PHYS115 Project Report

Henry Olling

June 2021

1 Introduction

This project demonstrates a 2D fluid dynamics simulation with support for basic chemical interactions among multiple reagents. These reagents are modeled as groups of tracers, each having a position and a chemical ID. Chemical simulations are processed in a random fashion in order to better emulate the stochastic processes of chemical reactions. This model can also be easily modified to work in three dimensions, but for the sake of simplicity of visualization and of debugging, that was not done in this project.

The 2D flow is confined to a rectangular area although the model permits periodic boundary conditions (i.e. fluid moving "upwards" and "out of bounds" will teleport to the "bottom" of the model) as well as placements of barriers. Furthermore, the 2D flow model also allows for starting conditions to be specified for the edges and the middle of the model and for separate boundary conditions to be specified as well.

The bulk of this project's implementation is concerned with the operation of the 2D flow model. This model is an implementation of the Lattice-Boltzmann fluid dynamics model, which discretizes flow direction and time. In this implementation, flow is given nine possible directions and takes place in two dimensions – these particular constraints are often referred to as the D2Q9 version of the Lattice-Boltzmann algorithm. This model is used to calculate densities and flow velocities at each point in the 2D grid it runs in, and these macroscopic properties are used to drive the tracers which then drive the chemical model.

For the chemical model, this project uses an implementation of the Gillespie algorithm. This algorithm defines a stochastic model of chemical reactions, and operates on an uneven time step. This clashes with the Lattice-Boltzmann model, where all time steps are the same length. Additionally, the Gillespie algorithm is not aware of the spatial distribution of particles and only understands the total counts of each reagent. This adds another aspect which distinguishes it from the Lattice-Boltzmann algorithm.

The remainder of this paper aims to explain how these two algorithms work, how they were connected, and what the result looks like.

2 Background

There are three important systems which make up this model: the Lattice-Boltzmann algorithm, Gillespie's algorith, and tracers. This section will provide a brief bird's-eye view of

each.

2.1 The Lattice-Boltzmann Algorithm

The Lattice Boltzmann algorithm is a discrete time and space fluid dynamics model which operates in a 2D grid and has discrete velocities. To begin our explanation of this algorith, we first must understand its structure and then we can understand its function.

The core of the Lattice-Boltzmann algorithm is a 2D grid. (For 3D models, a 3D grid is used, but this paper only discusses the 2D case for simplicity, although it is very easy to scale this all up into 3D). In this 2D grid we will permit only nine velocites (as shown in Figure 1) [8]. Note that one of the velocities corresponds to no movement at all. These velocities are given below:

$$\vec{e_0} = (0,0),$$
$$\vec{e_1} = (1,0), \vec{e_2} = (0,1), \vec{e_3} = (-1,0), \vec{e_4} = (0,-1)$$
$$\vec{e_5} = (1,1), \vec{e_6} = (-1,1), \vec{e_7} = (-1,-1), \vec{e_8} = (1,-1)$$

These will velocities allow us to model the movement of particles but we first need to create some weights. We know that for a 2D gas, the Boltzmann distribution of *thermal* velocities is given by

$$D(\vec{v}) = \frac{m}{2\pi kT} e^{\frac{-m|\vec{v}|^2}{2kT}}$$

[9]. However, we want to understand the distribution of thermal velocities in 2D *discrete* space, so we can discretize the Boltzman distribution into 2D. The result is show in Figure 2 and the weights are given below [9]:

$$w_0 = \frac{4}{9},$$
$$w_1 = \frac{1}{9}, w_2 = \frac{1}{9}, w_3 = \frac{1}{9}, w_4 = \frac{1}{9}$$
$$w_5 = \frac{1}{36}, e_6 = \frac{1}{36}, e_7 = \frac{1}{36}, w_8 = \frac{1}{36}$$

We must also make some assumptions here. Our fluid model will only work in the condition that the macroscopic flow $\vec{u}$ (which is different from the thermal velocity $\vec{v}$) is much smaller than the constant c, which is given by [9]:

$$c = \sqrt{\frac{3kT}{m}}.$$

We have not yet introduced this constant yet, so we will do so now. In our model c is set to be 1, but it is actually a constant involved in calculating the average values of the thermal velocities v_x and v_y (up to the fourth power). We will, for now, set aside what the exact interpretation of this parameter is, and instead mention its practical implications; we are constraining our flow velocities into nine possible values and must guarantee that $|\vec{u}| << 1$ in order to avoid the possibility that a particle moves more than one cell in a single time step [9].

What should be taken away from this rather hand-wavey discussion of the velocities and weights is that there are a finite number of velocities and each has a corresponding weight calculated by discretizing the Boltzmann distribution and that the macroscopic velocity of a bit of fluid must always be much less than one.

It is now possible to discuss the mechanics of the fluid flow itself. We begin by creating an array with some width w and some height h. At each cell, we wish to store nine values – one for each of our velocities. We will call this array n, and each of its velocity slices n_i. This array describes the densities (so to speak) of the number of particles flowing in a given direction. So n_0 describes the relative density of particles which are not moving, at every cell in our grid.

