High Performance Computing with Python

Final Report

Theodoros Athanasiadis

Matr. Number: 5365502

theodorathanasiadis@gmail.com

University of Freiburg

August 15, 2022

Contents

A note from the author

All of the code along with instructions on how to execute the experiments presented here is publicly availabe on Github at `https://github.com/theodorju/fr-hpcpy-pub`.

If further input is needed from my part, or if any error is encountered during the execution of the provided code, please contact me at theodorathanasiadis@gmail.com.

1

The Lattice Boltzmann Equation[1]

1.1 Boltzmann Transport Equation

The Boltzmann Transport Equation (BTE) can be written as

$$\frac{\partial f(\mathbf{r},\mathbf{v},t)}{\partial t} + \mathbf{v}\nabla_{\mathbf{r}} f(\mathbf{r},\mathbf{v},t) + \mathbf{a}\nabla_{\mathbf{v}} f(\mathbf{r},\mathbf{v},t) = C(f(\mathbf{r},\mathbf{v},t)) \tag{1.1}$$

On equation 1.1 the l.h.s. is the streaming part and the r.h.s. is the collision part. The distribution function f describes the probability of finding particles with a certain range of velocities at a certain range of locations in time [3].

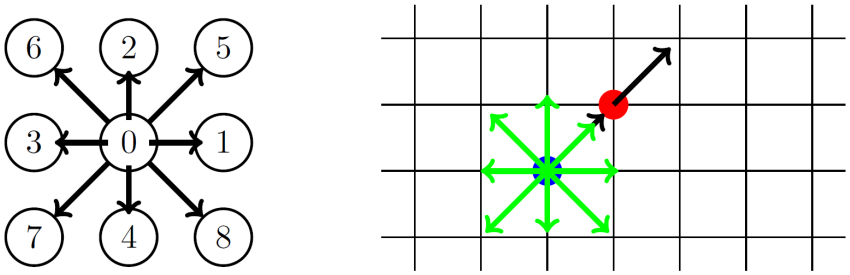
1.2 Discretization of BTE

In order to move from the continuous physical world to an approximation where we can run simulations, we use the discretized version of the Boltzmann Transport Equation. In this implementation we consider the D2Q9 discretization scheme, i.e. 2-Dimensional Space with 9-Dimensional velocities, as depicted in Figure 1.1

The nine different velocity channels of the D2Q9 discretization scheme are defined as

$$\mathbf{c} = \begin{pmatrix} 0 & 1 & 0 & -1 & 0 & 1 & -1 & -1 & 1 \\ 0 & 0 & 1 & 0 & -1 & 1 & 1 & -1 & -1 \end{pmatrix}^T.$$

[1]Unless cited otherwise, the theoretical background presented here is based on the lectures of Prof. Andreas Greiner on *High-Performance Computing: Fluid Mechanics with Python* at University of Freiburg as presented at Summer Semester of 2022.[2]



(a) (b)

Figure 1.1: D2Q9 discretization. (a) The 9 different velocity channels. (b) The lattice used for discretization [2]

The discretization means that in one time step Δt a point moves in a distance Δx given by $c_i \Delta t = \Delta x$.

Using this discretization scheme we can write the discretized version of the BTE, the Lattice Boltzmann Equation (LTE) as follows

$$f_i(\mathbf{r} + \mathbf{c}_i \Delta t, t + \Delta t) = f_i(\mathbf{r}, t) + C_i(\mathbf{r}, t). \tag{1.2}$$

2

Streaming Operator[1]

2.1 Streaming Operator

As a first step in the implementation of the LBE solver, we assume movement of particles in vacuum. This approximation leads to zero collision interactions in equation (1.2) which means we can focus solely on the streaming operator.

In this case the LBE equation becomes:

$$f_i(\mathbf{r} + \mathbf{c}_i \Delta t, t + \Delta t) = f_i(\mathbf{r}, t). \tag{2.1}$$

And the density and velocity at each point in the grid are given by:

$$\rho(\mathbf{x}_j, t) = \sum_i f_i(\mathbf{x}_j, t) \tag{2.2}$$

$$\mathbf{u}(\mathbf{x}_j, t) = \frac{1}{\rho(\mathbf{x}_j, t)} \sum_i \mathbf{c}_i f_i(\mathbf{x}_j, t) \tag{2.3}$$

2.2 Periodic Boundary Conditions

For the simple implementation of the streaming operator, we will also assume periodic boundary conditions (PBC). Periodic boundary conditions mean that the solution is periodic and that the fluid leaving the domain at one side will immediately re-enter at the opposite side[5][4]. As a result, mass and momentum are conserved at all times. Periodic boundary conditions are extremely useful when we want to isolate periodic flow patterns within a larger system.

Figure 2.1 depicts the Periodic Boundary Conditions. The update equation for periodic boundary conditions based on that figure is:

[1]Unless cited otherwise, the theoretical background presented here is based on the lectures of Prof. Andreas Greiner on *High-Performance Computing: Fluid Mechanics with Python* at University of Freiburg as presented at Summer Semester of 2022.[2]

$$f_i(\mathbf{x}_1, t) = f_i(\mathbf{x}_N, t). \tag{2.4}$$

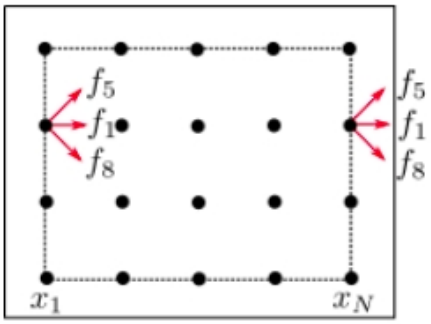


Figure 2.1: Representation of Periodic Boundary Conditions.[2]

2.3 Streaming Operator in Python

This chapter discusses the implementation of the Streaming Operator using PBC. Where necessary listings of code will be presented here. As a quick reminder, the full implementation is available at `https://github.com/theodorju/fr-hpcpy-pub`.

The main functionality required for Streaming Operator is implemented in file `src/lbm.py`, with methods that calculate the density and velocity at each point in the grid.

The streaming operator is implemented by looping over all velocity channels and using the method `np.roll`[2] as follows:

Listing 2.1: Streaming Operation

```
for i in range(n_channels):
    proba_density[i, :, :] = \
        np.roll(proba_density[i, :, :],
                velocity_channels[i],
                axis=(0, 1))
```

The implementation of streaming operator can be executed as[3]

Listing 2.2: Streaming Operation Execution

```
>>> python src/lbm.py
```

[2] `https://numpy.org/doc/stable/reference/generated/numpy.roll.html`

[3] The command presented here assumes that the user is in the cloned directory of the project

The default execution will artificially increase the mass at one point in the grid, perform 3 streaming operations, and generate and save 4 graphs (one corresponds to initial setup) containing the density at each point of the grid after each time step. The plots are named using the pattern: `density_<iteration_number>.png`.

2.4 Simulation Settings

The simulation settings that were used for the streaming operator are given in table 2.1 to ensure reproducibility of the results.

Table 2.1: Simulation settings

Setting	Value
Discretization Scheme	D2Q9
Grid size	15 x 15
Artificially increased density at	(7, 7)
Increased for channels	all
Increased by	1%
Initial probability distribution	Equilibrium
Initial Density	Equilibrium
Initial Velocity	Equilibrium

2.5 Simulation Results

The generated plots based on the streaming operator are provided here:

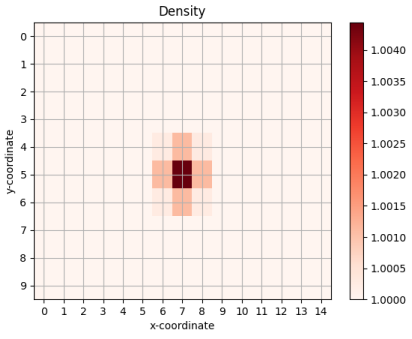


Figure 2.2: Initial density setup

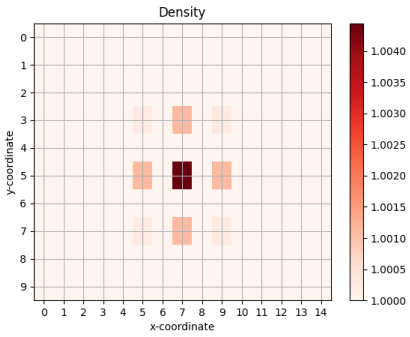


Figure 2.3: Density after 1 streaming step

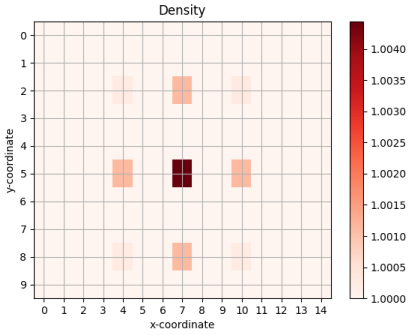


Figure 2.4: Density after 2 streaming steps

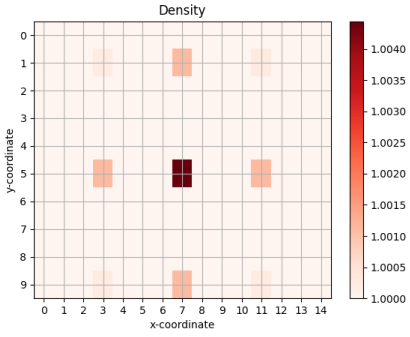


Figure 2.5: Density after 3 streaming steps

3

Collision Operator[1]

3.1 BGKT Approximation

To implement the collision operator in the LBE we use the BGKT approximation to create a simplified model that assumes the distribution function locally relaxes to an equilibrium function [3] [2].

Introducing the BGKT approximation, the BTE becomes

$$\frac{d}{dt} f(\mathbf{r}, \mathbf{v}, t) = -\frac{f(\mathbf{r}, \mathbf{v}, t) - f^{eq}(\mathbf{r}, \mathbf{v}, t)}{\tau} \tag{3.1}$$

$$= -\omega(f(\mathbf{r}, \mathbf{v}, t) - f^{eq}(\mathbf{r}, \mathbf{v}, t)) \tag{3.2}$$

Where $\omega = \frac{1}{\tau}$, with ω being the collision frequency and τ being the relaxation factor [3].

The equilibrium distribution is given as

$$f_i^{eq}(\rho(\mathbf{r}), \mathbf{u}(\mathbf{r})) = w_i \rho(\mathbf{r}) \left[1 + 3\mathbf{c}_i \cdot \mathbf{u}(\mathbf{r}) + \frac{9}{2}(\mathbf{c}_i \cdot \mathbf{u}(\mathbf{r}))^2 - \frac{3}{2}|\mathbf{u}(\mathbf{r})|^2\right] \text{[2]} \tag{3.3}$$

with the weights defined as

$$w_i = \left(\frac{4}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{36}, \frac{1}{36}, \frac{1}{36}, \frac{1}{36}\right) \text{[2]} \tag{3.4}$$

for the D2Q9 discretization.

