S.lattice_spacing = 1.0;
    S.initial_density = rho_f;
    S.initial_velocity = Vector3r::Zero();
    S.relaxation_time = 3.0 * nu + 0.5;
    S.collision_operator = "MRT";
    S.dem_coupling  = "IMB";
    S.GUO_fluid_forcing = Vector3r::Zero();
    double s8 = 2/(1 + 6*nu);
    S.SetRelaxationParamters(0, 1.4, 1.4, 0.75, 1.2, 1, 1.2, s8, s8);
```

```
    // ---------- DEM Parameters ----------
    const double mass = rho_p * M_PI * R * R;
    S.friction_angle = 30.0;
    S.local_damping = 0.0;
    S.normal_stiffness = 5.0e2;
    S.shear_stiffness = 2.5e2;
    S.border_stiffness = 5.0e3;
    S.gravity = Vector3r(0.0, -g, 0.0);

    // DEM subcycling
    const double dt_crit = 0.2 * std::sqrt(mass / S.normal_stiffness);
    const int dem_subcycles = std::max(1, (int)(S.dt_lbm / dt_crit) + 1);
    S.dt_dem = S.dt_lbm / dem_subcycles;

    // ---------- Grid ----------
    S.AddRectangularCanal();
    for (auto& C : S.cells) {
        C->set_initial_condition();
    }

    // ---------- Particle ----------
    Vector3r pos0(Lx * 0.5, Ly * 0.75, 0.0);
    S.AddDisk(pos0, R, rho_p);
    auto& B = S.bodies[0];

    B->state->vel    = Vector3r::Zero();
    B->state->rotVel = 0.0;

    BuildDiskPolygon(B, R);

    // ---------- Engines ----------
    S.engines.clear();
    S.engines.push_back(std::make_shared<LatticeSearch>());
    S.engines.push_back(std::make_shared<ImbBoundary>());
    S.engines.push_back(std::make_shared<FluidCollision>());
    S.engines.push_back(std::make_shared<FluidStreaming>());

    std::vector<std::shared_ptr<Engine>> dem_engines;
    dem_engines.push_back(std::make_shared<ContactResolution>());
    dem_engines.push_back(std::make_shared<InteractionLoop>());
    dem_engines.push_back(std::make_shared<BodyLoop>());
    dem_engines.push_back(std::make_shared<Integrator>());
    dem_engines.push_back(std::make_shared<UpdateContact>());

    // ---------- Output ----------
    std::ostringstream fname;
    fname << "stokes_2D_Ly_" << (int)Ly << "_R_" << (int)R << ".csv";
    std::ofstream csv(fname.str());
    csv << "time,y,vy,Re,Fy_hydro,Fy_net,F_net,res_force\n";
```

```
    // ---------- Robust terminal extraction ----------
    const int output_interval = 200;
    const int vtk_interval    = 200;
    const double max_time     = 60000.0;

    const int N_tail = 30;
    std::vector<double> vy_tail;
    vy_tail.reserve(N_tail);

    const double stop_y = 0.25 * Ly;
    const double start_collect_time = 8000.0;

    while (S.time < max_time) {

        BuildDiskPolygon(B, R);

        // LBM
        for (auto& E : S.engines) E->action();

        // DEM
        for (int sub = 0; sub < dem_subcycles; ++sub) {
            for (auto& E : dem_engines) E->action();
        }

        if (S.iter % output_interval == 0) {
            const double vy = -B->state->vel[1]; // Settling velocity
            const double Re = vy * D / nu;

            const double Fy_hydro = B->state->hydro_force[1];
            const double Fy_net   = B->state->force[1];

            double res_force = abs(Fy_hydro - F_net)/F_net;

            csv << S.time << ","
                << B->state->pos[1] << ","
                << vy << ","
                << Re << ","
                << Fy_hydro << ","
                << Fy_net << ","
                << F_net << ","
                << res_force << "\n";

            std::cout << "t=" << std::setw(7) << std::fixed <<
std::setprecision(0) << S.time
                      << "  y=" << std::setw(8) << std::setprecision(2) << B-
>state->pos[1]
                      << "  vy=" << std::scientific << std::setprecision(3) <<
vy
```

```
                      << "  vy_theory_2D=" << std::scientific <<
std::setprecision(3) << settling_velocity_2D
                      << "  vy_theory_3D=" << std::scientific <<
std::setprecision(3) << settling_velocity_3D
                      << "  Re=" << std::fixed << std::setprecision(4) << Re
                      << "  2d ratio=" << std::scientific <<
std::setprecision(3) << (vy / settling_velocity_2D)
                      << "  3d ratio=" << std::scientific <<
std::setprecision(3) << (vy / settling_velocity_3D)
                      << "  F_hydro_lbm=" << std::scientific << Fy_hydro
                      << "  F_net=" << std::scientific << F_net
                      << "  res_force=" << std::scientific << res_force
                      << std::endl;

            if (S.time >= start_collect_time) {
                if ((int)vy_tail.size() < N_tail) {
                    vy_tail.push_back(vy);
                } else {
                    vy_tail.erase(vy_tail.begin());
                    vy_tail.push_back(vy);
                }

                if ((int)vy_tail.size() == N_tail) {
                    const double m = Mean(vy_tail);
                    const double s = StdDev(vy_tail, m);
                    const double cv = (std::abs(m) > 1e-30) ? (s /
std::abs(m)) : 0.0;

                    if (cv < 0.01) { // 1%
                        std::cout << "\n*** Terminal regime (robust) detected
***\n"
                                  << "Ut_mean = " << std::scientific << m
                                  << "  Ut_std = " << s
                                  << "  (std/mean=" << std::fixed <<
std::setprecision(2) << (100.0*cv) << "%)\n\n";
                        break;
                    }
                }
            }
        }

        if (S.iter % vtk_interval == 0) {
            O.ExportFluidVtk("Stokes2D_Fluid_"+std::to_string(int(Ly))+"_");
            O.ExportParticleVtk("Stokes2D_Particle_"+std::to_string(int(Ly))+"
_");
        }

        if (B->state->pos[1] < stop_y) {
```

```
            std::cout << "\nParticle reached stop_y (avoid bottom effects).
Stopping.\n\n";
            break;
        }

        S.time += S.dt_lbm;
        ++S.iter;
    }

    csv.close();

    // ---------- Final robust metrics ----------
    if (!vy_tail.empty()) {
        const double m = Mean(vy_tail);
        const double s = StdDev(vy_tail, m);
        const double cv = (std::abs(m) > 1e-30) ? (s / std::abs(m)) : 0.0;

        std::cout << "\n===== Robust 2D summary =====\n";
        std::cout << "Ly = " << Ly << "   (Ly/D=" << (Ly/D) << ", ln(Ly/D)="
<< std::log(Ly/D) << ")\n";
        std::cout << "Ut_mean = " << std::scientific << m << "\n";
        std::cout << "Ut_std  = " << s << "  (std/mean=" << std::fixed <<
std::setprecision(2) << (100.0*cv) << "%)\n";
        std::cout << "F_net   = " << std::scientific << F_net << "\n";
        std::cout << "==============================\n\n";
    } else {
        std::cout << "\nNo tail samples collected (increase max_time or lower
start_collect_time).\n";
    }

    return 0;
}
```