UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE ENGENHARIA MECÂNICA
GRADUAÇÃO EM ENGENHARIA AERONÁUTICA

LUCAS LUTFFALLA ESTEVAM

MÉTODO LATTICE-BOLTZMANN APLICADO A ESCOAMENTOS EXTERNOS

UBERLÂNDIA - MG

2020

Lucas Lutffalla Estevam

MÉTODO LATTICE-BOLTZMANN APLICADO A ESCOAMENTOS EXTERNOS

Trabalho de Conclusão de Curso submetido à Faculdade de Engenharia Mecânica da Universidade Federal de Uberlândia, como parte das exigências para conclusão do curso de Graduação em Engenharia Aeronáutica.

Orientador: Prof. Dr. Francisco José de Souza

Uberlândia - MG, novembro de 2020

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Lucas Lutffalla Estevam

Esta Monografia foi julgada adequada para a obtenção do título de Bacharel em Engenharia Aeronáutica, sendo aprovada em sua forma final pela banca examinadora:

Orientador: Prof. Dr. Francisco José de Souza
Universidade Federal de Uberlândia - UFU

Prof. Dr. João Rodrigo Andrade
Universidade Federal de Uberlândia - UFU

Engº Pedro Paulo de Carvalho Brito

Uberlândia - MG, novembro de 2020

AGRADECIMENTOS

Em primeiro lugar, agradeço a minha família, especialmente a meus pais, Cristiane e Marcos, pelo privilégio da educação e pelo constante incentivo e apoio nessa caminhada, não medindo esforços para que eu persistisse com os estudos.

Sou grato também a todas as amizades que tive o prazer de cultivar no decorrer da graduação e que foram de enorme importância para a persistência e motivação em Uberlândia. Agradeço especialmente aos amigos Daniel e Dimas, que estiveram comigo ao longo de todo o percurso.

Sou grato também aos projetos de extensão dos quais pude fazer parte, oportunidades nas quais mais me desenvolvi, pessoal e profissionalmente. Em especial, agradeço a equipe Tucano Aerodesign, pelos conhecimentos trocados, pelos momentos inesquecíveis e, sobretudo, pelas amizades que começaram ali.

Agradeço também ao Profº Francisco, pela paciência e compreensão durante o desenvolvimento deste trabalho e por todo o conhecimento compartilhado comigo.

Por fim, gostaria de agradecer a Universidade Federal de Uberlândia e a Faculdade de Engenharia Mecânica, pela oportunidade de cursar Engenharia Aeronáutica e por todas as oportunidades de aprendizado ao longo desse período.

RESUMO

Em um contexto no qual o poder de processamento cresce de maneira consistente, persiste a busca por métodos numéricos mais eficientes capazes de melhor aproveitar tais recursos e que levem a soluções de engenhaira mais competitivas. Nessa conjuntura, o presente trabalho explora, dentro da Dinâmica dos Fluidos Computacional, o método Lattice-Boltzmann (LBM), visto como uma ferramenta promissora para análise de escoamentos incompressíveis devido a sua simplicidade de implementação e possibilidade extensiva de uso da computação paralelizada. Inicialmente, uma revisão bibliográfica de trabalhos pioneiros e recentes a respeito da solução de escoamentos ao redor de cilindros retangulares via LBM foi realizada, visando melhor compreender o nível de concordância dos resultados com métodos já bem estabelecidos no meio acadêmico e na indústria. Em seguida, o código implementado na linguagem $C++$ foi testado para o caso padrão de escoamento dentro de uma cavidade, obtendo concordância satisfatória. Finalmente, parâmetros como coeficiente de pressão, coeficiente de arrasto e número de Strouhal obtidos após simulação de um cilindro retangular exposto ao escoamento livre são comparados com resultados publicados obtidos via método de elementos finitos e com gerados via LBM pelo *openLB*, um software de código livre.

Palavras-chave: Método Lattice-Boltzmann, CFD, C++, openLB.

ABSTRACT

In the context which processing power grows consistently, the search for more efficient numerical methods, able to take advantage of such resources and that leads to more competitive engineering solutions persists. In this sense, the present work explores, in the scope of Computational Fluid Dynamics, the Lattice Boltzmann Method (LBM), currently seen as a promising tool for the analysis of incompressible flows given its ease of implementation and possibility of extensive use of parallelized computing. Firstly, a literature review of pioneer and recent works regarding the solution of flow around a square cylinder through LBM was carried out aiming to better understand the level of agreement with well stablished methods in the academia and industry. Then, the C++ developed algorithm was benchmarked with a lid driven cavity flow, obtaining a satisfactory agreement. Finally, parameters such as pressure coefficient, drag coefficient and the Strouhal number obtained by the simulation of a square cylinder in free flow were compared to published results obtained through finite element method and also via LBM, calculated using the open source software openLB.

Keywords: Lattice-Boltzmann Method, CFD, C++, openLB.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

BGK - Operador de Bhatnagar-Gross-Krook

CFD - Dinâmica dos Fluidos Computacional

c_s - Velocidade do som em unidades de lattice

DM - Dinâmica Molecular

f - Função de Distribuição

f^{eq} - Função de Distribuição no Equilíbrio

FDP - Função Densidade de Probabilidade

LBM - Método Lattice-Boltzmann

LES - *Large Eddy Simulation*, Simulação de Grandes Escalas

LGA - *Lattice Gas Automatta*

MEA - Algoritmo *momentum exchange*

MRT - *Multiple Relaxation Time*, tempo de relaxação múltiplo

MVF - Método de Volumes Finitos

TRT - *Two-relaxation time*, tempo de relaxação duplo

u - Velocidade de entrada do escoamento

α - Viscosidade em unidades de lattice

Ω - Operador de Colisão

τ - Tempo de Relaxação

ω - Frequência de Relaxação

SUMÁRIO

1 INTRODUÇÃO

Métodos computacionais, desde meados do século XX, têm se mostrado complementos poderosos aos procedimentos experimentais para a compreensão de fenômenos físicos. De maneira mais específica, atualmente a Fluido Dinâmica Computacional (CFD) ocupa papel essencial no desenvolvimento de novas soluções de engenharia, desde turbinas eólicas para geração de energia renovável até aeronaves mais eficientes para transporte de carga e de passageiros.

Além da aplicação prática, a Dinâmica dos Fluidos Computacional também se mostra como ferramenta importante para a investigação científica, permitindo a observação de fenômenos e obtenção de dados que se mostrariam impraticáveis em uma abordagem puramente experimental.

Dentro desse contexto, do ponto de vista matemático, um escoamento pode ser descrito em três níveis diferentes: microscópico, mesoscópico e macroscópico, vide Figura 1. O primeiro descarta a hipótese do contínuo e modela o fluido de modo discreto, descrevendo individualmente o movimento de cada uma das partículas que o compõe, exigindo, em contrapartida, um elevado e usualmente proibitivo poder de processamento. A abordagem macro, por sua vez, largamente utilizada na indústria, aplica as leis de conservação de massa, quantidade de movimento e energia em um dado volume de controle para estabelecer um conjunto de equações diferenciais parciais que governam o escoamento. No meio termo entre as duas escalas, a abordagem mesoscópica se utiliza da teoria cinética dos gases para descrever a microdinâmica de partículas fictícias através de modelos cinéticos simplificados, e surge como alternativa à solução de alguns problemas inerentes da modelagem contínua.

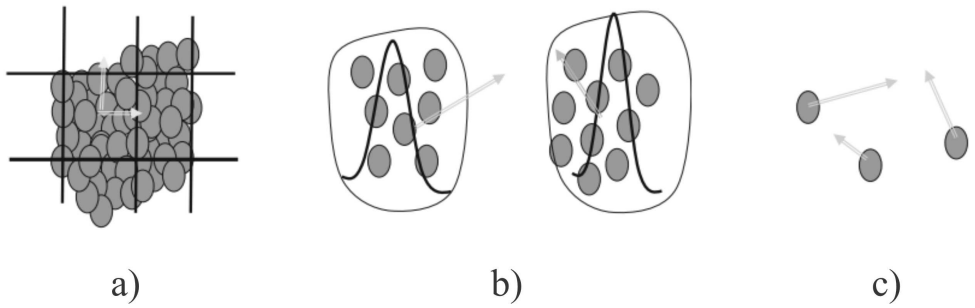

Figura 1 – Representação dos níveis de escala de um escoamento: (a) nível microscópico; (b) nível mesoscópico (método LB); (c) nível microscópico (modificado pelo autor a partir de [Mohamad 2011])

Um dos principais problemas dos métodos regularmente empregados em CFD advém da falta de uma base matemática sólida, uma vez que foram derivados nas décadas de 80-90 com base nas equações de Navier-Stokes [Brito 2017]. Uma consequência do

desenvolvimento quase que empírico de tais métodos por engenheiros que buscavam a solução prática de problemas foi o aumento da complexidade dos algoritmos, em razão do elevado grau de não-linearidade e do alto número de graus de liberdade de cada problema. Dessa forma, mesmo metodologias mais simples são ineficientes em razão do chamado acoplamento pressão-velocidade, originado das técnicas numéricas aplicadas para se obter uma aproximação da solução das equações de Navier-Stokes.

Ademais, na resolução de problemas em que a solução é da mesma ordem que a distância média das partículas do fluido, como um regime de voo hipersônico, a hipótese do contínuo, adotada nas técnicas mais tradicionais da Dinâmica dos Fluidos Computacional durante a derivação das equações de Navier-Stokes, pode levar a resultados não-realistas, em razão das descontinuidades que ocorrem, por exemplo, na formação de uma onda de choque.

Nesse contexto, o método Lattice-Boltzmann surge como uma alternativa à resolução das equações de Navier-Stokes [Succi 2001], atuando na escala mesoscópica. Historicamente, o método Lattice-Boltzmann é derivado de seu precursor, o Lattice Gas Automata (LGA). A ideia fundamental por trás do LGA é que interações microscópicas de partículas artificiais dentro de uma grade microscópica (*lattice grid*) pode levar às correspondentes equações macroscópicas que governam o mesmo escoamento. Tal ideia é baseada no princípio que a macro dinâmica de um fluido é o resultado do comportamento coletivo das partículas do sistema e interações microscópicas detalhadas podem ser negligenciadas. Durante essa interação, que consiste em sucessivos movimentos de colisão e propagação (*streaming*), a simetria interna de cada *lattice* é fundamental para garantir a conservação de massa e momento [Peng 2011].

Dessa forma, em um modelo LGA, grandezas como espaço, tempo e velocidade são discretas. Em cada um dos nós do *lattice grid* reside uma partícula, representada por uma variável booleana, dependente do espaço e do tempo. Dada uma condição inicial, a configuração de partículas, então, evolui através dos passos de colisão e propagação, respeitando o princípio da exclusão, no qual não mais que uma partícula pode ocupar o mesmo nó no mesmo instante de tempo [Hou et al. 1995]. O esquema LGA é estável, com condições de contorno de implementação simplificada e fácil aplicação em processamento paralelo, uma vez que a atualização das propriedades de um nó depende apenas de seus nós vizinhos. Apesar disso, o método apresenta alguns problemas intrínsecos, tais como ruído estatístico e a não conformidade com o princípio da invariância de Galileu, em razão de um coeficiente dependente da densidade no termo convectivo da equação de Navier Stokes [Hou et al. 1995].

A fim de solucionar alguns desses problemas, alguns modelos foram propostos [McNamara e Zanetti 1988], utilizando a equação de Boltzmann, no qual a principal característica é a substituição das variáveis booleanas por funções densidade de probalidade

(FDP) para cada uma das partículas. O modelo foi sendo sofisticado ao longo dos anos, distanciando-se do LGA original, através, por exemplo, da linearização do operador de colisão [Higuera e Jiménez 1989] e [Higuera, Succi e Benzi 1989], do desacoplamento entre pressão e velocidade [Qian, d'Humières e Lallemand 1992] e da implementação de esquemas de relaxação temporal baseados em [Bhatnagar, Gross e Krook 1954].

Devido a sua paralelização simplificada, o método pode ser extensivamente utilizado em computação de alto desempenho, em contraponto às equações diferenciais que descrevem o sistema contínuo.

2 OBJETIVOS

2.1 Objetivo Geral

Implementação e análise de modelo computacional baseado no método Lattice-Boltzmann para simulação de escoamentos externos.

2.2 Objetivos Específicos

- Comparar o código desenvolvido em C++ com o código *opensource* OpenLB;
- Avaliar as limitações do código desenvolvido;
- Comparar o coeficiente de arrasto, coeficiente de pressão no ponto de estagnação e número de Strouhal previstos no código desenvolvido com o método de volumes finitos.

3 FUNDAMENTAÇÃO TEÓRICA

3.1 Teoria Cinética de Partículas

3.1.1 Introdução

São duas as abordagens macro principais na simulação de escoamentos: contínua e discreta. Na hipótese contínua, equações diferenciais parciais ou ordinárias são desenvolvidas ao se aplicar as leis de conservação de energia, massa e momento para um elemento de fluido infinitesimal. Uma vez que a solução de tais equações é complicada devido, por exemplo, às não-linearidades e condições de contorno complexas, lança-se mão de diferentes métodos de elementos finitos para converter as equações diferenciais com suas respectivas condições iniciais e de contorno em um sistema de equações algébrico. O sistema, por sua vez, pode ser resolvido iterativamente, até que uma condição de convergência pré-estabelecida seja atingida. Considera-se que cada elemento de fluido contém uma coleção de partículas, de modo que a escala de resolução é considerada macroscópica.

Na escala oposta, o meio é considerado composto por pequenas partículas que colidem umas com as outras. Dessa forma, é preciso conhecer as forças intermoleculares para então aplicar a segunda lei de Newton e identificar a trajetória de cada partícula. A esse nível de grandeza, pressão e temperatura estão relacionadas à energia cinética e frequência das partículas. Tal método é denominado Dinâmica Molecular (DM) [Mohamad 2011]. Apesar de teoricamente simples, a abordagem da Dinâmica Molecular demanda um enorme poder de processamento para simulações em grande escala que não está disponível atualmente e provavelmente não estará em um futuro próximo, em razão da necessidade de se conhecer as forças intermoleculares e do passo de tempo precisar ser menor que o tempo médio de colisão entre as partículas, algo da ordem de pico-segundos ($10^{-12}s$) [Mohamad 2011].

Para resolver a escala macroscópica, no entanto, objeto de interesse da larga maioria dos problemas industriais, não é necessário conhecer o comportamento individual de cada partícula, mas sim o comportamento de um conjunto delas. E é justamente essa premissa que alicerça o método Lattice-Boltzmann. O intuito principal é aliar um pouco das duas abordagens citadas, caracterizando, então, uma abordagem em mesoescala, onde as propriedades de cada conjunto de partículas é representado por uma função de distribuição [Brito 2017].

3.1.2 Dinâmica da Partícula

Considerando moléculas como esferas que se movem aleatoriamente pelo espaço, de modo que o movimento obedeça as leis de conservação de massa, momento e energia, é possível aplicar a segunda lei de Newton (conservação de momento), em que a taxa de variação do momento linear é igual à força aplicada:

$$F = \frac{d(mc)}{dt} \tag{3.1}$$

onde F representa as forças intermoleculares e externa, m é a massa da partícula, c é o vetor velocidade e t é o tempo. Adotando uma massa constante, temos que

$$F = m\frac{dc}{dt} = ma \tag{3.2}$$

onde a é o vetor aceleração. A posição de cada partícula pode ser determinada a partir da definição de velocidade:

$$c = \frac{dr}{dt} \tag{3.3}$$

onde r é o vetor posição da partícula com relação à origem.

Na Dinâmica Molecular, as equações são resolvidas considerando que F é uma propriedade conhecida. Assim, se uma força F é aplicada a uma partícula de massa m, a velocidade da partícula muda proporcionalmente, de c para $c + Fdt/m$, e a posição muda de r para $r + cdt$. Na ausênsia de forças externas, a partícula se move livremente de uma posição para outra sem mudar de direção, assumindo que não ocorrem colisões.

A magnitude da velocidade da partícula e a interação entre elas cresce à medida que a energia interna cresce (aquecendo o sistema, por exemplo). O aumento da energia cinética em escala microscópica é traduzido em aumento de temperatura em escala macroscópica. Assim, com o aumento da energia cinética, aumenta também a probabilidade de colisão de partículas com a parede do recipiente que contém o gás, o que se traduz em um aumento na pressão. Dessa forma, estabelece-se uma relação entre pressão e temperatura, de forma que essas grandezas nada mais são do que uma escala da energia cinética das moléculas em escala microscópica. [Mohamad 2011].

3.1.3 Função de Distribuição

Uma vez que mapear a posição e velocidade de cada molécula em cada instante de tempo é impraticável, Maxwell propôs conhecer a função de distribuição para então caracterizar o efeito do comportamento das partículas, ou seja, qual porcentagem das

moléculas, em um dado instante de tempo e em determinado local, possui velocidades dentro de um determinado intervalo de valores.

Dessa forma, considerando um gás com N partículas, a quantidade de partículas na direção x com velocidades entre c_x e $c_x + dc_x$ é $Nf(c_x)dc_x$. A função $f(c_x)$ é a fração de partículas com velocidades entre c_x e $c_x + dc_x$, na direção x. Analogamente, pode-se definir a função f para as outras direções. Dessa forma, a probabilidade da velocidade de determinada partícula estar entre c_x e $c_x + dc_x$, c_y e $c_y + dc_y$ e c_z e $c_z + dc_z$ é igual a $Nf(c_x)f(c_y)f(c_z)dc_xdc_ydc_z$.

Assim, a equação anterior pode ser integrada entre todos os valores possíveis de velocidade, considerando todas as N partículas do sistema:

$$\iiint f(c_x)f(c_y)f(c_z)dc_xdc_ydc_z = 1. \tag{3.4}$$

Considerando que qualquer direção pode ser x, y ou z, a função de distribuição não deve depender da direção, mas somente da velocidade das partículas, de modo que podemos escrever:

$$f(c_x)f(c_y)f(c_z) = \phi(c_x^2 + c_y^2 + c_z^2), \tag{3.5}$$

onde ϕ é uma função desconhecida que precisa ser determinada. Uma possível função que possui as propriedades estabelecidas pela equação 3.5 é a expressão logarítmica ou exponencial, em que

$$logA + logB = log(AB) \tag{3.6}$$

ou

$$e^A e^B = e^{A+B}. \tag{3.7}$$

É possível demonstrar que uma expressão apropriada para a função de distribuição é:

$$f(c_x) = Ae^{-Bc_x^2}, \tag{3.8}$$

onde A e B são constantes [Mohamad 2011].

Resumidamente, Maxwell mostrou que um gás possui uma função de distribuição específica quando em equilíbrio e Boltzmann, mais tarde, propôs como tal equilíbrio é atingido.

3.2 Equação de Boltzmann

3.2.1 Equação de Transporte de Boltzmann

Como mencionado, a função de distribuição descreve um número de partículas em determinado tempo e local que possuem velocidades dentro de uma faixa estabelecida. Uma vez que o número de partículas antes e após a aplicação de uma força externa é constante, temos que

$$f(r + cdt, c + Fdt, t + dt)drdc - f(r, c, t)drdc = 0 \tag{3.9}$$

No entanto, se ocorrerem colisões entre as moléculas, existirá uma diferença entre o número de moléculas antes e após o intervalo $drdc$. A taxa de variação entre o estado inicial e final da função de distribuição é chamado de operador de colisão, Ω. Dessa forma, considerando a existência de colisões, a equação 3.9 pode ser escrita da seguinte forma:

$$f(r + cdt, c + Fdt, t + dt)drdc - f(r, c, t)drdc = \Omega(f)drdcdt \tag{3.10}$$

Dividindo a equação acima por $dtdrdc$ e aplicando o limite $dt \to 0$, temos que

$$\frac{df}{dt} = \Omega(f) \tag{3.11}$$

A equação acima afirma que a taxa total de variação da função de distribuição é igual a taxa de colisões. Uma vez que f é função de r, c e t, a taxa total de variação pode ser expandida para

$$df = \frac{\partial f}{\partial r}dr + \frac{\partial f}{\partial c}dc + \frac{\partial f}{\partial t}dt \tag{3.12}$$

Dividindo a equação 3.12 por dt e expressando r em um sistema cartesiano tridimensional onde $r = xi + yj + zk$, onde i, j e k são vetores unitários ao longo dos eixos x, y e z, respectivamente, temos que

$$df = \frac{\partial f}{\partial r}c + \frac{\partial f}{\partial c}a + \frac{\partial f}{\partial t} \tag{3.13}$$

onde a representa a aceleração ($a = dc/dt$) e pode ser relacionado com a força F pela segunda lei de Newton, $a = F/m$.

Dessa forma, a equação de transporte de Boltzmann pode ser escrita da forma:

$$\frac{\partial f}{\partial t} + \frac{\partial f}{\partial r}c + \frac{F}{m}\frac{\partial f}{\partial c} = \Omega \tag{3.14}$$

O operador Ω é função de f e precisa ser determinado para se resolver a equação de Boltzmann. Para um sistema na ausência de forças externas, a equação de Boltzmann pode ser escrito da seguinte forma:

$$\frac{\partial f}{\partial t} + c \cdot \nabla(f) = \Omega \tag{3.15}$$

onde c e $\nabla(f)$ são vetores.

3.2.2 A aproximação BGKW

O operador Ω, descrito pela equação 3.15, é uma função de f, e, portanto, difícil de ser resolvido em razão de sua natureza integral-diferencial. É possível, no entanto, aproximar a equação de Boltzmann através de um operador simplificado, sem prejuízo com relação aos resultados. Tal operador foi introduzido por Bhatnagar, Gross e Krook em 1954, e uma forma similar foi apresentado de maneira independente por Welander no mesmo ano. Desse modo, o operador Ω é substituído por:

$$\Omega = \omega(f^{eq} - f) = \frac{1}{\tau}(f^{eq} - f) \tag{3.16}$$

onde $\omega = 1/\tau$.

