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

Puripat Thumbanthu, Kunakorn Chaiyara

Started: November 11, 2024; Revision: November 19, 2024

Abstract

We developed a two-dimensional fluid simulation framework using the Lattice-Boltzmann method for optimizing airflow in a room. Our main goal was to find the best placement and direction for an air conditioner to achieve thermal balance efficiently. The three-dimensional airflow problem is simplified into a two-dimensional model, making it easier to analyze while still capturing key dynamics of fluids. We focused on boundary conditions that represent the room's shape and the air conditioner's position. Using convolutional neural networks, we've built a model which can predict the best position for an air conditioner with an accuracy of around 2 pixels. We also discussed the limitations of our model, such as challenges in simulating complex airflow patterns and accurately representing air conditioner behavior at the end.

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

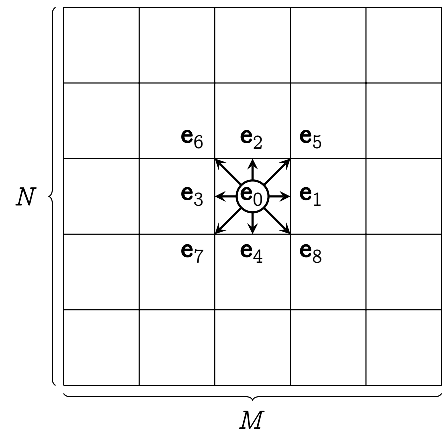


FIG. 1.1 | LATTICE OF FLUID

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

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

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

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

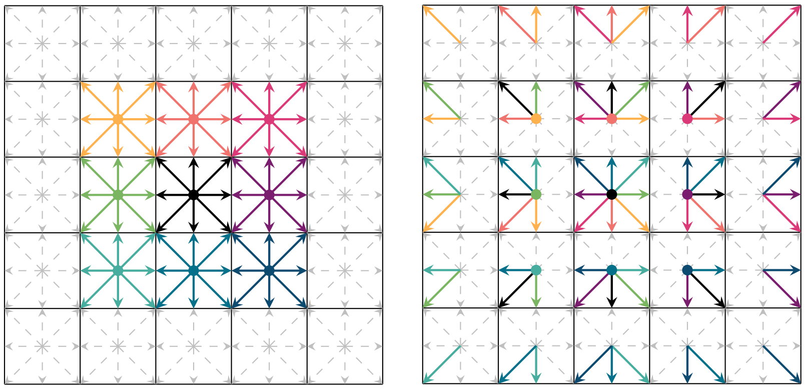
However, a fluid cannot possibly reach its own equilibrium in just one step; therefore, the Lattice-Boltzmann adjusts the fluid vector to approach the equilibrium vector. This behavior is captured by the relaxation time τ. For the set of fluid vector positioned at the cell (n,m) at time t, $f_i(n,m;t)$,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

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

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 [9].



(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:

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

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.

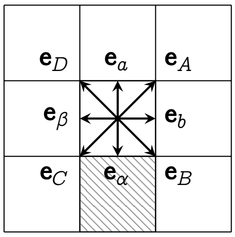


FIG. 1.3 | DIRECTIONS RELATIVE TO THE CONVENTION GIVEN BY THE WALL-VELOCITY BOUNDARY CONDITION. THE SHADED REGION IS THE WALL, AND THE CELL WITH VECTOR ARROWS IS THE TARGET CELL IN WHICH WALL-VELOCITY BOUNDARY CONDITION IS APPLIED.

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: [10]

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

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

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

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

CHAPTER 2

Building the simulation framework

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

This idea of implementing the simulation is actually an amalgamation of various ones. The article by Adams [1] and Schroeder [8] 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 Ortiz's video on Lattice-Boltzmann simulator [5]. However, that video is quite old and uses a rather strange technique of implementing the boundary conditions, which leads to many numerical instabilities.

Most of the time that's spent on this project is to make the boundary condition work. At the end, it did work, with the by product of sweat and tears. We don't want anyone to suffer through them with us. So, here is how we did it.

