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

Puripat Thumbanthu, Kunakorn Chaiyara

Started: November 11, 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 density boundary condition
3. **Imposed boundary condition**—shape of the room

It's then solved as follows:

1. Build a fluid simulator with wall and density 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 $\mathbf{e}_0, \dots, \mathbf{e}_8$ inside each cell, one respectively assign another set of vectors $\mathbf{f}_0, \dots, \mathbf{f}_8$. These vectors are scaled version of

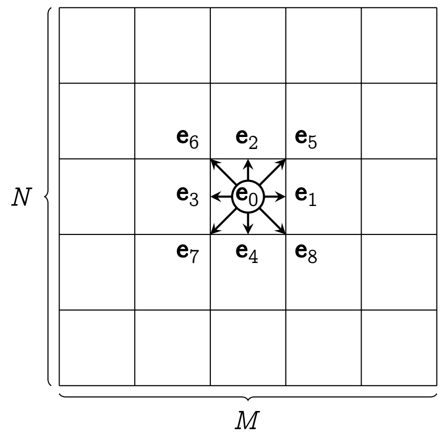


FIG. 1.1 | LATTICE OF FLUID

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 $\mathbf{f}_0, \dots, \mathbf{f}_8$, 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 that's given by the reduction of Boltzmann's distribution:

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

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

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

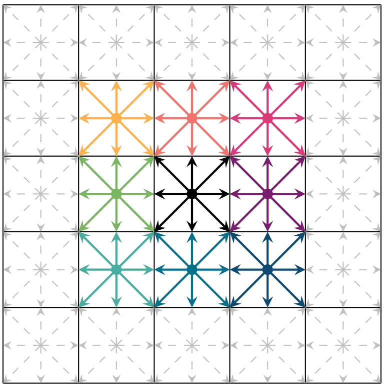
However, a fluid cannot possibly reach its own equilibrium in just one step; therefore, the Lattice-Boltzmann adjusts the fluid vector to approach the equilibrium vector. This behavior is captured by the relaxation time τ. For the set of fluid vector positioned at the cell (n,m) at time t, $f_i(n,m;t)$,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

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

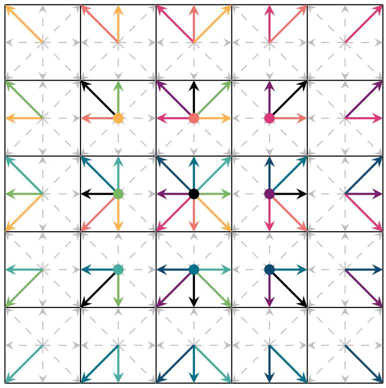
and that

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

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 density. 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.

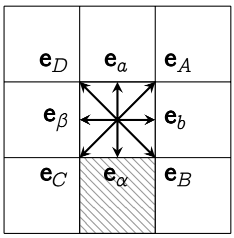


Fig. 1.3 | Directions relative to the convention given by the wall-velocity boundary condition. The shaded region is the wall, and the cell with vector arrows is the target cell in which wall-velocity boundary condition is applied.

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

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

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]

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

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

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

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. This chapter serves as an overview for self-implementing the model. The codes in this chapter are not the actual code used in the `GitHub` repository. It's liberated from the object-oriented paradigms for ease of understanding. All of these will be pieced together in chapter 3

The packages that are used throughout the project is imported as follows:

```
import numpy as np
import matplotlib.pyplot as plt
import itertools as itr
from scipy.ndimage import convolve
import copy
import random
import math
```

And, some useful arrays that are used throughout the project:

```
unitVect = np.array(
    [[0, 0], [1, 0], [0, 1], [-1, 0], [0, -1], [1, 1], [-1, 1], [-1, -1], [1,
    ↪  -1]]
)
unitX = np.array([0, 1, 0, -1, 0, 1, -1, -1, 1])
unitY = np.array([0, 0, 1, 0, -1, 1, 1, -1, -1])
```

The `unitVect` array represents the vector $\mathbf{e}_0, \ldots, \mathbf{e}_8$. `unitX` and `unitY` array represents the dot product of these vectors to $\hat{\mathbf{x}}$ and $\hat{\mathbf{y}}$ respectively.