O coeficiente ω é chamado de *frequência de colisão* e τ é chamado de *fator de relaxação*. A função de distribuição no equilíbrio é representada por f^{eq}, a função de distribuição de Maxwell-Botzmann.

Dessa forma, após a introdução do operador BGKW, a equação de Boltzmann pode ser aproximada por:

$$\frac{\partial f}{\partial t} + c \cdot \nabla(f) = \frac{1}{\tau}(f^{eq} - f) \tag{3.17}$$

Para aplicação no método Lattice-Boltzmann, a equação acima é discretizada e aplicada em cada uma das direções do arranjo de rede. Dessa forma, a equação discretizada para uma dada direção i pode ser representada por:

$$\frac{\partial f_i}{\partial t} + c_i \cdot \nabla(f_i) = \frac{1}{\tau}(f_i^{eq} - f_i) \tag{3.18}$$

A equação 3.18 é o cerne da implementação do método Lattice-Boltzmann e é a que substitui as equações de Navier-Stokes nas simulações CFD. O lado direito da equação representa a propagação *(streaming)* e o lado esquerdo representa o processo de colisão [Mohamad 2011]. O tipo de problema a ser resolvido é definido pela função de distribuição no equilíbrio e o tempo de relaxação.

3.3 Arranjos de Rede

A terminologia usada no LBM representa a dimensão do problema a ser resolvido (n) e o número de direções possíveis a serem percorridas pela partícula (m), na forma $DnQm$. Os arranjos de rede mais utilizados para modelos de uma, duas e três dimensões será brevemente discutido abaixo.

3.3.1 Arranjos Unidimensionais

Comumente, dois modelos são os mais utilizados para simulações unidimensionais, chamados D1Q3 e D1Q5, representados na figura 2

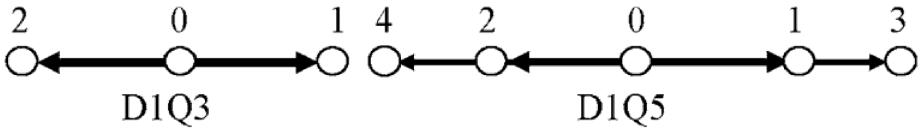

Figura 2 – Representação dos arranjos de rede D1Q3 e D1Q5 [Mohamad 2011])

Nos arranjos do LBM com uma partícula central, como os ilustrados acima, as partículas se propagam a partir do nó central para os nós vizinhos com uma velocidade específica, denominada *velocidade de lattice*.

No arranjo D1Q3, existem três vetores de velocidade, c_0, c_1 e c_2, para cada função de distribuição, f_0, f_1 e f_2, que correspondem aos valores de $0, 1$ e -1, respectivamente. Nesse arranjo, os pesos ω_i para cada função de distribuição são iguais a $4/6, 1/6$ e $1/6$, respectivamente.

Um arranjo de ordem maior pode ser utilizado, como é o caso do D1Q5. Nesse modelo, o número de partículas não pode exceder cinco em qualquer instante de tempo. Os pesos ω_i são iguais a $6/12, 2/12, 2/12, 1/12$ e $1/12$ para f_0, f_1, f_2, f_3 e f_4, respectivamente.

3.3.2 Arranjos Bidimensionais

Para arranjos bidimensionais, são três os modelos mais utilizados: D2Q4, D2Q5 e D2Q9. No D2Q4, existem quatro vetores de velocidade apontando para cada uma das quatro direções, norte, sul, leste e oeste, como ilustrado na figura 3. A diferença para o modelo D2Q5 é que neste último, uma partícula reside no nó central com velocidade nula.

O arranjo D2Q9, por sua vez, é o mais indicado para simulação de escoamentos, e foi o utilizado no presente trabalho. Nele, uma partícula reside no centro do arranjo como origem para oito vetores de velocidade, conforme a figura 4.

Os valores de velocidade para cada uma das direções, bem como os pesos associados a cada função de distribuição estão indicados na tabela 1.

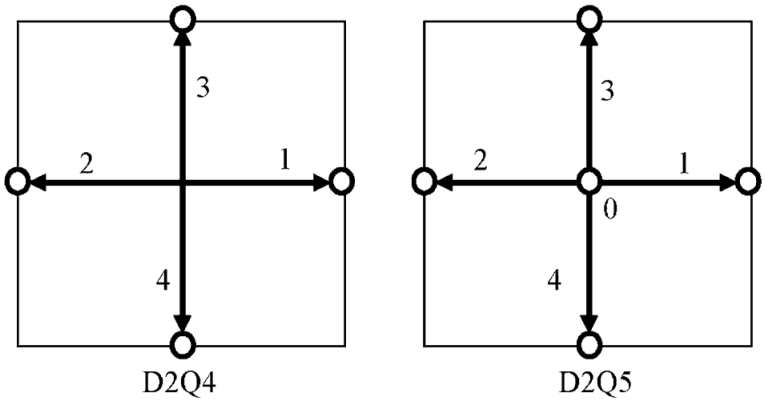

Figura 3 – Representação dos arranjos de rede D2Q4 e D2Q5 [Mohamad 2011])

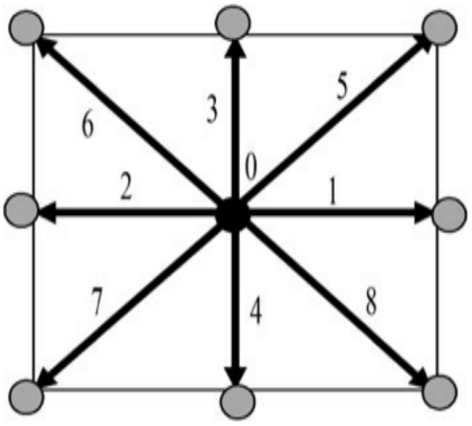

Figura 4 – Representação do arranjo de rede D2Q9 [Mohamad 2011])

Tabela 1 – Relação de velocidades e pesos para o arranjo D2Q9

Função de Distribuição	**Velocidade de Lattice** (c)	**Pesos** (ω_i)
f_0	$(0, 0)$	$4/9$
f_1	$(1, 0)$	$1/9$
f_2	$(0, 1)$	$1/9$
f_3	$(-1, 0)$	$1/9$
f_4	$(0, -1)$	$1/9$
f_5	$(1, 1)$	$1/36$
f_6	$(-1, 1)$	$1/36$
f_7	$(-1, -1)$	$1/36$
f_8	$(1, -1)$	$1/36$

3.3.3 Arranjos Tridimensionais

Os modelos mais utilizados para arranjos tridimensionais são o D3Q15, D3Q19 e D3Q27. Enquanto no modelo D3Q15, 14 vetores velocidade partem do centro, posição onde reside uma partícula, no modelo D3Q19, são 18 os vetores velocidade que partem da posição central. Os dois arranjos são representados na figura 5. O modelo D3Q27 não é comumente utilizado em razão de seu elevado custo computacional.

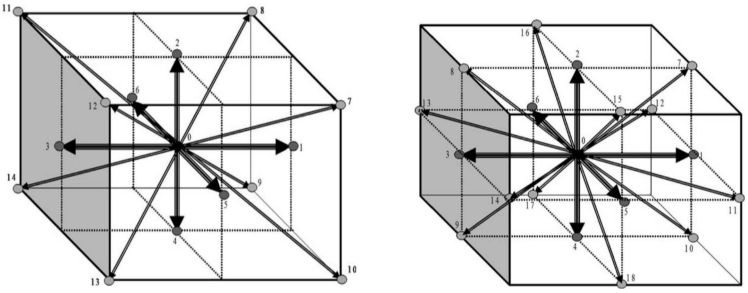

Figura 5 – Representação dos arranjos de rede D3Q15 e D3Q19 [Mohamad 2011])

3.4 Simulação em Fluidos Incompressíveis

Para a simulação de fluidos incompressíveis, o arranjo de rede D2Q9 é necessário para garantir a conservação de isotropia (propriedade de um meio no qual ele possui as mesmas propriedades físicas em todas as direções) e satisfazer a invariância de Galileu ou princípio da relatividade, segundo o qual as leis da física são as mesmas em todos os sistemas de referência inerciais.

3.4.1 Função Distribuição de Equilíbrio

O principal parâmetro que diferencia problemas de natureza diversa quando resolvidos via LBM é a *Função Distribuição de Equilíbrio*, f^{eq}. É possível mostrar que uma forma geral dessa função pode ser escrita da seguinte forma:

$$f^{eq} = \phi\omega_i[A + Bc_i \cdot u + C(c_i \cdot u)^2 + Du^2] \tag{3.19}$$

Na equação 3.19, u representa a velocidade macroscópica; A, B, C e D são constantes a serem determinadas a fim de satisfazer os princípio de conservação de massa, momento e energia, e ϕ representa um parâmetro escalar como densidade, temperatura ou concentração de uma dada espécie, a depender da natureza do problema a ser estudado. O parâmetro ϕ é tal que se iguala a soma de todas as f^{eq}:

$$\phi = \sum_{i=0}^{i=n} f_i^{eq} \tag{3.20}$$

em que n é igual ao número de direções dentro do arranjo de rede. Para a resolução de um escoamento isotérmico incompressível, a equação de Boltzmann pode ser escrita da seguinte forma:

$$f_k(x + \Delta x, t + \Delta t) = f_k(x, t)(1 - \omega) + \omega f_k^{eq}(x, t) \tag{3.21}$$

A função de distribuição de equilíbrio para esse tipo de problema, é escrita da forma:

$$f_k^{eq} = w_k \rho(x,t)[1 + \frac{c_k \cdot u}{c_s^2} + \frac{1}{2}\frac{(c_k \cdot u)^2}{c_s^4} - \frac{1}{2}\frac{u^2}{c_s^2}] \tag{3.22}$$

em que

$$c_s = \frac{c_k}{\sqrt{3}} \tag{3.23}$$

$$c_k = \frac{\Delta x}{\Delta t} i + \frac{\Delta y}{\Delta t} j, \tag{3.24}$$

e

$$u = ui + vj \tag{3.25}$$

É importante destacar que, para qualquer escoamento incompressível resolvido através do LBM, é preciso garantir o mesmo número de Reynolds e similaridade geométrica. Ademais, [Mohamad 2011] recomenda manter os valores de velocidade próximos da ordem de 0.1, a fim de reduzir o erro durante as simulações, em razão da natureza do método em resolver escomentos para baixos números de Mach. Dessa forma, para elevados valores de números de Reynolds, é possível aumentar o tamanho do domínio (número de *lattices*) ou diminuir o valor da viscosidade cinemática, v. Deve-se tomar cuidado, no entanto, pois valores muito baixos para v podem levar à instabilidades numéricas. A estabilidade do método será discutido com mais detalhes na seção 3.7.

3.4.2 Parâmetros de Similaridade

Assim como ocorre com a equação de Navier-Stokes, o erro no LBM é da ordem de Ma^2 [Mohamad 2011], o que reafirma a importância de se trabalhar com valores baixos para a velocidade característica U. A viscosidade real do fluido se relaciona com a frequência de relaxação através da equação:

$$v = \frac{\Delta x^2}{3\Delta t}\omega - 0.5 \tag{3.26}$$

Manipulando a equação 3.26 após dividi-la por UL, temos que

$$Ma = \frac{\Delta x}{L\sqrt{3}}(\omega - 0.5)Re \tag{3.27}$$

O número de Reynolds do lattice é definido por

$$Re = \frac{UN}{v} \tag{3.28}$$

onde U é a velocidade característica, N é o número de *lattices* na direção do comprimento característico e v é a viscosidade cinemática. O valor de U não se relaciona numericamente com o valor da velocidade macroscópica real do fluido, sendo importante para garantir a similaridade do número de Reynolds e evitar problemas de instabilidade na solução.

3.4.3 Conservação de Massa e Momento

Para o caso da simulação de fluidos incompressíveis, a soma das funções de distribuição em cada lattice representa a densidade macroscópica do fluido,

$$\rho = \sum_{k=0}^{8} f_k \tag{3.29}$$

O momento pode ser representado através da média das velocidades de lattice c_k, indicadas na tabela 1 para o arranjo D2Q9, ponderadas pela função de distribuição:

$$\rho u = \sum_{k=0}^{8} f_k c_k \tag{3.30}$$

ou, ainda,

$$u = \frac{1}{\rho} \sum_{k=0}^{8} f_k c_k \tag{3.31}$$

A pressão, por sua vez, é dada por

$$p = \frac{\rho}{3} \tag{3.32}$$

de forma que a velocidade do som pode ser considerada constante tal que

$$c_s = \frac{c_k}{\sqrt{3}} \tag{3.33}$$

Por fim, a viscosidade pode ser determinada a partir da equação 3.26.

3.5 Condições de Contorno

Um dos parâmetros principais em simulações via LBM é a correta modelagem das condições de contorno. Para cada uma das fronteiras, é preciso determinar o comportamento das funções de distribuição em relação ao restante do domínio através de equações específicas para cada condição imposta. Nos tópicos seguintes, serão detalhadas cada uma das condições utilizadas no desenvolvimento desse trabalho.

3.5.1 Bounce Back

A condição Bounce Back é utilizada para modelar fronteiras sólidas estacionárias ou móveis, bem como aplicar a condição de não escorregamento ou simular escoamento sobre obstáculos.

O princípio do método é simples. A ideia principal é que uma partícula de fluido ao encontrar a fronteira modelada pelo bounce back, salta de volta para o escoamento, garantindo a conservação de massa e de momento na fronteira. Três versões principais são encontradas na literatura, onde a fronteira se localiza à meia distância da parede, à meia distância dentro da parede e exatamente na parede. O último método é o mais simples, foi o utilizado no presente trabalho, e é ilustrado na figura 6.

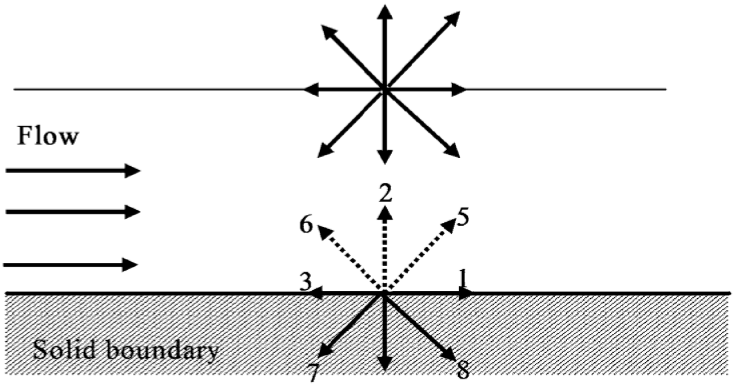

Figura 6 – Representação da condição de contorno bounce back [Mohamad 2011]

A implementação consiste em igualar as funções de distribuição na parede, tal que $f_5 = f_7, f_2 = f_4, f_6 = f_8$, sendo que f_7, f_4 e f_8 são conhecidos através do processo de propagação *(streaming)*.

3.5.2 Condição de Contorno com velocidade conhecida

Essa condição de contorno é útil quando se deseja impor uma entrada de velocidade na fronteira, caso comum em aplicações industriais. Para isso, utiliza-se o método proposto por Zhu e He para calcular o valor de três funções de distribuição assumindo a condição de equilíbrio na fronteira, resumido a seguir.

A equação 3.30 pode ser escrita da forma:

$$\rho = f_0 + f_1 + f_2 + f_3 + f_4 + f_5 + f_6 + f_7 + f_8, \tag{3.34}$$

A equação 3.31 pode ser escrita, para a componente x, como:

$$\rho u = f_1 + f_5 + f_8 - f_6 - f_3 - f_7 \tag{3.35}$$

e para a componente y como:

$$\rho v = f_5 + f_2 + f_6 - f_7 - f_4 - f_8 \tag{3.36}$$

Das equações acima, são quatro as incógnitas a serem determinadas: três funções de distribuição (as demais são conhecidas através do processo de propagação) e o valor de ρ na fronteira. Para tornar o sistema linear possível e determinado, assume-se a condição de equilíbrio na fronteira, normal à superfície, tal que

$$f_i - f_i^{eq} = f_j - f_j^{eq} \tag{3.37}$$

onde i e j dependem da localização da fronteira (norte, sul, leste ou oeste, por exemplo).

3.5.3 Condição de Contorno aberta

Em alguns tipos de problemas, a velocidade de saída não é conhecida, de forma que é comum utilizar uma extrapolação dos valores obtidos próximos a fronteira. [Mohamad 2011] cita, no entanto, que uma interpolação de primeira ordem apresenta resultados mais estáveis que os esquemas de segunda ordem, de modo que foi, então, o método adotado nesse trabalho. Para tal, assume-se um valor constante para a densidade $\rho_{saída}$ na fronteira, de modo que as equações seguintes podem ser utilizadas para determinar as funções de distribuições desconhecidas.

$$u_x = -1 + \frac{f_0 + f_2 + f_4 + 2(f_1 + f_5 + f_8)}{\rho_{saída}} \tag{3.38}$$

$$f_3 = f_1 - \frac{2}{3}\rho_{outlet} u_x \tag{3.39}$$

$$f_7 = f_5 + \frac{1}{2}(f_2 - f_4) - \frac{1}{6}\rho_{saída} u_x \tag{3.40}$$

$$f_6 = f_8 - \frac{1}{2}(f_2 - f_4) - \frac{1}{6}\rho_{saída} u_x \tag{3.41}$$

3.6 Método *momentum exchange* para cálculo do arrasto

Existem situações em que é necessário conhecer a força ou torque atuantes em uma superfície imersa em um fluido, como sustentação e arrasto.

Usualmente, a força total atuante em uma dada área A pode ser calculada através de uma integral de superfície do tensor de tensões na superfície multiplicado pelo vetor

normal, direcionado da superífice para o fluido, em toda a área A. A multiplicação do tensor de tensões pelo vetor normal é usualmente denominado vetor tensão.

O problema principal surge na avaliação dessa integral em uma simulação computacional. A abordagem convencional é calcular o tensor de tensões na superfície através do método de diferenças finitas, para aproximar o valor da integral. Dentro do LBM, contudo, é possível obter diretamente o tensor de tensões, conforme proposto por [Ladd 1994].

O método é fundamentado no fato de que o LBM é baseado em partículas, em que as populações f_i representam elementos de fluido com quantidade de movimento (momento) igual a $f_i c_i$. Assim, é preciso identificar as populações que cruzam a fronteira entre o fluido e o sólido e somar todas as contribuições correspondentes de momento para obter a variação total de momento na parede. É importante destacar que o *MEA (momentum exchange algorithm)* fornece diretamente o vetor tração, e não o tensor de tensões.

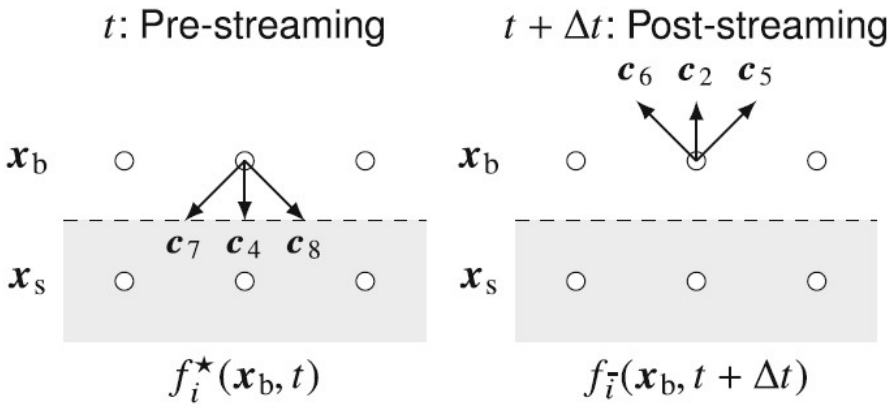

Figura 7 – Representação de uma parede plana inferior e as funções de distribuições envolvidas no esquema *bounce-back* entre os instantes t antes da propagação (esquerda) e $\Delta t + t$ pós propagação (direita). [Krüger et al. 2017]

Considerando uma parede plana inferior, conforme ilustrado na figura 7, as populações f_4, f_7 e f_8 se propagam do fluido para os nós sólidos. Elas atingem a parede e são refletidas, continuando o processo como f_2, f_5 e f_6. Na prática, é como se as populações f_4, f_7 e f_8 desaparecessem na fronteira e f_2, f_5 e f_6 surgissem da parede. Por um lado, momento é transportado por f_4, f_7 e f_8 para a parede, e ao mesmo tempo, é transportado por f_2, f_5 e f_6 para o fluido. Desse modo, a variação de momento total Δp em cada direção é dada por:

$$\Delta p = f_i^{entra} c_i - f_j^{sai} c_j \tag{3.42}$$

em que i e j representam as direções de entrada e saída. Para o caso do exemplo representado pela figura 7, $i = 2, 5$ ou7 e $j = 4, 7$ ou 8. As funções de distribuição que se movimentam

paralelas à parede não são relevantes pois não contribuem com a troca de quantidade de movimento entre o fluido e o sólido.

Dessa forma, a variação de momento total durante um passo de tempo é a soma da variação dos momentos em todos os nós da fronteira. Para um domínio bidimensional,

$$\Delta p = \Delta x^2 \sum_{x_i^w} (f_i^{entra} c_i - f_j^{sai} c_j). \tag{3.43}$$

A multiplicação prévia por Δx^2 garante que o resultado é quantidade de movimento, e não densidade de quantidade de movimento. Com a variação total de momento, é possível determinar a força F na fronteira atuante em cada instante de tempo Δt pela equação 3.44 [Krüger et al. 2017].

$$F = \frac{\Delta P}{\Delta t} \tag{3.44}$$

3.7 Estabilidade

Apresentados os fundamentos teóricos por trás do LBM, é importante destacar aspectos referentes a sua estabilidade numérica, objeto de vários estudos ([Kuzmin e Ginzburg], [Niu et al. 2004], [Siebert, Jr e Philippi 2008]).