2.1 Preliminary quantities

The packages that are used throughout the project are 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, here are the invariant 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])
```

```python
unitY = np.array([0, 0, 1, 0, -1, 1, 1, -1, -1])
weight = np.array([4/9, 1/9, 1/9, 1/9, 1/9, 1/36, 1/36, 1/36, 1/36])
```

The `unitVect` array represents the set of vectors $\mathbf{e}_i$ where $i = 1, 2, 3, \dots, 8$. `unitY` and `unitX` represents $\mathbf{e}_i \cdot \hat{\mathbf{y}}$ and $\mathbf{e}_i \cdot \hat{\mathbf{x}}$ respectively. Lastly, the `weight` array represents the weight that's used to calculate the fluid equilibrium ($\mathbf{E}_i$) in eqs. (1.5), (1.6) and (1.8).

2.2 Preliminary functions

2.2.1 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 $\mathbf{f}_0, \dots, \mathbf{f}_8$. 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:

```python
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 = copy.deepcopy(fluid)
```

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

For simplicity, we also define two arrays that are used throughout: `yIndex` and `xIndex`. They are arrays that are 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 2.3, 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)
```

The function `np.nan_to_num` is used to deal with infinities that might occur in the velocity calculation step. At the wall boundary condition where there's no fluid there; hence, zero density. Since the velocity is the ratio between momentum and fluid density, the velocity at the wall diverges to infinity, which it really shouldn't. So, we convert those infinities to zero by using the `np.nan_to_num`.

These arrays are then updated using the following functions:

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

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

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)
```

They are literally the calculation codes, but converted into a callable function. Since `updateSpeed` calls both `updateDensity` and `updateMomentum`, one doesn't have to call `updateDensity` and `updateMomentum` when the function `updateSpeed` is already called.

2.2.2 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.

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

There are four types of wall that's implemented in this framework: circular, border, and rectangular, and dot. Each of them can be created using their own functions.

The cylindrical wall function (`cylindricalWall`) takes in the boundary array (`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, and modify the boundary array such that there's a wall with cylinder with radius `cylinderRadius` centered at the point (y, x). The implementation is as follows:

```
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 (`borderWall`) takes in the boundary array (boundary), and the thickness of the border (thickness: int = 1) as an integer, then modifies the boundary array such that there is a border around the simulation. This function is called to create a box surrounding the fluid.

```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 is used to create a rectangular wall inside the simulation. It takes in the boundary array (boundary) and the position of the two corner points of the rectangle (cornerCoord1: tuple, cornerCoord2: tuple) as a tuple in the format (y, x). It's implemented as follows:

```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
```

Lastly, the dot wall (`dotWalls`) just takes in the boundary array (`boundary`) and some coordinates, then modifies the `boundary` array so that those coordinates are walls.

```python
def dotWalls(boundary, *args: tuple):
    for position in args:
        boundary[position[0], position[1]] = True
```

Since these four 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 favorable to use the indices of the boundaries directly. The `NumPy` array can be converted to a list of indices by the function `generateIndex`. This function takes in the boundary array (`boundary`), and outputs two arrays: `boundaryIndex` and `invertedBoundaryIndex`. The `boundaryIndex` list contains the indices of walls. Invert that list, and you get the `invertedBoundaryIndex`, which contains the indices of fluids. The function is implemented as follows:

```python
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:

```python
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 array (`boundary`) and return a list of possible air conditioner positions (`possibleACPos`). It works by shifting the `boundary` array along the four cardinal directions ($i = 1, 2, 3, 4$) by 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 an air conditioner that faces direction i can be put at point (n, m). All the possible points from all directions are combined using `np.logical_or` to obtain an array that contains all the point that can hold an air conditioner (`possibleACPos`).

Another requirement for gradient descent is that the parameters must be continuous; therefore, one must be able to stretch the two-dimensional contour of the possible air conditioner positions into a line. The function `indexPossibleACPos` will do just that. It's implemented doing breadth first search along a line until it comes back, or that the line terminates.

```
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
```

```python

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

2.2.3 Density boundary condition

The interpretation of this boundary condition isn't really to fix a density at a certain point. Rather, it is to make a point spew out the liquids continuously; therefore, the fluid vectors can just be set explicitly after each simulation iteration. Although I want this chapter to be liberated from object-oriented programming paradigm, this one just can't. Therefore, I shall introduce a new class: the `DensityBoundary` class, which is implemented as follows:

```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()
```

2.2.4 Wall-velocity boundary condition

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

```python
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
        reflectIndex = (direction - 2) if (direction in [3, 4]) else
         ↪  (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 velocity 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).

```python
velocityBoundaries = [] # A list of pressure boundaries point
def imposeVelocityBoundaryCondition(fluid, velocityBoundaries):
    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)
        )
```

2.3 Simulation functions

There are three 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.

Streaming step Earlier, we calculated the value of $\mathbf{e}_i \cdot \hat{\mathbf{y}}$ and $\mathbf{e}_i \cdot \hat{\mathbf{x}}$, and stored it into the array `unitX` and `unitY`. These quantities are then used to shift the array in their respective directions, representing the streaming step.