2.1 Preliminary functions

2.1.1 Representation of physical quantities

The whole lattice is represented as a `NumPy` array with dimensions $(N, M, 9)$. The first axis represents the y index, second represents the x index, and the third one with nine elements represent the fluid vectors $\mathbf{f}_0, \ldots, \mathbf{f}_8$. Since these fluid vectors actually represents the amount of fluid that's travelling inside a cell, the vectors cannot be zero. Thus, the array is set to be all ones even when the fluid at rest. One can simply initialize the array as follows:

```
yResolution = 24 # Configurable
xResolution = 36 # Configurable
fluid = np.ones(yResolution, xResolution, 9)
# The fluid array is to be modified according to the desired initial condition
initCondition = np.copy(fluid)
```

The array `initCondition` serves as a reference for future plotting.

For simplicity, we also define two arrays that are used throughout: `yIndex` and `xIndex` which is an array filled with numbers from 0 to $y-1$, and 0 to $x-1$ respectively.

```
yIndex = np.arange(yResolution)
xIndex = np.arange(xResolution)
```

In the simulation step that is documented later in section 2.2, there must be a function that updates the density, momentum density, and velocity density of the fluid every time the simulation runs. First, we initialize the arrays that contain the density, momentum density, and the velocity density of the fluid according to eqs. (1.2) to (1.4):

```
density = np.sum(fluid, axis=2)
momentumY = np.sum(fluid * unitY, axis=2)
momentumX = np.sum(fluid * unitX, axis=2)
speedY = momentumY / density
speedX = momentumX / density
speedY = np.nan_to_num(speedY, posinf=0, neginf=0, nan=0)
speedX = np.nan_to_num(speedX, posinf=0, neginf=0, nan=0)
```

These arrays are then updated using the following functions:

```
def updateDensity():
    density = np.sum(fluid, axis=2)

def updateMomentum():
    momentumY = np.sum(fluid * unitY, axis=2)
    momentumX = np.sum(fluid * unitX, axis=2)
```

```python

def updateSpeed():
    updateDensity()
    updateMomentum()

    speedY = momentumY / density
    speedX = momentumX / density
    speedY = np.nan_to_num(speedY, posinf=0, neginf=0, nan=0)
    speedX = np.nan_to_num(speedX, posinf=0, neginf=0, nan=0)
```

Since `updateSpeed` calls both `updateDensity` and `updateMomentum`, one doesn't have to call `updateDensity` and `updateMomentum` when the function `updateSpeed` is already called.

2.1.2 Wall boundary conditions

The wall boundaries condition is stored as another array with dimensions (N, M) filled with boolean elements. If a position (n, m) is `True`, then it is not a wall, else, it's a wall. This array can be used to easily impose the wall boundary condition on the `fluid` array. I.e., every point where there's a wall, there must be zero fluid; thus, the fluid vectors at those points shall be zero.

```python
boundary = np.full((yResolution, xResolution)) # Can be edited to be any shape
↪  desired.
fluid[boundary, :] = 0
```

There are two types of wall that's implemented in this framework: circular, border, and rectangular; with their own functions. The cylindrical wall function takes in the boundary array that's to be modified (`boundary`), the cylinder's center (`cylinderCenter`) as a tuple in the format (y, x), and the cylinder's radius (`cylinderRadius: float`) as a floating point number:

```python
def cylindricalWall(boundary, cylinderCenter: tuple, cylinderRadius: float):
    for yIndex, xIndex in itr.product(
        range(yResolution), range(xResolution)
    ):
        if math.dist(cylinderCenter, [yIndex, xIndex]) <= cylinderRadius:
            boundary[yIndex, xIndex] = True
```

The border wall function takes in the boundary array (`boundary`), and the thickness of the border (`thickness: int = 1`) as an integer:

```python
def borderWall(boundary, thickness: int = 1):
    boundary[0 : yResolution, -1 + thickness] = True
    boundary[0 : yResolution, xResolution - thickness] = True
    boundary[-1 + thickness, 0 : xResolution] = True
    boundary[yResolution - thickness, 0 : xResolution] = True
```

The rectangular wall function takes in the boundary array (`boundary`) and the position of the two corner points (`cornerCoord1: tuple, cornerCoord2: tuple`) as a tuple in the format (y, x).

```python
def filledStraightRectangularWall(
    boundary,
    cornerCoord1: tuple,
    cornerCoord2: tuple
):
```

```
    maxY = max(cornerCoord1[0], cornerCoord2[0])
    minY = min(cornerCoord1[0], cornerCoord2[0])
    maxX = max(cornerCoord1[1], cornerCoord2[1])
    minX = min(cornerCoord1[1], cornerCoord2[1])

    for yIndex, xIndex in itr.product(
        range(yResolution), range(yResolution)
    ):
        if (
            (xIndex <= maxX)
            and (xIndex >= minX)
            and (yIndex <= maxY)
            and (yIndex >= minY)
        ):
            boundary[yIndex, xIndex] = True
```

These functions directly modifies the `boundary` array. They must be called before imposing the wall boundaries to the `fluid` array.