No método Lattice-Boltzmann, para se realizar a simulação de um escoamento real, basta que sejam estabelecidas a similaridade geométrica e a correspondência do número de Reynolds. Este último, no entanto, quando calculado dentro das unidades de lattice, não pode assumir valores arbitrários de velocidade e viscosidade, uma vez que isso pode levar à instabilidade numérica da solução. De modo diferente dos métodos tradicionais de CFD, a observância da condição para que o número de *Courant*, que relaciona a velocidade com a qual a informação se propaga no modelo e a velocidade física de advecção do fluido real, seja menor que a unidade, não é único fator determinante no LBM [Krüger et al. 2017]. Mais especificamente, dentro do modelo de colisão *BGK*, utilizado no presente trabalho, uma condição suficiente para garantir a estabilidade é a manutenção positiva de todas as funções de distribuição no equilíbrio (f_{eq}), [Suga 2009] e [Ginzburg, d'Humières e Kuzmin 2010]. Essa condição é válida para todos os valores de tempo de relaxação tal que:

$$\frac{\tau}{\Delta t} > \frac{1}{2} \tag{3.45}$$

Assim, uma vez que f_{eq} é função da velocidade de lattice u, é possível expressar uma condição suficiente para estabilidade a partir da própria velocidade u. [Krüger et al.

2017] propõe uma condição de estabilidade em função da própria magnitude de u, expressa na equação 3.46, válida para os arranjos D2Q9, D3Q15, D3Q19 e D3Q27.

$$|u_{max}| < \sqrt{\frac{1}{3}}\frac{\Delta x}{\Delta t} \approx 0.577\frac{\Delta x}{\Delta t} \tag{3.46}$$

É possível encontrar na literatura algumas análises que afirmam a necessidade de assegurar positivas também as populações f_i, e não apenas as populações no equilíbrio f_{eq}.

De maneira geral, a fim de aumentar a estabilidade, partindo do modelo BGK com um determinado número de Reynods e tempo de relaxação τ arbitrário, primeiro é recomendado definir $\tau = \Delta t$ e ajustar $|u|$ para manter o Re. Caso $|u| \geq c_s$, ainda é possível reduzir τ para encontrar um novo valor de $|u|$ que atenda a essa condição. No caso negativo ou se a instabilidade ainda permanecer, é possível ainda abordar modelos de colisão mais avançados como o *TRT/MRT*.

4 REVISÃO BIBLIOGRÁFICA

O escoamento ao redor de corpos rombudos, especialmente cilindros, tem sido objeto de estudos dentro do campo da fluidodinâmica há muito tempo. A maioria destes trabalhos, no entanto, se concentra em cilindros de geometria circular expostos ao escoamento livre. Uma extensa revisão dos trabalhos publicados sobre a dinâmica de vórtices na esteira de cilindros foi realizada por [Williamson 1996] e [Zdravkovich 1997]. O escoamento ao redor de cilindros de seção retangular, por sua vez, apesar de alvo de uma menor quantidade de estudos, segundo [Breuer et al. 2000], possui vasta aplicação técnica como, por exemplo, no estudo da aerodinâmica de edifícios. Dessa forma, a crescente importância do método Lattice Boltzmann aliado a sua simplicidade de implementação em geometrias de aspecto retilíneo fez surgir ao longo das últimas décadas uma notável variedade de estudos sobre o escoamento ao redor da referida geometria utilizando o LBM.

A seguir, destacam-se alguns destes trabalhos, com a principais conclusões acerca dos resultados obtidos e comparações realizadas com metodologias mais canônicas, como o método de volumes finitos.

Em 2012, [Perumal, Kumar e Dass 2012] estudou as características do escoamento ao redor de cilindros retangulares variando os parâmetros de razão de bloqueio, definido como a relação entre a altura do canal e o diâmetro do cilindro, além do número de Reynolds. Foi utilizado o método Lattice Boltzmann com tempo de relaxação único e domínio com extensão de 800 x 128 *lattices*. Para uma razão de bloqueio $B = 8$ fixa e variação de Re, os resultados para os padrões de linhas de correntes estão indicados na figura 8

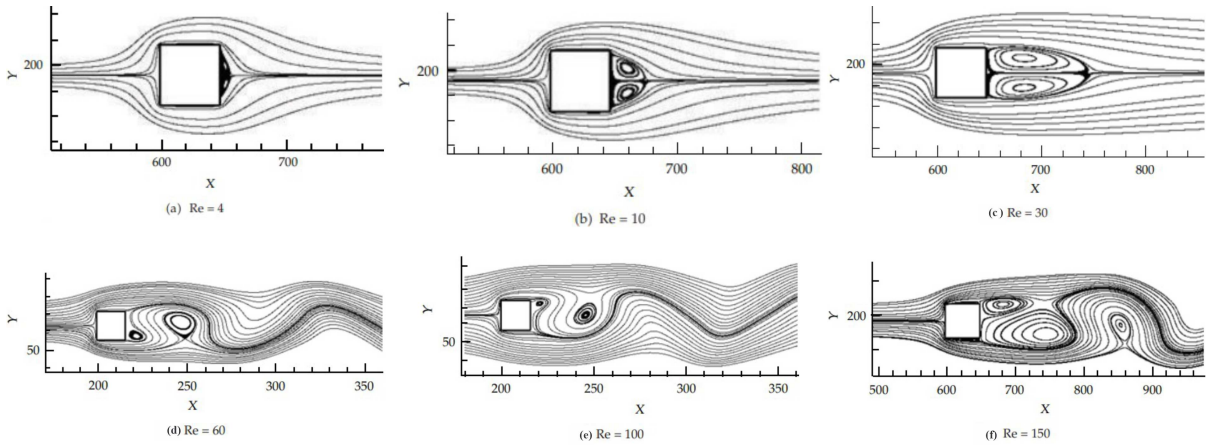

Figura 8 – Padrões de linha de corrente através de um cilindro retangular para diferentes números de Reynolds e uma razão de bloqueio $B = 8$: (a) Re = 4 (b) Re = 10 (c) Re = 30 (d) Re = 60 (e) Re = 100 (f) Re = 150. [Perumal, Kumar e Dass 2012]

Valores de coeficientes de sustentação e arrasto também foram obtidos e comparados com [Breuer et al. 2000]. Estão indicados na tabela 2 e na figura 9.

Tabela 2 – Coeficiente médio de arrasto para diferentes *Re*

Autores	**Re = 50**	**Re = 60**	**Re = 100**
[Breuer et al. 2000]	1,48	1,42	1,37
[Perumal, Kumar e Dass 2012]	2,21	2,02	1,80

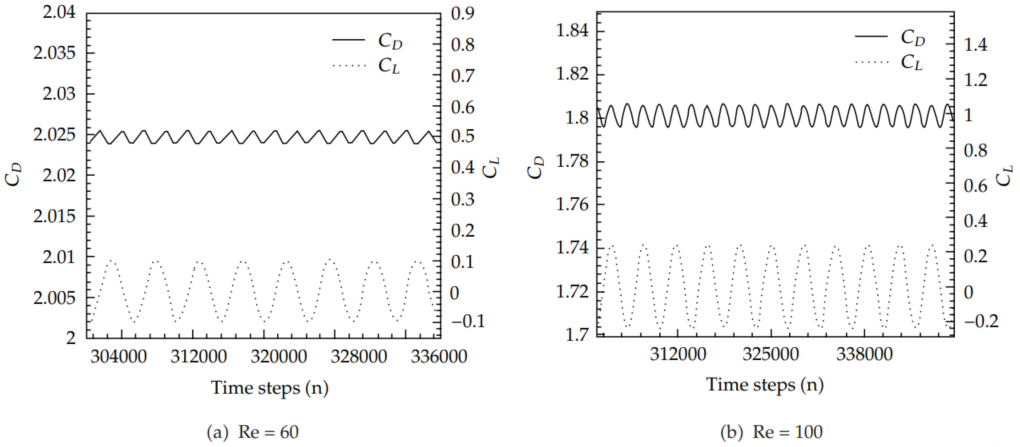

Figura 9 – Coeficientes de sustentação e arrasto variando com o tempo para (a) Re = 60 e (b) Re = 100. [Perumal, Kumar e Dass 2012]

No mesmo trabalho, ainda é realizado um estudo de diferentes condições de contorno de saída e um estudo da influência da posição horizontal do cilindro ao longo do domínio.

Utilizado como critério de comparação no estudo anterior, [Breuer et al. 2000] realizou uma extensa comparação dos resultados para escoamento laminar ao redor de um cilindro retangular entre o método LBM-BGK e o método de volumes finitos (MVF), encontrando excelente concordância.

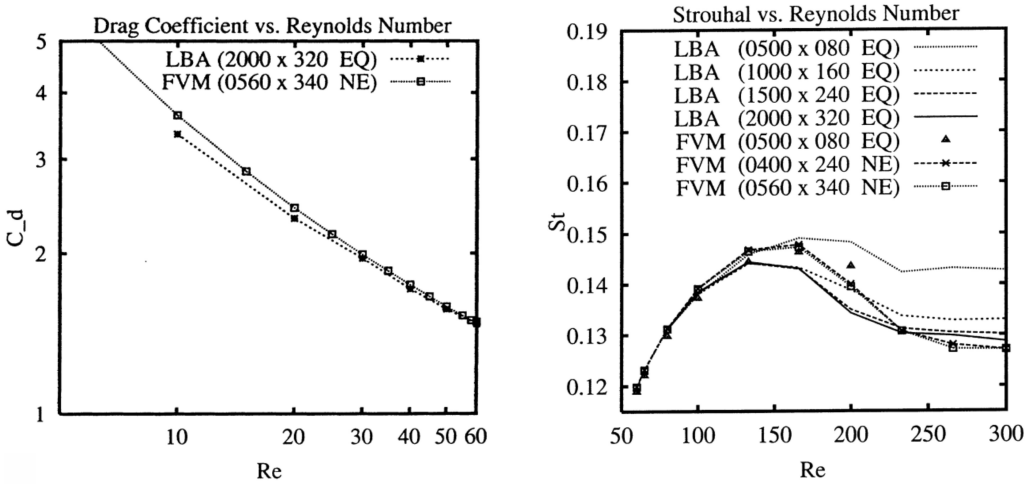

Figura 10 – Comparação entre Cd e St para diferentes *Re* [Breuer et al. 2000]

A figura 10 ilustra a comparação entre diferentes números de Reynolds para o coeficiente de arrasto obtidos através dos dois métodos. Também foi realizada uma comparação entre o número de Strouhal para diferentes resoluções do domínio entre os dois métodos.

Objetivando comparar resultados de coeficiente de arrasto, número de Strouhal e perfis de velocidade para $Re <= 300$, [Rowghani, Mirzaei e Kamali 2010] realizaram um estudo do escoamento ao redor de um cilindro retangular utilizando o modelo LBM-BGK com a razão de bloqueio $B = 1/8$ fixa. A figura 11 apresenta as *streaklines* obtidas para diferentes valores de Re.

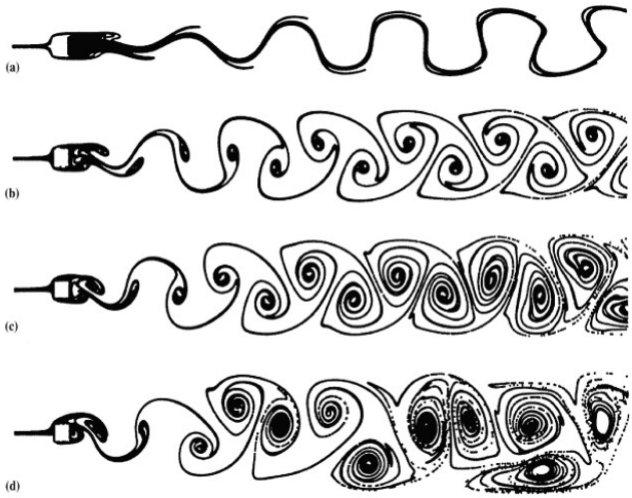

Figura 11 – *Streaklines* ao redor do cilindro retangular para (a) Re = 60, (b) Re = 100, (c) Re = 200, (d) Re = 300 [Rowghani, Mirzaei e Kamali 2010]

O autor também avaliou a variação do coeficiente de arrasto com Re, apresentada na figura 12. Os resultados foram comparados com os obtidos por [Breuer et al. 2000], apresentando significativa concordância.

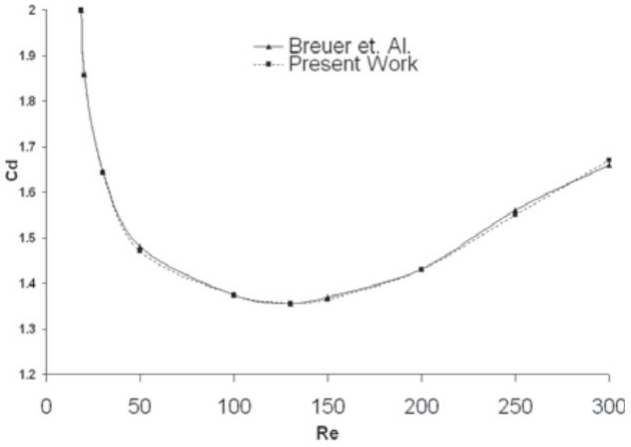

Figura 12 – Coeficiente de arrasto calculado para diferentes valores de Re. [Rowghani, Mirzaei e Kamali 2010]

A fim de avaliar a influência da distância do obstáculo até as paredes laterais do domínio nos valores de alguns parâmetros do escoamento, [Alam e Cheng 2010] compararam os valores dos seguintes parâmetros para uma mesma geometria imersa em domínios com alturas diferentes:

- Coeficiente médio de arrasto;
- Valor eficaz (RMS) do coeficiente de sustentação;
- Coeficiente de Pressão na base ($-C_{p,base}$);
- Coeficiente de Pressão no ponto de estagnação ($C_{p,stag}$);
- Numero de Strouhal (St)

O coeficiente de pressão na base é calculado na intersecção entre a linha de centro vertical do cilindro e a base do domínio. As dimensões dos domínios utilizados estão identificados na figura 13 e tabela 3.

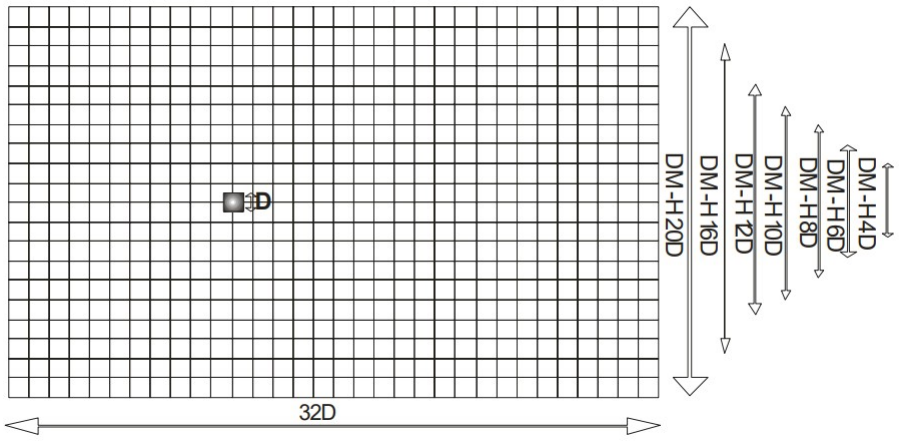

Figura 13 – Exemplo de layout dos grids para diferentes domínios, fora de escala [Alam e Cheng 2010]

Tabela 3 – Detalhamentos dos domínios utlizados - [Alam e Cheng 2010]

Domínio	**Distância**	**Malha**	**Bloqueio (%)**
DM-H4D	4D	512 x 64	25.00
DM-H6D	6D	512 x 96	16.67
DM-H8D	8D	512 x 128	12.50
DM-H10D	10D	512 x 160	10.00
DM-H12D	12D	512 x 240	8.33
DM-H16D	16D	512 x 256	6.25
DM-H20D	20D	512 x 320	5.00

A figura 14 mostra os resultados obtidos para $Re = 100$ nas diferentes configurações de domínio. É possível perceber que todos os parâmetros, com exceção do coeficiente

de pressão no ponto de estagnação, são sensivelmente influenciados pela localização das fronteiras laterais e tendem a convergir quando o obstáculo se posiciona suficientemente longe das fronteiras. As variações para o coeficiente de arrasto médio, valor eficaz (RMS) do coeficiente de sustentação, $-C_{p,base}$, $-C_{p,stag}$ e St são aproximadamente 11%, 9%, 13%, 1% e 5%, respectivamente, quando comparados os resultados entre os domínios DM-16D e DM-10D.

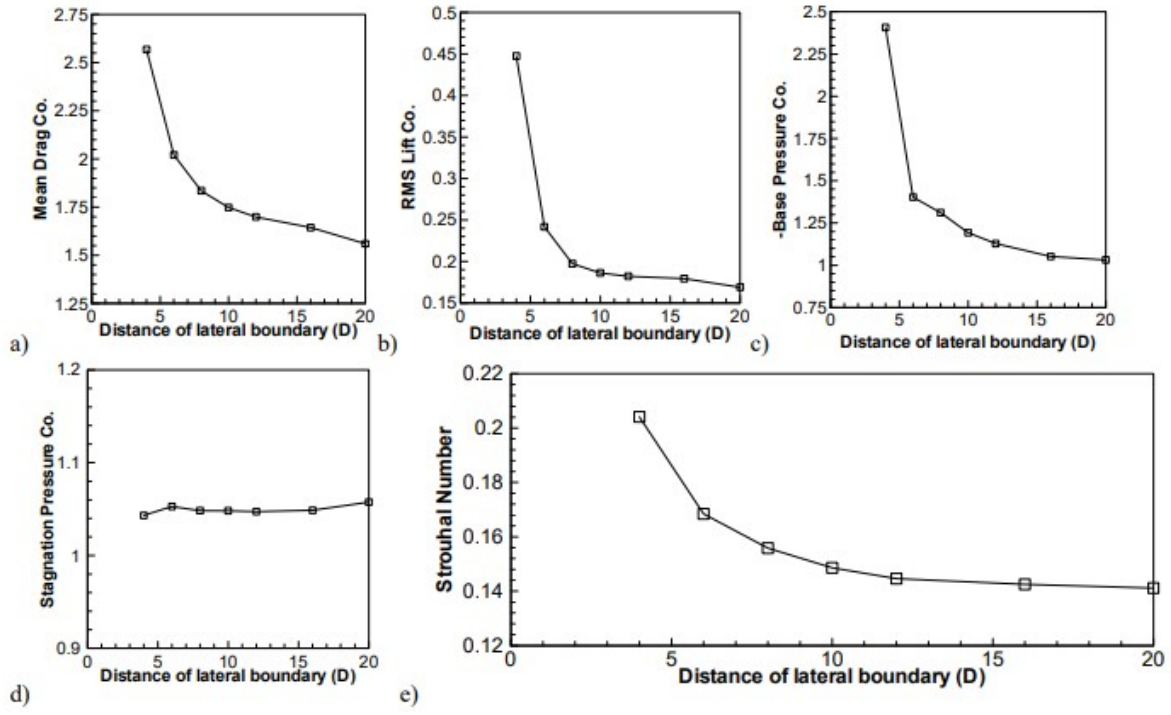

Figura 14 – Efeitos das fronteiras laterais no escoamento para $Re = 100$ [Alam e Cheng 2010]

O autor também apresenta os resultados para a influência do domínio em $Re = 1000$ e conclui que, apesar de também haver variação significativa entre diferentes domínios, as mudanças são menos proeminentes do que para $Re = 100$, conforme é possível depreender da tabela 4. A distância recomendada do obstáculo até as fronteiras laterais foi de $D = 16$.