This allows us to calculate densities and macroscopic velocities at each point [8]:

$$\rho = \Sigma_i n_i, \ \vec{u} = \frac{1}{\rho}\Sigma_i n_i e_i$$

From here, we can actually define the steps of our model. The Lattice-Boltzmann algorithm works in two major steps – firstly, we do the "streaming" step, and then we do the "collision" step [9]. The streaming step "moves" a piece of fluid along its associated velocity vector. This means that the fluid in n_1 would be moved "up" or "north" one cell since its associated vector is $\vec{e_1} = (1, 0)$. The collision step models the fluids interaction with itself: the drag, friction, etc which fluids exert on themselves.

Here is what the algorithm look like as a procedure:

1. Streaming step. n_i moved along e_i.
2. Handle barrier interactions. This involves reflecting fluid off of barriers.
3. Calculate macroscopic properties ρ, and $\vec{u}$.
4. Calculate collisions. This is where fluids with different velocity vectors interact.
5. Apply boundary conditions.
6. Repeat

How each of these steps is actually implemented will be discussed in section 5.2. For now, it should be enough to understand this simple outline. Fluid moves along its associated velocity vector (fluid is kept in nine seperate arrays), bounces off of walls, self interacts, and then repeats the cycle. It is, of course, both interesting and useful to discuss the theory of this algorithm in more detail, but as this section is already two pages long I will leave it for another time.

2.2 Gillespie Algorithm

The Gillespie Algorithm is a stochastic chemistry simulation. As inputs, the Gillespie Algorithm requires an initial population of chemicals, a set of reactions which can operate on that population of chemicals, and associated probabilities for those reactions [4].

The general procedure of the Gillespie Algorithm is as follows [6]:

1. Setup: Store initial populations, prepare chemical reaction table
2. Calculate probabilities.
3. Calculate tau.
4. Calculate mu.
5. Execute selected reaction.
6. Return to step two.

This straightforward algorithm is easy to understand. The variable τ is the amount of time to step forward on a given iteration. It is calculated as

$$\tau = \frac{1}{a_0} \ln(\frac{1}{r})$$

where a_0 is the sum of all the individual reaction probabilities a_i [6]. (It should be noted that the notation used in the source code follows that of the Karig lecture slides, and not the Roussel lecture slides).

The value of μ is given by:

$$\Sigma_{i=1}^{\mu-1} a_i < r < \Sigma_{i=1}^{\mu} a_i$$

where r is a random variable in the range $[0, 1)$ [4]. Solving this is actually quite fast to do in Python, and the specific implementation of this will be discussed in Section 5.2.

It is important to note that the initial population does not include information about placements or locations of molecules. This makes for a challenge which will be addressed in Section 5.3.

2.3 Tracers

Tracers are a very simple concept in fluid dynamics simulation. A tracer is a particle "floating" in the simulated fluid. Its movement is determined by the fluid it floats in and can be used as a powerful debugging tool. Tracers will be used to integrate the Gillespie and Lattice-Boltzmann algorithms later on in section 5.4 and 5.5.

3 Motivation

The primary motivation for doing this project was that it sounded like a satisfying mix of challenging and interesting. I've always found fluid mechanics interesting and have recently acquired an interest in the simulation of chemical reactions. Creating a model which can do both of these things (albeit rudimentary) struck me as a fun project idea.

While this project has to immediate practical use, and accomplishing any real science with it would require substantial modification, it still is useful as a toy model for learning about techniques used to solve fluid dynamics and chemical simulation problems.

4 Goals

The goals for this project have changed somewhat from the goals listed in the project proposal. The original goals were to "get a working model of one fluid inside of a box" (meaning get a simple fluid model working), get multiple fluids together inside of one box, then get those fluids to interact with each other, and then finally run some experiments with the resulting model. This only partially worked – while getting multiple fluids to work together inside of the model was successful, and the integration with the Gillespie Algorithm worked, not all of the proposed experiments were possible. The updated goals of this project are outlined in this section.

The first, and most important, goal of this project is to make a working 2D fluid model. This model will function inside of a rectangular grid, and will be an implementation of the Lattice-Boltzmann algorithm.

The second goal of this project was to make a working standalone implementation of the Gillespie algorithm. This is probably the easiest of the goals as the Gillespie algorithm is quite simple to implement.

The third goal of this project is to integrate the Lattice-Boltzmann algorithm and the Gillespie algorithm. This is challenging to do and will require the use of a third "algorithm" which will add tracers to the model. The completion of this step will mark the completion of this projects model.

The fourth, and final goal, of this project is to use the combined model to perform some basic experiments. These "experiments" will focus more on demonstration than actual science (largely due to time constraints).

5 Project Details

This section will explain how the actual project code is structured, why it is structured how it is, and how it works.

The model is implemented as a single Python class, named `Fluid_Chem_Algorithm`. This algorithm contains every method and variable needed to run the simulation and has flags which permit the user to disable certain features. The `is_chem_enabled` flag enables/disables all chemical simulation features, including creating tracer reference and count tables. The `is_tracers_enabled` flag enables/disables all tracer updates. The `is_react_enabled` enables/disables the actual reaction handling inside of the chemistry update code.

Inside of the implementation, the `@block` decorator is used. This decorator runs the function it is attached to and then deletes it. This allows the creation of C-style anonymous blocks and is used purely for organization purposes.