3.2 Collision Operator in Python

The main implementation for the Collision Operator is implemented on `src/lbm.py` with functions that calculate the equilibrium distribution and

[1] Unless cited otherwise, the theoretical background presented here is based on the lectures of Prof. Andreas Greiner on *High-Performance Computing: Fluid Mechanics with Python* at University of Freiburg as presented at Summer Semester of 2022.[2]

perform the collision and update step for each time step and each point in the grid.

No specific execution is implemented for the collision operator. As a result, this chapter concludes here without any specific implementation discussion, experimental setup or experimental results.

4

Shear Wave Decay[1]

4.1 Shear Wave Decay

The first experiment performed is the Shear Wave Decay with periodic boundary conditions (PBC), commonly used to calculate the kinematic viscosity of fluids.

The following two cases were simulated:

- Initial density equal to 1, $\rho(\mathbf{r}, 0) = 1$, and velocity initialized as $u_x(\mathbf{r}, 0) = \varepsilon \sin\left(\frac{2\pi y}{L_y}\right)$. Where L_y is the length of the domain in the y-direction.
- Density initialized as $\rho(\mathbf{r}, 0) = \rho_0 + \varepsilon \sin\left(\frac{2\pi x}{L_x}\right)$, and initial velocity equal to zero, $\mathbf{u}(\mathbf{r}, 0) = 0$. Where L_x is the lenght of the domain in the x-direction.

4.2 Shear Wave Decay in Python

The algorithm to simulate the cases presented above on the shear wave decay is implemented in `src/shear_wave_decay.py`. Where necessary, listings of code will be presented here. The code is available at `https://github.com/theodorju/fr-hpcpy-pub`.

The algorithm implements the following procedure for both cases and for each iteration step:

1. Perform the streaming procedure: In this step we apply the streaming operator.
2. Calculate the density based on the probability density function.

[1]Unless cited otherwise, the theoretical background presented here is based on the lectures of Prof. Andreas Greiner on *High-Performance Computing: Fluid Mechanics with Python* at University of Freiburg as presented at Summer Semester of 2022.[2]

3. Calculate the velocity based on the probability density function.

4. Calculate the equilibrium distribution using density, velocity, and the collision frequency ω.

5. Apply the collision & relaxation step.

Code listing 4.1 presents the python implementation corresponding to the steps mentioned above.

Listing 4.1: Shear Wave Decay Algorithm

```
# Loop over the simulation steps
for step in trange(steps, desc="Velocity Simulation"):

    # Perform the standard streaming procedure
    lbm.streaming(proba_density)

    # Calculate density
    density = lbm.calculate_density(proba_density)

    # Calculate velocity
    velocity = lbm.calculate_velocity(proba_density)

    # Perform collision and update
    proba_density = \
        lbm.collision_relaxation(proba_density,
                                 velocity,
                                 density,
                                 omega=omega)
```

The script for shear wave decay can be executed as follows[2]

Listing 4.2: Execution of Shear Wave Decay experiment with default arguments

```
>>> python src/shear_wave_decay.py
```

Note: The default implementation executes the shear wave decay experiment with sinusoidal velocity. The script supports multiple command line arguments that are described in detail on the Github repository and can be viewed by executing the command presented on listing 4.3.

Listing 4.3: Help on Shear Wave Decay execution

```
>>> python src/shear_wave_decay.py -h
```

[2]The command presented here assumes that the user is in the cloned directory of the project

4.3 Simulation Settings

The simulation settings that were used for the shear wave decay are given in tables 4.1, 4.2, and 4.3 to ensure reproducibility of the results.

The experiment could easily be executed with different settings provided as command line arguments.

Table 4.1: Simulation settings for sinusoidal velocity

Setting	Value
Discretization Scheme	D2Q9
Grid size	50 x 50
Simulation Steps	2000
Collision frequency	1.0
Epsilon multiplier at velocity	0.05
Velocity snapshot every	100 steps

Table 4.2: Simulation settings for sinusoidal density

Setting	Value
Discretization Scheme	D2Q9
Grid size	50 x 50
Simulation Steps	2000
Collision frequency	0.5
Initial density	0.8

Table 4.3: Simulation settings for varying collision frequency

Setting	Value
Discretization Scheme	D2Q9
Grid size	50 x 50
Simulation Steps	2000
Collision frequencies	(1., 1.2, 1.4, 1.8)
Setting used	Sinusoidal velocity
Epsilon multiplier at velocity	0.05
Snapshot taken every	10 steps

Table 4.4: Simulation settings for theoretical vs experimental viscosity for different collision frequency values

Setting	Value
Discretization Scheme	D2Q9
Grid size	50 x 50
Simulation Steps	2000
Collision frequencies	(0.1 to 2. with step 0.1)
Epsilon multiplier at velocity	0.05
Velocity snapshot every	10 steps

4.4 Simulation Results

4.4.1 Shear Wave Decay with sinusoidal velocity

In case of sinusoidal velocity the evolution of velocity over time as well as the rate of decay of the sinusoidal waves are depicted in Figure 4.1

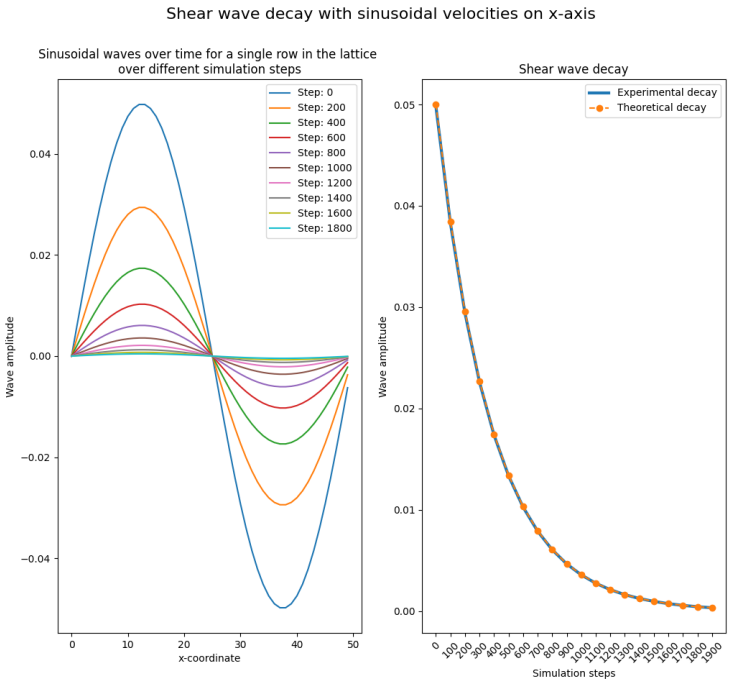


Figure 4.1: Shear Wave Decay initialized with sinusoidal velocity for $\omega = 1$.

Additionally, figure 4.2 depicts the decay of the amplitude of sinusoidal waves for different values of collision frequency ω.

Theoretical viscosity is calculated based on the formula:

$$\nu = \frac{1}{3}(\frac{1}{\omega} - \frac{1}{2}) \tag{4.1}$$

To calculate the experimental viscosity, the exponential decay formula was used:

$$A(t) = A_0 \cdot e^{(-\nu \cdot \frac{2\pi}{L_y} \cdot t)} \tag{4.2}$$

$$\nu = -\frac{ln(\frac{A(t)}{A_0})}{\frac{2\pi}{L_y} \cdot t} \tag{4.3}$$

During the experiments, the amplitude of the sinusoidal wave was measured every 100 time steps, which means that in equation (4.3), $t = 100$.

The experimental as well as the theoretical results for viscosity are presented in Table 4.5.

Table 4.5: Theoretical and Experimental Viscosity

Omega	Theoretical	Experimental
1	0.166666667	0.166666478
1.2	0.111111111	0.111146945
1.4	0.071428571	0.071278936
1.8	0.018518518	0.017900571

4.4.2 Shear Wave Decay with sinusoidal density

In case of sinusoidal density the evolution of density over time is depicted in Figure 4.3

4.4.3 Theoretical vs Experimental viscosity

Figure 4.4 depicts the theoretical viscosity calculated based on equations (4.1) compared to the one calculated using formula (4.3).

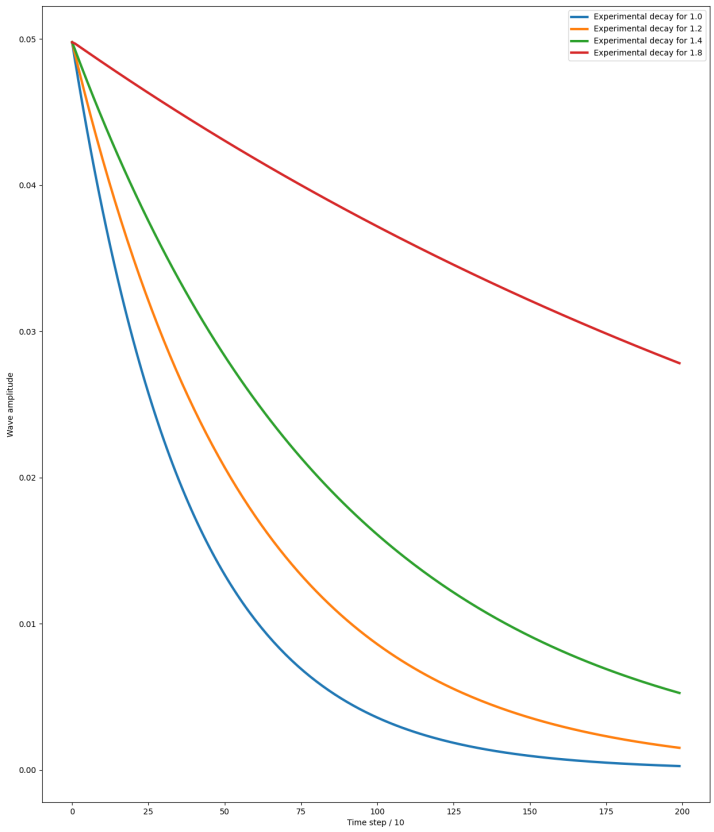


Figure 4.2: Shear Wave Decay for different values of ω.