Tabela 4 – Resumo dos resultados obtidos - [Alam e Cheng 2010]

Parâmetros	**Variação para D=20, Re=100 (%)**		**Variação para D=20, Re=1000 (%)**	
	DM-H10D	**DM-H16D**	**DM-H10D**	**DM-H16D**
Coef. Médio Arrasto	10.75	5.10	5.62	1.43
Coef. Sust. RMS	9.23	5.77	5.53	1.99
$-C_{p,base}$	13.48	1.90	8.15	1.28
$-C_{p,stag}$	0.89	0.84	0.87	0.67
St	5.04	0.98	1.61	1.67

Por fim, um extenso estudo do escoamento ao redor de dois cilindros retangulares foi realizado por [Agrawal, Djenidi e Antonia 2006]. Analisou-se a influência do *gap* entre

os dois cilindros, fator que influi no desenvolvimento de regimes onde a esteira de vórtices está em fase ou fora de fase, conforme observado na figura 15

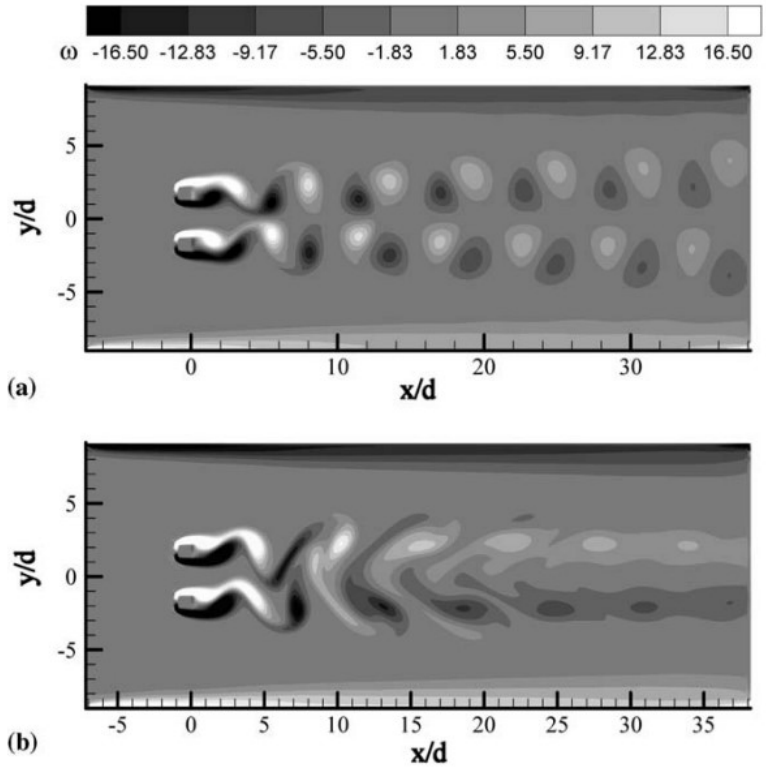

Figura 15 – Contornos de vorticidade para exemplificar o desprendimento de vórtices (a) fora de fase e (b) em fase para $Re = 73$. [Agrawal, Djenidi e Antonia 2006]

O modelo adotado também foi o LBM-BGK em que o lado de cada cilindro é representado por 22 *lattices*. O estudo detalha a influência do posicionamento relativo entre os dois cilindros, bem como a evolução da esteira de vórtices no contexto de regimes síncronos e assíncronos. É também estudada a trajetória do centro do vórtice mais próximo dos cilindros ao longo do domínio, bem como os valores de pressão ao longo do domínio.

5 METODOLOGIA

O código desenvolvido em C++ é uma adaptação do material apresentado por [Mohamad 2011] em *Fortran 90*. Foi inicialmente desenvolvido no Laboratório de Mecânica dos Fluidos da Universidade Federal de Uberlândia durante uma iniciação científica em 2019.

O código é estruturado de maneira orientada ao objeto, de modo a permitir a implementação de várias geometrias e condições iniciais de maneira mais simplificada. Inicialmente, buscou-se avaliar os resultados preliminares da simulação de uma cavidade para números de Reynolds variando entre 100 e 5000, de modo que a comparação com [Hou et al. 1995] e os resultados obtidos através do software *OpenLB* serão avaliados adiante. Em seguida, foi possível adaptar o código para a simulação de escoamentos com obstáculos, objetivo do presente trabalho, e compará-lo com o trabalho desenvolvido por [Norberg, Sohankar e Davidson 1995]. Os resultados para a simulação do escoamento com obstáculos serão apresentados e discutidos no próximo capítulo.

5.1 *OpenLB*

OpenLB é uma plataforma de simulação numérica para fluidodinâmica computacional baseada no método Lattice-Boltzmann. O projeto de código livre foi criado por estudantes e pesquisadores da área a partir de 2006. Destaca-se pela sua versatilidade, uma vez que pode ser usado tanto por programadores para simular escoamentos com geometrias específicas quanto por desenvolvedores para a implementação de novos modelos. A estrutura do código, escrito em C++, é orientada ao objeto a fim de facilitar a modularização.

Em relação aos recursos disponíveis, é possível encontrar arranjos de lattice do tipo D2Q9 e D3Q19 já compilados, e outros arranjos podem ser inseridos pelo programador. Estão implementados os modelos de colisão BGK e MRT com algumas variações que incluem velocidade do som variável para BGK além de BGK e MRT com o modelo de Smagorinsky para simulações LES. A maioria das condições de contorno também já estão implementadas. O funcionamento geral da plataforma está ilustrado na figura 16.

5.2 Estrutura do código desenvolvido

O código desenvolvido em C++ é baseado nas equações descritas na seção de fundamentação teórica. Utilizou-se a metodologia BGK para o operador de colisão, a

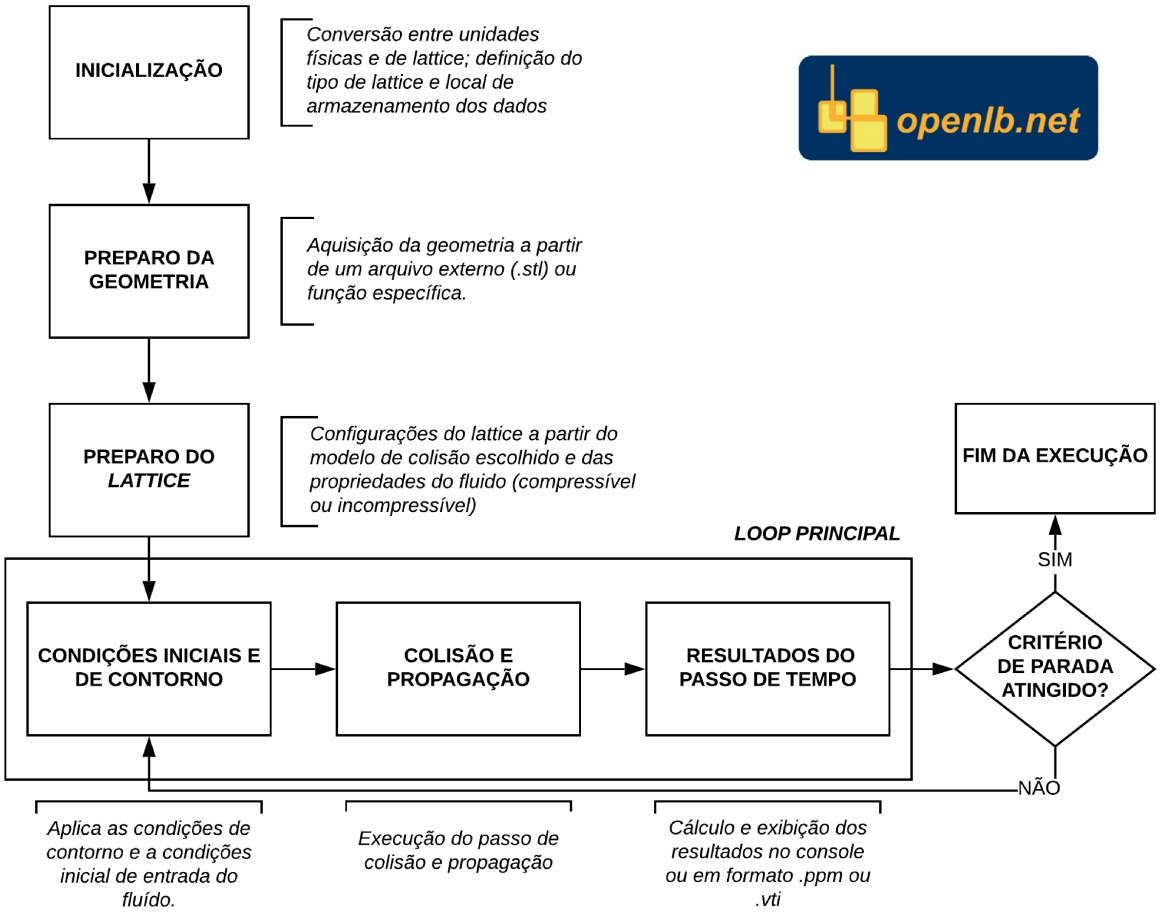

Figura 16 – Representação da estrutura da plataforma OpenLB

condição de contorno *bounce back* nas paredes, interpolação de primeira ordem para as fronteiras com velocidade conhecida e condição de contorno aberta para a saída.

O algoritmo é estruturado em métodos, todos partes do objeto *case* que representa o domínio a ser simulado e as condições de simulação. A estrutura geral do código é ilustrada na figura 17.

5.3 Simulação da Cavidade

A fim de verificar a validade dos métodos de colisão e propagação implementados, foi utilizada uma condição padrão e amplamente estudada na literatura visando comparação dos resultados. O problema consiste em um escoamento contido em uma cavidade bi-dimensional. O fluido é limitado por um compartimento quadrado e impulsionado por uma translação uniforme do topo. A simulação realizada utiliza coordenadas cartesianas com a origem localizada no canto inferior esquerdo. A fronteira superior se move da esquerda para a direita com velocidade-x unitária igual a U e velocidade-y igual a zero. A cavidade possui tamanho mxm e uma densidade uniforme inicial é imposta igual a

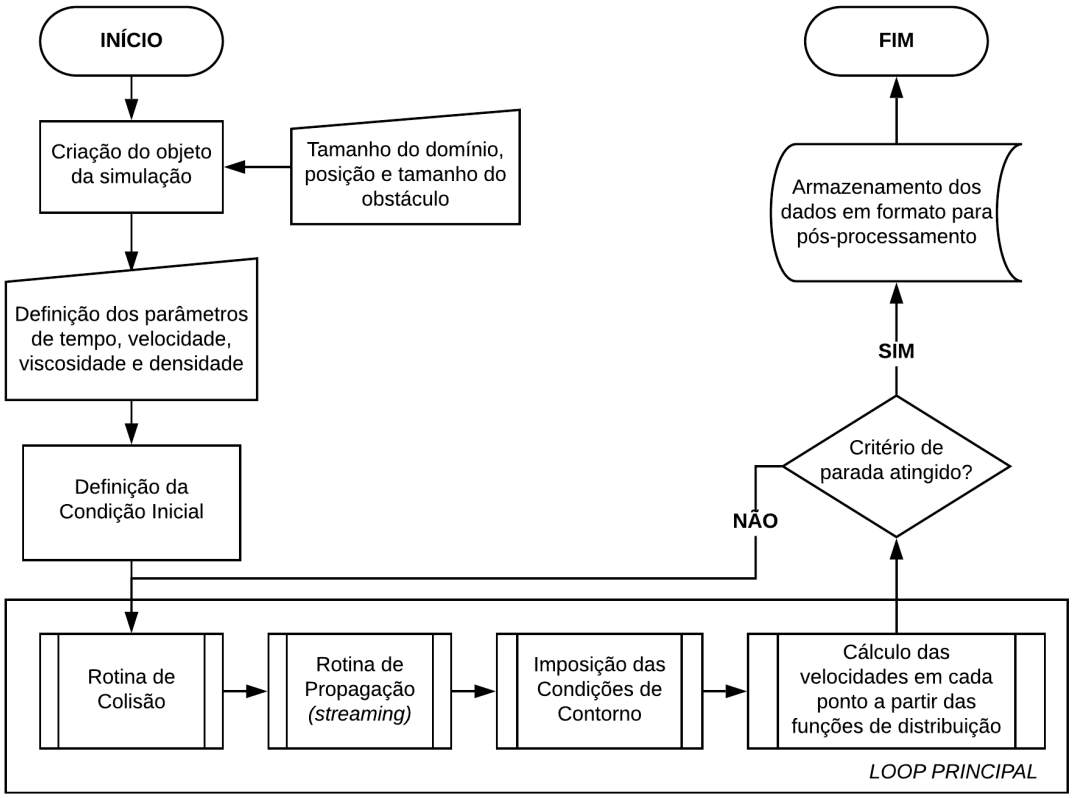

Figura 17 – Representação da estrutura do código implementado

ρ. A maioria das soluções numéricas de escoamento bidimensional em cavidade utiliza a equação de Navier-Stokes considerando o regime incompressível linear ou não linear discretizada por diferenças finitas, *multigrid* ou elementos finitos e suas variações. Os resultados apresentados, obtidos através do método Lattice-Boltzmann, serão comparados com o desenvolvido por [Hou et al. 1995], que obteve soluções numéricas via LBM-BGK para até $Re = 7500$. Serão comparados também com os resultados obtidos através do software de código livre *OpenLB*,

5.3.1 Parâmetros de Simulação

O método Lattice-Boltzmann é inerentemente transiente, de modo que é preciso definir um critério de parada adequado para a comparação de escoamentos em regime permanente. [Hou et al. 1995] considera como parâmetro para avaliação do erro relativo, ϵ, o valor máximo para função corrente no domínio a cada 10.000 passos de tempo, de modo que considera-se que o regime permanente é atingido quando $\epsilon \leq 10^{-6}$. Após testes com diferentes critérios de convergência, utilizou-se como método de avaliação do erro relativo no código desenvolvido a soma do campo de velocidade total, de tal modo que $\epsilon \leq 10^{-4}$, conforme proposto por [Mohamad 2019].

As condições para simulação de cada número de Reynolds estão descritas na tabela

5 e são os mesmos utilizados por [Hou et al. 1995].

O número de Reynolds é utilizado como parâmetro de entrada, de modo que o tempo de relaxação é calculado pela equação 3.26, observando-se as recomendações de valores de velocidade inicial e viscosidade recomendadas por [Mohamad 2011].

Tabela 5 – Condições de simulação da cavidade

Re	**Domínio**	**U**	ρ
100	257 x 257	0.05	2.7
400	257 x 257	0.1	2.7
1000	257 x 257	0.1	2.7
2000	257 x 257	0.1	2.7
5000	257 x 257	0.1	2.7

5.3.2 Resultados e Discussão

Na figura 18, é possível comparar os resultados dos perfis de velocidades ao longo do eixo x para uma linha vertical que passa pelo centro geométrico da cavidade com o obtido por [Hou et al. 1995] e pelo *openLB*.

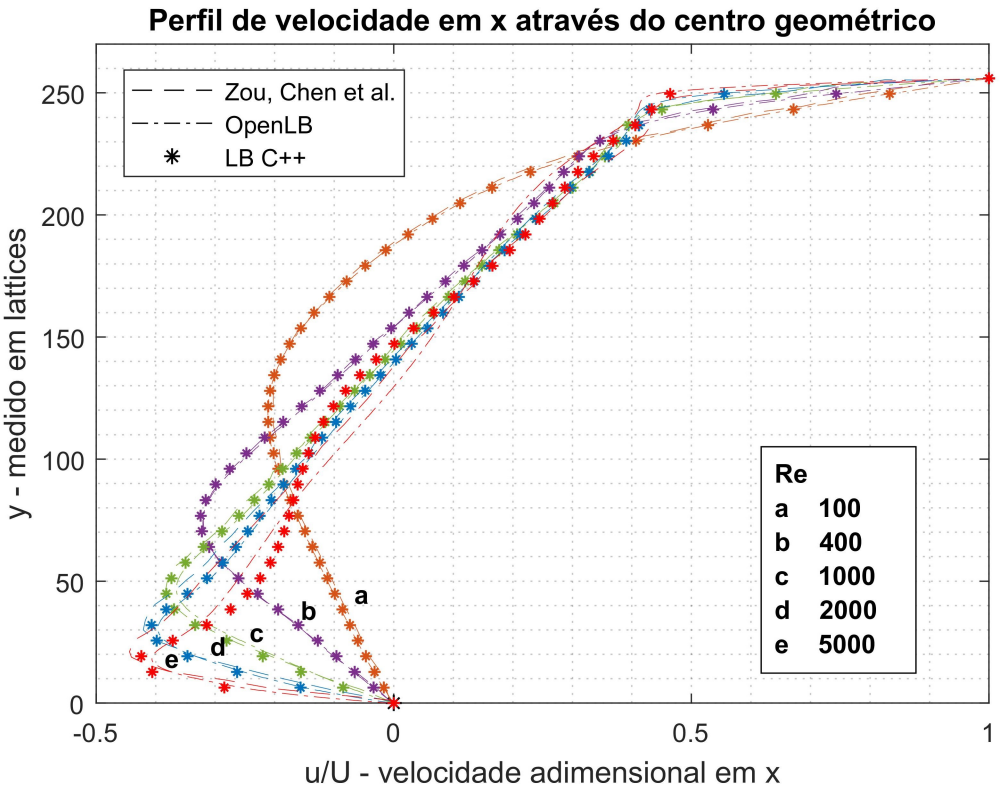

Figura 18 – Comparação do perfil de velocidade em x obtidos para Re = 100, 400, 1000, 2000 e 5000

Analogamente, a figura 19 apresenta o perfil de velocidades ao longo do eixo y para uma linha horizontal que passa pelo centro geométrico do domínio.

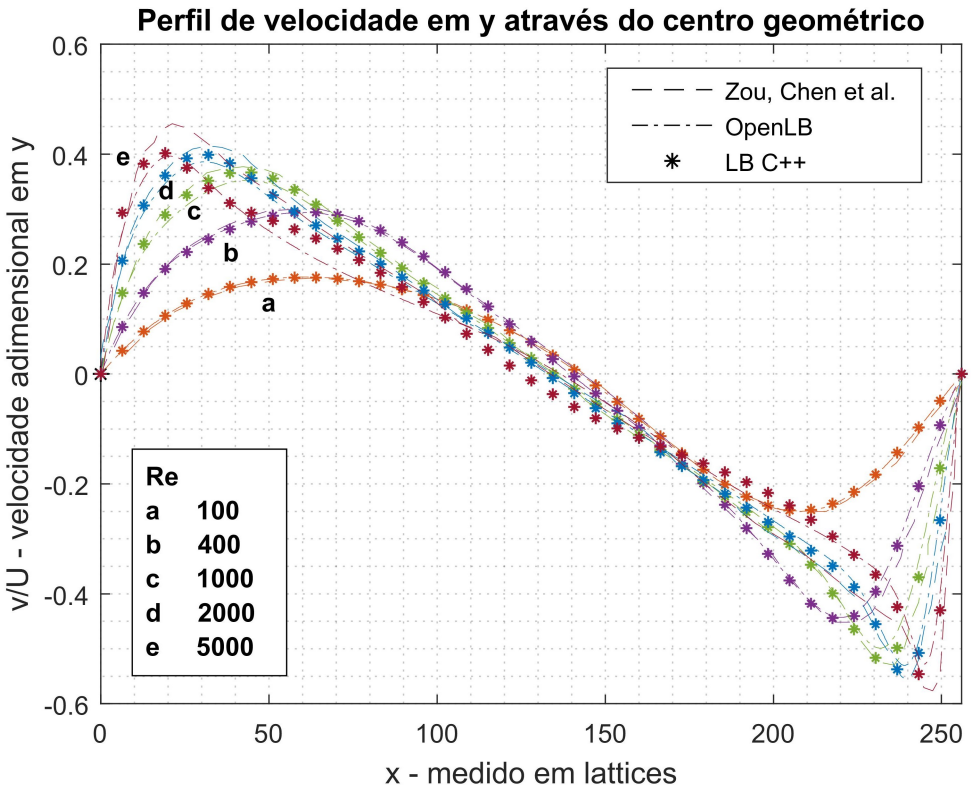

Figura 19 – Comparação do perfil de velocidade em x obtidos para Re = 100, 400, 1000, 2000 e 5000

A figura 20 apresenta os contornos de velocidade obtidos para cada Reynolds ao final simulação.

Uma vez que não existe solução analítica para a simulação do escoamento em cavidade, os resultados obtidos foram comparados com as soluções numéricas das fontes já mencionadas. Conforme é possível notar nas figuras 18 e 19, os valores obtidos para o perfil de velocidade em x e y foram bem próximos dos encontrados por [Hou et al. 1995], chegando a um erro máximo de aproximadamente 16% em $Re = 5000$. Tal diferença pode ser resultado do diferente critério de convergência utilizado entre o programa desenvolvido em C++ e o adotado por [Hou et al. 1995].

Considerando que o objetivo da simulação do escoamento em cavidade é verificar os métodos de colisão, propagação e condições de contorno implementados, bem como avaliar a robustez do código dentro de determinada faixa de Re, os resultados obtidos foram considerados satisfatórios. A partir de então foi possível adaptar as condições de contorno para a simulação do escoamento com obstáculos retangulares, objetivo principal do presente trabalho e descrito no próximo capítulo.