In some cases, it's more desirable to use the indices of the boundaries instead. We also write another function that's used to generate the indices of the boundaries. This function takes in the boundary array (`boundary`), and outputs two arrays: `boundaryIndex`, which is a list containing the indices of walls, and `invertedBoundaryIndex`, which is a list that contains the indices of fluids (invert of walls).

```
def generateIndex(boundary):
    boundaryIndex = []
    invertedBoundaryIndex = []
    for i, j in itr.product(
```

```
        range(yResolution), range(xResolution)
    ):
        if boundary[i, j] != False:
            boundaryIndex.append((i, j))
        else:
            invertedBoundaryIndex.append((i, j))
    return boundaryIndex, invertedBoundaryIndex
```

Since the end goal of this project is to simulate air conditioner placements, we also have to know the possible indices that the air conditioner can end up at. Therefore, we build a function `generateACPos` that can do so:

```
def generateACDirections(boundary):
    possibleACPos = []
    for shiftIndex, axisIndex in itr.product([-1, 1], [1, 0]):
        shiftedBoundary = np.roll(boundary, shift=shiftIndex, axis=axisIndex)
        possibleACPos = np.logical_or(
            possibleACPos,
            np.logical_not(boundary) & shiftedBoundary
        )
    return possibleACPos
```

This function takes in a boundary (`boundary`), then return a list of possible air conditioner positions (`possibleACPos`). It works by shifting the array `boundary` in the four cardinal directions ($i = 1, 2, 3, 4$) using the `np.roll` function, then comparing the shifted array to the original array. If a point (n, m) in the original array isn't a wall, but the point (n, m) on the shifted array along direction i isn't a wall, then the point (n, m) can hold an air conditioner that faces the direction i. All the possible points from all the shifted directions are combined using an or gate to obtain an array that contains all the point that can hold an air conditioner (`possibleACPos`).

The last function of the wall boundary condition is a function that turns the two-dimensional contour of the possible air conditioner position into a single continuous line called `indexPossibleACPos`. This has to be done because the gradient descent algorithm that is used to find the optimal air conditioner placement has to take in a continuous variable as its parameters. This can be done by a breadth first search along the line.

```
def indexPossibleACPos(possibleACPos, clear: bool = False):
    testArray = copy.deepcopy(possibleACPos)
    currentIndex = tuple()
    for yIndex, xIndex in itr.product(
        range(yResolution), range(xResolution)
    ):
        if testArray[yIndex, xIndex]:
            currentIndex = (yIndex, xIndex)
            break

    while testArray[currentIndex]:
        for latticeIndex in [1, 2, 3, 4, 5, 6, 7, 8, 0]:
            nextIndex = addTuple(
                currentIndex,
                (
                    unitX[latticeIndex],
                    unitY[latticeIndex],
                ),
            )
            if testArray[nextIndex]:
```

```python
                possibleACIndex.append(nextIndex)
                testArray[currentIndex] = 0
                currentIndex = nextIndex
                break
            else:
                pass
```

2.1.3 Density boundary condition

This is the easiest boundary condition to impose. One can just set the fluid vectors directly. Although I want this chapter to be liberated from object-oriented programming paradigm, this one just can't. Therefore, I shall introduce a new simple class: the `DensityBoundary` class:

```python
class DensityBoundary:
    def __init__(self, y: int, x: int, magnitude: float, direction: int):
        self.y = y
        self.x = x
        self.magnitude = magnitude
        self.direction = direction
```

This class contains the position of the density boundary condition (`y` and `x`), the magnitude of density, and the direction that the density is imposed. The density boundary condition is imposed as follows:

```python
velocityBoundaries = []
def imposeDensityBoundaryCondition(boundary, velocityBoundaries):
    for velocityBoundary in velocityBoundaries:
        fluid[
```

```
            velocityBoundary.y, velocityBoundary.x, velocityBoundary.direction
        ] = velocityBoundary.magnitude
    updateSpeed()
```

