Report: Lattice Boltzmann

Derrick Timmerman S3523799, Wesley Seubring S3529797, Group 06

October 12, 2018

Abstract

[General introduction]
[To be added as last paragraph of our report]

Contents

1 Introduction

Lattice Boltzmann method (LBM) is a very commonly used method for the simulation of fluid and can work with multi-phase elements, calculate flow through complex geometries and allows for the addition of calculation complex processes as a heat transfer in a flow [8]. When implementing LBM there are many lattice BGK models available to use for simulation for different dimension's of the simulation) [10]. Also there is already a lot of research done in more complex simulations of LBM, including the simulation of multiple-component, multiple-phase and heat transfer [12, 4] .

In this research paper we will discuss and analyze the Lattice Boltzmann Method (LBM) for simulating the emulsion of multiple fluids in a fixed environment. This research will first look at the implementation of a simulation of a single fluid and experiment with Reynolds number and look into the parallel property of the method. After the simulation with a single fluid, there will be a simulation of a emulsion of two fluids. Furthermore there will be looked into the strength's and weaknesses of the used LBM simulations.

In section 2 Theory, the methods and theory relevant to the experiment are discussed. In section 3 Research, there will be a description of the experiments done in this report. The results are showed in the the section 4 Results and are discussed in section 5 Discussion. In section 6 Conclusion, there will be a summary of the found results.

2 Theory

2.1 Lattice Boltzmann Method

The Lattice Boltzmann method is a numerical approach on the mechanic of fluid simulation. LBM looks at macroscopic properties (e.g. mass and momentum) of particles to compute the behavior. The implementation of the LBM is relatively easy and can make great use of parallel computing. On top of LBM there are plenty of models that extend on the LBM to compute more complex problems like multi-component flows and fluid simulation in complex geometries. [1, 7]

2.2 D2Q9

D2Q9 is one of the many different LBM-models that are being used [10]. The notation for these models is DxQy where x stands for the number of spatial dimensions, and y for the total number of velocities. In other words: D2Q9 represents a 2D-model with particles having the ability to move around in 9 different directions.

2.3 Boundary conditions

When using LBM, we have to deal with flow and therefor it is necessary to have techniques included that mimic boundary conditions. There has already been a lot of research into the behavior of several different types of boundary conditions for the LBM [9]. Boundary conditions that are very precise are not only required for simulations with complex geometry, but also for simulations with simple geometry like a sphere or in our case a cylinder. For fluid flowing around a cylinder we need no-slip boundary conditions, because we want to treat the cylinder as a solid object from which the particles bounce back and hence have a velocity of zero at the wall. Although there are several types of boundary conditions, we will limit ourselves to the bounce-back condition only, as we have used this type for our LBM implementation. Details for the bounce-back will be given in this thesis, following Bouzidi et al. [6].

2.3.1 Bounce-Back boundary condition

As already mentioned, bounce-back boundaries consist of a distribution function bounce-back scheme used at walls and objects to obtain a no-slip velocity condition. However, these boundaries do not guarantee perfect no-slip boundary conditions. During previous research, it has been noticed that the bounce-back scheme actually only gives first-order spatial accuracy in boundaries [2, 3].

Boundaries are created by letting particular nodes behave as solid obstacles. We can divide these solids into two groups: the ones that lie at the solid-fluid interface and the ones that have no contact with the fluid nodes, whereby the latter of these do not communicate with boundary nodes anymore. When using the full-way bounce-back, the fluid particles simply reverse their direction of motion. This can be seen in Figure 1.

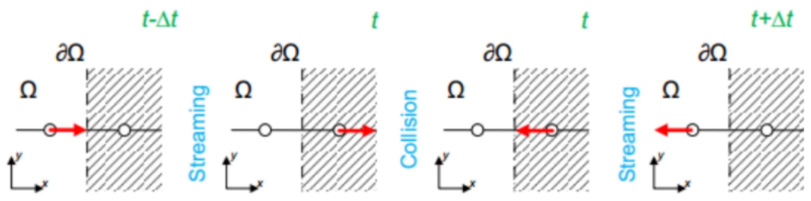

Figure 1: Overview of full-way bounce-back. Here Ω represents the fluid and ∂Ω represents the boundary. In the streaming step a fluid population leaves the fluid node and reaches the boundary solid node. Afterwards, in the collision step its velocity is reversed and in the next streaming step the fluid population leaves the boundary solid node and gets back to the fluid node, only having a reversed direction of motion. This process takes two time steps. Source: [11]

Fluid particles reversing their direction of motion can be expressed as follow:

$$n^*_{q'}(\vec{r}_j, t) = n_q(\vec{r}_j, t)$$

2.4 Parallel LBM

LBM can be optimized to compute more complex and or larger simulations. This can be done using multi-block and multi-grid LBM methods[8]. To deal with this problem the lattice grid is split in block and computed over multiple thread or machines. To encounter the missing information of the boundaries of these block, there need to be a follow up computation.