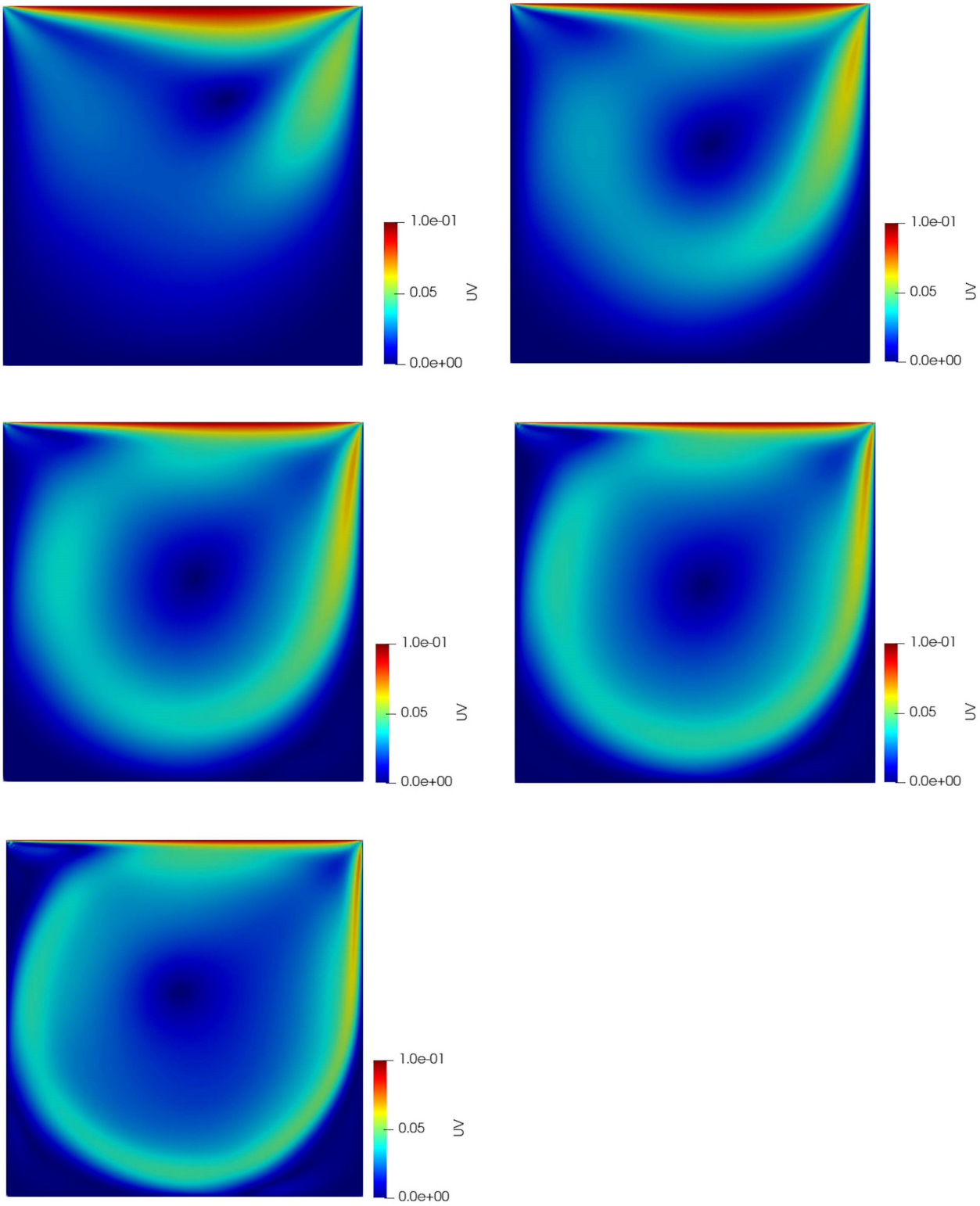

Figura 20 – Comparação dos contornos de velocidade obtidos para Re = 100, 400, 1000, 2000 e 5000 da esquerda para a direita, de cima para baixo. A escala representa a magnitude da velocidade de lattice. Pós processamento realizado no software de código livre *ParaView*

6 SIMULAÇÃO DE ESCOAMENTO SOBRE RETÂNGULOS

As simulações de escoamento sobre cilindros retangulares foram realizadas com o objetivo de comparar os resultados obtidos pelo código desenvolvido na Universidade Federal de Uberlândia com os resultados obtidos por [Norberg, Sohankar e Davidson 1995].

6.1 *Sohankar, Norberg e Davidson, 1995*

Sohankar, Norberg e Davidson publicaram, em 1995, um estudo a respeito do escoamento sobre cilindros retangulares com enfoque na influência do tamanho do domínio, ilustrado na figura 21, à jusante e à montante do corpo. A variável X_u representa a extensão do domínio à montante do corpo, X_d representa a extensão à jusante, H é a altura e d é o diâmetro do corpo. Todas as dimensões foram normalizadas com relação a d.

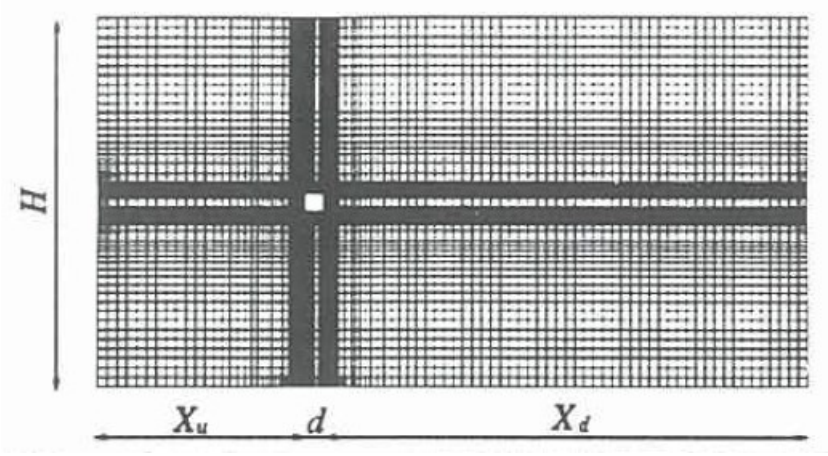

Figura 21 – Domínio 150 x 110 utilizado pelos autores. Fonte: [Norberg, Sohankar e Davidson 1995]

Também foi analisada a senbilidade dos resultados com a variação da razão de bloqueio, definida por:

$$B = \frac{d}{H} \tag{6.1}$$

Foram realizadas simulações utilizando um algoritmo *SIMPLEC* incompressível baseado em volumes finitos. Foi utilizado um esquema *QUICK* de terceira ordem e um esquema de segunda ordem de Van Leer para os termos convectivos. Resultados obtidos pelos autores são comparados com os obtidos pelo código LBM desenvolvido em C++

para B igual a 7.0% e 5.0%, $Re = 100$, $X_d = 22.65$ e $X_u = 18.3$. As comparações são apresentadas na seção 6.2.3.

A condição de contorno utilizada na entrada impõe uma velocidade $U = 1$ em x, e $V = 0$ em y. Na saída, a velocidade é calculada a partir da continuidade global em conjunto com um gradiente nulo na fronteira para U e V. A condição de não escorregamento é aplicada na superfície do corpo e a condição de simetria é adotada nas paredes superior e inferior.

6.2 Simulação sobre Cilindros Retangulares

Com o intuito de comparar os resultados obtidos através do código desenvolvido com o resultado originado de outro software com o mesmo modelo e uma simulação realizada via método de volumes finitos, realizou-se uma simulação para $Re = 100$ no software openLB, no código desenvolvido e ambos foram comparados entre si e com os resultados obtidos por [Norberg, Sohankar e Davidson 1995].

6.2.1 Parâmetros de Simulação

A geometria simulada representa um cilindro quadrado de lado $d = 0.1m$ em um domínio de comprimento $42d$ e altura variável conforme a razão de bloqueio adotada. A velocidade de entrada é $U = 1$ m/s.

O número de *Reynolds* é igual a 100, de modo que domínio possui a resolução de 1260 *lattices* ao longo de seu comprimento, com o cilindro retangular representando por 30 lattices em cada lado. Dessa forma, a velocidade de lattice de entrada foi definida de tal modo que $u = 0,1$ e $\alpha = 0,03$. Tais parâmetros estão dentro do indicado para manter a estabilidade no modelo BGK, conforme discutido na seção 3.7. A figura 22 e a tabela 6 resumem a configuração adotada, em que m representa a resolução na direção x e n a resolução em lattices na direção y.

Tabela 6 – Parâmetros da simulação - Re = 100

B	**H**	m	n
7.0	14d	1260	420
5.0	20d	1260	600

Um dos parâmetros adotados para comparação consiste no coeficiente de pressão no ponto de estagnação, $C_{p,s}$, considerando a pressão dinâmica do escoamento livre, tal que:

$$C_{p,s} = \frac{2(p_s - p_\infty)}{\rho U_\infty^2} \tag{6.2}$$

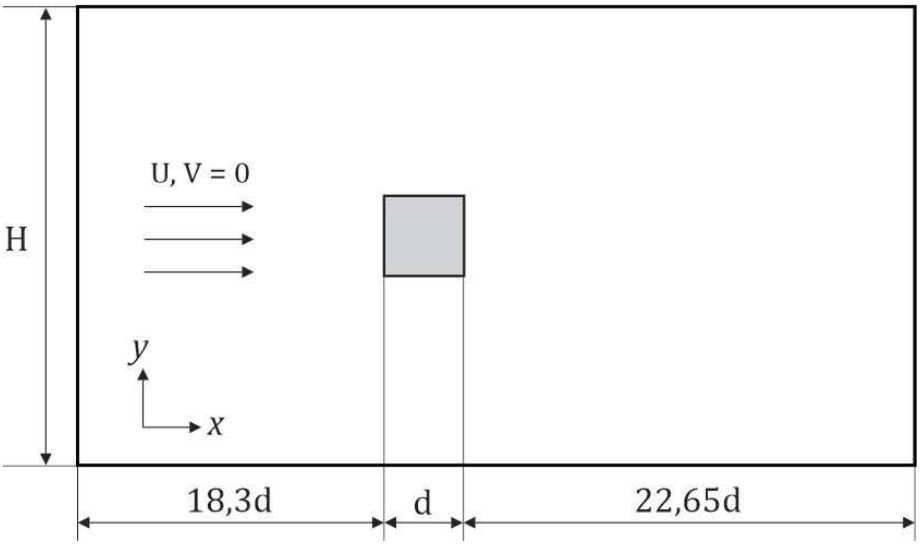

Figura 22 – Domínio utilizado nas simulações no openLB e no código desenvolvido em C++. Fonte: autor.

Também comparou-se o valor do número de Strouhal decorrente do desprendimento de vórtices na esteira do cilindro retangular, definido por:

$$St = \frac{fd}{U_\infty} \tag{6.3}$$

onde f é a frequência de emissão de vórtices, d é o comprimento do cilindro e U é a velocidade do escoamento livre.

Por fim, comparou-se também o valor do coeficente de arrasto, calculado através do método *momentum exchange*, discutido na seção 3.6.

6.2.2 Condições de Contorno

Na fronteira oeste, foi imposta a condição de velocidade de entrada. Para a saída, utilizou-se interpolação de segunda ordem para calcular as funções de distribuição desconhecidas, conforme proposto por [Mohamad 2011]. A fim de mitigar a influência da parede sobre a esteira do escoamento, foi utilizada a condição de simetria para as fronteiras norte e sul.

O obstáculo foi simulado considerando um esquema *bounce-back* exatamente na parede. Apesar de [Krüger et al. 2017] argumentar que esse posicionamento da fronteira possui acuracidade de primeira ordem, o método foi escolhido em razão da simplicidade de implementação em relação ao método de meia distância, discutido na seção de fundamentação teórica.

6.2.3 Resultados e Discussão

Visando padronização dos resultados, o tempo de simulação adotado no código C++ e no openLB foi de 100 segundos. Os parâmetros comparados foram o coeficiente de pressão no ponto de estagnação, $C_{p,s}$, o coeficiente de arrasto, C_d calculado através do método *momentum exchange* e o número de Strouhal, calculado a partir da frequência de emissão dos vórtices, determinada através do monitoramento do coeficiente de arrasto ao longo do tempo. A figura 28 indica o campo de velocidade ao final da simulação.

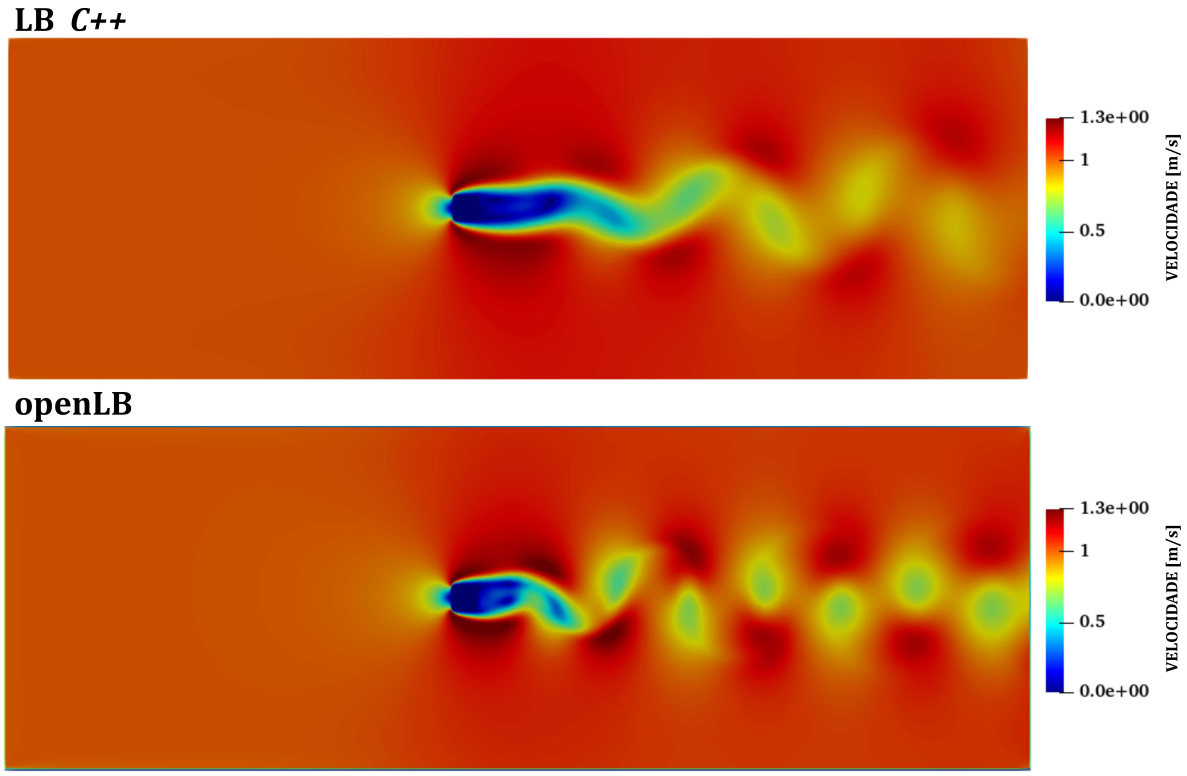

Figura 23 – Campo de velocidade e contornos de pressão para $B = 7\%$ e $Re = 100$ obtidos através do código desenvolvido e do *openLB*

A tabela 7 apresenta os resultados obtidos para o coeficiente de pressão calculado no ponto de estagnação do cilindro. A figura 24 apresenta os valores de pressão e velocidade normalizados utilizados no cálculo do $C_{p,s}$.

Tabela 7 – Comparação entre $C_{p,s}$ para Re = 100

	Sohankar et al.	**OpenLB**	**C++**
B = 7.0%	1,026	1,370	1,051
B = 5.0%	1,028	1,260	1,059

A diferença entre os valores obtidos pelo artigo tomado como base e o código desenvolvido em C++ foi de aproximadamente **2,5%**. Tais resultados estão de acordo com os obtidos por [Alam e Cheng 2010]. O erro observado entre o calculado pelo openLB e o código desenvolvido foi de aproximadamente **23,3%**. Destaca-se que a variação, dentro

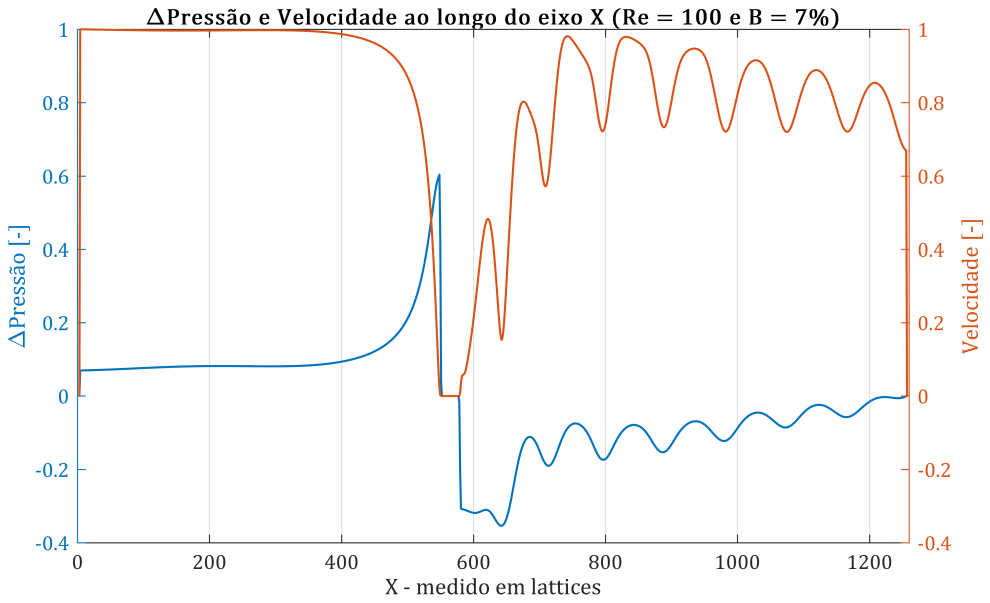

Figura 24 – Velocidade e variação de pressão ao longo do eixo X para $B = 7\%$ e $Re = 100$ ao final da simulação

do mesmo método, entre o $C_{p,s}$, para as duas razões de bloqueio é da mesma ordem de grandeza, variando entre **0,20%** e **0,76%** via MEF e o código desenvolvido em C++, respectivamente.

A tabela 8 apresenta a comparação entre os resultados obtidos para o coeficiente de arrasto.

Tabela 8 – Comparação entre C_d para Re = 100

	Sohankar et al.	**OpenLB**	**C++**
B = 7.0%	1,466	1,560	2,149
B = 5.0%	1,423	1,557	2,006

A diferença entre os valores obtidos pelo artigo tomado como base e o código desenvolvido em C++ foi de aproximadamente **46%**. Com relação ao openLB, o valor obtido foi, aproximadamente, **38%** maior. A figura 25 apresenta a comparação da evolução temporal do coeficiente de arrasto para $B = 7\%$.

A considerável diferença entre o valor obtido com o código desenvolvido e o apresentado por *Sohankar et al.* deve-se, em grande parte, a própria natureza dos métodos para a determinação do coeficiente de arrasto. Enquanto o primeiro não calcula o valor do tensor de tensões, atendo-se apenas a variação da quantidade de movimento obtida através das funções de distribuição entre o obstáculo e o fluido, o trabalho tomado como base aborda o problema de maneira mais canônica, calculando a força de arrasto através do tensor tensão de Cauchy via discretização em elementos finitos.

É fato que o método utilizado, *momentum exchange*, apresentado por [Ladd 1994],

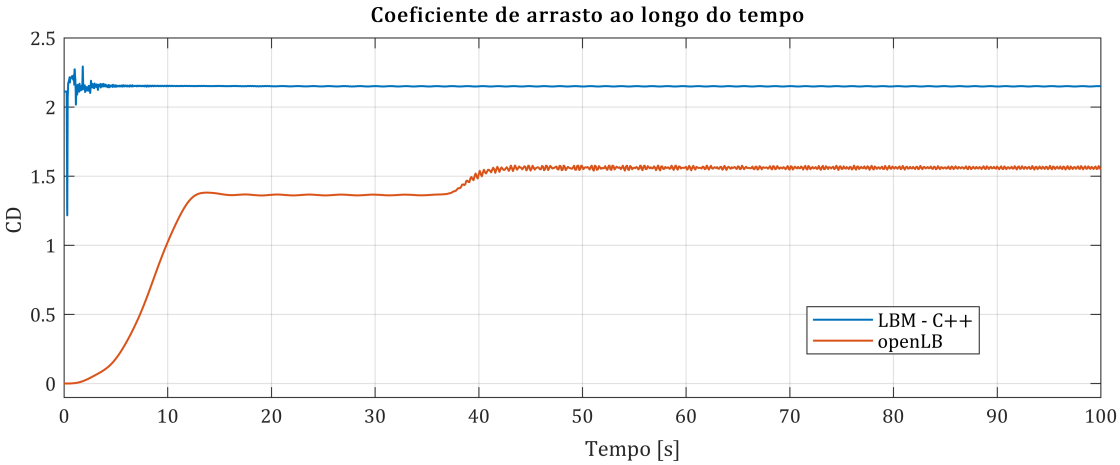

Figura 25 – Comparação da evolução temporal do coeficiente de arrasto para $B = 7\%$ e $Re = 100$

demonstra resultados mais próximos dos obtidos via MEF em outros trabalhos, como por exemplo [Breuer et al. 2000], detalhado na seção de revisão bibliográfica. Nesse contexto, sugere-se, para trabalhos futuros, um estudo de variação da posição do cálculo da troca de momento e a implementação de outros esquemas de *bounce-back*, como por exemplo o sugerido por [Krüger et al. 2017], a meia-parede, na tentativa de melhorar a previsão do coeficiente de arrasto.

Além do valor absoluto do coeficiente, é importante destacar a diferença entre a evolução temporal obtida através dos dois métodos. Devido a natureza subsônica do escoamento, sabe-se que ocorre o fenômeno de pré-organização das partículas a frente do corpo, de modo que o cilindro, na prática, é exposto a todos os Reynolds abaixo de 100 até o fim da pré-organização. Pelo gráfico, é possível depreender que esse período se passa em um intervalo de até 10 segundos. Da tabela 2, é possível notar que a tendência de variação do C_d é decrescente com o aumento de Re. Nesse contexto, o comportamento obtido pelo código desenvolvido está de acordo com o esperado, enquanto que a evolução temporal apresentada pelo *openLB* segue uma tendência crescente.

Finalmente, destaca-se que no resultado obtido via *openLB*, as instabilidades aparecem em aproximadamente 40 segundos, enquanto que no código desenvolvido, ocorrem logo após o perfil de velocidade ter atingido o cilindro, o que sugere uma fonte de instabilidade no código desenvolvido.

O valor do coeficiente de sustentação calculado ao longo do tempo é apresentado na figura 26. Como esperado, devido as condições de simulação, sua flutuação ocorre ao redor do valor nulo.

A frequência de emissão de vórtices foi calculada a partir do coeficiente de arrasto, como indicado na figura 27 entre $50s$ e $100s$. A tabela 9 compara os resultados obtidos para o número de Strouhal.