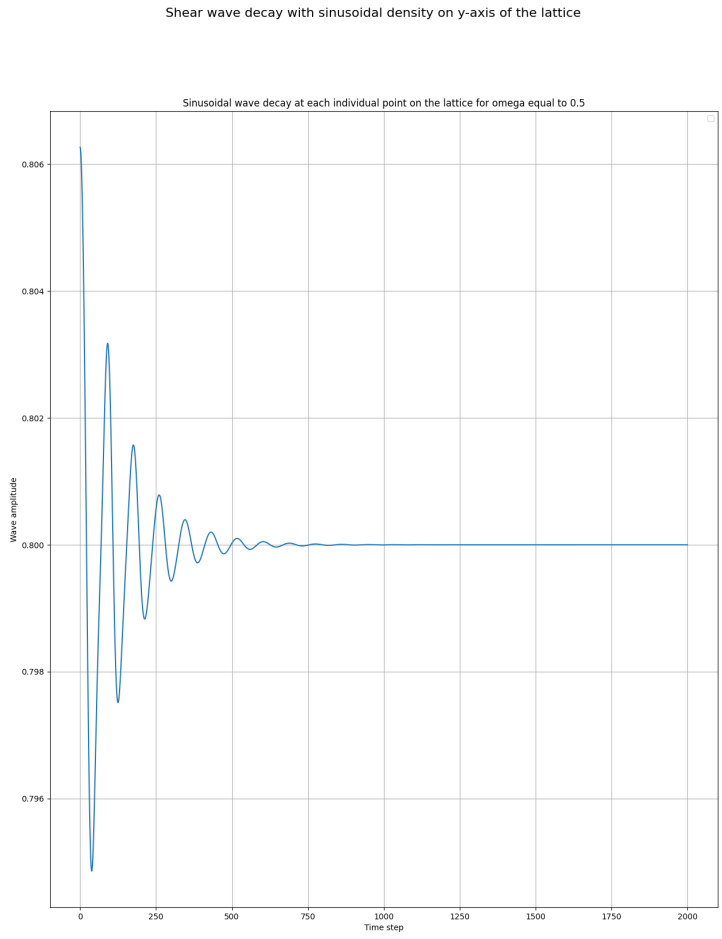


Figure 4.3: Shear Wave Decay initialized with sinusoidal density for $\omega = 0.5$.

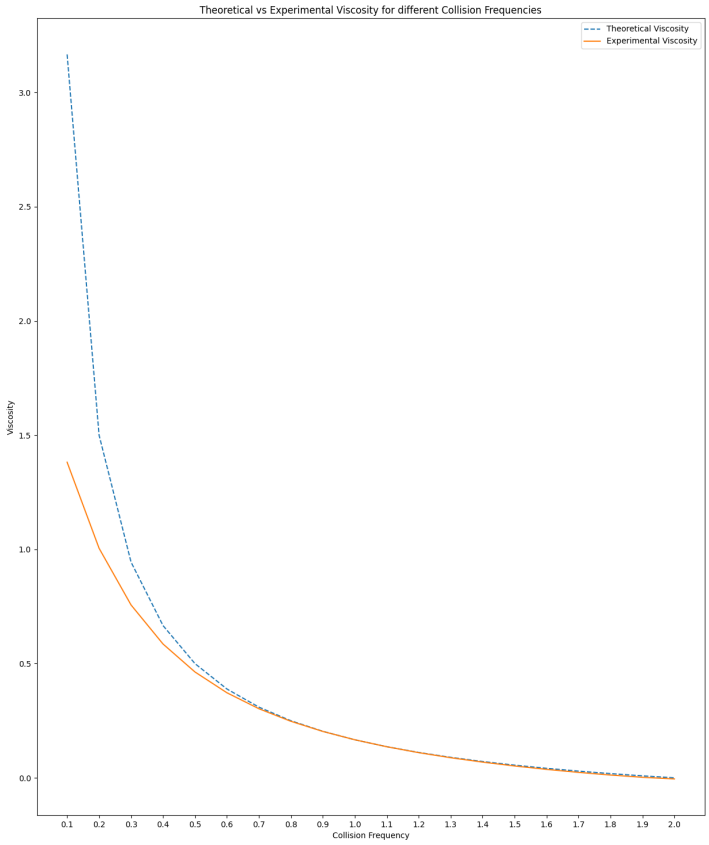


Figure 4.4: Theoretical viscosity compared with experimental viscosity for different collision frequencies.

5

Couette Flow[1]

The next experiment performed is the Couette Flow. In the Couette Flow, an incompressible fluid flows between two parallel plates, the top plate is moving perpendicular to the y-axis while the bottom plate is held still. No slip is present between the plates and the fluid, and no external force is applied in the system. This means that the velocity of the fluid near the wall is equal to the velocity of the wall[5].

This setting is depicted in Figure 5.1 where the top boundary will start moving to the right, perpendicular to the y-axis, and the bottom boundary will remain fixed. Periodic boundary conditions (PBC) are applied in the left and right boundaries.

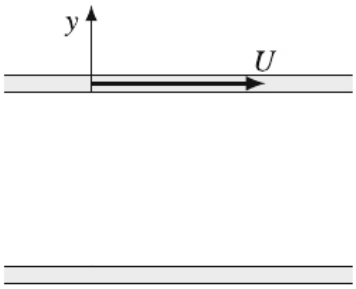


Figure 5.1: Couette flow initial setup[5].

The steady state of a Couette flow with upper moving boundary is depicted in Figure 5.2.

This chapter will also briefly discuss the *Dry nodes* approach for boundary conditions, the specific boundary condition equations for *Fixed Rigid*

[1]Unless cited otherwise, the theoretical background presented here is based on the lectures of Prof. Andreas Greiner on *High-Performance Computing: Fluid Mechanics with Python* at University of Freiburg as presented at Summer Semester of 2022.[2]

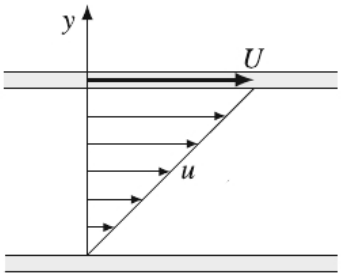


Figure 5.2: Couette flow steady state[5].

Wall and *Moving Rigid Wall*, the Python implementation of the Coutte flow, and finally, the experimental results of the implementation.

5.1 Dry Nodes

For the Couette flow boundary conditions we will use the *dry nodes* approach. This means that if the distance between nodes inside our domain is Δx then the boundary is located at distance $\frac{\Delta x}{2}$ from the nodes at the boundary, as depicted in Figure 5.3.

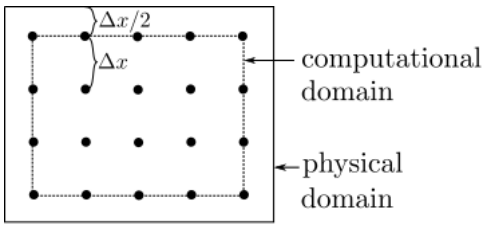


Figure 5.3: Representation of the dry-nodes boundary condition[2]

If we assume another pair of nodes that are outside of the computational domain, then the boundary is placed midway between those "ghost" nodes and our actual lattice nodes. This setting of the boundary makes the method *second-order accurate*[5].

5.2 Rigid Wall Boundary Conditions

With the placement of the boundary in distance $\frac{\Delta x}{2}$ from a node at the boundary, means that populations that are leaving the boundary node x_b at time t will reach the boundary after time $t + \frac{\Delta t}{2}$, will be reflected back with a velocity $c_{\bar{i}} = -c_i$ and will return at node x_b at time $t + \Delta t$ [5].

Note that $c_{\bar{i}}$ is the velocity of the "anti" channel of c_i. The relations between channel and "anti" channel are depicted in table 5.1

Table 5.1: Channels and their corresponding "anti" channels used in Rigid Wall boundary conditions

Channel	Anti Channel
0	0
1	3
2	4
3	1
4	2
5	7
6	8
7	5
8	6

This means that for the populations at the boundary the standard streaming step is replaced by

$$f_{\bar{i}}(\mathbf{x}_b, t + \Delta t) = f_i^\star(\mathbf{x}_b, t) \tag{5.1}$$

Where $f_i^\star$ is the probability density function before the streaming step is applied.

In our experiments, since the fixed rigid wall is at the bottom, the channels going out are 7, 4, and 8, and their corresponding channels going in are 5, 2, and 6 respectively. Applying equation (5.1) on those channels yield the following update equations:

$$f_{\bar{6}}(\mathbf{x}_b, t + \Delta t) = f_8^\star(\mathbf{x}_b, t) \tag{5.2}$$
$$f_{\bar{2}}(\mathbf{x}_b, t + \Delta t) = f_4^\star(\mathbf{x}_b, t) \tag{5.3}$$
$$f_{\bar{5}}(\mathbf{x}_b, t + \Delta t) = f_7^\star(\mathbf{x}_b, t) \tag{5.4}$$

5.3 Moving Wall Boundary Conditions

The case of the moving wall can be seen as an extension of the fixed rigid wall. For the moving wall, a small correction is needed in equation (5.1) to account for the momentum that will be either gained or lost by the bounced-back particles after hitting the wall. This correction is necessary so that the outcome respects Galilean invariance[5].

Assuming wall velocity u_w the updated equation is:

$$f_{\bar{i}}(\mathbf{x}_b, t + \Delta t) = f_i^\star(\mathbf{x}_b, t) - 2w_i\rho_w \frac{\mathbf{c}_i \cdot \mathbf{u}_w}{c_s^2} \tag{5.5}$$

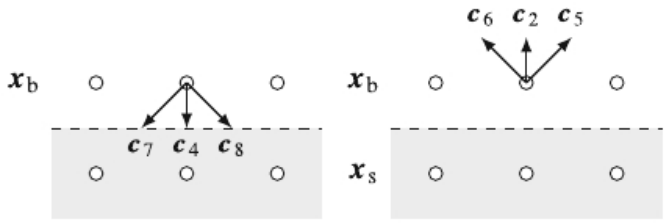


Figure 5.4: Bounce back populations at the bottom wall[5]

Where $f_i^\star$ is again the probability density function before the streaming step is applied, and c_s is the speed of sound, the square of which equals $c_s^2 = 1/3$ in lattice units.

In our experiments, since the moving wall is at the top moving perpendicular to the y-axis, the channels going out are 5, 2, and 6, and their corresponding channels going in are 7, 4, and 8 respectively. Applying equation (5.5) on those channels yield the following update equations:

$$f_{\bar{7}}(\mathbf{x}_b, t + \Delta t) = f_5^\star(\mathbf{x}_b, t) - 2w_5\rho_w \frac{\mathbf{c}_5 \cdot \mathbf{u}_w}{c_s^2} \tag{5.6}$$

$$f_{\bar{4}}(\mathbf{x}_b, t + \Delta t) = f_2^\star(\mathbf{x}_b, t) - 2w_2\rho_w \frac{\mathbf{c}_2 \cdot \mathbf{u}_w}{c_s^2} \tag{5.7}$$

$$f_{\bar{8}}(\mathbf{x}_b, t + \Delta t) = f_6^\star(\mathbf{x}_b, t) - 2w_6\rho_w \frac{\mathbf{c}_6 \cdot \mathbf{u}_w}{c_s^2} \tag{5.8}$$

5.4 Coutte Flow Implementation in Python

We will now discuss the algorithmic implementation of Coutte Flow. The experiment is implemented in file `/src/couette_flow.py`. Code snippets will be provided where necessary[2]

The algorithm that implements the Coutte flow closely follows the following steps:

1. Moment update or initial conditions: In this step density and velocity are calculated either from the initial conditions of from the probability density function.

2. Equilibrium distribution calculation: In this step we calculate the equilibrium distribution based on the density, velocity, and collision frequency.

[2]The full code is available online at `https://github.com/theodorju/fr-hpcpy-pub`.

