AI Builders: 2D Fluid simulation via the Lattice-Boltzmann method with conditional optimizers

Puripat Thumbanthu, Kunakorn Chaiyara

Started: 11/15/2024, Revision: November 17, 2024

Contents

CHAPTER 1

Introduction

1.1 Backgrounds

This project, submitted to the AI Builders X ESCK program, originated from a series of questions.

- What's the best way to blow on a liquid filled spoon to cool it?
- Given a room, what's the best place to place an air conditioner, and what direction must it face?
- What's the best place for a cooling fan in a CPU?,

etc. These problems are a set of problems that all fall in optimization problems in fluids:

> *"Given an imposed boundary condition on a system containing fluids, a boundary condition that's free to move, and a certain function, find the boundary condition that optimizes the function."*

E.g., in the first problem, the imposed boundary condition is the shape of the room; the free boundary condition is the placement and angle of the air conditioner, and the function is the time until the room reaches thermal equilibrium.

Due to the ten weeks time limit imposed by the `AI Builders X ESCK` program. We've decided to simplify various parts of the problem to make it more fathomable.

1.2 Problem statement and overview of solution

The problem that we've selected to tackle is the second problem due to phase homogeneity: "given a room, what's the best place to place an air conditioner, and what direction must it face." Due to complexity in three-dimensions, we've decided to simplify the problem to a room with boundaries in two-dimensions, and only allow the air conditioner to exist on a line around the border of the room.

The variables that are used in this problem is as follows:

1. **The function needed for optimization**—the time until equilibrium
2. **Free boundary condition**—placement of the air conditioner, represented as a velocity boundary condition
3. **Imposed boundary condition**—shape of the room

It's then solved as follows:

1. Build a fluid simulator with wall and velocity boundary condition,
2. Input the shape of the room, and the strength of the air conditioner,
3. Find the optimal air conditioner using gradient descent.

Originally, we planned to use the `OpenFOAM` simulator, as it's commonly used by researchers in computational fluid dynamics. However, the learning curve is too steep for just ten weeks. There's no clean way to connect the data from `OpenFOAM` into `Python` for post-processing. Most importantly, there aren't many great resources out there. So, we've decided to build our own simulation and optimization algorithm from scratch using one of the most accessible methods to do fluid simulation: the Lattice-Boltzmann method.

1.3 Overview of the Lattice-Boltzmann method

The Lattice-Boltzmann method is a fluid simulation method that doesn't require discretization of the Navier-Stokes equation. Instead, it models fluids as a collection of particles in a lattice filled with cells. In each step of the simulation, the particle moves from its own cell to its adjacent cells. Then, it interacts inside the cell through self-collisions. This cell-interpretation allow the derivation of the macroscopic fluid properties, e.g., density and velocity, to be derived from the particle distributions in each lattice directly. The process includes

1. **Streaming**—particles move into adjacent cells

2. **Collisions**—the densities in each cell is adjusted towards equilibrium inside the cell.

This method is very viable for parallel computing, making it very ideal for implementation in `NumPy`. However, it is numerically unstable for high-speed fluid flows near or above the speed of sound. Since we're not dealing with particles moving that fast, we should be fine.

Even though the Lattice-Boltzmann method is stable for the most part, it still has some numerical instabilities around boundary conditions especially anything to do with circles. These will become a problem in gradient descent, in which we have to implement an algorithm to work around these instabilities.

1.3.1 Representation

Coordinate convention Since `NumPy` indexes the y-axis (vertically) before the x-axis (horizontally), all pairs of coordinates from now on is to be read as (y, x), not (x, y)

A fluid simulation with resolution $N \times M$ illustrated in fig. 1.1, is represented as a rectangular lattice with $N \times M$ cells. Each cell in the lattice contains nine cell-invariant unit vectors, $\mathbf{e}_0$ to $\mathbf{e}_8$, which represents the eight possible direction that the fluid can travel in. The value for these vectors, respective to the Cartesian representation is given in table 1.1