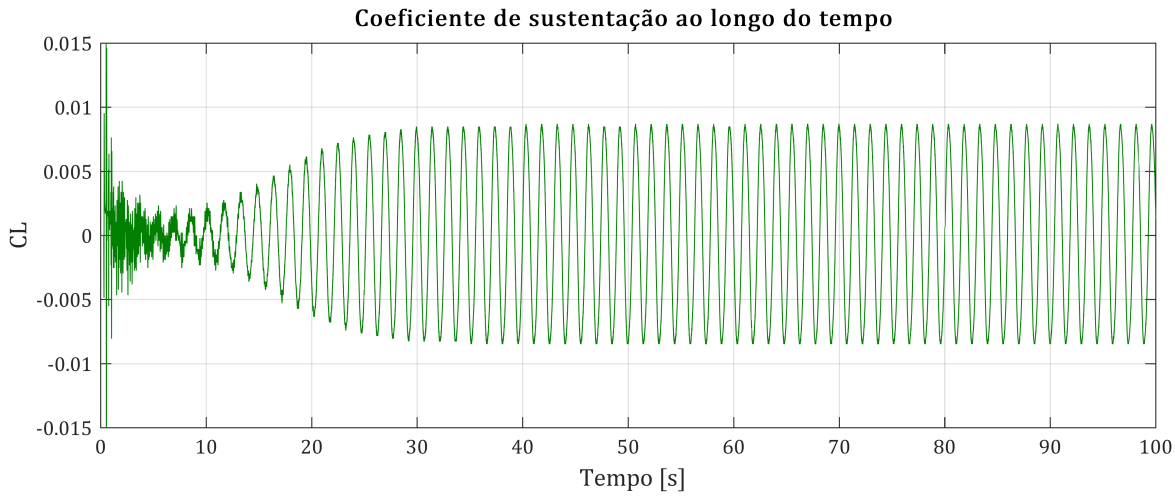

Figura 26 – Evolução temporal do coeficiente de sustentação $B = 7\%$ e $Re = 100$

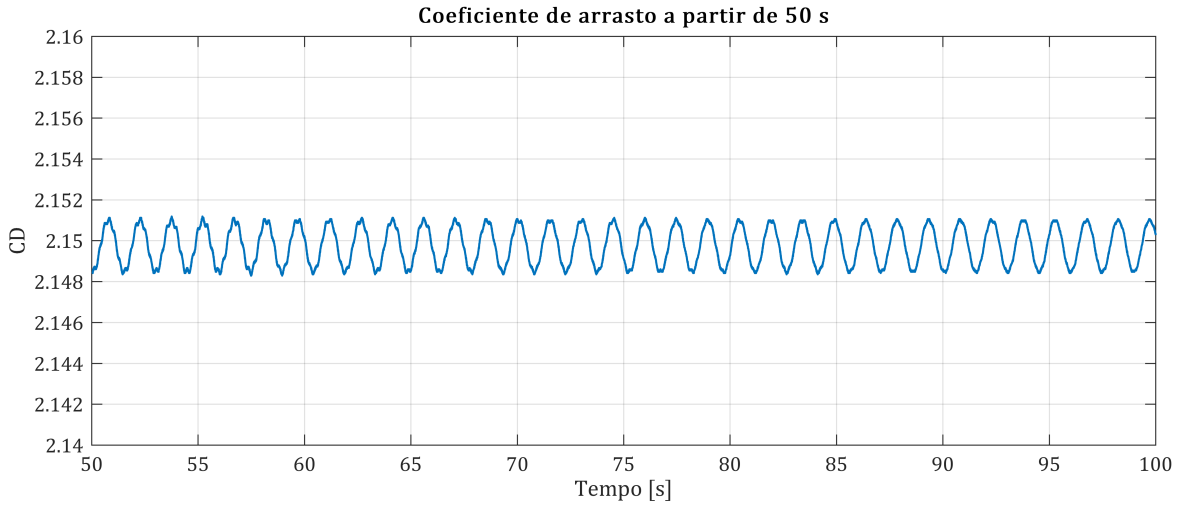

Figura 27 – Coeficiente de arrasto a partir de 50s para $B = 7\%$ e $Re = 100$

Tabela 9 – Comparação entre St para Re = 100

	Sohankar et al.	**OpenLB**	**C++**
B = 7.0%	0,142	0,2974	0,0679
B = 5.0%	0,139	0,2966	0,0659

A diferença entre os valores obtidos pelo artigo tomado como base e o código desenvolvido em C++ foi de aproximadamente **53%**. Em relação ao *openLB*, o valor obtido foi cerca de **77%** menor. Observa-se que tais diferenças podem ser explicadas pelo parâmetro utilizado para a determinação da frequência de emissão de vórtices, o coeficiente de arrasto, pelas razões elencadas anteriormente. Ademais, apesar da diferença entre os trabalhos ser considerável, destaca-se que a diferença entre as duas razões de bloqueio via MEF (**2,11%**) e o código desenvolvido (**2,93%**) é bem próxima, o que sugere que o método implementado pode ser capaz de captar a influência do tamanho do domínio de maneira semelhante ao MEF. A diferença entre as razões de bloqueio via *openLB*, por sua

vez (**0,27%**) é menor quando comparada com as duas outras abordagens apresentadas.

A figura 28 apresenta o perfil de velocidade ao longo do eixo horizontal central no domínio para $B = 7\%$.

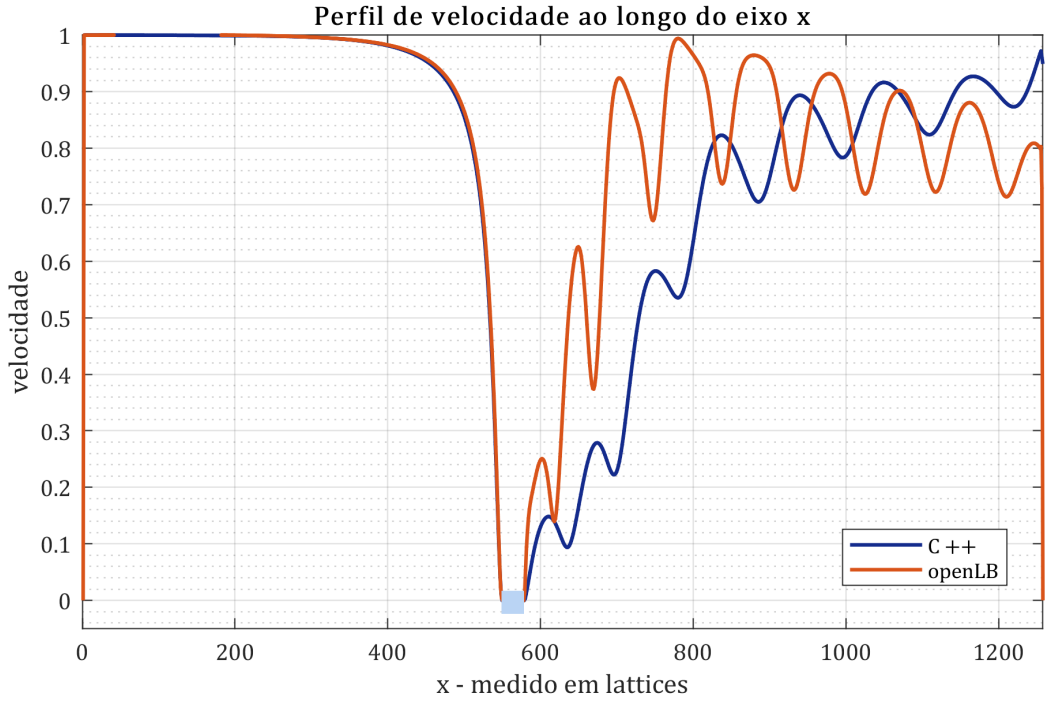

Figura 28 – Perfil de velocidade total ao longo do eixo horizontal central do domínio para $B = 7\%$ e $Re = 100$

Destaca-se a concordância entre o código desenvolvido e o *openLB* para a evolução espacial da velocidade entre a condição de contorno de entrada e o ponto de estagnação no obstáculo. O desvio observado a jusante do corpo pode ser explicado pela natureza transiente do escoamento. A boa concordância entre os perfis de velocidade está de acordo com o obtido para o caso padrão de cavidade, demonstrando a capacidade do código implementado em calcular de forma coerente tal grandeza macroscópica.

7 CONCLUSÃO

O interesse no método Lattice-Boltzmann vem aumentando desde que ele cresceu a partir dos modelos de *Lattice Gas Automata* no final dos anos 1980. Embora ambos os métodos simulem a dinâmica do escoamento a partir do comportamento básico de um gás, em que as moléculas se movem para frente e colidem umas com as outras, o LBM se destacou ao atenuar as desvantagens de seu predecessor enquanto manteve seus pontos fortes. Atualmente, o método é objeto extensivo de pesquisa em razão de sua simplicidade e escalabilidade em computação paralelizada, além da possibilidade de lidar com geometrias mais complexas.

Nesse contexto, o presente trabalho se propôs a avaliar a aplicabilidade do método Lattice-Boltzmann para a simulação de escoamentos externos em Reynolds intermediários através do cálculo de parâmetros específicos do escoamento, como coeficientes de pressão e arrasto, frequência de emissão de vórtices e perfis de velocidade e pressão ao longo do domínio. Os resultados obtidos através da implementação realizada em $C++$ foram comparados com dados publicados obtidos via Método de Elementos Finitos e via LBM através do software de código livre *openLB*.

Em um primeiro momento, contudo, visando avaliar as rotinas de colisão e propagação implementadas, simulou-se um caso padrão de escoamento em cavidade para diferentes valores de *Reynolds* e os resultados foram comparados tanto com a literatura disponível quanto com o *openLB*. Os perfis de velocidade obtidos mostraram excelente concordância, com desvios máximos da ordem de 16% para $Re = 5000$. Considerando os resultados satisfatórios, foi possível, então, alterar as condições de contorno para avaliar escoamentos externos com obstáculos.

O obstáculo foi avaliado dentro de dois domínios diferentes, com razões de bloqueio iguais a 7% e 5% visando compreender também a influência da distância entre a parede e as condições de contorno de simetria nos resultados. Com relação ao coeficiente de pressão no ponto de estagnação, a diferença entre o código desenvolvido e os resultados publicados obtidos via MEF foi da ordem de 2,5% para os dois casos, com diferenças entre as razões de bloqueio semelhantes nos dois métodos, da ordem de $0,76\%$ e $0,20\%$, para o código desenvolvido e o obtido da literatura, respectivamente. Os bons resultados obtidos tanto para o $C_{p,s}$ quanto para os perfis de velocidade na cavidade demonstram a capacidade do método em prever de maneira aceitável quantidades macroscópicas relacionadas a pressão e velocidade através de uma implementação mais simples e que demanda menos poder computacional em relação ao método de elementos finitos, uma vez que por se ater a escala mesoscópica, não necessita resolver o conjunto de equações diferenciais parciais que

descrevem o contínuo.

Com relação ao coeficiente de arrasto, o método utilizado, *momentum exchange*, já mostrou resultados satisfatórios em diversos trabalhos, como discutido no capítulo 4. Os resultados obtidos com o trabalho desenvolvido, por sua vez, mostraram desvios da ordem de $38 - 46\%$, considerando comparações com os resultados via MEF e com o próprio *openLB*. Tais resultados, no entanto, não devem ser encarados como indicativos de uma deficiência do método apresentado por [Ladd 1994], mas sim como motivadores de estudos mais detalhados para uma implementação mais assertiva.

Com relação a frequência de emissão de vórtices, os resultados obtidos via LBM não podem ser considerados satisfatórios. O número de Strouhal calculado é cerca de 53% menor para o código desenvolvido e aproximadamente 109% maior via *openLB*, quando ambos são comparados com os resultados obtidos via método de elementos finitos. É importante destacar, no entanto, que resultados mais coerentes já foram obtidos por outros autores, como [Breuer et al. 2000]. Ademais, apesar da comparação entre os valores absolutos não ter apresentado bons resultados, destaca-se que, dentro de um mesmo método, a variação entre as diferentes razões de bloqueio foram semelhantes para o código desenvolvido e o MEF. Tal fator, aliado as mesmas observações feitas para o coeficiente de pressão e de arrasto, indica que o código implementado pode ser capaz de capturar de maneira semelhante a influência da distância entre a parede e a condição de contorno de simetria na dinâmica do escoamento.

Por fim, conclui-se que os bons resultados apresentados obtidos para o coeficiente de pressão e perfis de velocidade ao longo do domínio indicam a exatidão do LBM para prever determinados parâmetros macroscópicos do escoamento em *Reynolds* intermediários. O código desenvolvido em C++ foi pensado de modo que a implementação de novas condições de contorno, alterações de geometria e condições de simulação fossem feitas de maneira simplificada. A íntegra do código desenvolvido está disponível no anexo A. Dessa forma, sugere-se para trabalhos futuros um estudo detalhado da posição de cálculo da troca de momento a fim de melhorar as previsões para o coeficiente de arrasto e número de Strouhal, em que o último é calculado em função da variação temporal do primeiro. Também, a partir dos resultados obtidos por [Sohankar, Norbergb e Davidson 1997] para cilindros retangulares com incidência, é possível alterar a condição de contorno de velocidade de entrada para verificar o comportamento do método quando a velocidade total não está alinhada com a direção dos lattices. Por fim, sugere-se a migração do código desenvolvido para uma plataforma de computação paralela, como por exemplo a proposta por [Krüger et al. 2017], de modo a explorar de forma mais abrangente as capacidades do método e aumentar a capacidade de testes de diferentes geometrias e parâmetros de simulação ao diminuir o tempo de execução do programa.

REFERÊNCIAS

AGRAWAL, A.; DJENIDI, L.; ANTONIA, R. Investigation of flow around a pair of side-by-side square cylinders using the lattice boltzmann method. *Computers & fluids*, Elsevier, v. 35, n. 10, p. 1093–1107, 2006. Citado 3 vezes nas páginas 7, 36 e 37.

ALAM, M.; CHENG, L. Blockage ratio and mesh dependency study for lattice boltzmann flow around cylinder. In: AMERICAN INSTITUTE OF PHYSICS. *AIP Conference Proceedings*. [S.l.], 2010. v. 1233, n. 1, p. 1453–1458. Citado 5 vezes nas páginas 7, 9, 35, 36 e 47.

BHATNAGAR, P. L.; GROSS, E. P.; KROOK, M. A model for collision processes in gases. i. small amplitude processes in charged and neutral one-component systems. *Physical review*, APS, v. 94, n. 3, p. 511, 1954. Citado na página 15.

BREUER, M. et al. Accurate computations of the laminar flow past a square cylinder based on two different methods: lattice-boltzmann and finite-volume. *International journal of heat and fluid flow*, Elsevier, v. 21, n. 2, p. 186–196, 2000. Citado 6 vezes nas páginas 7, 32, 33, 34, 49 e 53.

BRITO, P. P. d. C. Método Lattice-Boltzmann Aplicado a Aerodinâmica Externa. 2017. Disponível em: <https://repositorio.ufu.br/handle/123456789/20293>. Citado 2 vezes nas páginas 13 e 17.

GINZBURG, I.; D'HUMIÈRES, D.; KUZMIN, A. Optimal stability of advection-diffusion lattice boltzmann models with two relaxation times for positive/negative equilibrium. *Journal of Statistical Physics*, Springer, v. 139, n. 6, p. 1090–1143, 2010. Citado na página 30.

HIGUERA, F.; SUCCI, S.; BENZI, R. Lattice gas dynamics with enhanced collisions. *EPL (Europhysics Letters)*, IOP Publishing, v. 9, n. 4, p. 345, 1989. Citado na página 15.

HIGUERA, F. J.; JIMÉNEZ, J. Boltzmann approach to lattice gas simulations. *EPL (Europhysics Letters)*, IOP Publishing, v. 9, n. 7, p. 663, 1989. Citado na página 15.

HOU, S. et al. Simulation of cavity flow by the lattice boltzmann method. *Journal of computational physics*, Elsevier, v. 118, n. 2, p. 329–347, 1995. Citado 5 vezes nas páginas 14, 38, 40, 41 e 42.

KRÜGER, T. et al. The lattice boltzmann method. *Springer International Publishing*, Springer, v. 10, p. 978–3, 2017. Citado 6 vezes nas páginas 7, 29, 30, 46, 49 e 53.

KUZMIN, A.; GINZBURG, I. Note on the stability of the lattice boltzmann phase propagation equation for the binary-liquid model. Citado na página 30.

LADD, A. J. Numerical simulations of particulate suspensions via a discretized boltzmann equation. part 1. theoretical foundation. *Journal of fluid mechanics*, Cambridge University Press, v. 271, p. 285–309, 1994. Citado 3 vezes nas páginas 29, 48 e 53.

MCNAMARA, G. R.; ZANETTI, G. Use of the boltzmann equation to simulate lattice-gas automata. *Physical review letters*, APS, v. 61, n. 20, p. 2332, 1988. Citado na página 14.

MOHAMAD, A. *Lattice Boltzmann Method*. [S.l.]: Springer, 2011. v. 70. Citado 15 vezes nas páginas 7, 13, 17, 18, 19, 21, 22, 23, 24, 25, 27, 28, 38, 41 e 46.

MOHAMAD, A. *Lattice Boltzmann Method: Fundamentals and Engineering Applications with Computer Codes*. [S.l.]: Springer, 2019. Citado na página 40.

NIU, X. et al. Investigation of stability and hydrodynamics of different lattice boltzmann models. *Journal of statistical physics*, Springer, v. 117, n. 3-4, p. 665–680, 2004. Citado na página 30.

NORBERG, C.; SOHANKAR, A.; DAVIDSON, L. Numerical simulation of unsteady flows around a square two-dimensional cylinder. In: *Twelfth Australian Fluid Mechanics Conference*. [S.l.: s.n.], 1995. p. 517–520. Citado 4 vezes nas páginas 8, 38, 44 e 45.

PENG, C. The lattice boltzmann method for fluid dynamics: theory and applications. *M. Math, Department of Mathematics, Ecole Polytechnique Federale de Lausanne*, v. 2, n. 2, p. 2–4, 2011. Citado na página 14.

PERUMAL, D. A.; KUMAR, G. V.; DASS, A. K. Numerical simulation of viscous flow over a square cylinder using lattice boltzmann method. *ISRN Mathematical Physics*, Hindawi Publishing Corporation, v. 2012, 2012. Citado 3 vezes nas páginas 7, 32 e 33.

QIAN, Y.-H.; D'HUMIÈRES, D.; LALLEMAND, P. Lattice bgk models for navier-stokes equation. *EPL (Europhysics Letters)*, IOP Publishing, v. 17, n. 6, p. 479, 1992. Citado na página 15.

ROWGHANI, S.; MIRZAEI, M.; KAMALI, R. Numerical simulation of fluid flow past a square cylinder using a lattice boltzmann method. *Journal of Aerospace Science and Technology (JAST)*, v. 7, n. 4, p. 9–17, 2010. Citado 2 vezes nas páginas 7 e 34.

SIEBERT, D.; JR, L. H.; PHILIPPI, P. Lattice boltzmann equation linear stability analysis: Thermal and athermal models. *Physical Review E*, APS, v. 77, n. 2, p. 026707, 2008. Citado na página 30.

SOHANKAR, A.; NORBERGB, C.; DAVIDSON, L. Numerical simulation of unsteady low-reynolds number flow around rectangular cylinders at incidence. *Journal of Wind Engineering and Industrial Aerodynamics*, Elsevier, v. 69, p. 189–201, 1997. Citado na página 53.

SUCCI, S. *The lattice Boltzmann equation: for fluid dynamics and beyond*. [S.l.]: Oxford University Press, 2001. Citado na página 14.

SUGA, S. Stability and accuracy of lattice boltzmann schemes for anisotropic advection-diffusion equations. *International Journal of Modern Physics C*, World Scientific, v. 20, n. 04, p. 633–650, 2009. Citado na página 30.

WILLIAMSON, C. H. Vortex dynamics in the cylinder wake. *Annual review of fluid mechanics*, Annual Reviews 4139 El Camino Way, PO Box 10139, Palo Alto, CA 94303-0139, USA, v. 28, n. 1, p. 477–539, 1996. Citado na página 32.

ZDRAVKOVICH, M. M. *Flow around circular cylinders: Volume 2: Applications*. [S.l.]: Oxford university press, 1997. v. 2. Citado na página 32.

