Master Thesis Experiment Report IV : Optimisation and Bounceback

Jérémie Guy

October 18, 2024

Abstract

Code optimisation and fine-tuning

1 Context

In this experiment, we will focus on optimizing the code and fine-tuning the bounceback behavior. In particular, we will look at the streaming step and bounceback step of the main loop of the simulation. The details of both have been covered in previous reports, so we will not go into too much detail in this one.

2 Problem

The streaming up until now follows the standard by going through each node, and for each direction calculates the next location to which the PDFs have to go. We will instead use the Numpy roll function to do so. The bounceback nodes don't need to follow the same collision as the rest of the system and their initial definition can be changed.

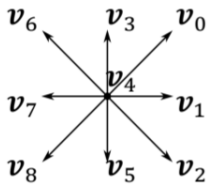


Figure 1: Directional velocity in a D2Q9 lattice, **Source :** [1]

3 Experiments

3.1 Experiment 1

3.1.1 Description

We laid the roll for every direction: we determined which axis is relevant for any direction and manually assigned them (based on the velocity vectors illustrated in Figure 1) to bring the PDFs to their next cell destination. Moreover, for the diagonal directions, we used two rolls in a row (one for horizontal movement and one for vertical movement). The details are shown in the next Python code section.

```python
# streaming step
# previous
for x in range(nx):
    for y in range(ny):
```

```
        for i in range(9):
            if flags[x,y] == 0 or flags[x,y]==1:
                next_x = x + v[i,0]
                next_y = y + v[i,1]

                if next_x < 0:
                    next_x = nx-1
                if next_x >= nx:
                    next_x = 0

                if next_y < 0:
                    next_y = ny-1
                if next_y >= ny:
                    next_y = 0

                fin[i,next_x,next_y] = fout[i,x,y]
```

```
# streaming step
# optimized
# i = 0
fin[0,:,:] = roll(roll(fout[0,:,:],1,axis=0),1,axis=1)
# i = 1
fin[1,:,:] = roll(fout[1,:,:],1,axis=0)
# i = 2
fin[2,:,:] = roll(roll(fout[2,:,:],1,axis=0),-1,axis=1)
# i = 3
fin[3,:,:] = roll(fout[3,:,:],1,axis=1)
# i = 4
fin[4,:,:] = fout[4,:,:]
# i = 5
fin[5,:,:] = roll(fout[5,:,:],-1,axis=1)
# i = 6
fin[6,:,:] = roll(roll(fout[6,:,:],-1,axis=0),1,axis=1)
# i = 7
fin[7,:,:] = roll(fout[7,:,:],-1,axis=0)
# i = 8
fin[8,:,:] = roll(roll(fout[8,:,:],-1,axis=0),-1,axis=1)
```

Additionally, we made a quick change to the initial velocity: previously the initial velocity was the expected curve injected into the left-side border. We changed it so that it would scale with the inlet, regardless of its size and placement on the left-side border of the lattice.

Finally, we changed the bounceback step behavior making it more efficient. Previously, the bounceback step went through every coordinate, and for the nodes that were defined as bounceback using a flag array, it changed for every direction the input PDFs of a cell into its output in the same direction (standard node-based bounceback behavior). Instead, we used the flags as a mask that allowed us to bypass the double *for* loops checking every coordinate, making the code significantly faster. Details are shown below.

```
# Bounce-back condition
# previous
for x in range(nx):
    for y in range(ny):
        if flags[x,y] == 1:
            for i in range(9):
                fout[i,x,y] = fin[8-i,x,y]
```

```python
# Bounce-back condition
# optimized
for i in range(9):
    fout[i, flags] = fin[8-i, flags]
```

The changes have been tested on a small 15x11 system. The code runs over 500 iterations. The system structure is detailed in Figure 2.