3. Perform the collision and relaxation step.

4. Copy necessary pre-streaming probability distribution values: In this step we keep a copy of the necessary probability distribution values before the streaming step since their values will be needed for the boundary conditions, as discussed earlier.

5. Perform the streaming step.

6. Apply the fixed rigid wall boundary conditions on the lower boundary.

7. Apply the moving rigid wall boundary conditions on the upper boundary.

The Python implementation of those steps is presented in listing 5.1. This process is performed for each simulation step.

Listing 5.1: Coutte Flow Algorithm

```
# Calculate density
density = lbm.calculate_density(proba_density)

# Calculate velocity
velocity = lbm.calculate_velocity(proba_density)

# Perform collision/relaxation
proba_density = \
    lbm.collision_relaxation(proba_density,
                             velocity,
                             density,
                             omega=omega)

# Keep the probability density function pre-streaming
pre_stream_proba_density = proba_density.copy()

# Streaming
lbm.streaming(proba_density)

# Apply boundary conditions on the bottom rigid wall
lbm.rigid_wall(proba_density,
               pre_stream_proba_density,
               "lower")

# Apply boundary condition on the top moving wall
lbm.moving_wall(proba_density,
                pre_stream_proba_density,
                wall_velocity,
                density, "upper")
```

The fixed and moving wall boundary conditions are implemented by using the values of the probability density function before the streaming.

In each case (lower or upper boundary), we iterate over the channels going in and directly overwrite their probability distribution value with the one of their corresponding "anti" channel. The process for the moving wall is presented in listing 5.2. Wherever possible, temporary variables are used to enhance readability.

Note that the boundary conditions are implemented in `/src/lbm.py`

Listing 5.2: Moving Wall Boundary Conditions

```
for i in range(len(in_channels)):

    # Set temporary variables for convenience
    temp_in, temp_out = in_channels[i], out_channels[i]

    # Calculate term due to velocity based on the channels
    # going out
    temp_term = \
        (-2 * weights[temp_out] * avg_density / c_s_squared) * \
        np.dot(velocity_channels[temp_out], wall_velocity)

    # Index of y's that are on the upper boundary is equal to the
    # size of the lattice - 1, for simplicity use "-1" to access
    proba_density[temp_in, :, -1] = \
        pre_streaming_proba_density[temp_out, :, -1] + temp_term
```

The fixed rigid wall boundary condition is implemented in a similar manner, without the momentum correction term.

The Coutte Flow experiment can be executed as presented in code listing 5.3[3]

Listing 5.3: Execution of Shear Wave Decay experiment with default arguments

```
>>> python src/couette_flow.py
```

The script supports multiple command line arguments that are described in detail on the Github repository and can be viewed by executing the command presented on listing 5.4.

Listing 5.4: Help on Shear Wave Decay execution

```
>>> python src/couette_flow.py -h
```

5.5 Simulation Settings

The simulation settings that were used for the simulation are given in table 5.2 to ensure reproducibility of the results.

[3]The command presented here assumes that the user is in the cloned directory of the project

Table 5.2: Couette flow Simulation settings

Setting	Value
Discretization Scheme	D2Q9
Grid size	100 x 100
Wall velocity	0.1
Simulation Steps	10000
Collision frequency (*omega*)	0.8
Initial probability distribution	Equilibrium
Initial Density	1.0
Initial Velocity	0.0
Results gathered every	500 steps
Speed of sound squared	1/3

5.6 Simulation Results

The results of the simulations are presented in Figures 5.5 and 5.6.

Figure 5.5 depicts the velocity field of the liquid in the steady state. The moving boundary is depicted in red on top, with an arrow at the direction of the movement. The fixed rigid boundary is depicted in black at the bottom. As already discussed, the boundaries are placed $\frac{\Delta x}{2}$ away from the boundary nodes.

Figure 5.6 depicts the evolution of the velocity at a particular slice of the lattice perpendicular to the x-axis for various simulation steps. The height of each lattice is presented on the x-axis of the figure, and the velocity at a specific step is presented on the y-axis. One can observe that for the points at the fixed wall at height 0 on the horizontal axis of the figure, the velocity remains equal to zero across all the simulation steps. We can additionally observe that the velocity of the points in the moving boundary, 100 in the horizontal axis of the figure, is equal to the velocity of the boundary.

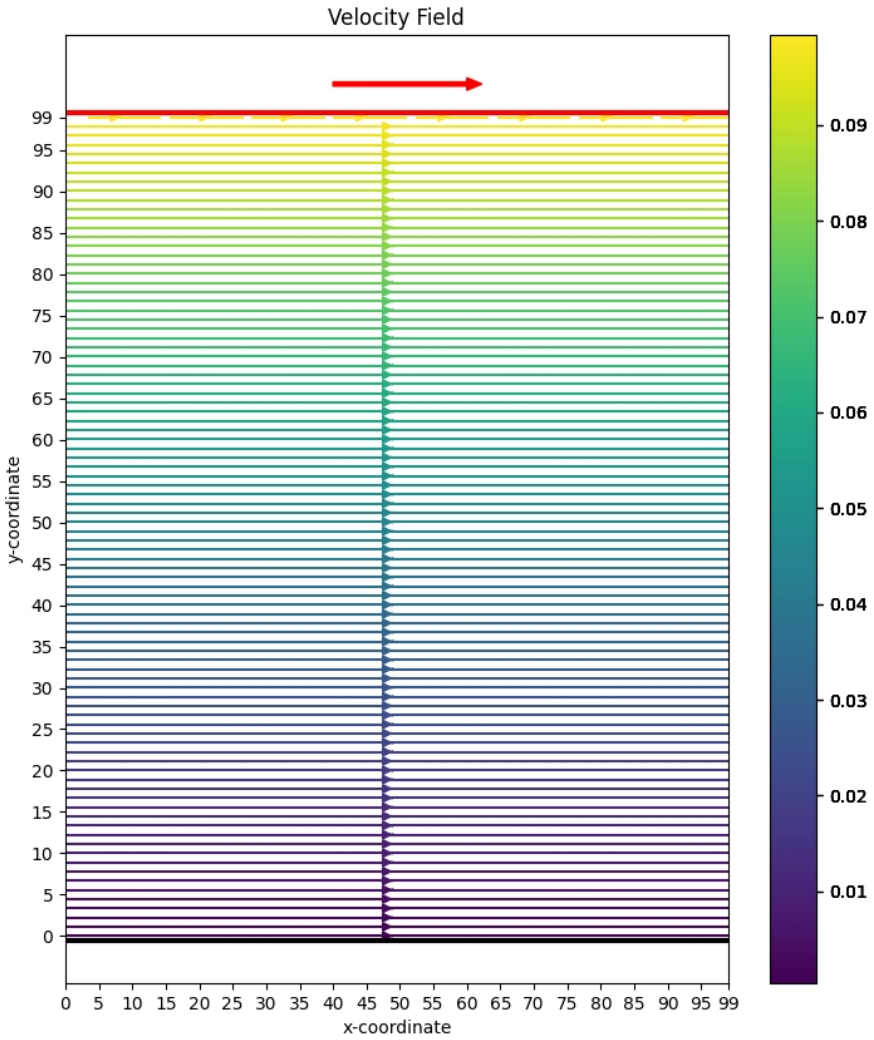


Figure 5.5: Couette Flow: Steady State Velocity profile. Higher brightness means higher velocity value.

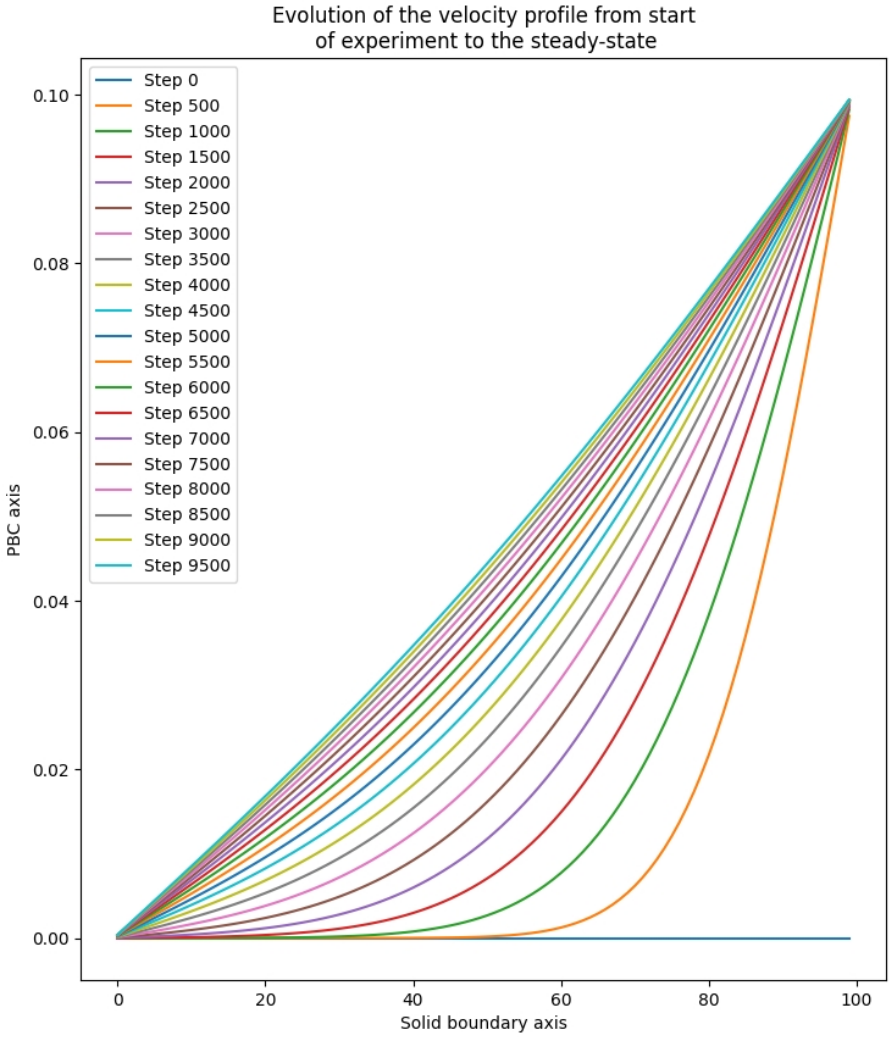


Figure 5.6: Couette Flow: Evolution of velocity

6

Poiseuille Flow[1]

The next experiment performed is the Poiseuille Flow. In the Poiseuille Flow, an incompressible fluid is flowing between two parallel plates as shown in Figure 6.1.

The flow can be driven either by a constant pressure gradient between inlet and outlet, or by an external force.[5]. In this experiment we assume that the flow is driven by a constant pressure gradient between inlet and outlet.