Apêndices

APÊNDICE A – CÓDIGO DESENVOLVIDO EM $C++$

```
//---------------------------//
// main file - main.cpp      //
//---------------------------//

#include <iostream>
#include "Case.h"
#include <stdio.h>
#include <time.h>
#include<stdlib.h>
using namespace std;

int main()
{
        Case cavidade;
        // B = 5
        cavidade.setObstacle(549, 285, 30, 30); // px, py, lx, ly [lattices]
        cavidade.SetMaxTimeStep(350000);
        cavidade.setRho(1.00);
        cavidade.setWestInitialVelocity(0.1, 0.00);
        cavidade.setRe(100.0);
        cavidade.setPhysicalDomainSize(4.195, 2); // x, y [m]
        cavidade.setPhysicalObstacleSize(0.1, 0.1); // x, y [m]

        cavidade.setDeltaPrint(0, 1); // initial timestep, timestep interval to save data
        bool convergence = false;
        double tol = 1.0e-4; // tolerancia para erro relativo do campo de velocidade total

        double error = 10.0; int t = 0;
        cavidade.setLatticeVelocities();
        cavidade.SetInitialCondition();
        clock_t tInicio, tFim, tDecorrido;

        tInicio = clock();
        if (convergence){
                while (error > tol) {
                        t = t + 1;
                        cout << "TIMESTEP =  " << t << endl;
                        cavidade.Collision();
                        cavidade.Streaming();
                        cavidade.Boundaries();
                        cavidade.rhouv();
                        error = cavidade.error(t);
                        cavidade.recordtime(t);

                }
        }
        else{
                for (t = 0; t <= cavidade.mstep; t++) {
                        cout << "TIMESTEP =  " << t << endl;
                        cavidade.Collision();
```

```
51                          cavidade.Streaming();
52                          cavidade.Boundaries();
53                          cavidade.rhouv();
54                          cavidade.recordtime(t);
55                  }
56          }
57
58          tFim = clock();
59          tDecorrido = ((tFim - tInicio) / (CLOCKS_PER_SEC / 1000));
60
61          cout << endl;
62          cout << endl;
63          cout << "SALVANDO ARQUIVOS ...." << endl;
64          cavidade.writeLog();
65
66          cavidade.calculatePressure();
67          cavidade.StreamFunction();
68          cavidade.recordfield(t);
69
70          cout << endl;
71          cout << "FIM DA EXECUCAO" << endl;
72          cout << "TEMPO DECORRIDO = " << tDecorrido/3600 << " minutos" << endl;
73          string a;
74          cin >> a;
75  }
76
77  //---------------------------//
78  // header file - Case.h      //
79  //---------------------------//
80  #pragma once
81  #include <vector>
82  #include <iostream>
83  #include <fstream>
84  #include <stdio.h>
85  #include <stdlib.h>
86  using namespace std;
87
88  class Case
89  {
90  public:
91          // DOMINIO
92          static const int nn = 600, mm = 1258, kk = 9;
93          int m = mm - 1, n = nn - 1;
94          int t;
95
96          // ERRO
97          double erro = 10.0;
98          double erso = 10.0;
99          double ers = 10.0;
100         double maior = -10000;
101
102         // VARIAVEIS
103         double w[9]{ 0 }, cx[9]{ 0 }, cy[9]{ 0 }, c[9]{ 0 };
104         double alpha, omega, Re, u0, v0, rhoo, vel, tau, mi;
105         double s1, s2;
106         double rhow;
107         int mstep;
108         int deltaPrint, iniT;
109         double nSpeed[2] = { 0, 0 };
110         double sSpeed[2] = { 0, 0 };
```

```
111         double eSpeed[2] = { 0, 0 };
112         double wSpeed[2] = { 0, 0 };
113         double u[mm][nn], v[mm][nn];
114         double p[mm][nn];
115         double rho[mm][nn];
116         double S[mm][nn];
117
118         double f[9][mm][nn];
119         double feq[9][mm][nn];
120
121         // DOMAIN SIZE [m]
122         double lx1, ly1;
123
124         // OBSTACULO
125         int px, py, lx, ly;
126
127         // MOMENTUM EXCHANGE
128         double deltat = 1.0;
129         double deltaX = 1.0;
130         double deltaY = 1.0;
131         double deltaP[4] = {0, 0, 0, 0}; // [0] north; [1] south; [2] east; [3] west
132         double deltaPx[4] = {0, 0, 0, 0};
133         double deltaPy[4] = {0, 0, 0, 0};
134         double deltaPtotal;
135
136         ofstream arquivocd;
137         string cdname;
138
139         // FUNCOES
140         Case();
141         void setObstacle(int _px, int _py, int _lx, int _ly);
142         void setPhysicalDomainSize(double _lx1, double _ly1);
143         void setPhysicalObstacleSize(double _s1, double _s2);
144         void setLatticeVelocities();
145         void SetMaxTimeStep(int _time);
146         void setRho(double _rho);
147         void setNorthInitialVelocity(double _nsx, double _nsy);
148         void setSouthInitialVelocity(double _ssx, double _ssy);
149         void setEastInitialVelocity(double _esx, double _esy);
150         void setWestInitialVelocity(double _wsx, double _wsy);
151         double getOmega();
152         double getRe();
153         int getTime();
154         void SetInitialCondition();
155         void setRe(double _re);
156         void setDeltaPrint(int ini, int _t);
157         int getDeltaPrint();
158         void Collision();
159         void Streaming();
160         void Boundaries();
161         void rhouv();
162         double error(int _t);
163         void momentumExchange(int _tt);
164         void StreamFunction();
165         void calculatePressure();
166         void recordvector();
167         void recordfield(int t);
168         void recordtime(int _tt);
169         void writeLog();
170 };
```

```
171
172 //---------------------------//
173 // Class file  - Case.cpp    //
174 //---------------------------//
175 #define _USE_MATH_DEFINES
176 #include <iostream>
177 #include <fstream>
178 #include <math.h>
179 #include <stdio.h>
180 #include <stdlib.h>
181 #include <string>
182 #include <vector>
183 #include "Case.h"
184 using namespace std;
185
186 Case::Case() {
187         cdname = "CD_Re=" + to_string(getRe()) + ".csv";
188         arquivocd.open(cdname, ios_base::app); // append instead of overwrite
189         arquivocd << "t, deltaP0, deltaP1, deltaP2, deltaP3, deltaPx0, deltaPx1, deltaPx2, deltaPx3,
          ↪   deltaPy0, deltaPy1, deltaPy2, deltaPy3" << endl;
190         arquivocd.close();
191 }
192
193 void Case::setObstacle(int _px, int _py, int _lx, int _ly) {
194         px = _px;
195         py = _py;
196         lx = _lx;
197         ly = _ly;
198 }
199
200 void Case::setLatticeVelocities(){
201         cx[0] = 0.0;
202         cx[1] = 1.0;
203         cx[2] = 0.0;
204         cx[3] = -1.0;
205         cx[4] = 0.0;
206         cx[5] = 1.0;
207         cx[6] = -1.0;
208         cx[7] = -1.0;
209         cx[8] = 1.0;
210
211         cy[0] = 0.0;
212         cy[1] = 0.0;
213         cy[2] = 1.0;
214         cy[3] = 0.0;
215         cy[4] = -1.0;
216         cy[5] = 1.0;
217         cy[6] = 1.0;
218         cy[7] = -1.0;
219         cy[8] = -1.0;
220
221         for (int i = 0; i < 9; i++)
222         {
223                 c[i] = sqrt(cx[i] * cx[i] + cy[i] * cy[i]);
224         }
225
226         w[0] = 4./9.;
227         for (int i = 1; i < 5; i++)
228         {
229                 w[i] = 1./9.;
```

```
230         }
231
232         for (int i = 5; i < 9; i++)
233         {
234                 w[i] = 1./36.;
235         }
236 }
237
238 void Case::SetMaxTimeStep(int _time){
239         mstep = _time;
240 }
241
242 void Case::setRe(double _re) {
243         Re = _re;
244         alpha = wSpeed[0] * lx / Re;
245         omega = 1.0 / (3.0 * alpha + 0.5);
246         tau = 1/omega;
247 }
248
249 void Case::setRho(double _rho){
250         rhoo = _rho;
251 }
252
253 void Case::setDeltaPrint(int _iniT, int _t){
254         iniT = _iniT;
255         deltaPrint = _t;
256 }
257
258 void Case::setPhysicalDomainSize(double _lx1, double _ly1){
259         lx1 = _lx1;
260         ly1 = _ly1;
261         mi = 1 * s1 / Re;
262 }
263
264 void Case::setPhysicalObstacleSize(double _s1, double _s2){
265         s1 = _s1;
266         s2 = _s2;
267 }
268
269 void Case::setNorthInitialVelocity(double _nsx, double _nsy){
270         nSpeed[0] = _nsx;
271         nSpeed[1] = _nsy;
272 }
273
274 void Case::setWestInitialVelocity(double _wsx, double _wsy){
275         wSpeed[0] = _wsx;
276         wSpeed[1] = _wsy;
277 }
278
279 int Case::getTime(){
280         return mstep;
281 }
282
283 double Case::getRe(){
284         return Re;
285 }
286
287 void Case::SetInitialCondition(){
288         double t1, t2;
289         for (int j = 0; j < nn; j++)
```

```
290         {
291                 for (int i = 0; i < mm; i++)
292                 {
293                         rho[i][j] = rhoo;
294                         u[i][j] = 0.0;
295                         v[i][j] = 0.0;
296                         p[i][j] = 0.0;
297                         S[i][j] = 0.0;
298                 }
299         }
300
301         for (int j = 0; j < nn; j++)
302         {
303                 for (int i = 0; i < mm; i++)
304                 {
305                         t1 = u[i][j] * u[i][j] + v[i][j] * v[i][j];
306                         for (int k = 0; k < 9; k++)
307                         {
308                                 t2 = u[i][j] * cx[k] + v[i][j] * cy[k]; // Velocidade em cada direcao
309                                 feq[k][i][j] = rho[i][j] * w[k] * (1.0 + 3.0 * t2 + 4.50 * t2 * t2 - 1.50 *
                                    ↪  t1);
310                                 f[k][i][j] = feq[k][i][j];
311                         }
312                 }
313         }
314
315         // WEST
316         for (int j = 0; j < nn ; j++)
317         {
318                 u[0][j] = wSpeed[0];
319                 v[0][j] = wSpeed[1];
320
321                 // EAST
322                 u[m][j] = eSpeed[0];
323                 v[m][j] = eSpeed[1];
324         }
325
326                 // SOUTH
327         for (int i = 0; i < mm ; i++)
328         {
329                 u[i][0] = sSpeed[0];
330                 v[i][0] = sSpeed[1];
331
332                 // NORTH
333                 u[i][n] = nSpeed[0];
334                 v[i][n] = nSpeed[1];
335         }
336 }
337
338 void Case::Collision(){
339         double t1, t2;
340         for (int i=0; i < mm; i++)
341         {
342                 for (int j=0; j < nn; j++)
343                 {
344                         t1 = u[i][j] * u[i][j] + v[i][j] * v[i][j]; // Velocidade total para cada ponto da
                            ↪  malha
345                         for (int k = 0; k < 9; k++)
346                         {
347                                 t2 = u[i][j] * cx[k] + v[i][j] * cy[k]; // Velocidade em cada direcao
```

```
348                                 feq[k][i][j] = rho[i][j] * w[k] * (double(1.0) + double(3.0) * t2 +
                                     ↪   double(4.50) * t2 * t2 - double(1.50) * t1);
349                                 f[k][i][j] = omega * feq[k][i][j] + (double(1.0) - omega) * f[k][i][j];
350                         }
351                 }
352         }
353 }
354
355 void Case::Streaming() {
356
357         for (int j = 0; j < nn; j++) {
358                 for (int i = m; i > 0; i--) {            // RIGHT 2 LEFT
359                         f[1][i][j] = f[1][i - 1][j];
360                 }
361
362                 for (int i = 0; i < mm - 1; i++) {         // LEFT 2 RIGHT
363                         f[3][i][j] = f[3][i + 1][j];
364                 }
365         }
366
367         for (int j = n; j > 0; j--) {               // TOP 2 BOTTOM
368                 for (int i = 0; i < mm; i++) {
369                         f[2][i][j] = f[2][i][j - 1];
370                 }
371
372                 for (int i = m; i > 0; i--) {
373                         f[5][i][j] = f[5][i - 1][j - 1];
374                 }
375
376                 for (int i = 0; i < mm - 1; i++) {
377                         f[6][i][j] = f[6][i + 1][j - 1];
378                 }
379         }
380
381         for (int j = 0; j < nn - 1; j++) { // BOTTOM TO TOP
382
383                 for (int i = 0; i < mm; i++) {
384                         f[4][i][j] = f[4][i][j + 1];
385                 }
386
387                 for (int i = 0; i < mm - 1; i++) {
388                         f[7][i][j] = f[7][i + 1][j + 1];
389                 }
390
391                 for (int i = m; i > 0; i--) {
392                         f[8][i][j] = f[8][i - 1][j + 1];
393                 }
394         }
395 }
396
397 void Case::Boundaries() {
398
399         int i, j;
400         for (j = 0; j < nn; j++) {
401                 // CONTORNO OESTE
402                 u0 = wSpeed[0];
403                 v0 = wSpeed[1];
404                 rhow = 1 / (1 - u0) * (f[0][0][j] + f[2][0][j] + f[4][0][j] + 2 * (f[3][0][j] + f[6][0][j]
                     ↪   + f[7][0][j]));
405                 f[1][0][j] = f[3][0][j] + 2./3. * rhow * u0;
```

```
406                 f[5][0][j] = f[7][0][j] - 0.5 * (f[2][0][j] - f[4][0][j]) + 1./6. * rhow * u0 + 0.5 * rhow
                     ↪   * v0;
407                 f[8][0][j] = f[6][0][j] + 0.5 * (f[2][0][j] - f[4][0][j]) + 1./6. * rhow * u0 - 0.5 * rhow
                     ↪   * v0;
408
409                 // CONTORNO LESTE
410                 f[1][m][j] = 2 * f[1][m - 1][j] - f[1][m - 2][j];
411                 f[5][m][j] = 2 * f[5][m - 1][j] - f[5][m - 2][j];
412                 f[8][m][j] = 2 * f[8][m - 1][j] - f[8][m - 2][j];
413         }
414
415         // BOUNCE-BACK
416         // for (i = 0; i < mm; i++) {
417                 //CONTORNO SUL
418                 // f[2][i][0] = f[4][i][0];
419                 // f[5][i][0] = f[7][i][0];
420                 // f[6][i][0] = f[8][i][0];
421
422                 //CONTORNO NORTE
423                 // f[4][i][n] = f[2][i][n];
424                 // f[8][i][n] = f[6][i][n];
425                 // f[7][i][n] = f[5][i][n];
426         // }
427
428         // SYMMETRY
429         for (i = 1; i < mm-1; i++) {
430                 //CONTORNO SUL
431                 f[2][i][0] = f[4][i][0];
432                 f[5][i][0] = f[8][i][0];
433                 f[6][i][0] = f[7][i][0];
434
435                 //CONTORNO NORTE
436                 f[4][i][n] = f[2][i][n];
437                 f[8][i][n] = f[5][i][n];
438                 f[7][i][n] = f[6][i][n];
439         }
440
441         /////////////// OBSTÁCULO (bounce-back)
442         //north boundary
443         for (int i = px; i < px + lx + 1; i++) {
444                 f[2][i][py + ly] = f[4][i][py + ly];
445                 f[5][i][py + ly] = f[7][i][py + ly];
446                 f[6][i][py + ly] = f[8][i][py + ly];
447         }
448
449         //east boundary
450         for (int j = py; j < py + ly + 1; j++) {
451                 f[1][px + lx][j] = f[3][px + lx][j];
452                 f[5][px + lx][j] = f[7][px + lx][j];
453                 f[8][px + lx][j] = f[6][px + lx][j];
454         }
455
456         //south boundary
457         for (int i = px; i < px + lx + 1; i++) {
458                 f[4][i][py] = f[2][i][py];
459                 f[7][i][py] = f[5][i][py];
460                 f[8][i][py] = f[6][i][py];
461         }
462
463         // west boundary
```

```
464         for (int j = py; j < py + ly + 1; j++) {
465                 f[3][px][j] = f[1][px][j];
466                 f[7][px][j] = f[5][px][j];
467                 f[6][px][j] = f[8][px][j];
468         }
469 }
470
471 void Case::rhouv() {
472
473         int i, j, k;
474         double somau, somav, soma;
475
476         for (j = 0; j < nn; j++) {
477                 for (i = 0; i < mm; i++) {
478                         soma = 0.0;
479                         for (k = 0; k < 9; k++) {
480                                 soma = soma + f[k][i][j];
481                         }
482                         rho[i][j] = soma;
483                 }
484         }
485
486         for (int j = 0; j < nn; j++) {
487                 u0 = wSpeed[0];
488                 rho[0][j] = (1 / (1 - u0)) * f[0][0][j] + f[2][0][j] + f[4][0][j] + 2. * (f[3][0][j] +
                    ↪  f[6][0][j] + f[7][0][j]);
489         }
490
491         for (i = 0; i < mm; i++) {
492                 for (j = 0; j < nn; j++) {
493                         somau = 0.0;
494                         somav = 0.0;
495                         for (k = 0; k < 9; k++) {
496                                 somau = somau + f[k][i][j] * cx[k];
497                                 somav = somav + f[k][i][j] * cy[k];
498                         }
499                         u[i][j] = somau / rho[i][j];
500                         v[i][j] = somav / rho[i][j];
501                 }
502         }
503
504         for (int j = 0; j < nn; j++) {
505                 v[m][j] = 0.0;
506         }
507
508
509         for (int i = px; i < px + lx + 1; i++) {
510                 for (int j = py; j < py + ly + 1; j++) {
511                         u[i][j] = 0.0;
512                         v[i][j] = 0.0;
513                         //rho[i][j] = 0.0;
514                 }
515         }
516 }
517
518 double Case::error(int _t) {
519         t = _t;
520         if (t > 0 && t%1000 == 0){
521                 StreamFunction();
522                 maior = -10;
```

```
523                 for (int i = 0; i < mm; i++) {
524                         for (int j = 0; j < nn; j++) {
525                                 if (abs(S[i][j]) > maior){
526                                         maior = S[i][j];
527                                 }
528                          }
529                  }
530                 ers = maior;
531                 erro = abs(ers - erso);
532                 erso = ers;
533         }
534         return erro;
535 }
536
537 void Case::calculatePressure() {
538
539         for (int i = 0; i < mm; i++) {
540                 for (int j = 0; j < nn; j++) {
541                         p[i][j] = rho[i][j] / 3;
542                 }
543         }
544
545         for (int i = px; i < px + lx + 1; i++) {
546                 for (int j = py; j < py + ly + 1; j++) {
547                         p[i][j] = 0.0;
548                 }
549         }
550 }
551
552 void Case::StreamFunction() {
553         //streamfunction calculations
554         S[0][0] = 0;
555         for (int i = 1; i < mm; i++) {
556                 double rhoav = 0.5 * (rho[i - 1][0] + rho[i][0]);
557                 if (i != 0) S[i][0] = S[i - 1][0] - rhoav * 0.5 * (v[i - 1][0] + v[i][0]);
558                 for (int j = 1; j < nn; j++) {
559                         double rhom = 0.5 * (rho[i][j] + rho[i][j - 1]);
560                         S[i][j] = S[i][j - 1] + rhom * 0.5 * (u[i][j - 1] + u[i][j]);
561                 }
562         }
563 }
564
565 void Case::recordvector() {
566         int i, j;
567
568         string filename = "Re=" + to_string(getRe()) + "_t=" + to_string(getTime()) + "_U.txt";
569         ofstream arquivo;
570         arquivo.open(filename);
571         for (j = n; j > -1; j--) {
572                 for (i = 0; i < mm; i++) {
573                         arquivo << u[i][j] << ";";
574                 }
575                 arquivo << endl;
576         }
577         arquivo.close();
578
579         filename = "Re=" + to_string(getRe()) + "_t=" + to_string(getTime()) + "_V.txt";
580         arquivo.open(filename);
581         for (j = n; j > -1; j--) {
582                 for (i = 0; i < mm; i++) {
```

```
583                         arquivo << v[i][j] << ";";
584                 }
585                 arquivo << endl;
586         }
587         arquivo.close();
588 }
589
590 void Case::recordfield(int _timestep) {
591         int i, j;
592         int time = _timestep;
593         string filename = "outFIELD_Re=" + to_string(getRe()) + ".dat." + to_string(time);
594         ofstream arquivo;
595         arquivo.open(filename);
596         arquivo << "VARIABLES = X, Y, U, V, UV, S, P, RHO" << endl;
597         arquivo << "ZONE " << "I=" << m + 1 << " J=" << n + 1 << "," << "F=BLOCK" << endl;
598
599         for (j = 0; j < nn ; j++) {
600                 for (i = 0; i < mm; i++) {
601                         arquivo << i << " ";
602                 }
603                 arquivo << endl;
604         }
605
606         for (j = 0; j < nn ; j++) {
607                 for (i = 0; i < mm; i++) {
608                         arquivo << j << " ";
609                 }
610                 arquivo << endl;
611         }
612
613         for (j = 0; j < nn ; j++) {
614                 for (i = 0; i < mm; i++) {
615                         arquivo << u[i][j] << " ";
616                 }
617                 arquivo << endl;
618         }
619
620         for (j = 0; j < nn ; j++) {
621                 for (i = 0; i < mm; i++) {
622                         arquivo << v[i][j] << " ";
623                 }
624                 arquivo << endl;
625         }
626
627         for (j = 0; j < nn; j++) {
628                 for (i = 0; i < mm; i++) {
629                         arquivo << (sqrt(u[i][j] * u[i][j] + v[i][j] * v[i][j])) << " ";
630                 }
631                 arquivo << endl;
632         }
633
634
635         for (j = 0; j < nn ; j++) {
636                 for (i = 0; i < mm; i++) {
637                         arquivo << S[i][j] << " ";
638                 }
639                 arquivo << endl;
640         }
641
642         for (j = 0; j < nn; j++) {
```

```
643                 for (i = 0; i < mm; i++) {
644                         arquivo << p[i][j] << " ";
645                 }
646                 arquivo << endl;
647         }
648
649         for (j = 0; j < nn; j++) {
650                 for (i = 0; i < mm; i++) {
651                         arquivo << rho[i][j] << " ";
652                 }
653                 arquivo << endl;
654         }
655
656         arquivo.close();
657 }
658
659 void Case::recordtime(int _tt) {
660         if (_tt > iniT && _tt % deltaPrint == 0 && _tt != getTime() ){
661                 calculatePressure();
662                 StreamFunction();
663                 recordfield(_tt);
664                 momentumExchange(_tt);
665                 momentumExchange_v2(_tt);
666         }
667 }
668
669 void Case::writeLog(){
670         double deltaX = s1 / lx;
671         double deltaT = (1/3) * (tau - 0.5) * deltaX * deltaX / mi;
672         double TT = mstep * deltaT;
673
674         ofstream logfile;
675         string logname = "logfile_Re =" + to_string(getRe()) + ".txt";
676         logfile.open(logname, ios_base::app); // append instead of overwrite
677
678         logfile << " --- SIMULATION PARAMETERS --" << endl;
679         logfile << endl;
680         logfile << "Re = " << Re << endl;
681         logfile << endl;
682         logfile << "--> LATTICE UNITS" << endl;
683         logfile << "Initial speed (west boundary), U = " << wSpeed[0] << endl;
684         logfile << "Lattice viscosity, alpha = " << alpha << endl;
685         logfile << "Domain size in x-direction = " << mm << " lattices" << endl;
686         logfile << "Domain size in y-direction = " << nn << " lattices" << endl;
687         logfile << "Obstacle Resolution" << lx << " by " << ly << " lattices" << endl;
688         logfile << "Obstacle Position px = " << px << " py = " << py << endl;
689         logfile << "Initial rho = " << rhoo << endl;
690         logfile << endl;
691
692
693         logfile << "--> PHYSICAL UNITS" << endl;
694         logfile << "Domain size = " << lx1 << " by " << ly1 << " [m]" << endl;
695         logfile << "Obstacle Size = " << s1 << " by " << s2 << " [m]" << endl;
696         logfile << "Time relaxation factor tau = " << tau << endl;
697         logfile << "deltaX = " << deltaX << " [m]" << endl;
698         logfile << "deltaT = " << deltaT << " [s]" << endl;
699         logfile << endl;
700
701         logfile << "--> TIME" << endl;
702         logfile << "TOTAL SIMULATION TIME = " << mstep << "timestep" << endl;
```

```
703         logfile << "TOTAL SIMULATION TIME = " << TT << "seconds" << endl;
704         logfile << "INITIAL TIMESTEP FOR SAVING DATA = " << iniT << endl;
705         logfile << "TIMESTEP INTERVAL FOR SAVING DATA = " << deltaPrint << endl;
706         logfile << endl;
707 
708         logfile << "--- END OF FILE ---" << endl;
709         logfile << endl;
710 
711         logfile.close();
712 }
713 
714 void Case::momentumExchange(int _tt){
715         int time = _tt;
716         double b = cos(45*M_PI/180);
717         deltaP[0] = 0.0;
718         deltaP[1] = 0.0;
719         deltaP[2] = 0.0;
720         deltaP[3] = 0.0;
721 
722         deltaPx[0] = 0.0;
723         deltaPx[1] = 0.0;
724         deltaPx[2] = 0.0;
725         deltaPx[3] = 0.0;
726 
727         deltaPy[0] = 0.0;
728         deltaPy[1] = 0.0;
729         deltaPy[2] = 0.0;
730         deltaPy[3] = 0.0;
731 
732         int pxf_w = px-1;
733         int pxf_e = px + lx + 1;
734         int pyf_s = py-1;
735         int pyf_n = py + ly + 1;
736 
737         // north - IN - OUT
738         for (int i = px-1; i < px + lx + 2; i++){
739                 deltaP[0] = deltaP[0] + c[4]*(f[2][i][py+ly-1] + f[4][i][pyf_n]) + c[7]*(f[5][i][py+ly-1] +
                    ↪   f[7][i][pyf_n]) +
740                         c[8]*(f[6][i][py+ly-1] + f[8][i][pyf_n]);
741 
742                 deltaPy[0] = deltaPy[0] +  c[4]*(f[2][i][py+ly-1] + f[4][i][pyf_n]) +
                    ↪   b*c[7]*(f[5][i][py+ly-1] + f[7][i][pyf_n]) +
743                         b*c[8]*(f[6][i][py+ly-1] + f[8][i][pyf_n]);
744 
745         // south
746                 deltaP[1] = deltaP[1] + c[2]*(f[4][i][py+1] + f[2][i][pyf_s]) + c[5]*(f[7][i][py+1] +
                    ↪   f[5][i][pyf_s]) +
747                         c[8]*(f[8][i][py+1] + f[6][i][pyf_s]);
748 
749                 deltaPy[1] = deltaPy[1] + c[2]*(f[4][i][py+1] + f[2][i][pyf_s]) +b* c[5]*(f[7][i][py+1] +
                    ↪   f[5][i][pyf_s]) +
750                         b*c[6]*(f[8][i][py+1] + f[6][i][pyf_s]);
751         }
752 
753         // east
754         for (int j = py-1; j < py + ly + 2; j++){
755                 deltaP[2] = deltaP[2] + c[3]*(f[1][px+lx-1][j] + f[3][pxf_e][j]) + c[7]*(f[5][px+lx-1][j] +
                    ↪   f[7][pxf_e][j]) +
756                         c[3]*(f[1][px+lx-1][j] + f[3][pxf_e][j]);
757 
```

```
758                 deltaPx[2] = deltaPx[2] + c[3]*(f[1][px+lx-1][j] + f[3][pxf_e][j]) +
                     ↪   b*c[7]*(f[5][px+lx-1][j] + f[7][pxf_e][j]) +
759                       b*c[3]*(f[1][px+lx-1][j] + f[3][pxf_e][j]);
760
761         // west
762                 deltaP[3] = deltaP[3] + c[1]*(f[3][px+1][j] + f[1][pxf_w][j]) + c[5]*(f[7][px+1][j] +
                     ↪   f[5][pxf_w][j]) +
763                       c[8]*(f[6][px+1][j] + f[8][pxf_w][j]);
764
765                 deltaPx[3] = deltaPx[3] + c[1]*(f[3][px+1][j] + f[1][pxf_w][j]) + b*c[5]*(f[7][px+1][j] +
                     ↪   f[5][pxf_w][j]) +
766                       b*c[8]*(f[6][px+1][j] + f[8][pxf_w][j]);
767         }
768
769         arquivocd.open(cdname, ios_base::app); // append instead of overwrite
770         arquivocd << time <<  ";" << deltaP[0] << ";" << deltaP[1] << ";" << deltaP[2] << ";" << deltaP[3]
             ↪   << ";" << deltaPx[0] << ";" << deltaPx[1] << ";" << deltaPx[2] << ";" << deltaPx[3] << ";" <<
             ↪   deltaPy[0] << ";" << deltaPy[1] << ";" << deltaPy[2] << ";" << deltaPy[3] << endl;
771         arquivocd.close();
772 }
```