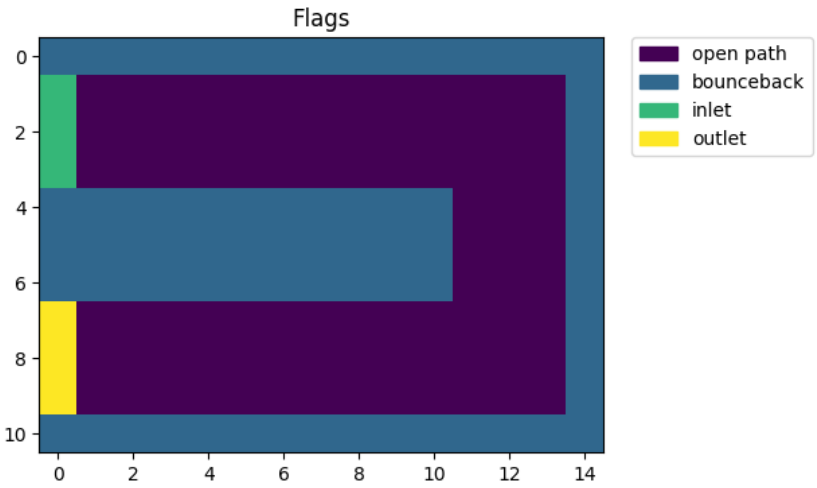
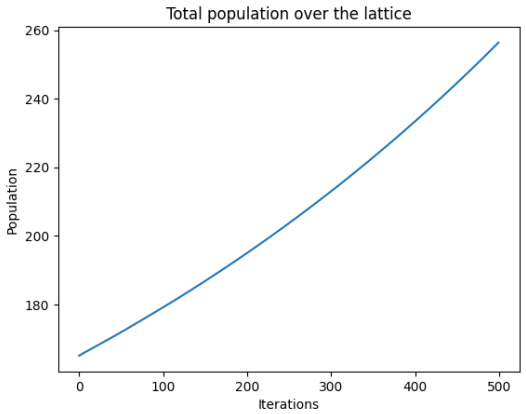


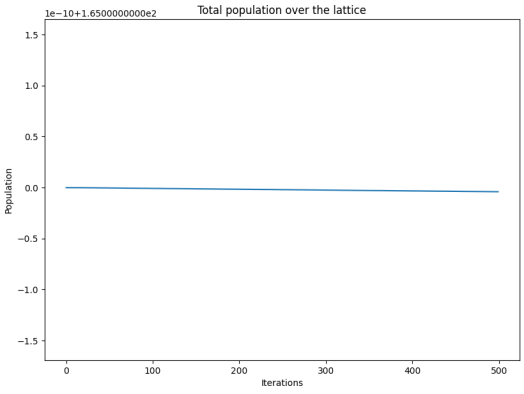
Figure 2: System Structure

3.1.2 Results

The modifications mentioned above resulted in a normal flow state. However, to check the results, we looked at the total sum of the population of the lattice throughout the iterations. The results are shown in Figure 3a. As expected, there is a slow and steady increase in population induced by the Zu-He border condition. As a precautionary measure, we also tested the same system, but without any inlet or outlet. The results shown in Figure 3b show that the results are almost constant: There is a very slight decrease of the PDF values at magnitude e-10, which can be considered negligible.



(a) Population sum with inlet & outlet



(b) Population sum with no inlet & outlet

Figure 3: Population Sums

3.2 Experiment 2

3.2.1 Description

During the streaming step, both the previous code and the new rolls possess a cyclic behavior: what goes out of the system from one side comes back on the opposite side. To counteract that effect, we first started by initializing the bounceback nodes differently. For every bounceback node, we set the PDfs values of the directions facing away from the open path of the system to 0 : this means that no values should cycle (only zeros). We tried as well to compensate for the cyclical roll behavior in specific directions, given the system detailed in figure 2. In this case, that means, for directions 6,7 and 8 (as detailed in Figure 1), the PDFs that leave the inlet and outlet site are expected to turn around and end at the bounceback nodes on the right border of the system. To counteract that, we set the incoming PDFs of said border for the three relevant directions to 0. Details are given below.