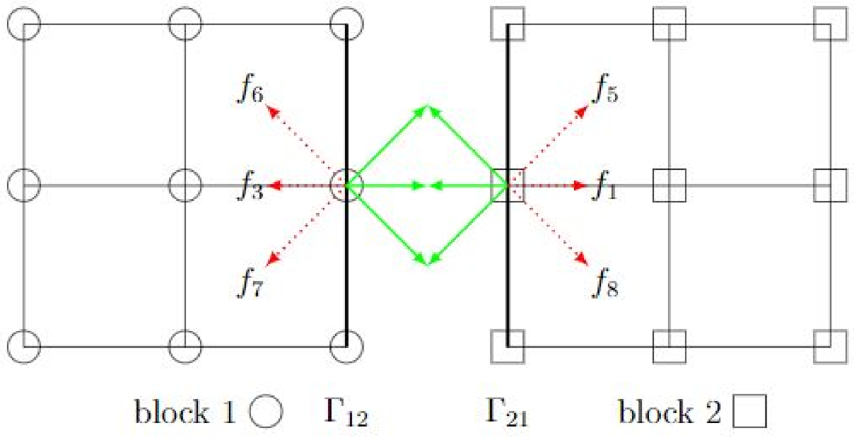

Figure 2: Sketch for multilblock. Source [8]

2.5 LBM width multiple components

3 Research

Using the theory of LMB, the first simulation of LBM can be done. The research starts by setting up a simulation of one fluid using LBM with D2Q9, which is used for a 2D simulation. To visualize the workings of LBM within the experiment, the speed of the fluid will be shown as it changes due to flow or object interference. The implementation (See **??**) is written in Python and is an altered version of the available on Palabos [5].

The first simulation can be used to look at the impact of Reynolds number (Re). Changing the values of the Reynolds number will change the flow condition. The effects of different Reynold numbers will be tested and studied.

Our implementation initially performed all tasks on a single thread, but it turned out soon that calculating the collision and streaming steps were compute-intensive tasks, causing the simulation to become relatively slow. To give it a more realistic and real-time feeling, we started making adjustments to our code which could lead to a significant increase in performance. Based on [7], we tried to split the particles into multiple so-called blocks. This gives us the ability to calculate the collision and streaming steps for each block separately on multiple threads. After doing so, we would only have to merge the blocks to complete an iteration.

4 Results

4.1 Single fluid

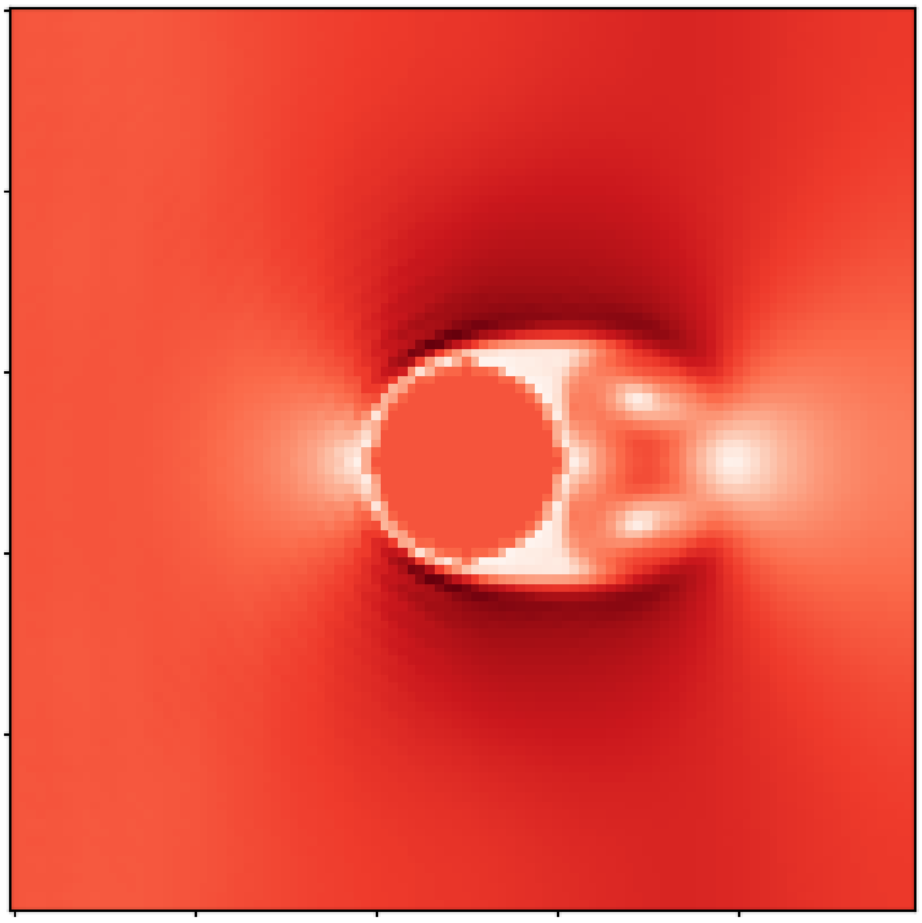
Figure 3: Speed of the fluid with a flow from left to right