Unit vector	Representation
$\mathbf{e}_0$	$\mathbf{0}$
$\mathbf{e}_1$	$\hat{\mathbf{x}}$
$\mathbf{e}_2$	$\hat{\mathbf{y}}$
$\mathbf{e}_3$	$-\hat{\mathbf{x}}$
$\mathbf{e}_4$	$-\hat{\mathbf{y}}$
$\mathbf{e}_5$	$\hat{\mathbf{x}} + \hat{\mathbf{y}}$
$\mathbf{e}_6$	$-\hat{\mathbf{x}} + \hat{\mathbf{y}}$
$\mathbf{e}_7$	$-\hat{\mathbf{x}} - \hat{\mathbf{y}}$
$\mathbf{e}_8$	$\hat{\mathbf{x}} - \hat{\mathbf{y}}$

TABLE 1.1 | UNIT VECTORS USED IN A CELL OF FLUID

From the set of vectors $\mathbf{e}_0, \ldots, \mathbf{e}_8$ inside each cell, one respectively assign another set of vectors $\mathbf{f}_0, \ldots, \mathbf{f}_8$. These vectors are scaled version of

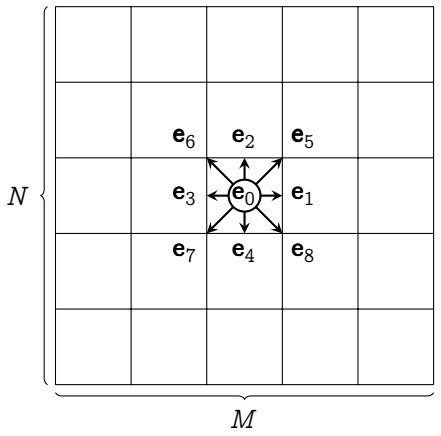


FIG. 1.1 | LATTICE OF FLUID

the unit vectors, i.e.,

$$\mathbf{f}_i = f_i \mathbf{e}_i, \tag{1.1}$$

where the scalar f_n represents the amount of fluid that's moving in the direction $\mathbf{e}_n$. From this representation alone, the density, momentum, and speed of the fluid at a certain point (n, m) can be found. The density of fluid at the cell (n, m), $\rho(n, m)$, is the sum from of all f_i's inside the cell:

$$\rho(n, m) \equiv \sum_i f_i(n, m). \tag{1.2}$$

The momentum density, $\mathbf{U}(n, m)$, traditionally given by the product between velocity and mass, can be calculated as the sum of product between f_n and their respective unit vectors:

$$\mathbf{U}(n, m) \equiv \sum_n f_n \mathbf{e}_n. \tag{1.3}$$

The velocity density at a certain cell is just the ratio between the momentum density and the fluid density:

$$\mathbf{u}(n, m) \equiv \frac{\mathbf{U}(n, m)}{\rho(n, m)} = \frac{\sum_n f_n \mathbf{e}_n}{\rho}. \tag{1.4}$$

1.3.2 Self-collision step

The self-collision step represents the relaxation of fluid that happens inside a cell. In each of the fluid vectors $\mathbf{f}_0, \ldots, \mathbf{f}_8$, a corresponding equilibrium vector is assigned by

$$\mathbf{E}_i(n,m) = w_i \rho \left(1 + 3\mathbf{e}_i \cdot \mathbf{u}(n,m) + \frac{9}{2}\big(\mathbf{e}_i \cdot \mathbf{u}(n,m)\big)^2 - \frac{3}{2}|\mathbf{u}(n,m)|^2\right)\mathbf{e}_i \tag{1.5}$$

where w_i is a weighting factor:

$$w_i = \begin{cases} \frac{4}{9} & \text{if } i = 0, \\ \frac{1}{9} & \text{if } i = 1,2,3,4, \\ \frac{1}{36} & \text{if } i = 5,6,7,8. \end{cases} \tag{1.6}$$