APÊNDICE B – TEMPLATE UTILIZADO NO OPENLB

```
/*  Lattice Boltzmann sample, written in C++, using the OpenLB library

 Copyright (C) 2006-2014 Jonas Latt, Mathias J. Krause, Vojtech Cvrcek, Peter Weisbrod
 E-mail contact: info@openlb.net
 The most recent release of OpenLB can be downloaded at <http://www.openlb.net/>

 This program is free software; you can redistribute it and/or modify it under the terms of the GNU General
 ↪  Public License as published by the Free Software Foundation; either version 2 of the License, or (at
 ↪  your option) any later version.

 This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the
 ↪  implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public
 ↪  License for more details.

 square2d.cpp:
 This example examines a steady flow past a cylinder placed in a domain.
 The square cylinder is offset somewhat from the center of the flow to make the steady-state symmetrical
 ↪  flow unstable. At the inlet, a uniform profile is imposed on the velocity, whereas the outlet
 ↪  implements a Dirichlet pressure condition set by p = 0. In the upper and lower boundaries, a slip
 ↪  boundary condition is imposed to simulate an external flow (symmetry condition).

 The following code was adapted by Lucas Lutffalla Estevam (lutffallae@gmail.com)  considering the example
 ↪  'cylinder2d' avaiable in the standard instalation  package of openLB.
 */

#include "olb2D.h"
#ifndef OLB_PRECOMPILED // Unless precompiled version is used,
#include "olb2D.hh"     // include full template code
#endif
#include <vector>
#include <cmath>
#include <iostream>
#include <fstream>

using namespace olb;
using namespace olb::descriptors;
using namespace olb::graphics;
using namespace olb::util;
using namespace std;

typedef double T;
#define DESCRIPTOR D2Q9<>

// Parameters for the simulation setup
const int N = 30;                                  // resolution of the model
const T Re = 100.;                                 // Reynolds number
const T maxPhysT = 150;                            // max. simulation time in s, SI unit

// Geometry definition
const T lengthX = 4.195;                           // width of domain
```

```
43  const T lengthY = 1.4;                              // height of domain
44  const T sideSquareX   = 0.1;                        // square cylinder size [m]
45  const T sideSquareY   = sideSquareX;
46  // Position of cylinder along the x-direction:
47  const T centerSquareX = 1.83;
48  // Position of cylinder along y-direction:
49  const T centerSquareY = lengthY/2 - sideSquareY/2;
50  const T L = sideSquareX/N;                          // lattice length
51
52  // Stores geometry information in form of material numbers
53  void prepareGeometry( UnitConverter<T, DESCRIPTOR> const& converter,
54                        SuperGeometry2D<T>& superGeometry )
55  {
56    OstreamManager clout( std::cout,"prepareGeometry" );
57    clout << "Prepare Geometry ..." << std::endl;
58    Vector<T,2> extend( lengthX,lengthY );
59    Vector<T,2> cylinder( sideSquareX, sideSquareY );
60    Vector<T,2> square0( centerSquareX,centerSquareY );
61    Vector<T,2> origin;
62    IndicatorCuboid2D<T> square( cylinder, square0 );
63    superGeometry.rename( 0,2 );
64    superGeometry.rename( 2,1,1,1 );
65
66    // Set material number for inflow
67    extend[0] = 2.*L;
68    origin[0] = -L;
69    IndicatorCuboid2D<T> inflow( extend, origin );
70    superGeometry.rename( 2,3,1,inflow );
71    // Set material number for outflow
72    origin[0] = lengthX-L;
73    IndicatorCuboid2D<T> outflow( extend, origin );
74    superGeometry.rename( 2,4,1,outflow );
75    // Set material number for cylinder
76    superGeometry.rename( 1,5,square );
77
78    // Removes all not needed boundary voxels outside the surface
79    superGeometry.clean();
80    superGeometry.checkForErrors();
81
82    superGeometry.print();
83
84    clout << "Prepare Geometry ... OK" << std::endl;
85  }
86
87  // Set up the geometry of the simulation
88  void prepareLattice( SuperLattice2D<T,DESCRIPTOR>& sLattice,
89                       UnitConverter<T, DESCRIPTOR> const& converter,
90                       Dynamics<T, DESCRIPTOR>& bulkDynamics,
91                       sOnLatticeBoundaryCondition2D<T,DESCRIPTOR>& sBoundaryCondition,
92                       sOffLatticeBoundaryCondition2D<T,DESCRIPTOR>& offBc,
93                       SuperGeometry2D<T>& superGeometry )
94  {
95
96    OstreamManager clout( std::cout,"prepareLattice" );
97    clout << "Prepare Lattice ..." << std::endl;
98
99    const T omega = converter.getLatticeRelaxationFrequency();
100
101   // Material=0 -->do nothing
102   sLattice.defineDynamics( superGeometry, 0, &instances::getNoDynamics<T, DESCRIPTOR>() );
```

```
103
104   // Material=1 -->bulk dynamics
105   // Material=3 -->bulk dynamics (inflow)
106   // Material=4 -->bulk dynamics (outflow)
107   auto bulkIndicator = superGeometry.getMaterialIndicator({1, 3, 4});
108   sLattice.defineDynamics( bulkIndicator, &bulkDynamics );
109
110   // Material=2 -->bounce back
111   sLattice.defineDynamics( superGeometry, 2, &instances::getNoDynamics<T, DESCRIPTOR>() );
112   sBoundaryCondition.addSlipBoundary(superGeometry, 2);
113
114   // Material=5 -->bounce back
115   sLattice.defineDynamics(superGeometry, 5, &instances::getBounceBack<T, DESCRIPTOR>());
116
117   // Setting of the boundary conditions
118   sBoundaryCondition.addVelocityBoundary( superGeometry, 3, omega );
119   sBoundaryCondition.addPressureBoundary( superGeometry, 4, omega );
120
121   // Initial conditions
122   AnalyticalConst2D<T,T> rhoF( 1 );
123   std::vector<T> velocity( 2,T( 0 ) );
124   AnalyticalConst2D<T,T> uF( velocity );
125
126   // Initialize all values of distribution functions to their local equilibrium
127   sLattice.defineRhoU( bulkIndicator, rhoF, uF );
128   sLattice.iniEquilibrium( bulkIndicator, rhoF, uF );
129
130   // Make the lattice ready for simulation
131   sLattice.initialize();
132
133   clout << "Prepare Lattice ... OK" << std::endl;
134 }
135
136 // Generates a slowly increasing inflow for the first iTMaxStart timesteps
137 void setBoundaryValues( SuperLattice2D<T, DESCRIPTOR>& sLattice,
138                         UnitConverter<T, DESCRIPTOR> const& converter, int iT,
139                         SuperGeometry2D<T>& superGeometry )
140 {
141
142   OstreamManager clout( std::cout,"setBoundaryValues" );
143
144   // No of time steps for smooth start-up
145   int iTmaxStart = converter.getLatticeTime( maxPhysT*0.10 );
146   int iTupdate = 5;
147
148   if ( iT%iTupdate==0 && iT<= iTmaxStart ) {
149
150     // Smooth start curve, polynomial
151     PolynomialStartScale<T,T> StartScale( iTmaxStart, T( 1 ) );
152
153     // Creates and sets the Poiseuille inflow profile using functors
154     T iTvec[1] = {T( iT )};
155     T frac[1] = {};
156     StartScale( frac,iTvec );
157
158         AnalyticalConst2D<T,T> u0(converter.getCharLatticeVelocity()*frac[0],0);
159         sLattice.defineU(superGeometry, 3,  u0);
160   }
161 }
162
```

```cpp
// Computes the pressure drop between the voxels before and after the cylinder
void getResults( SuperLattice2D<T, DESCRIPTOR>& sLattice,
                 UnitConverter<T, DESCRIPTOR> const& converter, int iT,
                 SuperGeometry2D<T>& superGeometry, Timer<T>& timer,
                 CircularBuffer<T>& buffer )
{

  OstreamManager clout( std::cout,"getResults" );

  SuperVTMwriter2D<T> vtmWriter( "square2d" );
  SuperLatticePhysVelocity2D<T, DESCRIPTOR> velocity( sLattice, converter );
  SuperLatticePhysPressure2D<T, DESCRIPTOR> pressure( sLattice, converter );
  vtmWriter.addFunctor( velocity );
  vtmWriter.addFunctor( pressure );

  const int vtkIter  = converter.getLatticeTime( .3 );
  const int statIter = converter.getLatticeTime( .1 );

  T point[2] = {};
  point[0] = centerSquareX + 3*sideSquareX;
  point[1] = centerSquareY;
  AnalyticalFfromSuperF2D<T> intpolateP( pressure, true );
  T p;
  intpolateP( &p,point );
  buffer.insert(p);

  if ( iT == 0 ) {
    // Writes the geometry, cuboid no. and rank no. as vti file for visualization
    SuperLatticeGeometry2D<T, DESCRIPTOR> geometry( sLattice, superGeometry );
    SuperLatticeCuboid2D<T, DESCRIPTOR> cuboid( sLattice );
    SuperLatticeRank2D<T, DESCRIPTOR> rank( sLattice );
    vtmWriter.write( geometry );
    vtmWriter.write( cuboid );
    vtmWriter.write( rank );
    vtmWriter.createMasterFile();
  }

  // Writes the vtk files
  if ( iT%vtkIter == 0 && iT > 0 ) {
    vtmWriter.write( iT );

    SuperEuklidNorm2D<T, DESCRIPTOR> normVel( velocity );
    BlockReduction2D2D<T> planeReduction( normVel, 600, BlockDataSyncMode::ReduceOnly );
    // write output as JPEG
    heatmap::write(planeReduction, iT);
  }

  // Gnuplot constructor (must be static!)
  // for real-time plotting: gplot("name", true) // experimental!
  static Gnuplot<T> gplot( "drag" );

  // write pdf at last time step
  if ( iT == converter.getLatticeTime( maxPhysT )-1 ) {
    // writes pdf
    gplot.writePDF();
  }

  // Writes output on the console
  if ( iT%statIter == 0 ) {
```

```
222       // Timer console output
223       timer.update( iT );
224       timer.printStep();
225       clout << "Circular buffer test: moving average pointwise value=" << buffer.average() << std::endl;
226
227       // Lattice statistics console output
228       sLattice.getStatistics().print( iT,converter.getPhysTime( iT ) );
229
230       // Drag, lift, pressure drop
231       AnalyticalFfromSuperF2D<T> intpolatePressure( pressure, true );
232       SuperLatticePhysDrag2D<T,DESCRIPTOR> drag( sLattice, superGeometry, 5, converter );
233
234
235       T point1[2] = {};
236       T point2[2] = {};
237
238       point1[0] = centerSquareX - sideSquareX;
239       point1[1] = centerSquareY;
240
241       point2[0] = centerSquareX + sideSquareX;
242       point2[1] = centerSquareY;
243
244       T p1, p2;
245       intpolatePressure( &p1,point1 );
246       intpolatePressure( &p2,point2 );
247
248       clout << "pressure1=" << p1;
249       clout << "; pressure2=" << p2;
250
251       T pressureDrop = p1-p2;
252       clout << "; pressureDrop=" << pressureDrop;
253
254       int input[3] = {};
255       T _drag[drag.getTargetDim()];
256       drag( _drag,input );
257       clout << "; drag=" << _drag[0] << "; lift=" << _drag[1] << endl;
258
259       // set data for gnuplot: input={xValue, yValue(s), names (optional), position of key (optional)}
260       gplot.setData( converter.getPhysTime( iT ), {_drag[0]}, {"drag(openLB)", "drag(schaeferTurek)"},
          ↪   "bottom right", {'l','l'} );
261       // writes a png in one file for every timestep, if the file is open it can be used as a "liveplot"
262       gplot.writePNG();
263
264       // every (iT\%vtkIter) write an png of the plot
265       if ( iT\%( vtkIter ) == 0 ) {
266         // writes pngs: input={name of the files (optional), x range for the plot (optional)}
267         gplot.writePNG( iT, maxPhysT );
268       }
269     }
270   }
271
272   int main( int argc, char* argv[] )
273   {
274
275     // === 1st Step: Initialization ===
276     olbInit( &argc, &argv );
277     singleton::directories().setOutputDir( "./tmp/" );
278     OstreamManager clout( std::cout,"main" );
279     // display messages from every single mpi process
```

```
280   //clout.setMultiOutput(true);
281
282   UnitConverterFromResolutionAndRelaxationTime<T, DESCRIPTOR> const converter(
283     int {N},                        // resolution: number of voxels per charPhysL
284     (T)   0.59,                     // latticeRelaxationTime: relaxation time, have to be greater than 0.5!
285     (T)   sideSquareY,               // charPhysLength: reference length of simulation geometry
286     (T)   1.0,                      // charPhysVelocity: maximal/highest expected velocity during
        ↪   simulation in __m / s__
287     (T)   1.0*sideSquareY/Re,        // physViscosity: physical kinematic viscosity in __m^2 / s__
288     (T)   1.0                       // physDensity: physical density in __kg / m^3__
289   );
290   // Prints the converter log as console output
291   converter.print();
292   // Writes the converter log in a file
293   converter.write("square2d");
294
295   // === 2rd Step: Prepare Geometry ===
296   Vector<T,2> extend( lengthX,lengthY );
297   Vector<T,2> origin;
298   IndicatorCuboid2D<T> cuboid( extend, origin );
299
300   // Instantiation of a cuboidGeometry with weights
301 #ifdef PARALLEL_MODE_MPI
302   const int noOfCuboids = singleton::mpi().getSize();
303 #else
304   const int noOfCuboids = 7;
305 #endif
306   CuboidGeometry2D<T> cuboidGeometry( cuboid, L, noOfCuboids );
307
308   // Instantiation of a loadBalancer
309   HeuristicLoadBalancer<T> loadBalancer( cuboidGeometry );
310
311   // Instantiation of a superGeometry
312   SuperGeometry2D<T> superGeometry( cuboidGeometry, loadBalancer, 2 );
313
314   prepareGeometry( converter, superGeometry );
315
316   // === 3rd Step: Prepare Lattice ===
317   SuperLattice2D<T, DESCRIPTOR> sLattice( superGeometry );
318
319   BGKdynamics<T, DESCRIPTOR> bulkDynamics( converter.getLatticeRelaxationFrequency(),
      ↪   instances::getBulkMomenta<T, DESCRIPTOR>() );
320
321   // choose between local and non-local boundary condition
322   sOnLatticeBoundaryCondition2D<T,DESCRIPTOR> sBoundaryCondition( sLattice );
323   // createInterpBoundaryCondition2D<T,DESCRIPTOR>(sBoundaryCondition);
324   createLocalBoundaryCondition2D<T,DESCRIPTOR>( sBoundaryCondition );
325
326   sOffLatticeBoundaryCondition2D<T, DESCRIPTOR> sOffBoundaryCondition( sLattice );
327   createBouzidiBoundaryCondition2D<T, DESCRIPTOR> ( sOffBoundaryCondition );
328
329   prepareLattice( sLattice, converter, bulkDynamics, sBoundaryCondition, sOffBoundaryCondition,
      ↪   superGeometry );
330
331   // === 4th Step: Main Loop with Timer ===
332   CircularBuffer<T> buffer(converter.getLatticeTime(.2));
333   clout << "starting simulation..." << endl;
334   Timer<T> timer( converter.getLatticeTime( maxPhysT ), superGeometry.getStatistics().getNvoxel() );
335   timer.start();
336
```

```
337   for ( int iT = 0; iT < converter.getLatticeTime( maxPhysT ); ++iT ) {
338     // === 5th Step: Definition of Initial and Boundary Conditions ===
339     setBoundaryValues( sLattice, converter, iT, superGeometry );
340
341     // === 6th Step: Collide and Stream Execution ===
342     sLattice.collideAndStream();
343
344     // === 7th Step: Computation and Output of the Results ===
345     getResults( sLattice, converter, iT, superGeometry, timer, buffer );
346   }
347
348   timer.stop();
349   timer.printSummary();
350 }
```