```python
def streamFluid(fluid):
    for latticeIndex, shiftY, shiftX in zip(range(9), unitY, unitX):
        fluid[:, :, latticeIndex] = np.roll(
            fluid[:, :, latticeIndex], shiftY, axis=0
        )
        fluid[:, :, latticeIndex] = np.roll(
            fluid[:, :, latticeIndex], shiftX, axis=1
        )
```

Self-collision step The code follows from eqs. (1.5) and (1.8). It iterates through every fluid vectors in a lattice and update them accordingly.

```python
def collideFluid(fluid):
    fluidEquilibrium = np.zeros(fluid.shape)
    for latticeIndex, cy, cx, w in zip(
        range(9), unitY, unitX, weight,
    ):
        fluidEquilibrium[:, :, latticeIndex] = (
            density
            * w
            * (
                1
                + 3 * (cx * speedX + cy * speedY)
                + 9 * (cx * speedX + cy * speedY) ** 2 / 2
                - 3 * (speedX**2 + speedY**2) / 2
            )
        )
    fluid += (fluidEquilibrium - fluid) / relaxationTime
```

Imposing wall boundary condition We define a dictionary `reflectIndices` to convert from the index that's pointing into the wall to the index that points opposing the wall. The elements of the dictionary directly reflect the relation given by eq. (1.12). The part that updates the fluid directly reflects eq. (1.11).

```python
boundaryIndex, invertedBoundaryIndex = generateIndex(boundary)
reflectIndices = {0: 0, 1: 3, 2: 4, 3: 1, 4: 2, 5: 7, 6: 8, 7: 5, 8: 6}
def bounceBackFluid(fluid):
    for y, x in boundaryIndex:
        for latticeIndex in range(9):
            if fluid[y, x, latticeIndex] != 0:
                bounceIndexY = y - unitY[latticeIndex]
                bounceIndexX = x - unitX[latticeIndex]
                if (bounceIndexY ≥ 0 and bounceIndexY < yResolution)
                ↪  and (
                    bounceIndexX ≥ 0 and bounceIndexX < xResolution
                ):
                    fluid[
                        bounceIndexY,
                        bounceIndexX,
                        reflectIndices[latticeIndex],
                    ] = fluid[y, x, latticeIndex]
                    fluid[y, x, latticeIndex] = 0
    updateSpeed()
```

2.4 Simulation loop

We first define a counter that counts the amount of times that we ran the simulation: `step`. For further comparison, we deep-copy the state of the fluid before updating to the variable `lastStepFluid`. Then, the fluid vector is updated in order: stream, bounce back, self-collide, impose velocity, and impose density.

```python
step = 1
def stepSimulation(fluid):
    lastStepFluid = copy.deepcopy(fluid)
    streamFluid()
    bounceBackFluid()
    collideFluid()
    imposeVelocityBoundaryCondition()
    imposePressureBoundaryCondition()
    step += 1
```

Then, another function is implemented as a way to loop over the step function:

```python
def simulate(fluid, step: int = 1):
    [stepSimulation(fluid) for i in range(step)]
```

2.5 Metrics and baseline function

A more explicit form of the problem is

Find the air conditioner position that makes the system reaches the equilibrium the fastest.

So, it's necessary to have a function that will simulate the fluid until equilibrium. Here, the word *equilibrium* is defined as the point in which the difference in the sum of fluid vector between the last simulation step (lastStepFluid), and the current step (fluid), is smaller than a set threshold (equilibriumThreshold), which is normally set to be 0.5. The function that determines whether the system has reached equilibrium or not is called isEquilibrium. This function takes in a fluid state and its past state, then outputs a boolean. If it's true, then the system has reached equilibrium, otherwise, it hasn't.

```python
def isAtDensityEquilibirum(
    fluid, lastStepFluid, threshold: float = 0.5
):
    error = np.sum(np.abs(self.lastStepFluid - self.fluid))
    return not (error ≥ threshold)
```

The function to simulate until equilibrium also needs some care. Firstly, some initial condition where the simulation is numerically unstable, so the simulation has to be terminated before it uses up all the memory. This can be done by terminating the simulation when the sum of the fluid vectors reaches a certain threshold: the explodeThreshold. Secondly, some initial condition isn't numerically unstable, but doesn't reach an equilibrium either, so the amount of simulation passes limit (limit) must be set to control how many steps the simulation can take before forcing it to terminate. Altogether, the simulateUntilEquilibrium function is implemented as follows:

```python
def simulateUntilEquilibrium(
    fluid,
    limit: int = 5000,
    equilibriumThreshold: float = 0.5,
    explodeThreshold: float = 11 * xResolution * yResolution,
):
    step = 0
    isStable = True
    for _ in range(limit):
        stepSimulation()
        step += 1
        if np.sum(self.fluid) > explodeThreshold:
            isStable = False
            break
        if isAtDensityEquilibirum(equilibriumThreshold):
            break

    return step, isStable
```

2.6 Plotting the results

Plotting various aspects of the fluid simulation is quite straightforward with the framework that we've built. The density plot can be achieved directly by using the `imshow` function. [6]