Initial bounceback conditions :

```
def setBBnodeDirToZero():
    # top border
    fin[[1,2,4,5,7,8],:,0] = 0
    # right border
    fin[[3,4,5,6,7,8],nx-1,:] = 0
    #  bottom border
    fin[[0,1,3,4,6,7],:,ny-1] = 0

    # obstacle Top except rightmost
    fin[[0,1,3,4,6,7],0:10,4] = 0
    # obstacle bottom except rightmost
    fin[[1,2,4,5,7,8],0:10,6] = 0
    # obstacle right border except corners
    fin[[0,1,2,3,4,5],10,5] = 0
    # obstacle top right corner
    fin[[0,1,4,3],10,4] = 0
    # obstacle bottom right corner
    fin[[1,2,4,5],10,6] = 0
    # obstacle inside
    fin[:,0:10,5] = 0
```

Roll compensation during streaming step :

```
# compensate roll
fin[[6,7,8],nx-1,:] = 0
```

3.2.2 Results

Unfortunately, the results are not satisfactory. For every case we describe above, the system PDFs grow exponentially due to some numerical instability. We will need to further assess if its relevant to continue to compensate the cyclic behavior and initial conditions or if it does not influence the system badly enough.

3.3 Experiment 3

3.3.1 Description

We started by removing the initial conditions on the BB nodes stated in the previous experiment. The decreasing population mentioned in the first experiment could be explained by the collision step. The collision step computes how incoming PDFs from every direction in a cell collide with each other and how they distribute themselves to go out of the cell during the streaming step. The collision in Lattice Boltzmann is formulated using the following equation :

$$fout = fin - omega * (fin - feq) \quad (1)$$

where $fout$ is the PDFs going out of a cell, fin the PDFs coming in, feq the equilibrium state of the fluid and *omega* the relaxation parameter which indicate how fast the particles converge back to their equilibrium state. Up until now the collision step was made over the whole system using *numpy*'s arrays :

```
    fout = fin - omega * (fin - feq)
```

Instead we limited the collision step only to the nodes that are not bounceback using the flags mask mentioned earlier :

```
    fout[:,invFlags] = fin[:,invFlags] - omega * (fin[:,invFlags] - feq[:,invFlags])
```

3.3.2 Results

We again tested the total population of the system when removing inlet & outlet. The results illustrated in Figure 4 show that expect the initial perturbation, the total PDFs state constant, which is the what we expected.

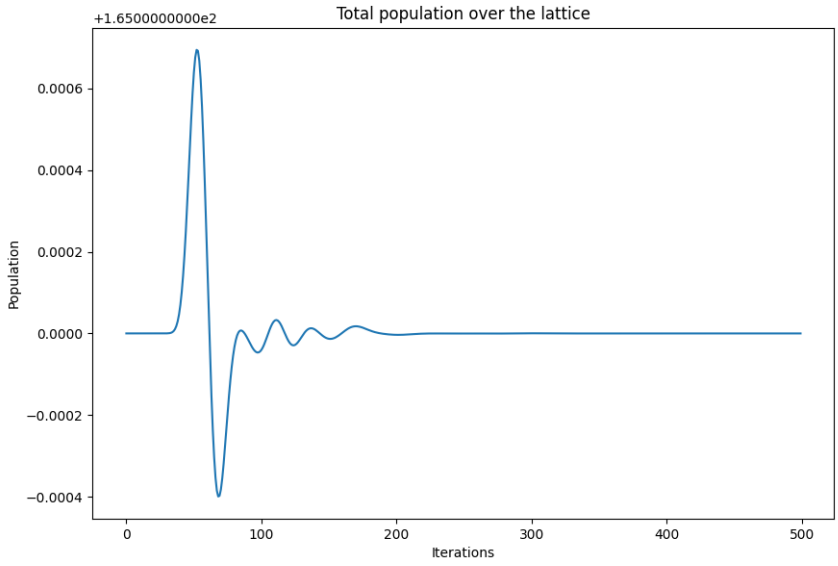


Figure 4: Total population with no inlet or outlet, new collision definition, no initial BB conditions

3.4 Experiment 4

3.4.1 Description

We tried once more to set the initial values of the BB nodes to 0. This time, instead of manually choosing the relevant directions that have to be set to 0, we set all the directions of the BB nodes to 0 :

```
def setBBNodeToZero():
    # top border
    fin[:,:,ny-1] = 0
    # bottom border
    fin[:,:,0] = 0
    # right border
```

```
    fin[:,nx-1,:] = 0

    # obstacle
    fin[:,0:11,4:7] = 0

     # top border
    fout[:,:,ny-1] = 0
    # bottom border
    fout[:,:,0] = 0
    # right border
    fout[:,nx-1,:] = 0

    # obstacle
    fout[:,0:11,4:7] = 0
```

3.4.2 Results

We tested this using the same population benchmark as before, illustrated in Figure 5. The initial behavior is different, but the system stabilized once more to a constant total PDF value. However, this sparked another problem : the fluid seems to come to a standstill. When looking at the norm of the velocities, we can see that the borders have a constant velocity, which is much superior as compared to the open path velocity. The phenomenon is illustrated in Figure 6. The system stays exactly the same for the entire simulation. We need to dig deeper in perhaps the equilibrium calculation part as it is not necessarily relevant to include the bounceback nodes. We will come back to it after further discussion.