5.1 Lattice-Boltzmann Algorithm

If we ignore, for a moment, the fact that the Lattice-Boltzmann algorithm is implemented in a class, then the listing shown below (Listing 1) shows the setup of the LB algorithm. The first two variables `es` and `ws` define the velocity vectors and their associated weights respectively. The variable `ns` defines the grid whereupon the actual algorithm runs. This

variable contains the densities of fluid at each point, split into each of the nine directions of flow. The `rhos` and `us` variables hold the information about density and macroscopic flow. It should be noted that `us[:,:,0]` is they velocity in the "Y" direction and that `us[:,:,1]` is the velocity in the "X" direction.

```
es = np.array([[0,0], [1,0], [0,1], [-1,0], [0,-1], [1,1], [-1,1],
    [-1,-1], [1,-1] ] , dtype=np.int64)
ws = np.array([4.0 /  9.0, 1.0 /  9.0, 1.0 /  9.0, 1.0 /  9.0, 1.0 /  9.0,
     1.0 / 36.0, 1.0 / 36.0, 1.0 / 36.0, 1.0 / 36.0 ], dtype=np.float64)
ns = np.zeros((HEIGHT, WIDTH, 9), dtype=np.float64)
rhos = np.zeros((HEIGHT, WIDTH), dtype=np.float64)
us = np.zeros((HEIGHT, WIDTH, 2), dtype=np.float64)
```

Listing 1: Setup of Lattice-Boltzmann Algorithm

The Lattice-Boltzmann algorithm is initalized in the `reset_model()` function. This function resets the whole model (chemistry, tracers, and fluids). We are interested in just two functions this calls: `initalize_fluid()` and `initalize_walls()`. The `initalize_walls()` function simply clears the array which contains the walls and then rasters a cricle into it. The `initalize_fluid()` function, however, is much more interesting.

If you look at the source of `initalize_fluid()`, you will notice that it has five almost identical functions called with the `@block` decorator. Each of these functions initializes a different region of the model. Lets examine the first sub-function of the fluid initalization routine.

```
def initalize_fluid(self):
  @block
  def setupall():
    # initalize the whole model to have zero fluid flow
    u_0 = self.boundary_conds[0]
    u0magsq = u_0[0]**2 + u_0[1]**2
    for i in range(9):
      edotu = np.dot(self.es[i], u_0)
      self.ns[:,:,i] = self.ws[i] * 1 * (1 + (3 * edotu) + (9 / 2 * edotu
   **2) - (3 / 2 * u0magsq))
```

Listing 2: Fluid Initalization

The `setupall()` function effects the entire model and all of the "sub-fluids" inside of it. If we look at it one step at a time, it follows this procedure [9]:

1. Boundary condition for body of model $\rightarrow u_0$.
2. $n_i = w_i \rho(1 + 3(\vec{e_i} \cdot \vec{u_0}) + \frac{9}{2}(\vec{e_i} \cdot \vec{u_0})^2 - \frac{3}{2}|\vec{u_0}|^2)$ for every i.

What step two does is calculate the equilibrium "density." (For the actual derivation of this, see [9]). We must initialize every cell in the model to have a non-zero sum of n_i in order to avoid a divide by zero in an upcoming step.

After the initialization step is complete, the model begins the actual simulation. This is achieved through calling the `tick()` method, which calls several functions which are of interest to us. First are the `update_stream()` and `update_barriers()` method. The listings for these functions are shown below, in Listing 3.

```
def update_stream():
  for i in range(1, 9):
    ns[:,:,i] = np.roll(ns[:,:,i], es[i], axis=(0, 1))

# lookup table for indices of opposite "sub-fluid"
opp = np.array([0, 3, 4, 1, 2, 7, 8, 5, 6])

def update_barriers():
  for i in range(1, 9):
    ns[:,:,i][np.roll(walls, es[i], axis=(0, 1))] = ns[:,:,opp[i]][walls]
```

Listing 3: Streaming and Barrier Steps

These two steps are quite similar. The first one moves the entire fluid grid for each "sub-fluid" along its velocity vector. This is called the streaming step. Notice how only the "sub-fluids" with non-zero velocity vectors are updated. The second function is usually bundled in with the streaming step, but was separated here for clarity and ease of debugging. This function is a bit more complex. It check if there is a wall "in front of" the "sub-fluid" and then swaps makes it swap values with the "sub-fluid" which points in the opposite direction. The speed of this is greatly increased by using `numpy`'s masking abilities.

Next in the operation of this algorithm, the densities and velocities are calculated. The code for these steps is provided in Listing 4, along with the collision step.

```
def calc_velocity():
  us[:,:,0] = np.sum([ns[:,:,i] * es[i,0] for i in range(9)], axis=0) /
   rhos
  us[:,:,1] = np.sum([ns[:,:,i] * es[i,1] for i in range(9)], axis=0) /
   rhos

def calc_density():
  rhos[:,:] = np.sum(ns[:,:,:], axis=2)

def update_collide():
  umag = np.linalg.norm(us, axis=2)
  const_1 = 1 - 1.5 * umag * umag
  for i in range(9):
    edotu = np.dot(us, es[i])
    ns[:,:,i] = ((1 - omega) * ns[:,:,i]) + (omega * ws[i] * rhos * ((3 *
   edotu) + (9 / 2 * edotu**2) + const_1))

```

Listing 4: Velocity and Density Calculation and Collision Step

Density is calculated *before* velocity, and is used in the calculation of velocity. The equations which describe these two functions are provided in Section 2.1. Of great interest to us now is the `update_collide()` method. This method handles the self interaction of the fluid and the formula it uses is provided below [9]:

$$n_i = n_i(1-\omega) + \omega w_i \rho(1 + 3(\vec{e_i} \cdot \vec{u}) + \frac{9}{2}(\vec{e_i} \cdot \vec{u})^2 - \frac{3}{2}|\vec{u}|^2)$$

