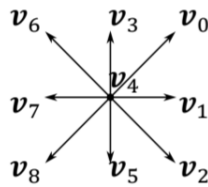


Jérémie Guy
October 6, 2024

Code optimisation and fine-tuning

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.

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.



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.

1

```

5     for i in range(9):
6         if flags[x,y] == 0 or flags[x,y]==1:
7             next_x = x + v[i,0]
8             next_y = y + v[i,1]
9
10            if next_x < 0:
11                next_x = nx-1
12            if next_x >= nx:
13                next_x = 0
14
15            if next_y < 0:
16                next_y = ny-1
17            if next_y >= ny:
18                next_y = 0
19
20            fin[i,next_x,next_y] = fout[i,x,y]

```

```

1  # streaming step
2  # optimized
3  # i = 0
4  fin[0,:,:] = roll(roll(fout[0,:,:],1,axis=0),1,axis=1)
5  # i = 1
6  fin[1,:,:] = roll(fout[1,:,:],1,axis=0)
7  # i = 2
8  fin[2,:,:] = roll(roll(fout[2,:,:],1,axis=0),-1,axis=1)
9  # i = 3
10 fin[3,:,:] = roll(fout[3,:,:],1,axis=1)
11 # i = 4
12 fin[4,:,:] = fout[4,:,:]
13 # i = 5
14 fin[5,:,:] = roll(fout[5,:,:],-1,axis=1)
15 # i = 6
16 fin[6,:,:] = roll(roll(fout[6,:,:],-1,axis=0),1,axis=1)
17 # i = 7
18 fin[7,:,:] = roll(fout[7,:,:],-1,axis=0)
19 # i = 8
20 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.

```

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

```

```

1 # Bounce-back condition
2 # optimized
3 for i in range(9):
4     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.

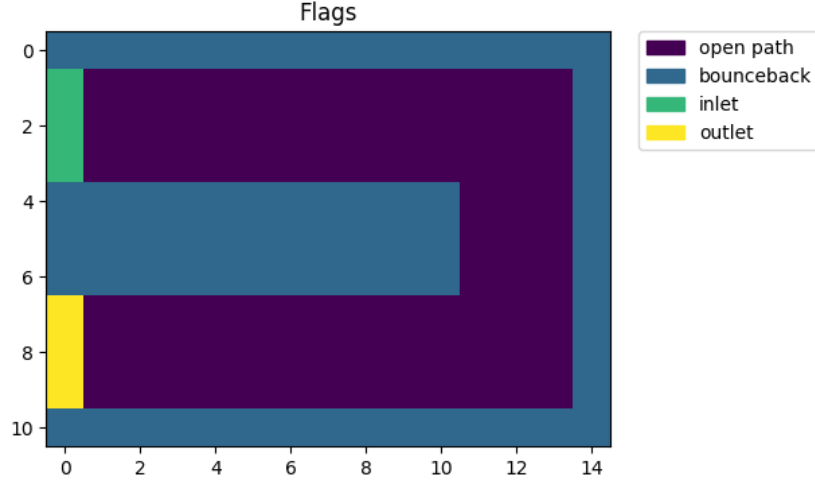


Figure 2: System Structure

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.

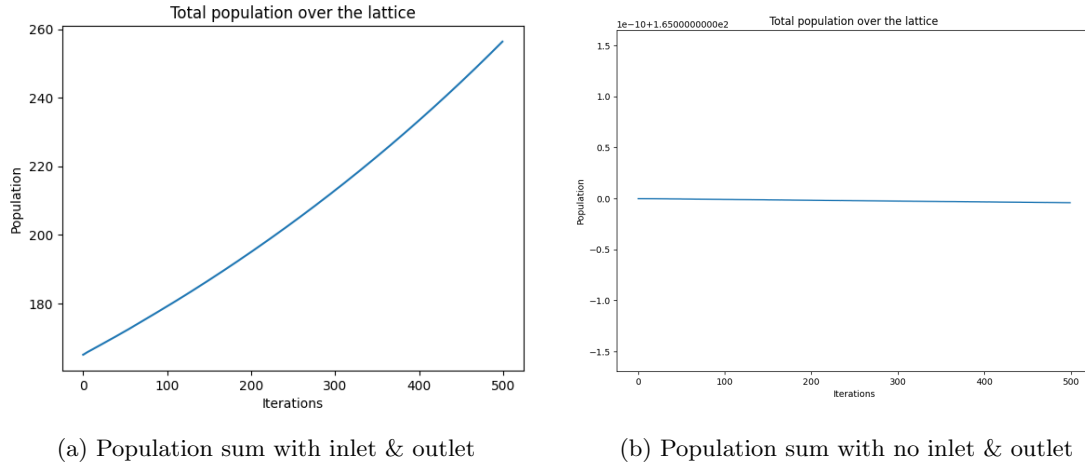


Figure 3: Population Sums

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 :

```
1 def setBBnodeDirToZero():
2     # top border
3     fin[[1,2,4,5,7,8],:,0] = 0
4     # right border
5     fin[[3,4,5,6,7,8],nx-1,:] = 0
6     # bottom border
7     fin[[0,1,3,4,6,7],:,ny-1] = 0
8
9     # obstacle Top except rightmost
10    fin[[0,1,3,4,6,7],0:10,4] = 0
11    # obstacle bottom except rightmost
12    fin[[1,2,4,5,7,8],0:10,6] = 0
13    # obstacle right border except corners
14    fin[[0,1,2,3,4,5],10,5] = 0
15    # obstacle top right corner
16    fin[[0,1,4,3],10,4] = 0
17    # obstacle bottom right corner
18    fin[[1,2,4,5],10,6] = 0
19    # obstacle inside
20    fin[:,0:10,5] = 0
```