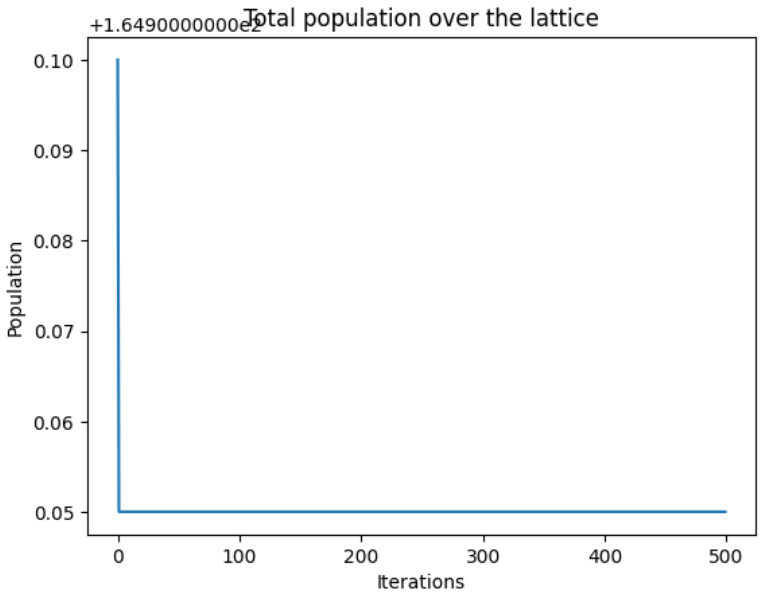


Figure 5: Total population with no inlet or outlet, with the new collision definition

3.5 Experiment 5

3.5.1 Description

To further correct the behavior of the fluid, we changed the computation of the macroscopic variables as well as the equilibrium state. Previously, the computation was performed on the entire system, including the bounceback nodes. As their behavior differs from the rest of the fluid, we changed the calculation of the variables and equilibrium state by limiting it only to the nodes that were considered as open path (not bounceback) using the same flags as explained above. We went from :

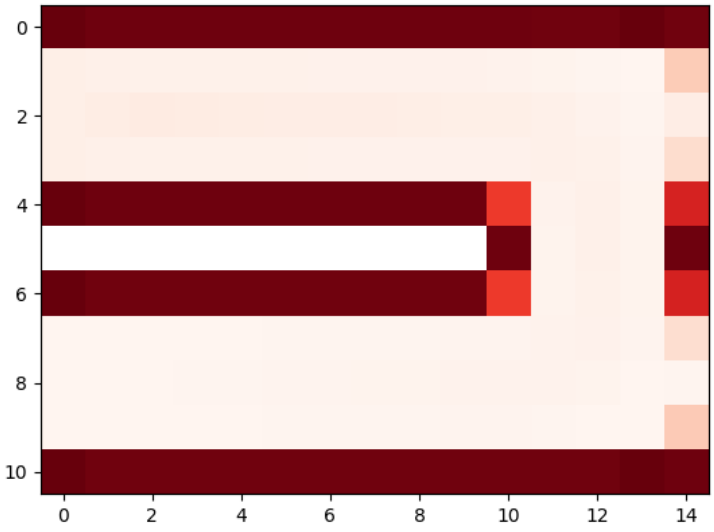


Figure 6: Norm of the velocities of the fluid at iteration 130

```python
# macroscopic variable computation
def macroscopic(fin):
    rho = sum(fin, axis=0)
    u = zeros((2, nx, ny))
    for i in range(9):
        u[0,:,:] += v[i,0] * fin[i,:,:]
        u[1,:,:] += v[i,1] * fin[i,:,:]
    u /= rho
    return rho, u

# Equilibrium distribution function.
def equilibrium(rho, u):
    usqr = 3/2 * (u[0]**2 + u[1]**2)
    feq = zeros((9,nx,ny))
    for i in range(9):
        cu = 3 * (v[i,0]*u[0,:,:] + v[i,1]*u[1,:,:])
        feq[i,:,:] = rho*t[i] * (1 + cu + 0.5*cu**2 - usqr)
    return feq
```