This bears a striking resemblance to the formula used to calculate the equilibrium value, but instead includes an omega term. This omega term is a unitless number which defines

roughly how long it will take for the various n_i values to reach equilibrium. This value is, in this model, calculated from the viscosity, like so [7]:

$$\omega = \frac{1}{3 * \eta + \frac{1}{2}}$$

The final part of the Lattice-Boltzmann implementation is the setting of the boundary conditions. Every tick, the edges of the model are clamped to their equilibrium value. This is done to prevent fluid from "wrapping around" the model – from going out of bounds on one side and appearing at the other edge. This is all accomplished in the `update_boundaries()` function, which is almost identical to the initialization function except that it only sets the edges of the model.

5.2 Gillespie's Algorithm

Two Gillespie Algorithm implementations are provided with this project. The first, `Gillespie_Algorithm_Demo.ipynp` is a Jupyter notebook which provides a very basic implementation of the algorithm. We will discuss this implementation first, before moving onto the implementation found in `fca.py`.

The first block of code inside the `do_gillespie()` function prepares the initial populations as well as some containers in which to record the history of the process. The tuple `(n_steps, p0.shape[0])` simply sets up the `populations` container to have 1 slot for every chemical type for every time step.

It is also interesting to note that since, in this particular implementation, the probabilities of reaction do not depend on the state of the model, a_0 and a_u are calculated before iteration begins, instead of once every step.

The most interesting part of this code sample is the part which calculates τ and μ. This part is shown in Listing 5. The second line of the listing shows that τ is calculated exactly as it is described in Section 2.2. However, the procedure for calculating μ is more involved. While the `scipy` package has a function which enables the user to select an element from a list based off of associated probabilities, the solution shown in Listing 5 is much faster. This implementation was borrowed from a Caltech lecture on "Stochastic simulation of biological circuits" [10].

```
# calculate our time step tau
tau = (1 / a0) * np.log(1 / np.random.rand())

# calculate mu
_q = np.random.rand(); mu = 0; _p = 0.0;
while _p < _q:
    _p += au[mu]
    mu += 1
mu -= 1
```

Listing 5: Calculation of τ and μ

The implementation of the Gillespie algorithm is much more complex in the `fca.py` module. This is because integrating the Gillespie algorithm into the Lattice-Boltzmann

algorithm was quite challenging. The `fca` implementation operates per-cell on the populations of tracers inside the given cell. This requires two things:

1. A grid which, for every cell and every chemical in that cell, contains a list of references to tracers of the given chemical type.
2. A grid which, for every cell and every chemical in that cell, contains the total number of tracers of the given chemical type in that cell.
3. A grid which, for every cell, contains the amount of time remaining in the current reaction.

The solution to the above two problems are shown below, in Listing 6.

```
# In the initalization of the model
refs = np.full((HEIGHT, WIDTH, 3, 4), -1, dtype=np.int64)
counts = np.zeros((HEIGHT, WIDTH, 3), dtype=np.int64)
times = np.zeros((HEIGHT, WIDTH), dtype=np.int64)

...

# Inside the chemistry update function
counts[:,:,:] = 0; refs[:,:,:,:] = -1

# create our counts table and our references tables
xs = tracers[:,0].astype(np.int64)
ys = tracers[:,1].astype(np.int64)
mask = (ys < WIDTH) & (xs < HEIGHT)
xs = xs[mask]; ys = ys[mask]
for i in range(tracers[mask].shape[0]):
  g = counts[xs[i], ys[i], chemids[mask][i]]
  if (g < 4):
    refs[xs[i], ys[i], chemids[mask][i], g] = i
  counts[xs[i], ys[i], chemids[mask][i]] += 1
```

Listing 6: Calculation of References and Counts

In the above function, the condition that `g` must be less than four exists purely to reduce the memory requirements of the references table, since we only will ever need four references per chemical per cell. This snippet of code accounts for a *significant* amount of time during the model's tick function.

The way that all of this works is that every tick, the references and counts tables are updated. Then, τ is calculated for every cell in the grid and μ is calculated for every cell where the `times` grid is less than or equal to zero. Cells in the `times` grid with values less than or equal to zero have their value upated to be equal to the corresponding τ value. All cells with a time value which is greater than zero are skipped. For each of the remaining cells, the specific reaction is effected upon the grids population in the following manner (the python code for this procedure is visible in the `update_chemistry()` function but is far too lengthy to list here):

1. Select the current reaction by fetching the μ value for the current cell.

2. Iterate over each entry in the reaction.

3. If the current entry's coefficient is less than zero, remove that many tracers of the correct type by randomly picking that many tracers from the references list and queuing them for deletion.

4. If the current entry's coefficient is greater than zero, create that many new tracers of the current chemical type scattered uniformly in the current cell.

This accounts for the entire implementation of the Gillespie Algorithm in this model. The next Section will briefly provide context as to why this complexity was necessary.

5.3 Integration Challenges

There were a lot of challenges to overcome while combining the Gillespie Algorithm and the Lattice-Boltzmann algorithm. Firstly, the Lattice-Boltzmann algorithm does not really handle individual particles of fluid, which the Gillespie algorithm requires to operate. Additionally, the Gillespie algorithm does not care about where particles are placed. Both of these issues were solved by introducing tracers (the implementation of which is explained in the next section). Large volumes of tracers can be injected into the model to simulate the existence of actual particles. This is why the counts and references grids are necessary.