Roll compensation during streaming step :

```
1 # compensate roll
2 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 :

$$f_{out} = f_{in} - \omega * (f_{in} - f_{eq}) \quad (1)$$

where f_{out} is the PDFs going out of a cell, f_{in} the PDFs coming in, f_{eq} the equilibrium state of the fluid and ω 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 :

```
1 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 :

```
1 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.

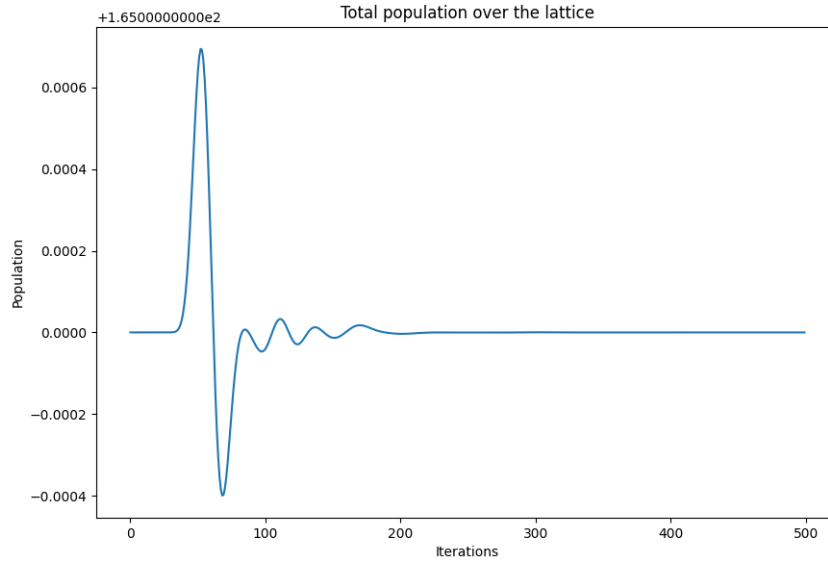


Figure 4: Total population with no inlet or outlet, new collision definition, no initial BB conditions

3.4 Experiment 4

3.4.1 Description

We tried once more to set initial values of 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 :

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

```

8
9     # obstacle
10    fin[:,0:11,4:7] = 0
11
12    # top border
13    fout[:,ny-1] = 0
14    # bottom border
15    fout[:,0] = 0
16    # right border
17    fout[:,nx-1,:] = 0
18
19    # obstacle
20    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 instigated 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 whole 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.

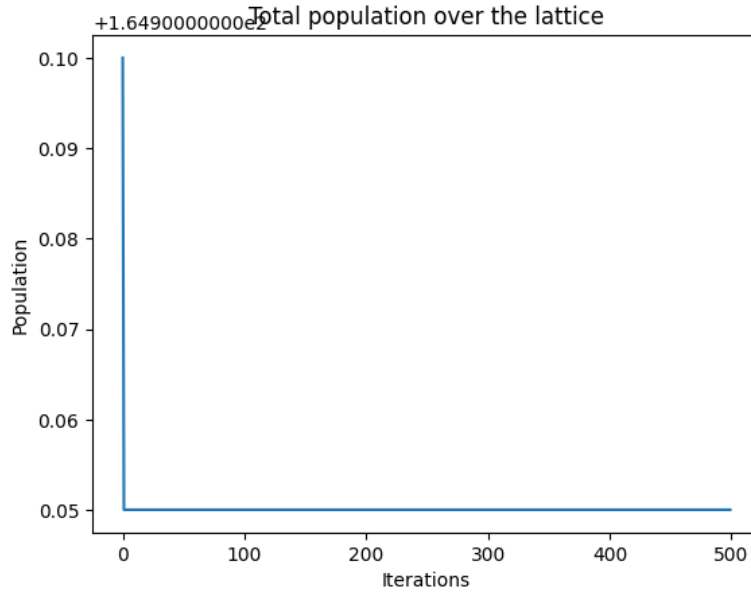


Figure 5: Total population with no inlet or outlet, with the new collision definition

4 Conclusion

5 References

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

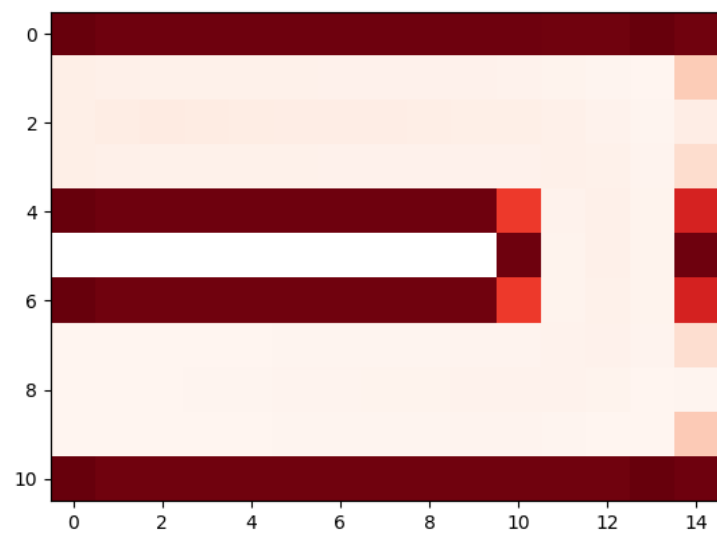


Figure 6: Norm of the velocities of the fluid at iteration 130