2.1.4 Wall-velocity boundary condition

The wall-velocity boundary condition is also implemented as a class which is initialized with the (y, x) position of the boundary condition and the velocity along direction $\mathbf{e}_a, \mathbf{e}_b$.

```
class VelocityBoundary:
    indices = [[1, 8, 5], [2, 5, 6], [3, 6, 7], [4, 7, 8]]

    def __init__(self, y: int, x: int, ux, uy, direction: int):
        self.y = y
        self.x = x
        self.uy = uy
        self.ux = ux
        self.direction = direction

        # For calculating the ua and ub
        if direction in [3, 4]:
            reflectIndex = direction - 2
        else:
            reflectIndex = direction + 2

        self.mainVelocity = ux if direction in [1, 3] else uy
        self.minorVelocity = uy if direction in [1, 3] else ux
        self.setIndices = VelocityBoundary.indices[direction - 1]
```

```
        self.getIndices = VelocityBoundary.indices[reflectIndex - 1]
```

All the pressure boundaries point are then stored as an object of the class `VelocityBoundaries` in a list called `velocityBoundaries`. We then iterate over the list to update the fluid simulation grid according to eqs. (1.21) to (1.23).

```
velocityBoundaries = [] # A list of pressure boundaries point
def imposeVelocityBoundaryCondition(fluid):
    for velocityBoundary in velocityBoundaries:
        for latticeIndex in range(9):
            fluid[velocityBoundary.y, pressureBoundary.x, latticeIndex] = 0
        densityAtIndex = density[velocityBoundary.y, pressureBoundary.x]
        fluid[
            velocityBoundary.y, pressureBoundary.x,
            ↪ pressureBoundary.setIndices[0]
        ] = fluid[
            velocityBoundary.y, pressureBoundary.x,
            ↪ pressureBoundary.getIndices[0]
        ] + (
            2 / 3
        ) * (
            velocityBoundary.mainVelocity
        )
        fluid[
            velocityBoundary.y, velocityBoundary.x,
            ↪ velocityBoundary.setIndices[1]
        ] = (
            fluid[
                velocityBoundary.y,
```

```
                velocityBoundary.x,
                velocityBoundary.getIndices[1],
            ]
            - (
                0.5
                * (
                    fluid[
                        velocityBoundary.y,
                        velocityBoundary.x,
                        (
                            4
                            if velocityBoundary.direction - 1 == 0
                            else velocityBoundary.direction - 1
                        ),
                    ]
                    - fluid[
                        velocityBoundary.y,
                        velocityBoundary.x,
                        (
                            1
                            if velocityBoundary.direction + 1 == 5
                            else velocityBoundary.direction + 1
                        ),
                    ]
                )
            )
            + (0.5 * densityAtIndex * velocityBoundary.minorVelocity)
            + (1 / 6 * densityAtIndex * velocityBoundary.mainVelocity)
        )
        fluid[
```

```
            velocityBoundary.y, velocityBoundary.x, velocityBoundary.setIndices[2]
        ] = (
            fluid[
                velocityBoundary.y,
                velocityBoundary.x,
                velocityBoundary.getIndices[2],
            ]
            + (
                0.5
                * (
                    self.fluid[
                        velocityBoundary.y,
                        velocityBoundary.x,
                        (
                            4
                            if velocityBoundary.direction - 1 == 0
                            else velocityBoundary.direction - 1
                        ),
                    ]
                    - self.fluid[
                        velocityBoundary.y,
                        velocityBoundary.x,
                        (
                            1
                            if velocityBoundary.direction + 1 == 5
                            else velocityBoundary.direction + 1
                        ),
                    ]
                )
```

```
            )
            - (0.5 * densityAtIndex * velocityBoundary.minorVelocity)
            + (1 / 6 * densityAtIndex * velocityBoundary.mainVelocity)
        )
```

2.2 Simulation functions

Here, it's time to write the actual code for simulating the fluid. There will be three main functions that are used: `streamFluid`, `bounceBackFluid`, and `collideFluid`. All of which follows the main steps of the Lattice-Boltzmann method: streaming, self-collision, and wall boundary.

This idea of implementing the simulation is actually an amalgamation of various ones. The article by Adams [1] and **schroeder-2012** [**schroeder-2012**] gave us a very comprehensive overview of the Lattice-Boltzmann method with boundaries condition, and also provided us with the intuition for creating our own ways of implementing. The inspiration for using the roll function is from **matias-2022**'s video on Lattice-Boltzmann simulator [**matias-2022**]. However, that video is quite old and uses a rather strange technique of implementing the boundary conditions, which leads to many numerical instabilities. So, most of the boundary conditions code for us to head scratch. It did work at the end, and here is how we did it.

Firstly, the fluid is streamed by using `np.roll` along the various axes.

2.3 Main loop

CHAPTER 3

Model structure

CHAPTER 4

Optimization algorithm

4.1 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).