The `times` grid is necessary because the Lattice-Boltzmann algorithm operates on a fixed-length time step. The Gillespie algorithm does not; the length of its time steps changes each iteration. This difference, combined with the fact that each cell would need a different length time step necessitated the existence of a separate grid to keep track of the time remaining for each Gillespie reaction to finish. The `times` grid is decremented by one every iteration, and cells which have grid values less than zero have their value updated when a reaction is ready to effect them. This `times` grid enables multiple Gillespie algorithms to be running at once, each with a different length time step.

5.4 Tracers

The implementation of the tracers is quite simple. The positions of the tracers is kept in an array with shape `(N_TRACERS,2,)` named `tracers`. An additional array named `chemids` exists alongside it, and has the same *length* and keeps track of the chemical identity of each tracer. Tracers are updated in the `update_tracers()` function where additional tracers are injected every `INJECT_DELAY` number of ticks, tracers positions are updated via the fluid's macroscopic velocity $\vec{u}$ and tracers which are out of bounds or inside walls are deleted.

6 Results

After the model was completed and seemed to work reasonably correctly, several experiments/demonstrations were created to showcase its capabilities. Only three will make an appearance in this write up as several of the early ones revealed bugs and were discarded (after being used to fix the model). The three demonstrations are: a demonstration of vortex

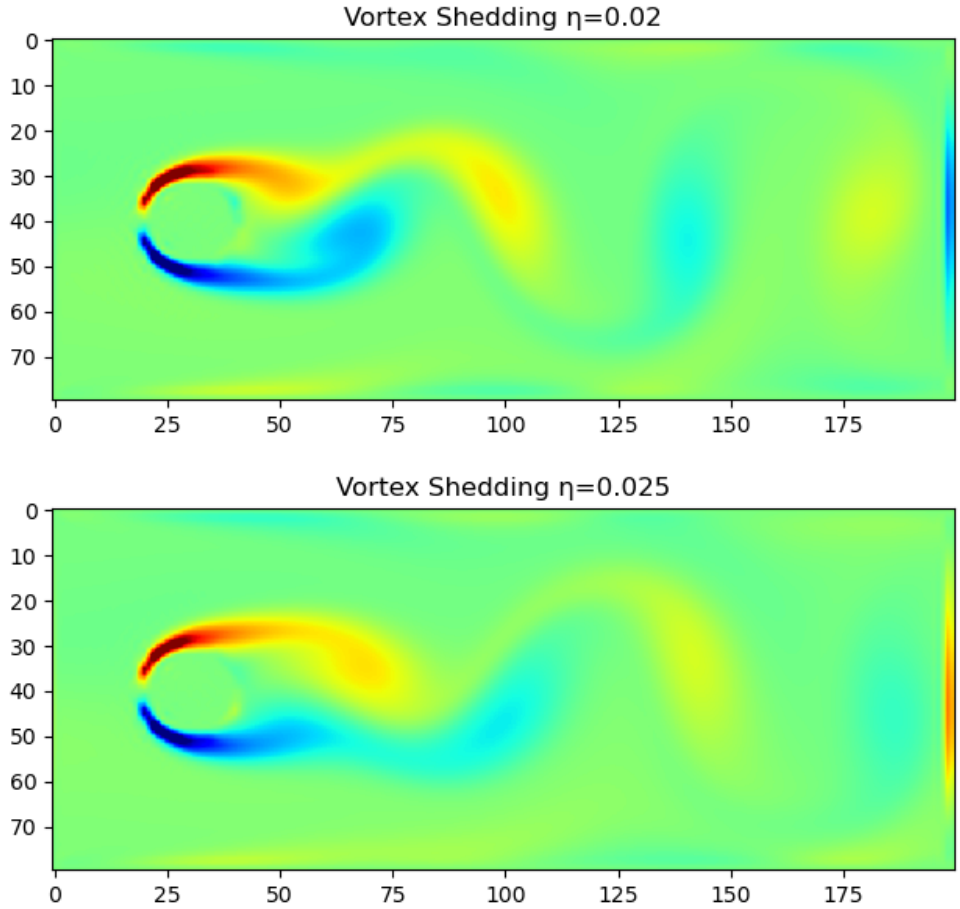
shedding, a demonstration of the effect of wall grid density on the distance tracers can get, and a demonstration of chemical reaction phenomena.

