PDE based traffic model

$\rho = \rho(x,t)$ density of cars at position x at time t $[\#\text{ vehicles}/\text{km}]$

$v = v(x,t)$ velocity of a car at position x at time t $[\text{km}/\text{h}]$

$f = f(x,t)$ flow of cars at position x at time t $[\#\text{ vehicles}/\text{h}]$

$$f(x,t) = \rho(x,t)\, v(x,t)$$

By the LWR model,

$$\frac{\partial \rho}{\partial t} = -\frac{\partial f}{\partial x} \qquad (1)$$

We now make 2 simplifying assumptions: $f = f(\rho)$ and $f(\rho)$ is logistic of the form:

$$f(\rho) = V_{max}\,\rho\left(1 - \frac{\rho}{\rho_{max}}\right) \qquad (2)$$

Now,

$$\frac{\partial f}{\partial x} = \frac{df}{d\rho}\frac{\partial \rho}{\partial x} = V_{max}\left(1 - 2\frac{\rho}{\rho_{max}}\right)\frac{\partial \rho}{\partial x} \qquad (3)$$

Plugging this into (1)

$$\frac{\partial \rho}{\partial t} = -V_{max}\left(1 - 2\frac{\rho}{\rho_{max}}\right)\frac{\partial \rho}{\partial x} \qquad (4)$$

This is the system we are trying to solve.

Let us discritize space and time into points $x_1, x_2, x_3, \dots, x_i, \dots$ with step size $x_{i+1} - x_i \equiv s$ and points $t_1, t_2, t_3, \dots, t_j, \dots$ with step size $t_{j+1} - t_j \equiv h$. Then let's define ρ_{ij} to be $\rho_{ij} \approx \rho(x_i, t_j)$ and $f_{ij} = f(\rho_{ij})$. Now, the approximate partial derivatives are

$$\left.\frac{\partial \rho}{\partial t}\right|_{x=x_i,\,t=t_j} = \frac{\rho_{i,j+1} - \rho_{i,j}}{h} \qquad \left.\frac{\partial f}{\partial x}\right|_{x=x_i,\,t=t_j} = \frac{f_{i+1,j} - f_{i,j}}{s} \qquad (5)$$

Now, we can plug this into (1) and solve for the update rule as follows:

$$\frac{\rho_{i,j+1}-\rho_{i,j}}{h} = -\frac{f_{i+1,j}-f_{i,j}}{s}$$

$$\rho_{i,j+1} = \rho_{i,j} - \frac{h}{s}\left(f_{i+1,j} - f_{i,j}\right)$$

Then using (2)

$$\rho_{i,j+1} = \rho_{i,j} - \frac{h v_{max}}{s}\left(\left(1-\frac{\rho_{i+1,j}}{\rho_{max}}\right)\rho_{i+1,j} - \left(1-\frac{\rho_{i,j}}{\rho_{max}}\right)\rho_{i,j}\right)$$

Then, we shift by 1 in time space for our update:

$$\rho_{i,j} = \rho_{i,j-1} - \frac{h v_{max}}{s}\left(\left(1-\frac{\rho_{i+1,j-1}}{\rho_{max}}\right)\rho_{i+1,j-1} - \left(1-\frac{\rho_{i,j-1}}{\rho_{max}}\right)\rho_{i,j-1}\right)$$

Lex Friedrichs:

$$\rho_{i,j+1} = \frac{\rho_{i+1,j}+\rho_{i-1,j}}{2} - \frac{h}{s}\,\frac{f_{i+1,j}-f_{i-1,j}}{2}$$

where $f_{i,j} = V_{max}\,\rho_{i,j}\left(1-\frac{\rho_{i,j}}{\rho_{max}}\right)$

Again, applying a shift in time space:

$$\rho_{i,j} = \frac{\rho_{i+1,j-1}+\rho_{i-1,j-1}}{2} - \frac{h}{s}\,\frac{f_{i+1,j-1}-f_{i-1,j-1}}{2}$$

Plugging in (2),

$$\rho_{i,j} = \frac{1}{2}\left(\rho_{i+1,j-1}+\rho_{i-1,j-1}\right) - \frac{h v_{max}}{2s}\left(\left(1-\frac{\rho_{i+1,j-1}}{\rho_{max}}\right)\rho_{i+1,j-1} - \left(1-\frac{\rho_{i-1,j-1}}{\rho_{max}}\right)\rho_{i-1,j-1}\right)$$

Note, that in code, we can store the computation of $f_{i,j}$ to avoid recomputation. It also makes the update formulas simpler.