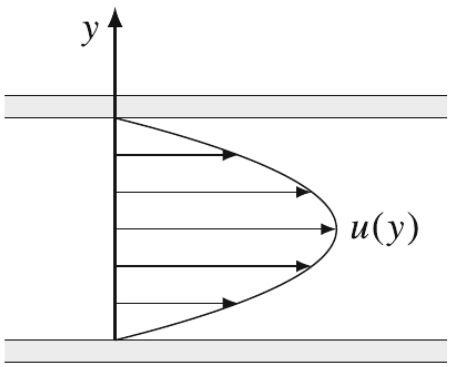


Figure 6.1: Poiseuille flow [5]

As we can observe from Figure 6.1 the Rigid Wall Boundary Conditions will be applied in the top and bottom boundaries. Due to the pressure difference between inlet and outlet, a special case of Periodic Boundary Conditions will be applied there, namely Periodic Boundary Conditions with pressure gradient.

[1]Unless cited otherwise, the theoretical background presented here is based on the lectures of Prof. Andreas Greiner on *High-Performance Computing: Fluid Mechanics with Python* at University of Freiburg as presented at Summer Semester of 2022.[2]

6.1 Period Boundary Conditions with pressure gradient

Assuming a pressure difference between the inlet pressure p_{in} and outlet pressure p_{out}, we can express the boundary conditions as follows:

$$p(\mathbf{x}, t) = p(\mathbf{x} + L, t) + \Delta p, \tag{6.1}$$
$$\mathbf{u}(\mathbf{x}, t) = \mathbf{u}(\mathbf{x} + L, t). \tag{6.2}$$

and keeping in mind that for LBM the pressure is directly related to the density through the ideal gas equation of state $p = c_s^2 \rho$ with $c_s^2 = 1/3$, the boundary conditions can also be expressed as:

$$\rho(\mathbf{x}, t) = \rho(\mathbf{x} + L, t) + \Delta \rho, \tag{6.3}$$
$$\mathbf{u}(\mathbf{x}, t) = \mathbf{u}(\mathbf{x} + L, t). \tag{6.4}$$

In order to simulate those boundary conditions we are going to need an artificial extra layer of nodes outside of the physical domain as shown in Figure 6.2.

Now due to periodicity we can easily see that what takes place in nodes x_0 -the last nodes of the previous domain- should be identical to what takes place in nodes x_N -the last nodes in our domain. With the same reasoning we can also deduce that the situation in nodes x_1 and x_{N+1} should also be similar.

In order to formulate the boundary conditions we are going to split the probability density function into an equilibrium part and a non-equilibrium part. The non-equilibrium part is given by $f_i^{neq} = f_i^\star - f_i^{eq}$. Keeping in mind that there are now different velocities in the inlet and outlet nodes the boundary conditions are given by Equations (6.5) and (6.6).

$$f_i^\star(x_0, y, t) = f_i^{eq}(\rho_{in}, \mathbf{u}_N) + (f_i^\star(x_N, y, t) - f_i^{eq}(x_N, y, t)) \tag{6.5}$$
$$f_i^\star(x_{N+1}, y, t) = f_i^{eq}(\rho_{out}, \mathbf{u}_1) + (f_i^\star(x_1, y, t) - f_i^{eq}(x_1, y, t)) \tag{6.6}$$

6.2 Poiseuille Flow Implementation in Python

We will now discuss the algorithmic implementation of the Poiseuille Flow. The experiment is implemented in `/src/poiseuille_flow.py`. Code snippets will be provided where necessary.[2]

[2]The full code is available online at `https://github.com/theodorju/fr-hpcpy-pub`.

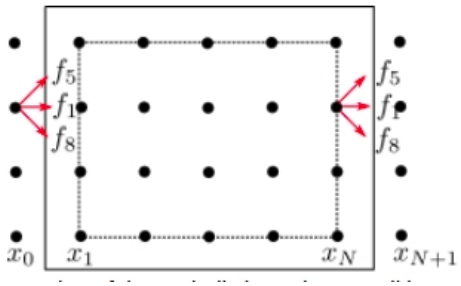


Figure 6.2: Periodic Boundary Conditions with Pressure Gradient[2]

The algorithm that implements Poiseuille flow closely follows the following steps:

1. Moment update or initial conditions: In this step, density and velocity are calculate either from the initial conditions or from the probability density function.

2. Perform the collision and relaxation step.

3. Copy the necessary pre-streaming probability distribution values: In this step we keep a copy of the necessary probability distribution values before the streaming step since their values are neeeded for the boundary conditions.

4. Apply periodic boundary conditions with pressure gradient. Note that the boundary conditions in this case, must be applied before the streaming step[5].

5. Perform the streaming step.

6. Apply fixed rigid wall boundary conditions on the upper and lower boundaries.

These steps are performed in each simulation step. The python implementation of those steps is presented in listing 6.1.

Listing 6.1: Poiseuille Flow main algorithm

```
# Calculate density
density = lbm.calculate_density(proba_density)

# Calculate velocity
velocity = lbm.calculate_velocity(proba_density)

# Perform collision/relaxation
proba_density = \
```

```
    lbm.collision_relaxation(proba_density,
                             velocity,
                             density,
                             omega=omega)

# Keep the probability density function pre-streaming
pre_streaming_proba_density = proba_density.copy()

# Apply periodic boundary conditions with pressure gradient
lbm.pressure_gradient(proba_density,
                      pre_streaming_proba_density,
                      density,
                      velocity,
                      density_input,
                      density_output,
                      flow="left_to_right")
# Streaming
lbm.streaming(proba_density)

# Apply boundary conditions on the bottom wall
lbm.rigid_wall(proba_density,
pre_streaming_proba_density, "lower")

# Apply boundary conditions on the top wall
lbm.rigid_wall(proba_density,
pre_streaming_proba_density, "upper")
```

The algorithm that handles periodic boundary conditions with pressure gradient is implemented in `src/lbm.py` and in high level it is as follows:

1. Calculate the equilibrium distribution for all datapoints and channels. In equations (6.5) and (6.6) this calculates the term f_i^{eq}

2. Calculate the equilibrium distribution using the input density and output velocity. In equations (6.5) and (6.6) this calculates the term $f_i^{eq}(\rho_{in}, \mathbf{u}_N)$.

3. Calculate the equilibrium distribution using the output density and input velocity. In equations (6.5) and (6.6) this calculates the term $f_i^{eq}(\rho_{out}, \mathbf{u}_1)$.

4. Calculate the probability density at the inlet, using the necessary values before the streaming. In equations (6.5) and (6.6) this calculates the term $f_i^{\star}(x_0, y, t)$.

5. Calculate the probability density at the outle, using the necessary values before the streaming. In equations (6.5) and (6.6) this calculates the term $f_i^{\star}(x_{N+1}, y, t)$.

These steps are performed in each simulation step. The python implementation of those steps is presented in listing 6.2.

Listing 6.2: Periodic Boundary Conditions with pressure gradient

```
proba_equilibrium = \
    calculate_equilibrium_distro(density, velocity)

equil_din_vout = \
    calculate_equilibrium_distro(density_input, velocity[:, -2, :])


equil_dout_vin = \
    calculate_equilibrium_distro(density_output, velocity[:, 1, :])

proba_density[:, 0, :] = \
    equil_din_vout[:, :] + (proba_density[:, -2, :]
                            - proba_equilibrium[:, -2, :])

proba_density[:, -1, :] = \
    equil_dout_vin[:, :] + (proba_density[:, 1, :]
                            - proba_equilibrium[:, 1, :])
```

6.3 Theoretical Velocity Profile on Poiseuille flow

Under the assumption of laminar flow, the Navier-Stokes equations can be simplified as

$$\frac{\partial p(x)}{\partial x} = \frac{1}{\mu}\frac{\partial^2 u_x(y)}{\partial y^2}. \tag{6.7}$$

Integating twice along the wall direction and applying the boundary conditions $u(0) = 0$ and $u(h) = 0$ we get the velocity profile:

$$u(y) = -\frac{1}{2\mu}\frac{dp}{dx}y(h - y) \tag{6.8}$$

where h is the diameter of the pipe, $\mu = \rho\nu$ is the dynamic viscosity[2].

This equation is used to calculate the theoretical velocity profile in the steady state on the experiments. The implementation can be found under `/src/utils.py`.

For the calculation of dynamic viscosity the average density was used, and since this is set equal to 1, we get $\mu = \nu$. Additionally, dp was calculated as the difference of the outlet minus the inlet densities multiplied by the speed of sound squared, based on relation of density and pressure thought the ideal gas equation of state $p = c_s^2\rho$:

$$dp = (\rho_{outlet} - \rho_{inlet})c_s^2 \tag{6.9}$$

Finally, dx is calculated as $x_N - x_0$ which is equal to the size of the lattice in the x dimension.

6.4 Simulation Settings

The simulation settings that were used for the simulation are given in table 6.1 to ensure reproducibility of the results.

Table 6.1: Poiseuille flow Simulation settings

Setting	Value
Discretization Scheme	D2Q9
Grid size	50 x 50
Simulation Steps	10000
Collision frequency (*omega*)	0.8
Initial probability distribution	Equilibrium
Initial Density	1.0
Initial Velocity	0.0
Inlet Density	1.0 + 1%
Outlet Density	1.0 - 1%
Results gathered every	1000 steps
Speed of sound squared	1/3

6.5 Simulation Results

The results of the simulations are presented in Figures 6.3, 6.4, 6.5, 6.6, 6.7, and 6.8.

Figure 6.3 depicts the evolution of the velocity profile from the beginning of the simulation until the stedy state for every 1000 simulation steps.

Figure 6.4 depicts a streamplot of the velocity profile in the steady state. The brighter the lines, the higher the velocity values in that area.

Figure 6.5 compares the theoretical with the experimental velocity profiles in the steady state. The experimental solution deviates the most from the theoretical one in the middle points of the pipe, on the area of maximum velocity. The deviation seemed to decrease with the increase of the diameter of the pipe, as it can be seen on Figure 6.6, and increase with the decrease of the diameter of the pipe, as it can be seen on Figure 6.7.

Finally, Figure 6.8 gives a comparison of the velocity profiles calculated at the inlet, middle and outlet of the pipe.

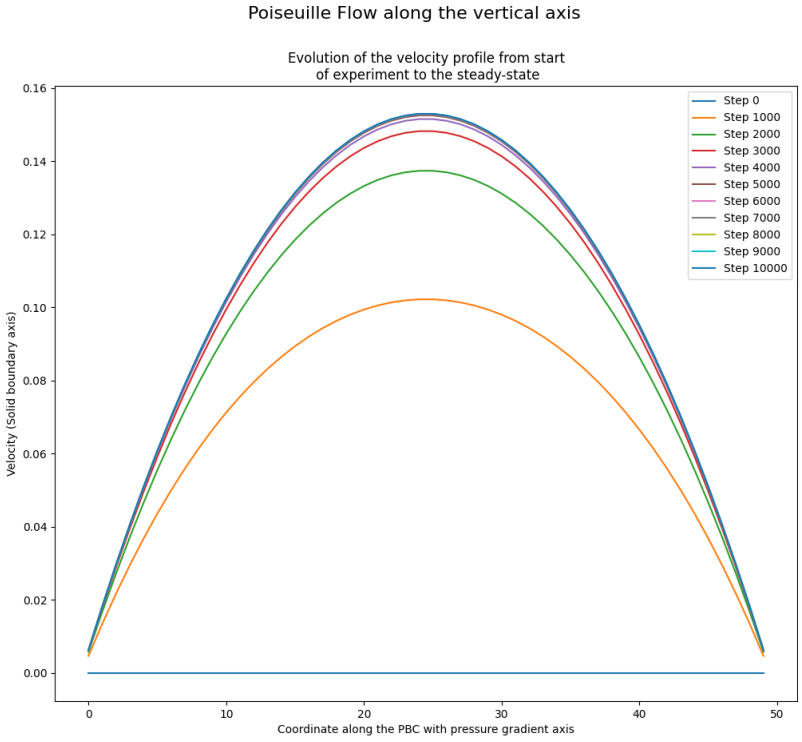


Figure 6.3: Evolution of velocity on Poiseuille flow from the beginning of the experiment to the steady state.