```
plt.imshow(density, cmap="hot", interpolation="hermite")
```

where `cmap` is the color map, and `interpolation` is the color map smoothing algorithm. The full list of options for both of these can be found in Matplotlib's documentation: [2] (`cmap`), and [4] (`interpolation`).

The momentum density and the velocity density can be directly plotted along with the `quiver` function from Matplotlib. [7]

```python
fig, ax = plt.subplots()
heatmap = ax.imshow(
    densityPlot, cmap = "hot", interpolation = "hermite"
)
quivermap = ax.quiver(xIndex, yIndex, momentumX, -momentumY)
# Change momentumX, -momentumY to speedX, and speedY to plot the
↪  velocity instead of momentum.
fig.colorbar(heatmap)
plt.show()
```

The `momentumY` has to have a negative sign upfront because `matplotlib` libraries doesn't invert the y and x axes like the `NumPy` convention. Further modifications of the plot can be found in Matplotlib's quiver documentation: [7].

CHAPTER 3

Framework structure

As said, we shall enforce the object-oriented paradigm of `Python` by turning everything into an object one by one

3.1 Class `WallBoundary`

The `WallBoundary` class is used to store the position of walls, and is initialized with the following variables

- `yResolution: int`
- `xResolution: int`
- `invert: bool = False`

Along with these constants that are baked into the class

```
class WallBoundary:
    unitVect = np.array(
        [
```

```python
            [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])
    directions = [
        (1, -1), (1, 1), (1, 1), (-1, 1),
        (1, -1), (1, 1), (-1, 1), (-1, -1)
    ]
```

The `invert` variable is used in case the walls needed to be inverted. E.g., instead of generating a cylinder in the middle, we want the cylinder to enclose the fluid instead. If `invert` is true, then the wall boundaries is inverted.

The object variables are then evaluated as follows:

```python
self.yResolution = yResolution
self.xResolution = xResolution
self.invert = invert
self.boundary = np.full((yResolution, xResolution), invert)
self.invertedBoundary = np.invert(self.boundary)
self.boundaryIndex = []
self.invertedBoundaryIndex = []

self.possibleACPos = np.full((yResolution, xResolution), False)
self.possibleACIndex = []
self.possibleACDirections = None
```

The class contains the following function.

- `generateIndex(self)`
- `generateACPos(self)`
- `indexPossibleACPos(self)`
- `cylindricalWall(self, cylinderCenter: tuple, cylinderRadius: float)`
- `filledStraightRectangularWall(self, cornerCoord1: tuple, cornerCorod2: tuple)`
- `borderWall(self, thickness: int = 1)`
- `dotWalls(self, *args: tuple)`

The top three functions are used by the simulation to generate various indices for placing the air conditioner. The lower five are used to modify the boundaries directly. All the coordinates that are used in these class functions are tuples in the form (y, x). These functions must be called before initializing the simulation using the `Simulation` class, documented in section 3.4.

3.2 Class `DensityBoundary`

This class is used to model density boundary condition and is implemented as follows:

```
class VelocityBoundary:
    def __init__(
        self, y: int, x: int, magnitude: float, direction: int
    ):
        self.y = y
        self.x = x
        self.magnitude = magnitude
        self.direction = np.array(direction)
```

This class does not have any methods or functions.

If a simulation has multiple density boundary condition, they must be stored in a list and passed to the variable `densityBoundaries` of the `Simulation` class, documented in section 3.4.

3.3 Class `VelocityBoundary`

This class is used to model velocity boundary condition and is implemented as follows:

```
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
        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 = PressureBoundary.indices[direction - 1]
        self.getIndices = PressureBoundary.indices[reflectIndex - 1]
```

If a simulation has multiple velocity boundary condition, they must be stored in a list and passed to the variable `velocityBoundaries` of the `Simulation` class, documented in section 3.4.

3.4 Class `Simulation`

The class `Simulation` is used to initialize a simulation and store all the physical variables and boundaries condition of a simulation. It's initialized with the following variables:

- `yResolution: int`
- `xResolution: int`
- `initCondition: np.array`
- `wallBoundary: WallBoundary`
- `densityBoundaries: list = []`
- `velocityBoundaries: list = []`
- `relaxationTime: float = 0.8090`
- `initialStep: int = 0`

The following constants are baked into the class:

```python
class Simulation:
    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])
    weight = np.array(
        [
            4 / 9, 1 / 9, 1 / 9, 1 / 9, 1 / 9,
            1 / 36, 1 / 36, 1 / 36, 1 / 36
        ]
    )
    latticeSize = 9
    reflectIndices = {
        0: 0, 1: 3, 2: 4, 3: 1, 4: 2, 5: 7, 6: 8, 7: 5, 8: 6
    }
```

Then, the following class variables are calculated:

```python
self.yResolution = yResolution
self.xResolution = xResolution
self.yIndex = np.arange(yResolution)
self.xIndex = np.arange(xResolution)
self.initCondition = copy.deepcopy(initCondition)
self.fluid = copy.deepcopy(initCondition)
self.lastStepFluid = initCondition
self.relaxationTime = relaxationTime
self.wallBoundary = wallBoundary
self.wallBoundary.generateIndex()
self.fluid[self.wallBoundary.boundary, :] = 0
self.pressureBoundaries = pressureBoundaries
self.velocityBoundaries = velocityBoundaries
self.step = initialStep

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

The variable `initCondition` must be a `NumPy` array shaped `(yResolution, xResolution, 9)`. A wall boundary is required to initialize this class. If the simulation contains no walls, pass in an object of class `WallBoundary` (section 3.1) without calling any functions on it. Optionally, one might pass in a list of velocity boundary conditions and pressure boundary condition. A common way to initialize this class is.

```python
yResolution = 24  # Can be modified
xResolution = 36  # Can be modified
initCondition = np.ones((yResolution, xResolution,
↪  Simulation.latticeSize)) / 9
walls = WallBoundary(yResolution, xResolution)
# Call the WallBoundary class methods here to generate the desired
↪  walls.
walls.cylindricalWall((12, 10), 5) # Ex: Generate a cylinder at (12, 10)
↪  with radius 5
walls.borderWall() # Ex: Generate a border for the simulation

# Examples of density and velocity boundary conditions.
densityInlet = [DensityBoundary(12, 2, 1, 1)]
velocityInlet = [VelocityBoundary(12, 2, 1, 0, 1)]

# Initializing the simulation
simulation = Simulation(
    yResolution,
    xResolution,
    initCondition,
    walls,
    densityBoundaries = densityInlet,
    velocityBoundaries = velocityInlet
)
```

This class contains the following functions:

- `updateDensity(self)`
- `updateMomentum(self)`
- `updateSpeed(self)`
- `streamFluid(self)`
- `bounceBackFluid(self)`
- `collideFluid(self)`
- `imposeVelocityBoundaryCondition(self)`
- `imposePressureBoundaryCondition(self)`
- `stepSimulation(self)`
- `simulate(self, step: int = 1)`
- `isAtDensityEquilibrium(self, threshold: float = 0.5)`
- `simulateUntilEquilibrium(self, limit: int = 5000, equilibriumThreshold: float = 0.5, explodeThreshold: float = 400)`

CHAPTER 4

Optimization algorithm

This section is still under active development and is quite incomplete both theoretically and computationally due to the time limit that has been unexpectedly imposed. Somehow, the god of AI has our back. So, enjoy the chapter!

4.1 Overview

Originally, we planned to use the stochastic gradient descent algorithm straight on the possible air conditioner position. However, it falls short because most of the best points are all scattered throughout the rooms. The time until equilibrium of most half-way point of a straight wall is mostly similar, and is the optimal position. Meaning that there are many hills and valleys in our loss function; thus, gradient descent is basically out of the window.

It was depressing. It took us four weeks before realizing that both the original gradient descent and its best cousin, `Adam`, literally doesn't work because the room's geometry. When there's a long straight wall, the function is smooth, and the local minima usually sits at the center of the straight wall. At the end of these walls are usually sharp corners, and these corners creates a spike in the function, so gradient descent wouldn't work. We hypothesized that a better algorithm would probably be a detection of all straight walls, find the midpoint of each one, then compare. However, we're already reaching the time limit of AI Builders. So that is out of the window as well.

Recognizing that the visual geometry of the room is significant to optimizing the air conditioner placement, we settled for a convolutional neural network: a machine-learning method which can easily recognize the shape of a room. The overview of the plan is as follows:

1. Generate a data set that captures
   a) **Data**—The shape of the room in as `.png` files
   b) **Labels**—The best air conditioner position as a tuple (Y, X)
2. Import the data into a `tensorflow` compatible format, then do train-test split
3. Design the convolutional neural networks layer
4. Train the model using the square of the mean squared error loss function

Generation of best position The best position is determined via brute-forcing through every possible positions that has been indexed by the `indexPosibleACPos` function and measuring the fastest time until equilibrium. The point that makes the room reaches an equilibrium the fastest is said to be "the best position." However, this method has its own pitfalls.

In that, the air conditioner is supposed to distribute the air evenly, so a more proper indexing must be

> *The best air conditioner position that makes the difference of velocity of air along the boundaries the least.*

However, the algorithm to determine this is too complex to be done in the time limit imposed. It will be changed to a correct one after the `AI Builders X ESCK` is done.