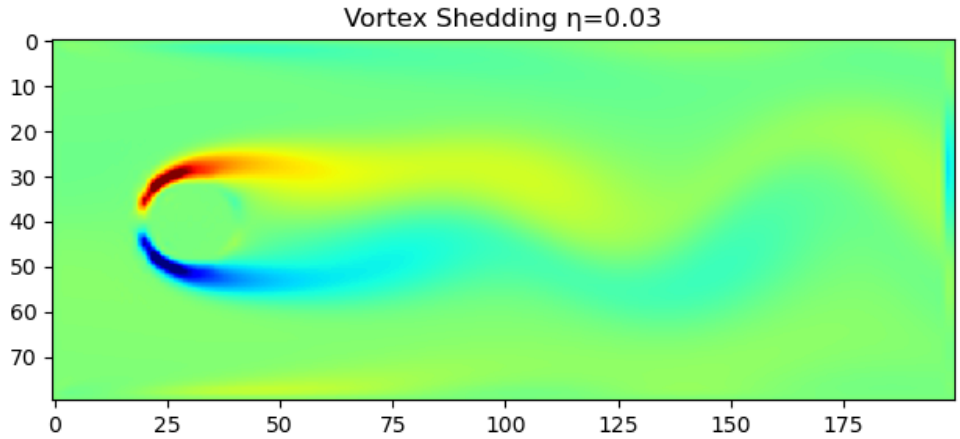
6.1 Vortex Shedding

Vortex shedding is a common and complex phenomenon. Creating a simulation of vortex shedding is, however, quite easy. Simply initialize all flow to move rightwards and place a solid symmetric object in the middle of the flow. This will usually lead to some sort of vortex shedding (and is not the only way to produce it).

This experiment demonstrates how vortex shedding varies with viscosity. The model was initialized to a constant rightward flow and was configured to have a single circle of barriers towards the left-hand side of the model. This circle of barriers was centered vertically. Then, the model was run for 12000 ticks and the resulting state was saved to an image. This process was repeated two more times, for a total of three runs, one for each viscosity chosen.

The results are shown below. It is easy to see how, as viscosity increases, the frequency and strength of the vortexes decreases. For much larger viscosity, on the order of 0.1, the model produces a steady-state of laminar flow. Producing vortex shedding under these conditions requires changing the rate of flow.



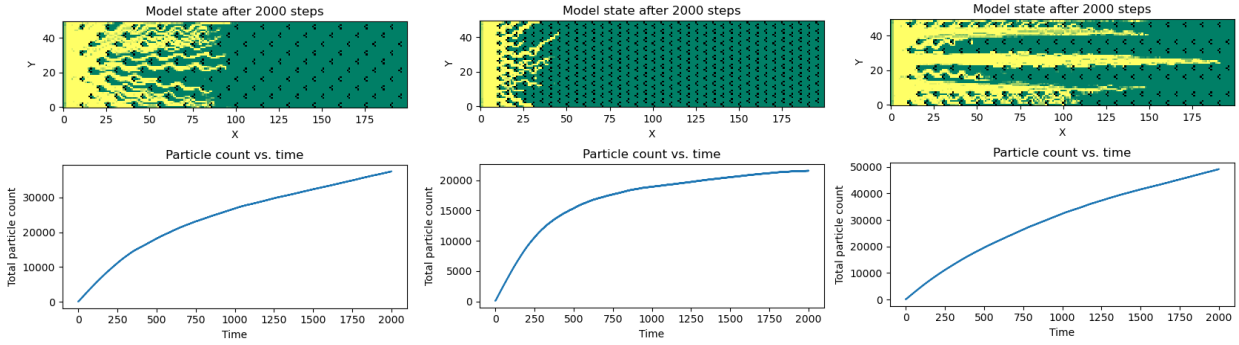


It should be noted that this simulation was relatively fast and it took only a handful of seconds to process all three scenarios. This was made possible my disabling all tracer and chemical functionality.

6.2 Varying Wall Layout

While testing the model, I was curious about how far particles could get through grids of walls of varying tightness. This experiment involved initializing the fluid to a uniform rightward flow, and then introduce a grid of walls. The simulation was then run for 2000 ticks and the number of particles was recorded. Once the simulation was completed, a snapshot was taken, and the process was repeated for a different grid. In total, the experiment was run three times.

The results are shown below.

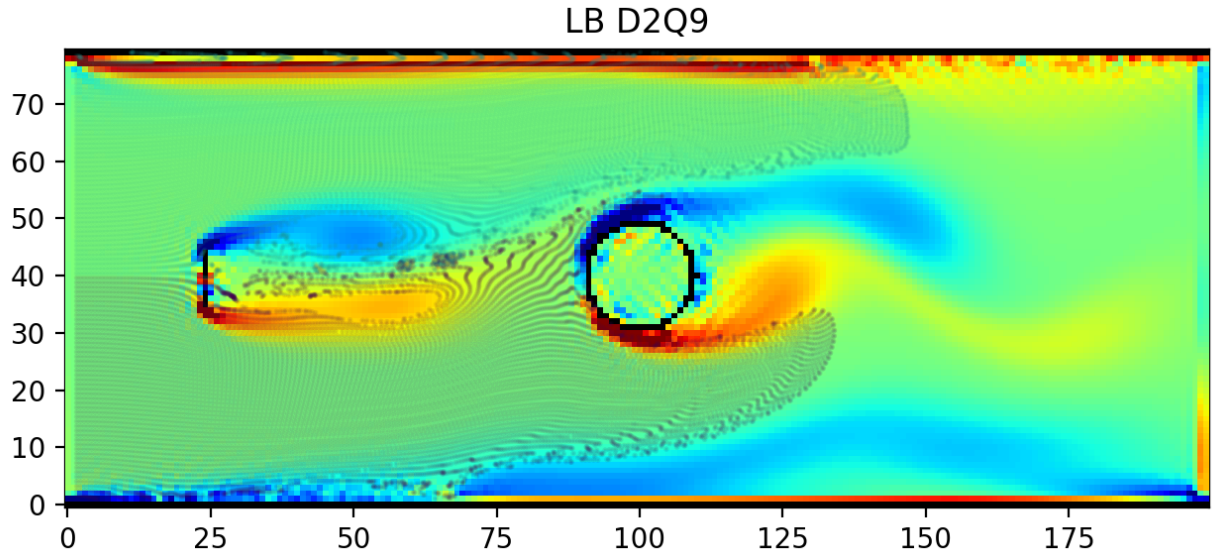


From these figures, it becomes quite apparent that the denser the walls are together, the less distance particles can travel. Additionally, since the model deletes particles which are clipping through walls, a flatter curve for particle count over time means that more particles are being destroyed. This also makes sense, since the densest grid has the flattest slope at the end of the simulation while the least dense grid has the steepest slope.