to, with applied changes :

```python
# macroscopic variable computation
def macroscopic(fin):
    rho = zeros((nx,ny))
    rho[openPath] = sum(fin[:,openPath], axis=0)

    u = zeros((2, nx, ny))
    for i in range(9):
        u[0,openPath] += v[i,0] * fin[i,openPath]
        u[1,openPath] += v[i,1] * fin[i,openPath]
    u[:,openPath] /= rho[openPath]
    return rho, u

# Equilibrium distribution function (rho = array)
def equilibrium(rho, u):
```

```
    usqr = 3/2 * (u[0,openPath]**2 + u[1,openPath]**2)
    feq = zeros((9,nx,ny))
    for i in range(9):
        cu = 3 * (v[i,0]*u[0,openPath] + v[i,1]*u[1,openPath])
        feq[i,openPath] = rho[openPath]*t[i] * (1 + cu + 0.5*cu**2 - usqr)
    return feq
```

Additionally, we increased the size of the system back to something more viable. Figure 7 shows the new system. We tested the changes over 60'000 iterations.

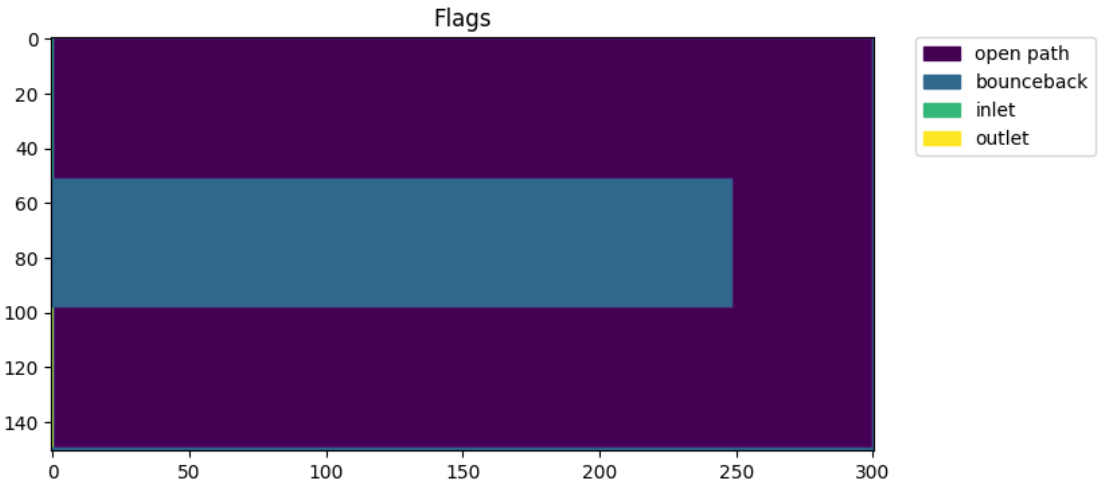


Figure 7: Increased system

3.5.2 Results

We first have to address an error mentionned above. The PDFs staying at the same values as described in experiment 4 was erronous : we confirmed that when removing inlet and outlet, the system still loses some particles. This has been reduced down to the BGK collision step : whatever happens we still lose some particles at this step. As the loss is greatly compensated with with the PDFs increase induced by Zu-He, we decided for now to leave it as is.

To test the results of this experiment, we looked at the overall PDFs throughout iterations as well as 3 velocity profiles spread in the various sections of the system. Figure 8 shows the total PDFs, Figure 9a shows where the velocity profiles have been taken and Figure 9b the resulting profiles.

As expected, the total PDFs is increase over iterations. The resulting velocity profiels have all the expected curve. The max velocity however seems to be decreasing over the iterations : this is certainly due to the viscosity of the fluid. The further we go from the inlet site, the less energy the fluid carries and its velocity is subsequently reduced.

4 Conclusion

5 References

[1] Jonas Lätt. "La méthode de Boltzmann sur réseau pour la simulation des fluides". In: ().