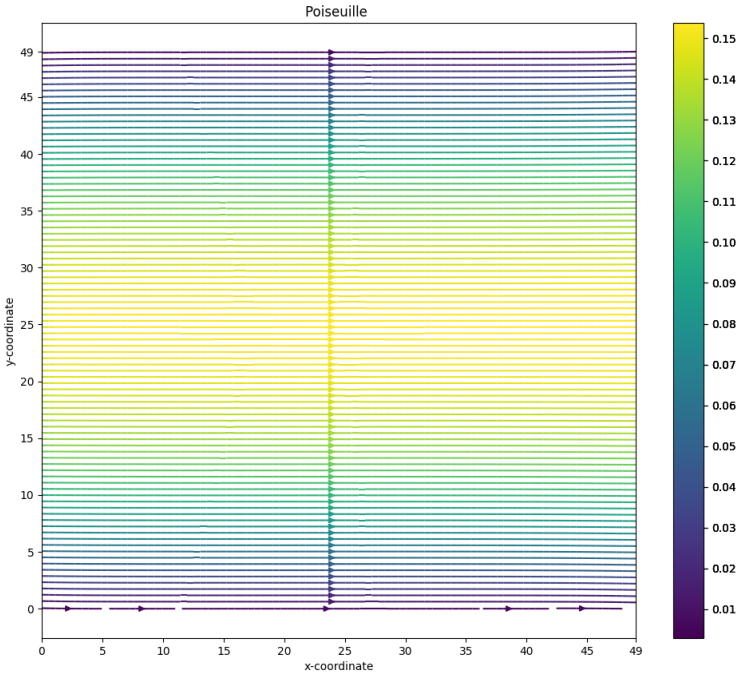


Figure 6.4: Streamplot of velocity profile in the steady state of Poiseuille flow. High brightness means higher velocity.

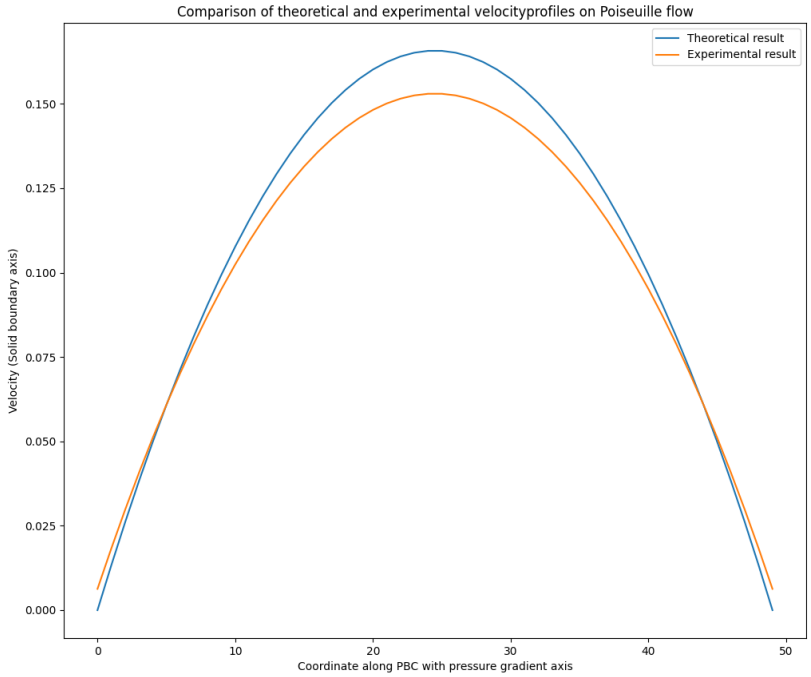


Figure 6.5: Comparison of experimental velocity profile against theoretical velocity profile for the 50 x 50 grid.

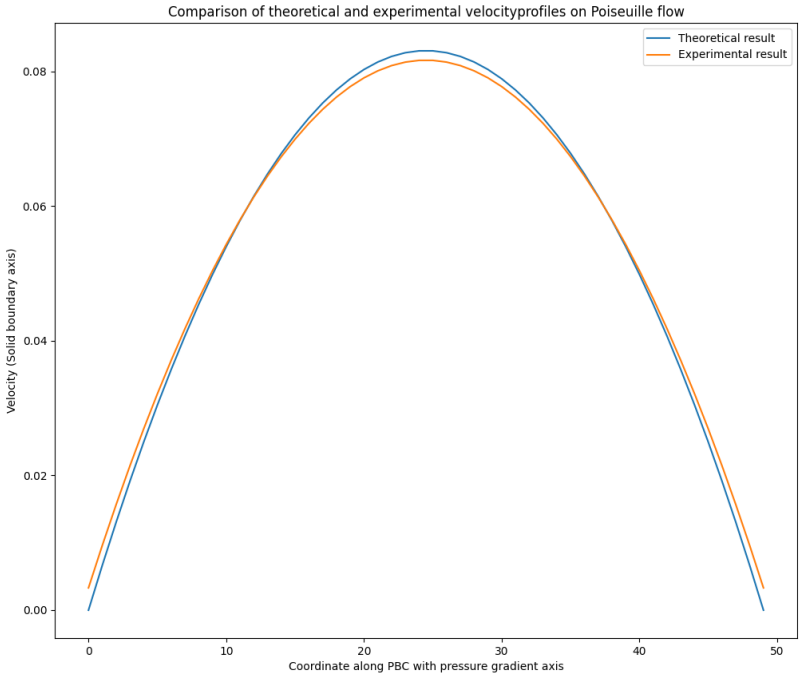


Figure 6.6: Comparison of experimental velocity profile against theoretical velocity profile for the 100 x 50 grid.

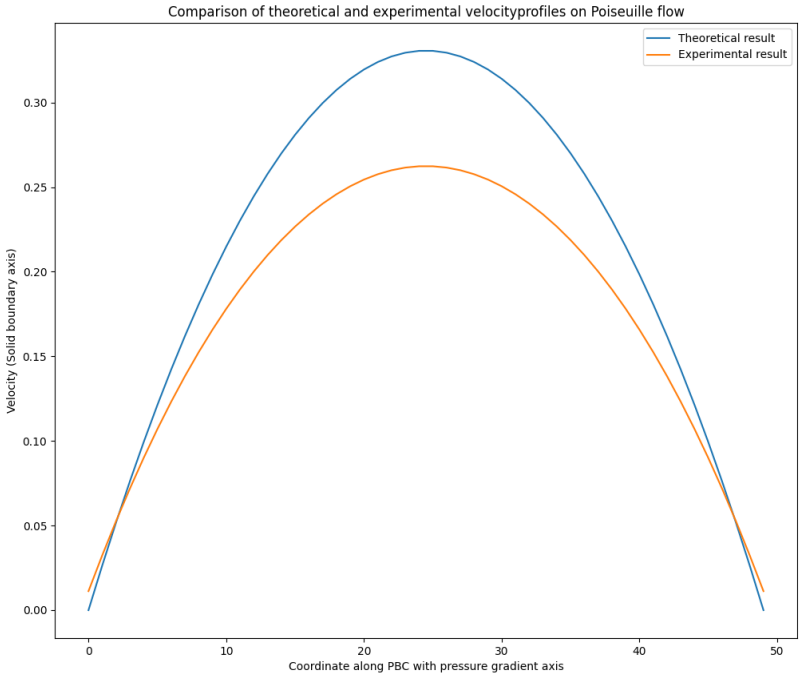


Figure 6.7: Comparison of experimental velocity profile against theoretical velocity profile for the 25 x 50 grid.

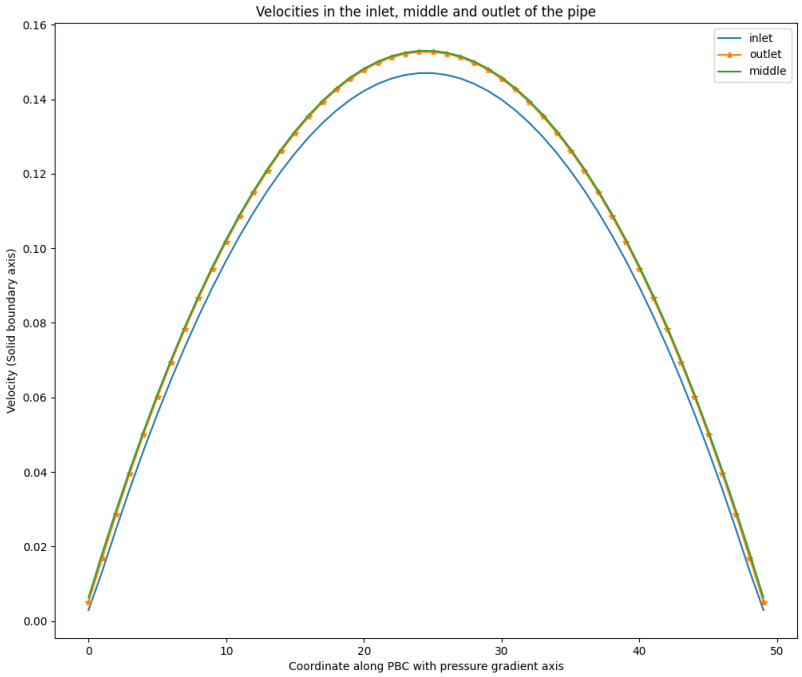


Figure 6.8: Velocity profile in the middle compared with the profiles at inlet and outlet.

7

Sliding Lid[1]

The next experiment performed is the Sliding Lid. In the sliding lid experiment the lid of a closed box moves with a constant velocity to the right and as a result, creates a flow in the liquid itself.

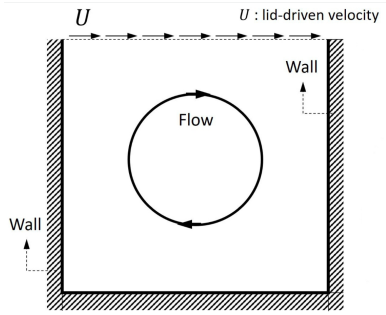


Figure 7.1: Setup and velocity profile of the sliding lid experiment.