The image above shows the speed (from white slow to red fast) of a fluid with a outflow to the right side of the screen.

[Reynolds number]
[Parallel]
[Multiple component simulation]

5 Discussion

[Rewriting the code very useful for understanding]
[Single fluid standard]
[Reynolds number]
[Parallel]
[Multiple component simulation]

6 Conclusion

[Single fluid standard]
[Reynolds number]
[Parallel]
[Multiple component simulation]

7 Work load

8 Reference

9 Appendix

Listing 1: D2Q9 LBM simulation code

```
from numpy import *;
from numpy.linalg import *
import matplotlib.pyplot as plt;
from matplotlib import cm

#https://en.wikipedia.org/wiki/Reynolds_number
# low == more smooth fluid | High == more chaotic and ↩
    turbulantflow
n_iterations = 20000  # Total number of time iterations.
Reynolds_number = 220.0
water_density = 1.0

plot_width = 400
plot_height = 400
```

```

ly = plot_height - 1.0 #TBD


# number of lattice velocities
q = 9

#Cylinder coordinates
cylinder_x = plot_width / 2
cylinder_y = plot_height / 2
cylinder_radius = plot_height / 9

#Velocity in lattice units
velocity_lattice_units = 0.04

#todo Define vars
nulb = velocity_lattice_units * cylinder_radius / ↩
    Reynolds_number
relaxation_time = 1.0 / (3. * nulb + 0.5)  # Relaxation ↩
    parameter.

###### Lattice Constants ######
D2Q9_FAST_WEIGHTING_VECTOR = 1. / 36.
D2Q9_SLOW_WEIGHTING_VECTOR = 1. / 9.
D2Q9_REST_WEIGHTING_VECTOR = 4. / 9.
D2Q9_SPEED = 1. / 3.

lattice_directions = array([(x, y) for x in [0, -1, 1] for ↩
    y in [0, -1, 1]])

#Intit all weights to fast
lattice_weights = D2Q9_FAST_WEIGHTING_VECTOR * ones(q)

#Init the xaxis and yaxis horizontal with slow weights
lattice_weights[asarray([norm(direction) == 1. for ↩
    direction in lattice_directions])] = ↩
    D2Q9_SLOW_WEIGHTING_VECTOR;

#Set center particle to a rest weight
lattice_weights[0] = D2Q9_REST_WEIGHTING_VECTOR

#Find al the inverse indexes of the directions of the ↩
    lattice #aka: noslip
inverse_direction_indexes = []
for direction in lattice_directions:
    inverse_direction_index = lattice_directions.tolist().↩
        index((-direction).tolist())
    inverse_direction_indexes.append(↩
        inverse_direction_index)

```

```python
#Unknow indexes to the walls. todo: Find a nice way to find the indexes of lattice_directions
index_range = arange(q)
left_direction_indexes = index_range[asarray([direction[0] < 0 for direction in lattice_directions])]
vertical_direction_indexes = index_range[asarray([direction[0] == 0 for direction in lattice_directions])]
right_direction_indexes = index_range[asarray([direction[0] > 0 for direction in lattice_directions])]

sumpop = lambda fin: sum(fin, axis=0)  # Helper function for density computation.

def equilibrium(density, velocity):  # Equilibrium distribution function.
    # To find out: why multiplying with magic number 3.0
    # Calculates velocity for each direction of each element (e.g. 2x2 plane results in 9x2x2 arrays)
    direction_velocities = 3.0 * dot(lattice_directions, velocity.transpose(1, 0, 2))

    # Calculating squared velocity by the D2Q9_SPEED property
    squared_velocity = (velocity[0] ** 2 + velocity[1] ** 2) / (2 * D2Q9_SPEED)

    # Initialising zero-matrices for each direction
    equilibrium_distribution = zeros((q, plot_width, plot_height))

    # Equilibrium mass distribution
    for i in range(q): equilibrium_distribution[i, :, :] = density * lattice_weights[i] * (1. + direction_velocities[i] + 0.5 * direction_velocities[i] ** 2 - squared_velocity)
    return equilibrium_distribution

###### Setup: cylindrical obstacle and velocity inlet with perturbation ########
obstacle = fromfunction(lambda x, y: (x - cylinder_x) ** 2 + (y - cylinder_y) ** 2 < cylinder_radius ** 2, (plot_width, plot_height))
lattice_velocities = fromfunction(lambda d, x, y: (1 - d) * velocity_lattice_units * (1.0 + 1e-4 * sin(y / ly * 2 * pi)), (2, plot_width, plot_height))
equilibrium_distribution = equilibrium(water_density, lattice_velocities)
fin = equilibrium_distribution.copy()