6.3 Chemical Interactions

One of the aspects of the model which was very difficult to test was the behavior of chemical reactions inside a flowing fluid. This experiment is much more of a demonstration of a reaction taking place – simulating a reaction for more than a few hundred time steps takes an astonishing amount of time.

This demonstration was created by initializing the model to a completely rightward flow and then emitting tracers from the rightward edge. Half of the tracers were of chemical A and the other half were chemical B. As they flowed through the model, they began to mix and form clusters of reacting material. (I do not remember how many ticks this was simulated for – it did take about an hour to run). The full result is shown below.



It is quite apparent that this model did not run for very long. Of greater interest, however, is the clumping of particles near the middle of the region between the vertical wall and the ring. The reaction which was tested here is provided below, along with its associated probabilities, and it has a tendency to form these clusters of reacting matter that grow and move with the flow of the fluid.

$$\begin{aligned} A + 2B &\rightarrow C; P = 0.1; h_\mu = 2 \\ C &\rightarrow A + 2B; P = 0.2; h_\mu = 1 \\ B &\rightarrow; P = 0.4; h_\mu = 1 \end{aligned}$$

This setup was chosen specifically to make it more probable for molecules to be destroyed (and thereby reduce the load on the computer while running the simulation). It does, however, still produce a significant amount of reactions and is very slow to simulate.

7 Discussion

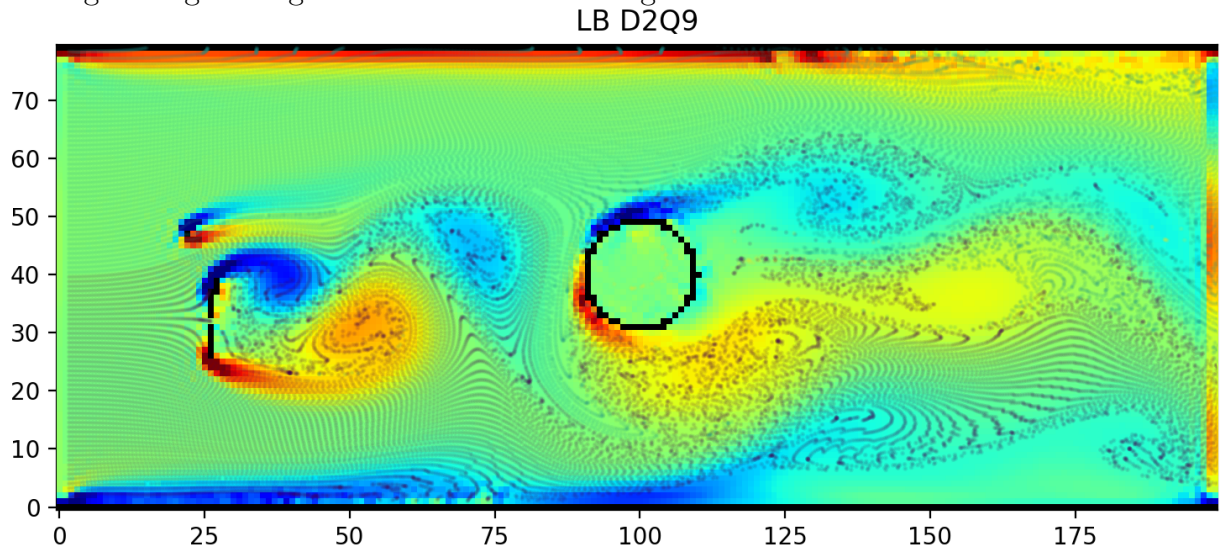
Combining these two algorithms was quite difficult and took several days of experimenting to get right. I think the current solution works quite well, but that it does have some practical

problems.

Mainly, the issues it has are related to the realism of the chemistry – since particles are spawned on one edge and flow with the fluid, the chemistry can not influence the behavior of the fluid and the densities and viscosities of the various fluids are completely discarded. I did make a serious attempt to solve this but struggled. Ultimately, I concluded that the Lattice-Boltzmann algorithm can not support fluids of varying density (at least not as I had implemented it).

Additionally, simulating the mixing of fluids was somewhat challenging as you must create significant enough turbulence to get the tracers to mix correctly, and initalizing the model to have two fluids collide together in an empty region causes major numeric issues since the Lattice-Boltzmann algorithm has a serious aversion to a density of zero. Also, the performance of the chemistry algorithm is really really abysmal, and get worse quadratically (I think) with the number of tracers. It is not immediately clear how to resolve this, but I do speculate about it in Section 8.2.

Overall, I think the results of the project were promising, and I learned a siginificant amount about the intracacies of fluid simulation. The model is really enjoyable to mess with in interactive mode provided you are prepared to wait for things to happen once the number of tracers gets large enough. I think the below image is the best to come out of the model.