The corresponding equilibrium scalar $E_i(n,m)$, is given by the relation

$$\mathbf{E}_i(n,m) = E_i(n,m)\mathbf{e}_i. \tag{1.7}$$

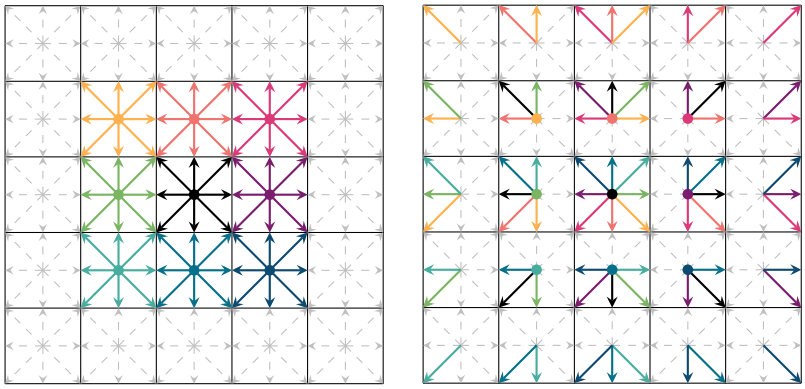
However, a fluid cannot possibly reach its own equilibrium in just one step; therefore, the Lattice-Boltzmann adjusts the fluid vector to approach the equilibrium vector. This behavior is captured by the relaxation time τ. For the set of fluid vector positioned at the cell (n,m) at time t, $f_i(n,m;t)$, the fluid vector at the next time step, $t + \Delta t$ is given by

$$f_i(n,m;t+\Delta t) = f_i(n,m;t) + \frac{1}{\tau}\big(E_i(n,m;t) - f_i(n,m;t)\big), \tag{1.8}$$

and that

$$\mathbf{f}_i(n,m;t+\Delta t) = f_i(n,m;t+\Delta t)\mathbf{e}_i. \tag{1.9}$$

The relaxation value that's used throughout this project is $\tau = 0.8070$, which is said to be the most numerically stable [2].



(A) BEFORE (B) AFTER

FIG. 1.2 | FLUID VECTORS BEFORE AND AFTER THE STREAMING STEP HIGHLIGHTED IN COLOR. GRAY VECTORS ARE NOT CONSIDERED.

1.3.3 Streaming step

Using the vector that's adjusted to the equilibrium from the self-collision step, that vector is streamed to the adjacent cells, given by

$$\mathbf{f}_i\left(n + (\hat{\mathbf{y}} \cdot \mathbf{e}_i), m + (\hat{\mathbf{x}} \cdot \mathbf{e}_i)\right) = \mathbf{f}_i(n, m; t + \Delta t). \tag{1.10}$$

Basically, this equation moves the fluid from one cell to the other as illustrated in fig. 1.2

1.3.4 Boundary conditions

There are two boundaries condition that needs to be implemented in this problem: wall and velocity. Since we want this to be a complete framework for two-dimensional fluid simulation, we also implemented the density boundary condition for completeness' sake.

Directional density boundary condition This boundary condition can be achieved by explicitly setting the value of f_0 to f_8 after a complete simulation step.

Wall boundary condition This boundary condition is sometimes referred off as the bounce-back boundary condition. If the fluid from an adjacent cell is streamed into a wall located at (n, m), the wall simply reflects the fluid vector back:

$$f_j\big(n - (\mathrm{e}_i \cdot \hat{\mathbf{y}}), m - (\mathrm{e}_i \cdot \hat{\mathbf{x}})\big) = f_i(n, m) \tag{1.11}$$

where

$$j = \begin{cases} i + 2 & \text{if } i = 1, 2, 5, 6, \\ i - 2 & \text{if } i = 3, 4, 7, 8. \end{cases} \tag{1.12}$$

Since the fluid cannot possibly stream into the center of the wall, j doesn't have to be defined at $i = 0$. [1]

Wall-velocity boundary condition Given a wall that's located at position (n, m), and an exposed fluid cell located at position $\big(n + (\mathbf{e}_a \cdot \hat{\mathbf{y}}), m + (\mathbf{e}_b \cdot \hat{\mathbf{x}})\big)$, the velocity boundary condition can be defined by two variables: velocity along $\mathbf{e}_a$, and along its clockwise perpendicular, $\mathbf{e}_b$ where

$$b = \begin{cases} a + 3 & \text{if } a = 1, \\ a - 1 & \text{if } a + 2, 3, 4. \end{cases} \tag{1.13}$$

Here, we define the other directions that are relative to direction a:

$$\alpha = \begin{cases} a + 2 & \text{if } a = 1, 2, \\ a - 2 & \text{if } a = 3, 4, \end{cases} \tag{1.14}$$

$$\beta = \begin{cases} a + 1 & \text{if } a = 1, 2, 3, \\ a - 3 & \text{if } a = 4, \end{cases} \tag{1.15}$$

$$A = \begin{cases} a + 7 & \text{if } a = 1, \\ a + 3 & \text{if } a = 2, 3, 4, \end{cases} \tag{1.16}$$

$$B = \begin{cases} a + 6 & \text{if } a = 1, 2, \\ a + 2 & \text{if } a = 3, 4, \end{cases} \tag{1.17}$$

$$C = \begin{cases} a + 5 & \text{if } a = 1, 2, 3, \\ a - 1 & \text{if } a = 4, \end{cases} \tag{1.18}$$

$$D = a + 4. \tag{1.19}$$

These directions live on a grid relative to direction a as illustrated in fig. 1.3.