The Reynolds number Re is used to characterize the phenomenon and is calculated as

$$\mathrm{Re} = \frac{Lu}{\nu} \tag{7.1}$$

where L is the length of the box, u is the velocity of the moving lid and ν is the kinematic viscosity. This experiment was performed for a Reynolds number close to 1000.

[1]Unless cited otherwise, the theoretical background presented here is based on the lectures of Prof. Andreas Greiner on *High-Performance Computing: Fluid Mechanics with Python* at University of Freiburg as presented at Summer Semester of 2022.[2]

7.1 Boundary conditions

Since the domain contains rigid boundaries on the left, right, and bottom, bounce back rigid wall boundary conditions are applied there. The equations describing these conditions are similar to the ones described in the the Couette Flow experiment.

Based on the channel-antichannel table 5.1 the conditions applied in the bottom wall are already discussed and are given by equations (5.2), (5.3), and (5.4).

Similar equations describe the left and right boundaries as well, more specifically for the left rigid boundary:

$$f_{\bar{1}}(\mathbf{x}_b, t + \Delta t) = f_3^\star(\mathbf{x}_b, t) \tag{7.2}$$
$$f_{\bar{5}}(\mathbf{x}_b, t + \Delta t) = f_7^\star(\mathbf{x}_b, t) \tag{7.3}$$
$$f_{\bar{8}}(\mathbf{x}_b, t + \Delta t) = f_6^\star(\mathbf{x}_b, t) \tag{7.4}$$

and for the right rigid boundary:

$$f_{\bar{3}}(\mathbf{x}_b, t + \Delta t) = f_1^\star(\mathbf{x}_b, t) \tag{7.5}$$
$$f_{\bar{7}}(\mathbf{x}_b, t + \Delta t) = f_5^\star(\mathbf{x}_b, t) \tag{7.6}$$
$$f_{\bar{6}}(\mathbf{x}_b, t + \Delta t) = f_8^\star(\mathbf{x}_b, t) \tag{7.7}$$

Finally, since the top boundary is moving to the right, moving wall boundary conditions are applied there, similar to the ones described in the Couette Flow.

7.2 Sliding Lid Implementation in Python

The experiment is implemented in file `/src/sliding_lid.py`. Code snippets will be provided where necessary[2]

The algorithm that implements the sliding lid flow closely follows the following steps:

1. Moment update or initial conditions: In this step density and velocity are calculated either from the initial conditions of from the probability density function.

2. Equilibrium distribution calculation: In this step we calculate the equilibrium distribution based on the density, velocity, and collision frequency.

3. Perform the collision and relaxation step

[2]The full code is available online at `https://github.com/theodorju/fr-hpcpy-pub`.

4. Copy necessary pre-streaming probability distribution values: In this step we keep a copy of the necessary probability distribution values before the streaming step since their values will be needed for the boundary conditions, as discussed earlier.

5. Perform the streaming step.

6. Apply the fixed rigid wall boundary conditions on the lower, left, and right boundaries.

7. Apply the moving rigid wall boundary conditions on the upper boundary.

The Python implementation of those steps is presented in listing 7.1. This process is performed for each simulation step.

Listing 7.1: Sliding Lid Algorithm

```
# Calculate density
density = lbm.calculate_density(proba_density)
# Calculate velocity
velocity = lbm.calculate_velocity(proba_density)
# Perform collision/relaxation
lbm.collision_relaxation(proba_density,
                         velocity,
                         density,
                         omega=omega)
# Keep the probability density function pre-streaming
pre_stream_proba_density = proba_density.copy()
# Streaming
lbm.streaming(proba_density)
# Apply boundary conditions on the bottom rigid wall
lbm.rigid_wall(proba_density,
               pre_stream_proba_density,
               "lower")
# Apply boundary condition on the top moving wall
lbm.moving_wall(proba_density,
                pre_stream_proba_density,
                lid_velocity,
                density, "upper")
# Apply boundary conditions on the left rigid wall
lbm.rigid_wall(proba_density,
               pre_stream_proba_density,
               "left")
# Apply boundary conditions on the right rigid wall
lbm.rigid_wall(proba_density,
               pre_stream_proba_density,
               "right")
```

7.3 Simulation Settings

The simulation settings that were used for the simulation are given in table 7.1 to ensure reproducibility of the results.

Additionally, the following variants where also executed for 10000 steps:

1. Collision frequency of 0.8 on a grid size of 300 x 300 and lid velocity 0.1 (Reynolds 120)
2. Collision frequency of 0.8 on a grid size of 100 x 100 and lid velocity 0.1 (Reynolds 40)
3. Collision frequency of 1.7 on a grid size of 100 x 100 and lid velocity 0.1 (Reynolds 340)
4. Collision frequency of 1.0 on a grid size of 300 x 300 and lid velocity 0.5 (Reynolds 900)

For the extra variants, all other settings were fixed to the values mentioned in table 7.1.

Table 7.1: Sliding lid Simulation settings

Setting	Value
Discretization Scheme	D2Q9
Grid size	300 x 300
Wall velocity	0.1
Simulation Steps	10000
Collision frequency (*omega*)	1.7
Initial probability distribution	Equilibrium
Initial Density	1.0
Initial Velocity	0.0
Results gathered every	500 steps
Reynolds number	1020
Speed of sound squared	1/3

7.4 Simulation Results

The results of the simulations are presented in Figures 7.2, 7.3, 7.4, 7.5, and 7.6.

In addition to the figures presented here, GIFs were generated for each case and are available in the public Github repository accompanying this report, at `https://github.com/theodorju/fr-hpcpy-pub/tree/main/src/data/gifs`

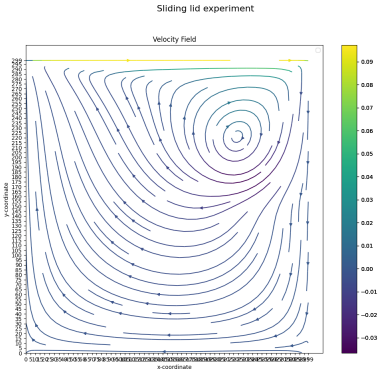


Figure 7.2: Sliding lid after 10000 iteration steps for collision frequency 1.7 on a 300x300 grid with lid velocity 0.1 (Reynolds 1020). Colorbar denotes the magnitude of the velocity field.

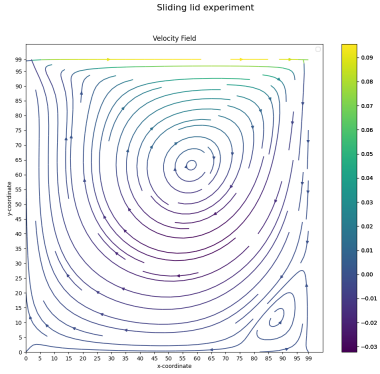


Figure 7.3: Sliding lid after 10000 iteration steps for collision frequency 1.7 on a 100x100 grid with lid velocity 0.1 (Reynolds 340). Colorbar denotes the magnitude of the velocity field.

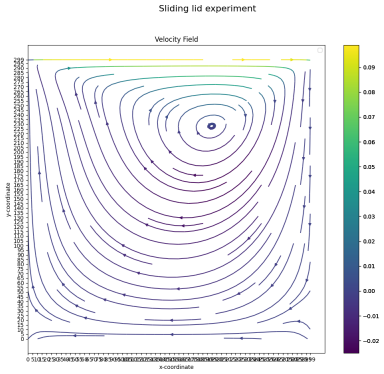


Figure 7.4: Sliding lid after 10000 iteration steps for collision frequency 0.8 on a 300x300 grid with lid velocity 0.1 (Reynolds 120). Colorbar denotes the magnitude of the velocity field.

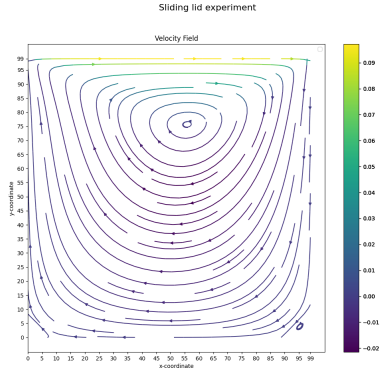


Figure 7.5: Sliding lid after 10000 iteration steps for collision frequency 0.8 on a 100x100 grid with lid velocity 0.1 (Reynolds 40). Colorbar denotes the magnitude of the velocity field.

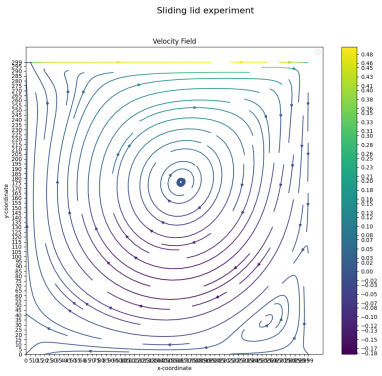


Figure 7.6: Sliding lid after 10000 iteration steps for collision frequency 1.0 on a 300x300 grid with lid velocity 0.1 (Reynolds 900). Colorbar denotes the magnitude of the velocity field.

8

Parallelization using the Message Passing Interface[1]

As a final experiment, the sliding lid implementation was parallelized using the unofficial python bindings for the Message Passing Interface (MPI), `mpi4py` and a two-dimensional domain decomposition.

The `mpi4py` package is available online at `https://github.com/mpi4py/mpi4py`. It is a mature library and one of the most used Python bindings for the Message Passing Interface (MPI)[1].

8.1 Domain Decomposition

In order to enable parallel execution of the code, the domain was spatially decomposed and each subdomain was assigned to a separate processor, similar to the way depicted in Figure 8.1.

In order to effectively and seamlessly parallelize the experiment, a cartesian topology was used. Extra care had to be taken for subdomains that correspond to the topmost and rightmost parts of the original lattice domain. More specifically their size had to be calculated differently than all other domains, in cases where the domain was not exactly divisible with the chosen decomposition.

8.2 Communication

Due to the nature of the experiment, occupation numbers need to move from the boundaries of one domain to the neighboring one. As a result the size of each subdomain was increased to accommodate ghost cells that were used as receive buffers for those occupation numbers.

[1]Unless cited otherwise, the theoretical background presented here is based on the lectures of Prof. Andreas Greiner on *High-Performance Computing: Fluid Mechanics with Python* at University of Freiburg as presented at Summer Semester of 2022.[2]

In total, four communications steps are necessary to populate those ghost cells with the values from the neighboring subdomains, namely:

1. Send to top and receive from down
2. Send down and receive from top
3. Send to the left and receive from the right
4. Send to the right and receive from the left

Those steps were implemented with the `Sendrecv` method of `mpi4py`.