Generation of best air conditioner position The *best air conditioner placement in a room* is generated by brute-forcing through all the possible positions of air conditioner that's indexed by the function **`indexPossibleACPos`**.

Reasons for the mean-squared loss function Since we want to predict the optimal position of the air conditioner in a room, we want the AI to minimize the distance between the prediction and the actual position. This feature is captured by the Pythagorean distance:

$$d = \sqrt{(y-y')^2 + (x-x')^2} \tag{4.1}$$

where (y, x) is the actual value and (y', x') is the predicted value. If we take the Pythagorean distance and square it, then throw in many data points and calculate the error between them, what we get is exactly the mean squared error.

$$\bar{d^2} = \frac{1}{n}\sum_{i=1}^{n}\left((y_i - y_i')^2 + (x_i - x_i')^2\right), \tag{4.2}$$

where n is the amount of data points that's used to validate the AI. Since the system that we are dealing with is continuous, and there are no classifications involved, the loss function directly reflects the accuracy of the model, and can be used as a metric for the model as well.

4.2 Data generation algorithm

The room generator works on a set of predetermined coordinates determined based on room size. A random amount of obstacles (Obstacles here are defined as areas that are inaccessible, whether that be furniture, columns, or wall curvature) are then placed randomly onto these positions with position harboring more than one obstacle. All the possible positions are stored in the `WallBoundary` class, and is initialized as follows:

```python
self.possiblePositions = [
    (int(self.yResolution / 3), self.xResolution - 1),
    (int(2 * self.yResolution / 3), self.xResolution - 1),
    (0, int(self.xResolution / 3)),
    (0, int(2 * self.xResolution / 3)),
    (int(self.yResolution / 3), 0),
    (int(2 * self.yResolution / 3), 0),
    (self.yResolution - 1, int(self.xResolution / 3)),
    (self.yResolution - 1, int(2 * self.xResolution / 3)),
    (0, self.xResolution - 1),
    (0, 0),
    (self.yResolution - 1, 0),
    (self.yResolution - 1, self.xResolution - 1),
]
```

There are two types of obstacles: rectangular ones and cylindrical ones. The type of obstacle placed is determined by a random float generator, picking circles and rectangles on a ratio of 4:6 respectively.

Then, we define a function `generateRoom` in the class `WallBoundary` as follows:

```python
def generateRoom(self):
    for i in random.sample(range(12), k=random.randint(1, 12)):
        wallPos = self.possiblePositions[i]
        if random.random() < 0.4:
            if random.random() < 0.1:
                maxSize = int(min(self.yResolution, self.xResolution) * 0.7)
                minSize = int(min(self.yResolution, self.xResolution) * 0.5)
            else:
                maxSize = int(min(self.yResolution, self.xResolution) * 0.3)
                minSize = int(min(self.yResolution, self.xResolution) * 0.2)
            sizeR = random.randint(minSize, maxSize)
            self.cylindricalWall(wallPos, sizeR)
        else:
            if random.random() < 0.2:
                maxSize = int(min(self.yResolution, self.xResolution) * 0.6)
                minSize = int(min(self.yResolution, self.xResolution) * 0.3)
            else:
                maxSize = int(min(self.yResolution, self.xResolution) * 0.4)
                minSize = int(min(self.yResolution, self.xResolution) * 0.2)

            sizeX = random.randint(minSize, maxSize)
            sizeY = random.randint(minSize, maxSize)
            rectPos = (
                wallPos[0] - ((sizeY * WallBoundary.directions[i]
                ↪  [0])/2),
                wallPos[1] - ((sizeX * WallBoundary.directions[i]
                ↪  [1])/2),
            )
            endPos = (
                wallPos[0] + ((sizeY * WallBoundary.directions[i]
                ↪  [0])/2),
                wallPos[1] + ((sizeX * WallBoundary.directions[i]
                ↪  [1])/2),
            )
            self.filledStraightRectangularWall(rectPos, endPos)
```

The cylindrical wall is added to the room contours using a function that takes in the center-point, and the radius. The size of the circle is picked in two steps. First, we see if we'll get a big circle or a small circle, at a chance of 40% and 60% respectively. This is due to the fact that, during testing, it was found that too many large circles often created small 1-pixel channels with little to no airflow. Increasing the size range of all circles didn't work, since those channels would still appear, and we would also have to increase the size of the rectangular walls to prevent creating pockets with no air-flow at all. Thus, it was decided that if a circle was going to be big, it was going to be very big, but also very rare, making it take up as much space as possible to prevent space between it and other obstacles.