```

```
###### Main time loop ##########################################################
for time in range(n_iterations):
    #fin[left_direction_indexes, -1, :] = fin[left_direction_indexes, -2, :]  # Right wall: outflow condition.
    density = sumpop(fin)  # Calculate macroscopic density and velocity.
    velocity = dot(lattice_directions.transpose(), fin.transpose((1, 0, 2))) / density

    velocity[:, 0, :] = lattice_velocities[:, 0, :]  # Left wall: compute density from known populations.
    density[0, :] = 1. / (1. - velocity[0, 0, :]) * (sumpop(fin[vertical_direction_indexes, 0, :]) + 2. * sumpop(fin[left_direction_indexes, 0, :]))

    equilibrium_distribution = equilibrium(density, velocity)  # Left wall: Zou/He boundary condition.
    #fin[right_direction_indexes, 0, :] = fin[left_direction_indexes, 0, :] + equilibrium_distribution[right_direction_indexes, 0, :] - fin[left_direction_indexes, 0, :]
    fin[right_direction_indexes, 0, :] = equilibrium_distribution[right_direction_indexes, 0, :]
    fout = fin - relaxation_time * (fin - equilibrium_distribution)  # Collision step.
    for i in range(q): fout[i, obstacle] = fin[inverse_direction_indexes[i], obstacle]
    for i in range(q):  # Streaming step.
        fin[i, :, :] = roll(roll(fout[i, :, :], lattice_directions[i, 0], axis=0), lattice_directions[i, 1], axis=1)

    #   if (time % 100 == 0):  # Visualization
    plt.clf();
    plt.imshow(sqrt(velocity[0] ** 2 + velocity[1] ** 2).transpose(), cmap=cm.Reds)
    plt.pause(0.001)
```

References

[1] Meskas, J. Bill, Bao, Y. *Lattice Boltzmann Method for Fluid Simulations.* 2011. URL: https://www.math.nyu.edu/~billbao/report930.pdf.

[2] J. G. Georgiadis R. O. Buckius D. R. Noble S. Chen. "A consistent hydrodynamic boundary condition for the lattice boltzmann method". In: *Physics of fluids* 7 (203 1995).

[3] J. G. Georgiadis R. O. Buckius D. R. Noble S. Chen. "An evaluation of the bounce-back boundary condition for lattice boltzmann simulations". In: *International journal for numerical methods in fluids* 25 (249 1997).

[4] Zhixin, L. Jinku, W. Moran, W. *A lattice Boltzmann algorithm for fluid–solid conjugate heat transfer.* 2006.

[5] *Lattice Boltzmann in various languages.* Accessed on: 11-10-2018. URL: http://wiki.palabos.org/numerics:codes.

[6] P. Lallemand M. Bouzidi M. Firdaouss. *Momentum transfer of a boltzmann-lattice fluid with boundaries.* 2001.

[7] E. Magnus Viggen. *The lattice Boltzmann method: Fundamentals and acoustics.* 2014. URL: https://www.researchgate.net/profile/Zoltan_Horvat/post/What_is_lattice_boltzmann_methodLBM_all_about/attachment/59d630ed79197b807798eb98/AS%5C%3A363442914512896%5C%401463662951192/download/Erlend+Magnus+Viggen+-+The+lattice+Boltzmann+method.pdf.

[8] Peng, C. *The Lattice Boltzmann Method for Fluid Dynamics: Theory and Applications.*

[9] Li-Shi Luo Pierre Lallemand. *Lattice Boltzmann method for moving boundaries.* 2002.

[10] Lallemand, P. Qian, Y. H. D'humiéres, D. *Lattice BGK Models for Navier-Stokes Equation.* 1992.

[11] S.C. Wetstein. *IMPLEMENTING THE LATTICE-BOLTZMANN METHOD.* 2014.

[12] Hudong, C. Xiaowen, S. *Lattice Boltzmann model for simulating flows with multiple phases and components.* 1992.