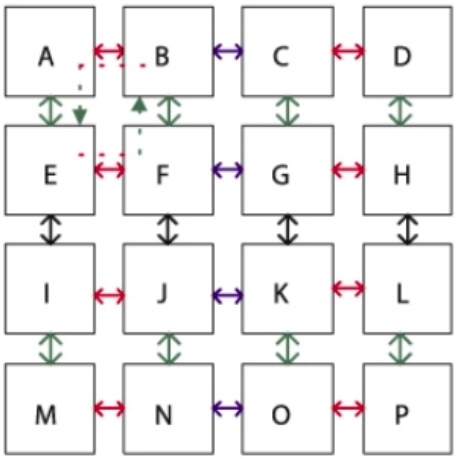


Figure 8.1: Domain decomposition example with communication steps. Image taken from https://stackoverflow.com/questions/43857497/mpi-partition-and-communication-in-2d-topology-velocity-directions

The communication between the processes plays a crucial role in the total speedup achieved by increasing the number of processes. Keeping the grid size constant, one might expect that as the number of processes increases, the execution time will decrease. However, as the number of processes increases, the total overhead needed for those processes to communicate their ghost cells also increases. At one point, the overhead that communication adds will overcome the gain due to the increased number of processes. From that point onward, more processes might even lead to an increase in the execution time. This behavior is also illustrated on the experimental results and plots presented in the next chapters.

8.3 Parallel Implementation in Python

This section will discuss some of noteworthy implementation details. As always, the full implementation is available online at `https://github.com/theodorju/fr-hpcpy-pub`.

The MPI Cartesian communicator was used in order to have an easy mapping between the rank number of each processor and its corresponding responsibility subdomain in the lattice. The instantiation of the Cartesian communicator is presented in listing 8.1. The keyword argument `periods` is given as `(False, False)` to denote the solid boundaries both in x as well as in y direction. This essentially sets invalid ranks of sources and destinations to a negative number (-2) to avoid sending and receiving from them.

Listing 8.1: MPI Send-Receive operation

```
# Create a cartesian communicator
cartcomm = comm.Create_cart(dims=[nsub_x, nsub_y],
                            periods=(False, False),
                            reorder=False
                            )
```

As already mentioned, communication between the neighboring processes is done with the use of `Sendrecv` method from `mpi4py`. One downside of this, is that the buffers passed in `Sendrecv` command need to be contiguous in memory. For that reason, the method `np.ascontiguousarray`[2] was used wherever necessary in combination with a buffer. As expected, this leads to an increase in the memory footprint of the application. A send-receive step is presented in listing 8.2

Listing 8.2: MPI Send-Receive operation

```
buffer = np.ascontiguousarray(proba_density[:, :, 0])
comm.Sendrecv(np.ascontiguousarray(proba_density[:, :, -2]),
              u_dest,
              recvbuf=buffer,
              source=u_source)
proba_density[:, :, 0] = buffer
```

The communication step needs to happen before the streaming. All other operations continue after the communication step, similarly to the serial execution of the sliding lid experiment.

8.4 Simulation Settings

The simulation settings that were used for the simulation are given in table 8.1 to ensure reproducibility of the results.

Additionally, experiments with 1000 x 1000 and 3000 x 3000 lattices were also executed in `bwUniCluster`, in order to observe how the size of the lattice affects scaling. All other settings were left as in table 8.1.

[2]https://numpy.org/doc/stable/reference/generated/numpy.ascontiguousarray.html

Table 8.1: Sliding lid Simulation settings

Setting	Value
Discretization Scheme	D2Q9
Grid size	300 x 300
Wall velocity	0.1
Simulation Steps	10000
Collision frequency (*omega*)	1.7
Initial probability distribution	Equilibrium
Initial Density	1.0
Initial Velocity	0.0
Reynolds number	1020
Speed of sound squared	1/3

The number of processes in the parallel execution ranged from 2 to 2048. Namely, the first execution form each different lattice dimension was done using 2 processes and that number was doubled until it reached 2048 processes. The domain decomposition was modified accordingly.

8.5 Execution in bwUniCluster

All benchmarking experiments for the parallel execution were executed in the `bwUniCluster`[3].

The experiments were executed using `sbatch` and varying the number of nodes and the number of tasks per node in order to increase the total number of cores. Each node had access to 12GB of memory.

Both the file that was used to create each individual job `run_sbatch_job.sh` as well as the one that was used to submit all the jobs `submit_jobs.sh` are available in the projects repository[4] under the `/src` directory.

8.6 Simulation Results

Figure 8.2 depicts the resulting velocity field from a parallel execution of the sliding lid experiment after 10,000 simulation steps with the settings presented in table 8.1. This figure is presented as proof that the simulation runs correctly in parallel.

Figure 8.3 depicts the Million Lattice Updates Per Second (MLUPS) as a function of the number of processes used to parallelize the experiment for 300x300, 1000x1000 and 3000x3000 grids. In the 300x300 grid one can observe that, even though the gain is almost linear at the beginning, this grad-

[3] `https://wiki.bwhpc.de/e/Main_Page`
[4] `https://github.com/theodorju/fr-hpcpy-pub`

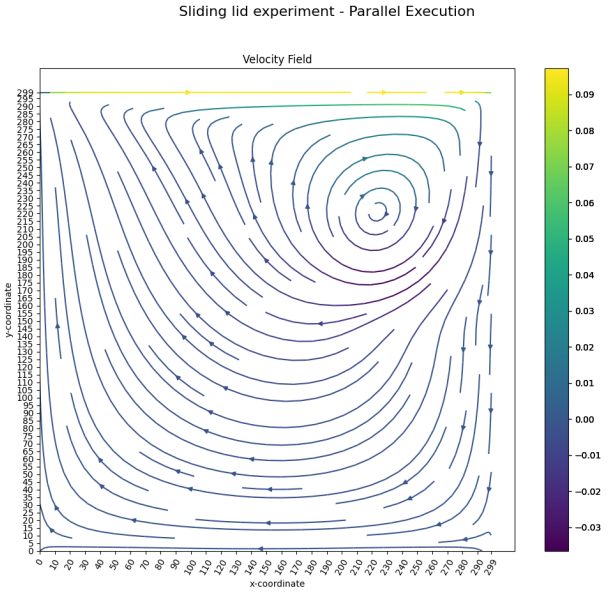


Figure 8.2: Velocity field after 10000 simulation steps in parallel. A 300 x 300 grid was used with collision frequency 1.7 and lid velocity 0.1. The experiment was executed in parallel using 4 processes and a 2 x 2 domain decomposition. The colorbar maps the magnituted of the velocity in different parts of the grid.

ually becomes sublinear and at the end reaches a point where adding more processes hurts the performance. The results are simular for the 1000x1000 and 3000x3000 cases. As expected, the number of processes where the gain becomes sublinear in those cases increases, due to the increase of the lattice size.

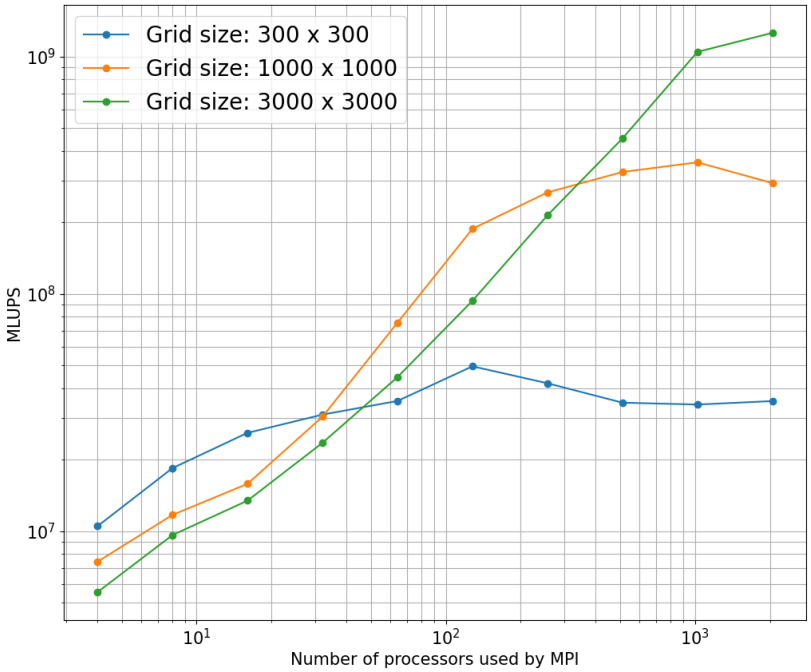


Figure 8.3: MLUPS (Million Lattice Updates Per Second) as a function of the number of processes for different grid sizes. Both axes are in log-scale. Additionally, when calculating the timings used in this experiment, all commands that wrote data to a file were excluded from execution. The exact timings are presented in the Appendix.

Appendix A

Parallel execution timings and MLUPs

The exact timings and MLUPs as a function of the number of processes for different grid size from the `bwUniCluster` execution are presented in table A.1, A.2, A.3

Table A.1: Execution timings and MLUPs for 300x300 grid

Number of cores	Wall clock time (min:sec)	MLUPS
4	1:25.783	10491589.24
8	0:48.824	18433557.27
16	0:34.680	25951557.09
32	0:29.053	30977868.03
64	0:25.445	35370406.76
128	0:18.184	49494060.71
256	0:21.427	42003080.23
512	0:25.875	34782608.7
1024	0:26.347	34159486.85
2048	0:25.474	35330140.54

Table A.2: Execution timings and MLUPs for 1000x1000 grid

Number of cores	Wall clock time (min:sec)	MLUPS
4	22:24.813	7435978.088
8	14:13.824	11712015.59
16	10:31.842	15826741.5
32	5:29.451	30353527.54
64	2:12.326	75570938.44
128	0:53.175	188058298.1
256	0:37.373	267572846.7
512	0:30.658	326179137.6
1024	0:27.906	358345875.4
2048	0:34.153	292800046.8

Table A.3: Execution timings and MLUPs for 3000x3000 grid

Number of cores	Wall clock time (min:sec)	MLUPS
4	271:6.271	5532921.467
8	155:47.209	9628542.595
16	111:35.103	13442661
32	63:40.397	23557761.14
64	33:39.139	44573454.33
128	16:2.737	93483474.72
256	7:0.132	214218388.5
512	3:19.297	451587329.5
1024	1:26.009	1046402121
2048	1:11.516	1258459645

Table A.4: Domain decomposition based on the number of processes

Number of cores	Domain Decomposition
4	2x2
8	4x2
16	4x4
32	8x4
64	8x8
128	16x8
256	16x16
512	32x16
1024	32x32
2048	64x32

Bibliography

[1] Lisandro Dalcin and Yao-Lung L. Fang. mpi4py: Status update after 12 years of development. *Computing in Science & Engineering*, 23(4):47–54, 2021.

[2] Andreas Greiner. Lecture notes in High Performance Computing: Fluid Mechanics with Python, Summer Semester 2022. University of Freiburg.

[3] A. A. Mohamad. *Lattice Boltzman Method: Fundamentals and Engineering Applications with Computer Codes*. Springer: London, 2011.

[4] Krüger Timm. Introduction to lattice boltzmann method. `https://www.youtube.com/watch?v=jfk4feD7rFQ`, 2021.

[5] Krüger Timm, H Kusumaatmaja, A Kuzmin, O Shardt, G Silva, and E Viggen. *The lattice Boltzmann method: principles and practice*. Springer: Berlin, Germany, 2016.