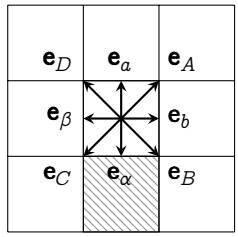


FIG. 1.3 | DIRECTIONS RELATIVE TO THE CONVENTION GIVEN BY THE WALL-VELOCITY BOUNDARY CONDITION. THE SHADED REGION IS THE WALL, AND THE CELL WITH VECTOR ARROWS IS THE TARGET CELL IN WHICH WALL-VELOCITY BOUNDARY CONDITION IS APPLIED.

a	b	α	β	A	B	C	D
1	4	3	2	8	7	6	5
2	1	4	3	5	8	7	6
3	2	1	4	6	5	8	7
4	3	2	1	7	6	5	8

TABLE 1.2 | DIRECTIONS RELATIVE TO THE CONVENTION GIVEN BY THE WALL-VELOCITY BOUNDARY CONDITION.

Given that

$$\mathbf{u}_a\big(n + (\mathbf{e}_a \cdot \hat{\mathbf{y}}), m + (\mathbf{e}_b \cdot \hat{\mathbf{x}})\big) \quad \text{and,} \quad \mathbf{u}_b\big(n + (\mathbf{e}_a \cdot \hat{\mathbf{y}}), m + (\mathbf{e}_b \cdot \hat{\mathbf{x}})\big) \tag{1.20}$$

is fixed by the boundary condition, the surrounding velocities in the cell $\big(n + (\mathbf{e}_a \cdot \hat{\mathbf{y}}), m + (\mathbf{e}_b \cdot \hat{\mathbf{x}})\big)$ can be updated as follows: [3]

$$\mathbf{f}_a = \mathbf{f}_\alpha + \frac{2}{3}\rho\big(n + (\mathbf{e}_a \cdot \hat{\mathbf{y}}), m + (\mathbf{e}_b \cdot \hat{\mathbf{x}})\big)\mathbf{f}_a, \tag{1.21}$$

$$\mathbf{f}_A = \mathbf{f}_C - \frac{1}{2}(\mathbf{f}_b - \mathbf{f}_\beta) + \rho\big(n + (\mathbf{e}_a \cdot \hat{\mathbf{y}}), m + (\mathbf{e}_b \cdot \hat{\mathbf{x}})\big)\Big(\frac{\mathbf{u}_b}{6} + \frac{\mathbf{u}_a}{6}\Big), \tag{1.22}$$

$$\mathbf{f}_D = \mathbf{f}_B - \frac{1}{2}(\mathbf{f}_b - \mathbf{f}_\beta) + \rho\big(n + (\mathbf{e}_a \cdot \hat{\mathbf{y}}), m + (\mathbf{e}_b \cdot \hat{\mathbf{x}})\big)\Big(-\frac{\mathbf{u}_b}{6} + \frac{\mathbf{u}_a}{6}\Big). \tag{1.23}$$

For the rest of the directions, use the wall boundary condition (bounce-back) to calculate the fluids vector.

CHAPTER 2

Model building

The implementation of the model in `Python` is different from the theoretical model due to some `NumPy` functions.

2.1 Components of simulation

2.2 Equilibrium step

2.3 Colliding step

2.4 Boundary conditions

2.5 Equilibrium detection

CHAPTER 3

Optimization algorithm

3.1 Implementation

Bibliography

[1]V. Adams, *Lattice-boltzmann in python, c, and verilog*, https://vanhunteradams.com/DE1/Lattice_Boltzmann/Lattice_Boltzmann.html (visited on 11/15/2024).

[2]Zhao, "Optimal relaxation collisions for lattice boltzmann methods", Computers & Mathematics with Applications **65**, 172–185 (2013).

[3]Q. Zou and X. He, "On pressure and velocity boundary conditions for the lattice boltzmann bgk model", Physics of Fluids **9**, 1591–1598 (1997).