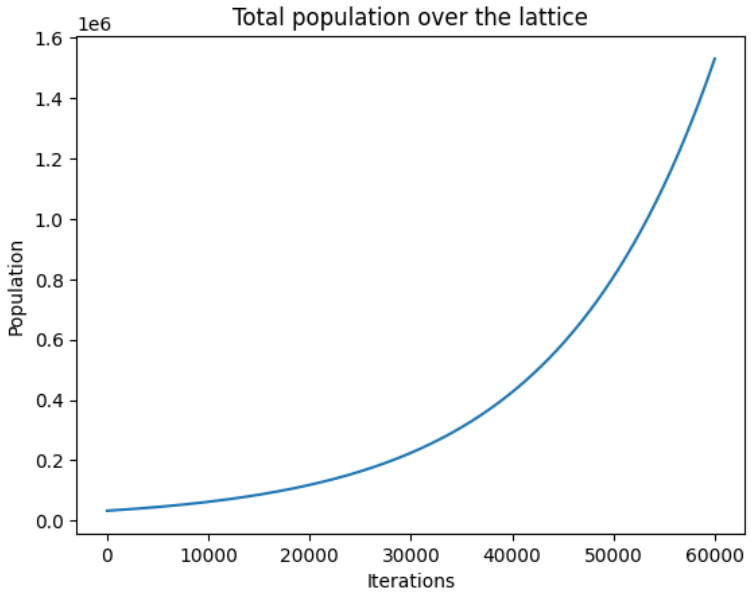
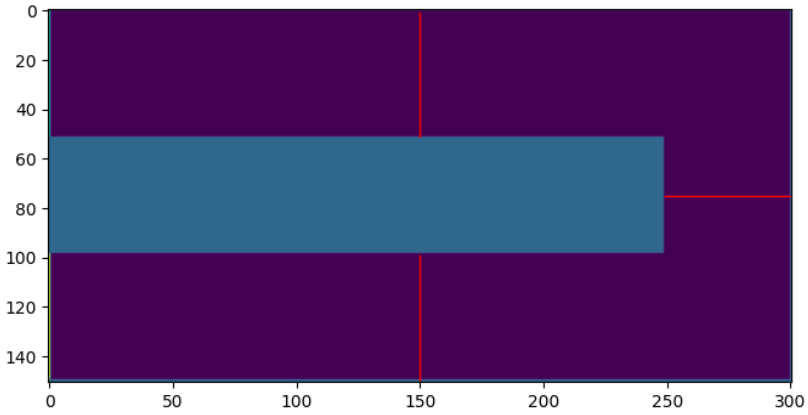
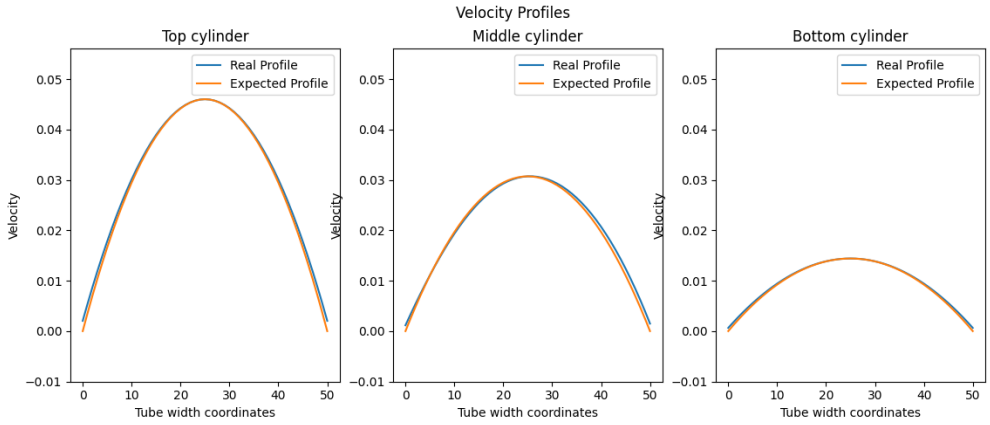


Figure 8: PDFs across 60'000 iterations



(a) System with velocity profiles sites



(b) Velocity profiles

Figure 9: Velocity profiles across system, 60'000 iterations