This image shows off the results of running the model for about an hour with chemistry disabled and tracers enabled. The mixing of the two chemicals (light colored tracers and dark colored tracers) is quite apparent, as is the swirling pattern of the vortex shedding.

Overall, I am satisfied with how this project turned out.

8 Going Further

There are many different ways to go continue experimenting with this project. Most which come to mind immediately are various methods of optimization, but expanding the model up into 3D and improving the chemistry simulation would also be interesting routes to take.

8.1 Upgrade to 3D

The easiest method of improving this project is to expand it into three dimensions. This is quite simple to do and involves only changing a handful of constants and making some minor changes to some of the methods. The shape of `ns` needs to be changed to `(HEIGHT, WIDTH, 27)` and the shape of `us` needs to be changed to `(HEIGHT, WIDTH, 3)`. Additionally `es` would need to be updated to have 3-component vectors and have 27 of them. Similar modifications would need to be made to the tracers algorithm as well.

8.2 Optimizations

There are quite a number of optimizations which can be done to improve the performance of this model.

8.2.1 Translation into C

The most efficient way (at least in my opinion) would be to rewrite a substantial amount of the model in C and call the C functions via python. Doing this for the entire model would be excessive and "C-ifying" the `update_chemistry()` and `update_tracers()` methods would probably be the most effective as these functions take up a substantial amount of time to run.

8.2.2 GPU Acceleration

The way to get the maximum amount of speed out of this model (at the cost of the maximum amount of work) would be to modify it to work on a GPU. Modern GPUs are SIMD type CPUs, meaning that they operate on multiple data at once. This, combined with their massively parallel architecture would make it possible to achieve incredible speeds. This could be done with the `OpenGL` or `Vulcan` APIs and would have to be written entirely in C or C++, although bindings could be created to connect between the model implementation and a python environment. If I were to implement one of these improvements, this would be the one I would choose.

8.3 Better Chemistry

There are some substantial problems with the implementation of the Gillespie algorithm, mainly that it only permits fixed values for reaction probabilities. In reality, these probabilities depend on a host of factors including concentration, temperature, and pressure. Improving the code-base to support this would be quite trivial, although I have not tried it.

8.4 Heat Flow

Currently, the model does not have any understanding of temperature. This makes simulation of some processes impossible. Implementing heat flow into a Lattice-Boltzmann simulation has been done in several papers. The 2018 paper by Fei and Luo provides a good

starting point for implementing heat flow, which would greatly improve the utility of the model.

8.5 Varying Viscosity and Zero Densities

Another interesting improvement to this project would be to make the Lattice-Boltzmann component properly support varying viscosities and areas with zero density. This would be quite challenging though as the viscosity field would have to be effected by the flow of the fluid, while simultaneously effecting the fluid flow. I have not found much in the way of actual published efforts to accomplish this and I doubt that it is feasible.

The introduction of zero density would make for more interesting initialization and could facilitate interesting mixing scenarios, but should not occur outside of initialization and once regions with zero density disappear, no more should appear.

9 Conclusion

Overall, this project was a success. All of the goals outlined in section 4 ultimately worked out in some way. This model works well as a toy for experimenting with fluid dynamics, tracers, and primitive stochastic chemical simulation.

While there is significant room for improvement, this is not a downside and is merely an opportunity to explore more.

References

[1] Gillespie algorithm. URL: https://en.wikipedia.org/wiki/Gillespie_algorithm/.

[2] Shiyi Chen and Gary D Doolen. "Lattice Boltzmann Method for Fluid Flows". In: *Annual Review of Fluid Mechanics* 30.1 (1998), pp. 329–364. DOI: 10.1146/fluid.2002.34.issue-1.

[3] Linlin Fei and Kai Hong Luo. "Cascaded lattice Boltzmann method for incompressible thermal flows with heat sources and general thermal boundary conditions". In: *Computers & Fluids* 165 (2018), pp. 89–95. ISSN: 0045-7930. DOI: https://doi.org/10.1016/j.compfluid.2018.01.020. URL: https://www.sciencedirect.com/science/article/pii/S0045793018300288.

[4] David Karig. *Introduction to Stochastic Simulation with the Gillespie Method*. University Lecture. Apr. 2005. URL: https://www.cs.princeton.edu/picasso/seminarsS05/Karig_slides.pdf.

[5] Timm Kruger et al. *The Lattice Boltzmann Method. Principles and Practice*. Springer, 2017.

[6] Marc R. Roussel. *The Gillespie stochastic simulation algorithm*. University Lecture. URL: http://people.uleth.ca/~roussel/C4000foundations/slides/25stochsim.pdf.

[7] Dan Schroeder. *Fluid Dynamics Simulation*. URL: https://physics.weber.edu/schroeder/fluids/.

[8] Dan Schroeder. "Fluid Simulations for Undergrads". URL: https://physics.weber.edu/schroeder/fluids/FluidSimulationsForUndergrads.pdf.

[9] Dan Schroeder. *Lattice-Boltzmann Fluid Dynamics*. Weber State University, 2012. URL: https://physics.weber.edu/schroeder/javacourse/LatticeBoltzmann.pdf.

[10] *Stochastic simulation of biological circuits*. Caltech University, 2019. URL: http://be150.caltech.edu/2019/handouts/12_stochastic_simulation_all_code.html.