The rectangular ones operate on the same principles as the cylindrical ones, only with probabilities tweaked for the average size to be smaller, as larger and longer rectangles protruding from the walls would further increase the chance of air pockets. The generator function for this takes in the top-left, and bottom-right position of the rectangles, and thus additional math is needed to convert center, height, and length into that format.

The generators are then written as a dataframe as follows:

```
new = pd.DataFrame(
    {
        "BestX" : [],
        "BestY" : [],
        "BestTime" : [],
        "WorstTime" : [],
        "ImageLink": [],
        "SizeX": [],
        "SizeY": [],
        "NumberOfCuts": [],
        "TypesOfCuts": [],
        "CutPositionsX": [],
        "CutPositionsY": [],
        "CutSizesX":  [],
        "CutSizesY": []
    }
)
for i in range(1000000):
    yResolution = 32
    xResolution = 32
^^I# Room size is fixed to 32x32
    ACspeed = 1

    walls = WallBoundary(yResolution, xResolution)
    walls.borderWall()
    conditions = walls.generateRoom()
    walls.generateIndex()
    walls.generateACDirections()
    walls.indexWithoutCare()
```

Then, another function is used to generate the room based on the previous function, and index all the possible positions the air conditioner can go on.

```
for index, pos in enumerate(walls.possibleACIndex):
    print(f"Simulation batch {start}/{total}")
    simTimes = []
    for directionIndex, direction in enumerate(WallBoundary.unitVect):
        #print(f"Trying direction: {direction}...")
        pathOfFlow = [i+j for i, j in zip(pos, direction)]
        if (not walls.boundary[pathOfFlow[0], pathOfFlow[1]]):
            #print("Works! Testing now...")
            velocityInlet = [VelocityBoundary(pos[0], pos[1], ACspeed,
            ↪  directionIndex)]
            initCondition = np.ones((yResolution, xResolution,
             ↪  Simulation.latticeSize)) / 9
            simulation = Simulation(
                yResolution, xResolution, initCondition, walls,
                 ↪  velocityBoundaries=velocityInlet,
                 ↪  pressureBoundaries=[]
            )
            simTime, stable = simulation.simulateUntilEquilibrium(
                equilibriumThreshold=2.5, limit=500
            )
            #print("Done!")
            if not stable or simTime == 500:
                #print("It exploded :() or didn't equalize")
                break
            simTimes.append(simTime)
        #else:
            #print("Doesn't work! Moving on...")
        #print("\n")
    results.append(np.average(simTimes))
    start += 1
    #print("-------------")}
```

Then, the generator iterate through all positions and viable (facing into open air) directions. For position, its result is saved as the average result from all viable directions.

```python
if(len(results) == 0):
    continue

full_frame()
plt.imshow(walls.invertedBoundary, cmap="hot", interpolation="nearest")
imgpath = f'data/images/{count}.png'
plt.savefig(imgpath, bbox_inches="tight", pad_inches=0)
# Save image of room.

results = np.nan_to_num(results, copy=False, nan=501)
# In case some positions averages equate to nan, covert to 501.
lowest = min(results)
most = max(results)
ansIndex = [index for index, result in enumerate(results) if result == lowest][0]
# Get the index with the lowest result.
ansPos = walls.possibleACIndex[ansIndex]

conditions["BestX"] = [ansPos[1]]
conditions["BestY"] = [ansPos[0]]
conditions["BestTime"] = [lowest]
conditions["WorstTime"] = [most]
conditions["ImageLink"] = [imgpath]

print(f"position: {ansPos} Momentum: {results[ansIndex]}")
df_dictionary = pd.DataFrame(conditions)
```

```
    new = pd.concat([new, df_dictionary], ignore_index=True)
    count += 1
    # Save everything to data.
```

4.3 Convolutional neural network

We used `Tensorflow`'s Convolutional Neural Network framework to train our model [3]. The images of the room were scaled down from their original 480×480 size to a more manageable 32×32 size. Its grayscale values were also inverted for more efficiency. We had generated 1438 possible room layouts and air conditioner positions, and used it to train and test our model with a train to test ratio of $7:3$. Our model has ten layers, consisting of `Conv2D`, `MaxPooling2D`, and Dense layers arranged as follows:

1. `Conv2D(32, (4, 4), activation="relu", input_shape=(32, 32, 1))`
2. `MaxPooling2D((2, 2))`
3. `Conv2D(64, (4, 4), activation="relu")`
4. `MaxPooling2D((2, 2))`
5. `Conv2D(64, (4, 4), activation="relu")`
6. `Flatten()`
7. `Dense(64, activation="relu")`
8. `Dense(32, activation="relu")`
9. `Dense(16, activation="relu")`

10. `Dense(2, activation="relu")`

The first five is used to extract features that are relevant to our system, and the last five is used to calculate those features and output it into numbers while keeping computational overhead low. In the end, we used the `Adam` optimizer to increase the training efficiency, and the best error (using MSE) we could achieve was 2.56, meaning that the predicted position was on average 1.5 pixels away from the optimal position.

Our baseline was roughly calculated based on the assumption that since most rooms are rectangular, ours being a square of size 32×32 in particular, the minimal distance from the optimal position where cooling starts to be affected should be about 1 side or 32 pixels, since from optimal testing it appears the optimal placement tended towards corners. This means that the highest acceptable error is around 45, since maximum distance where cooling time would be minimally affected. Thus, comparing this to our actual error of 2.56, our AI is a success.

4.4 Discussion

Written by Puripat Thumbanthu as strongly advised by Assoc. Prof. Dr. Wiwat Ruenglertpanyakul

Here, I'd like to address the pitfalls of the model. By my standards, this project is more than complete. We have built a full-fledged two-dimensional fluid simulation in less than ten weeks. From knowing literally nothing about the field to being able to understand the equation that underlies it. It is a huge achievement in terms of learning. The convolutional neural networks also worked extremely well for the room shapes in two dimensions. Too well almost. However, this project has many major pitfalls that I feel too guilty to not state it before ending this report. So, I shall address the elephant in the room here.

Firstly, the fluid simulation currently applies only partially to three-dimensional systems. Most of the airflow in a room is driven by secondary and tertiary flows, which are responsible for creating vortices. While secondary flows occur in two dimensions, tertiary flows do not. These tertiary flows contribute significantly to the complexity of simulating fluids in three dimensions. Addressing this issue would require extending the Lattice-Boltzmann method to three dimensions—a feasible task, but one that would significantly delay data processing.

Secondly, the boundary conditions in the Lattice-Boltzmann method do not fully capture air conditioner dynamics. An actual air conditioner functions as both a velocity and a pressure boundary condition. The reason a room doesn't explode under the influence of the air conditioner is that walls and windows allow air to escape. Those walls and leaks aren't included in our model yet. Instead, we rely on the assumption that a density boundary condition can approximate the effects of velocity and pressure boundary conditions, which introduces some inaccuracies. These inaccuracies are also doubled down by the fact that the function that we used are the time until fluid equilibrium, not thermal differences on the wall.

Thirdly, there isn't a standard metric that can capture the exact thermal units of an air conditioner in the Lattice-Boltzmann method. From my current knowledge, the Lattice-Boltzmann method is quite a mathematical method, and there isn't a well-defined unit for measurement. The room that we used is 32×32, but the size of the density boundary condition is only 1×1. We don't know the exact magnitude of the fluid vector that can imitate a home air conditioner with $12,000$ BTU. If a 1×1 density boundary actually represents a square meter of air conditioning unit, let's say $12,000$ BTU, that would be like trying to cool down a closed basketball field with a single air conditioner, which is absurd. Still, we don't know if this is true or not.

Nonetheless, despite the limitations of our computational model as well as certain time constraints imposed on us, I have achieved the goal I set out to achieve during my time in this camp, and still attest to the level of quality that our model has. This would not be possible if not for P. Jenta Wonglertsakul and P. Nattavee Sunitsakul, our mentor that has guided us all the way through, and our colleague, Kritpatchara Wongkwan, for providing us with this opportunity to learn much about fluid dynamics and artificial intelligence by contacting `AI Builders` team to host this `AI Builders X ESCK` event in the first place.

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]*Choosing colormaps in matplotlib — matplotlib 3.9.2 documentation*, https://matplotlib.org/stable/users/explain/colors/colormaps.html (visited on 11/17/2024).

[3]*Convolutional neural network (cnn)*, https://www.tensorflow.org/tutorials/images/cnn.

[4]*Interpolations for imshow — matplotlib 3.9.2 documentation*, https://matplotlib.org/stable/gallery/images_contours_and_fields/interpolation_methods.html (visited on 11/17/2024).

[5]Matias Ortiz, *Simple lattice-boltzmann simulator in python | computational fluid dynamics for beginners*, Apr. 15, 2022.

[6]*Matplotlib.pyplot.imshow — matplotlib 3.9.2 documentation*, https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.imshow.html (visited on 11/17/2024).

[7]*Matplotlib.pyplot.quiver — matplotlib 3.9.2 documentation*, https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.quiver.html.

[8]V. Schroeder, *Lattice-boltzmann fluid dynamics, Physics 3300, weber state university, spring semester, 2012*, (2012) `https://physics.weber.edu/schroeder/javacourse/LatticeBoltzmann.pdf` (visited on 11/17/2024).

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

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