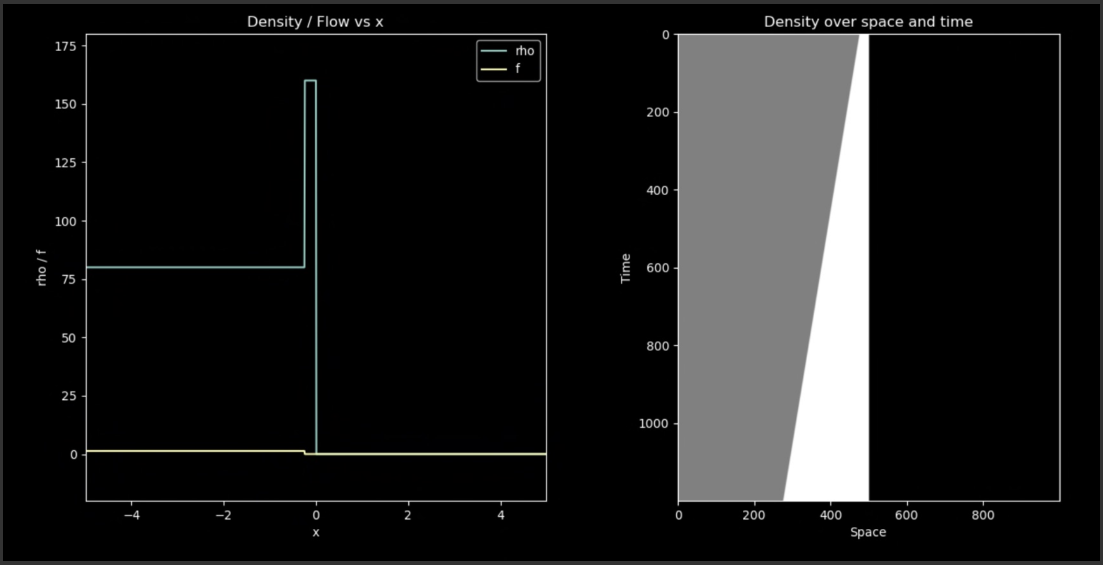
Task 1.1:

1) The implementation I've written is likely correct as it produces viable results given that the approximation is a first order Euler approximation as shown in the next question. Further, the code written breaks everything into separate modular functions. The update functions written are shown below and follow directly from the formula's written above. The PDE from above has been previously devised to model traffic and we are just using a derivative approximation on top of it.

```
def update_textbook(self, sol, j):
    rho = sol[:, j - 1]
    f = self.compute_f(rho)
    sol[:-1, j] = rho[:-1] - self.h[j - 1] / self.s * (f[1:] - f[:-1])
```

The compute_f function just calculates the flow function from the density from the previous time step (j-1). Then, the code sets the density for the current time step (j).

2) Below is the initial conditions (density and flow) of the simulation. On the right is the density heatmap over space and time. To see an animation of the left graph moving in real time through time space in the solution, run the code and it should use a matplotlib animation to show it.



3) These results make decent sense since we are just using a first order approximation. It is showing certain qualities that appear in real life as well: where the traffic jam is occurring, traffic is getting backed up from there and it seems to be propagating backwards from there. This is clear in the heatmap since the bright white region is expanding as it the time increases. These results are not that accurate since it doesn't seem to have cars move past the traffic jam and instead just keeps them stationary.

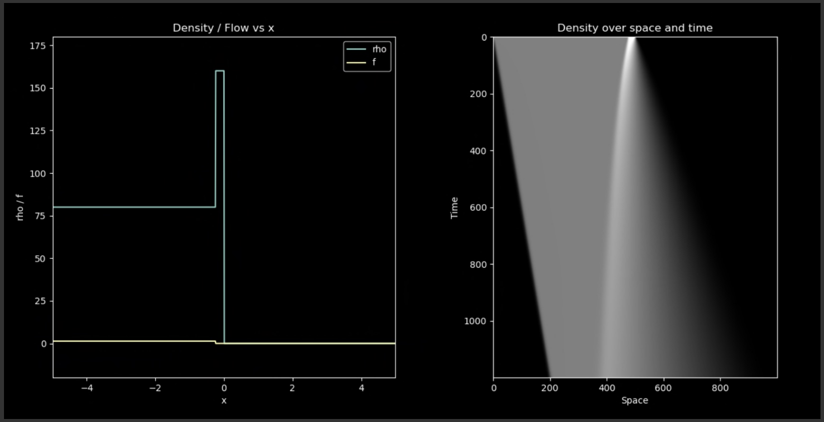
Task 1.2:

1) This implementation is likely correct since most of the code is reused from Task 1.1. The part that is different is the update function, which was written directly from the formula's above and is shown below. This is just using the same model as above, but with a better derivative approximation.

```
def update_lax_friedrichs(self, sol, j):
    rho = sol[:, j - 1]
    f = self.compute_f(rho)
    sol[1:-1, j] = (
        (rho[2:] + rho[:-2]) / 2
        - self.h[j - 1] / self.s[1:] * (
            f[2:] - f[:-2]
        ) / 2
    )
```

The details of this function are pretty similar to the update_textbook function. The compute_f function calculates the flow, which is then used to update the density.

2) Below is the initial conditions (density and flow) of the simulation. On the right is the density heatmap over space and time. To see an animation of the left graph moving in real time through time space in the solution, run the code and it should use a matplotlib animation to show it.



3) These results look better than the Euler approximation results since the traffic spreads out over time and doesn't just back up more and more over time. The density propagates forward (as it should since the cars are moving forward) as well as traffic backing up due to the traffic. Further, after the initial bit of time, the density never reaches back to the maximum density again and it instead slowly spreads out over the space, as traffic usually would without any external factors. Overall, this approximate seems to yield more accurate results than the Euler approximate and shows features of real traffic spread.

CA Model

Steps:

1) Acclerate:

$$\forall i \quad v_i := \max\{v_i + 1, v_{max}\}$$

2) Decelerate:

$$\forall i \quad v_i := \min\{d(i, i+1), v_i\}$$

3) Randomize (optional):

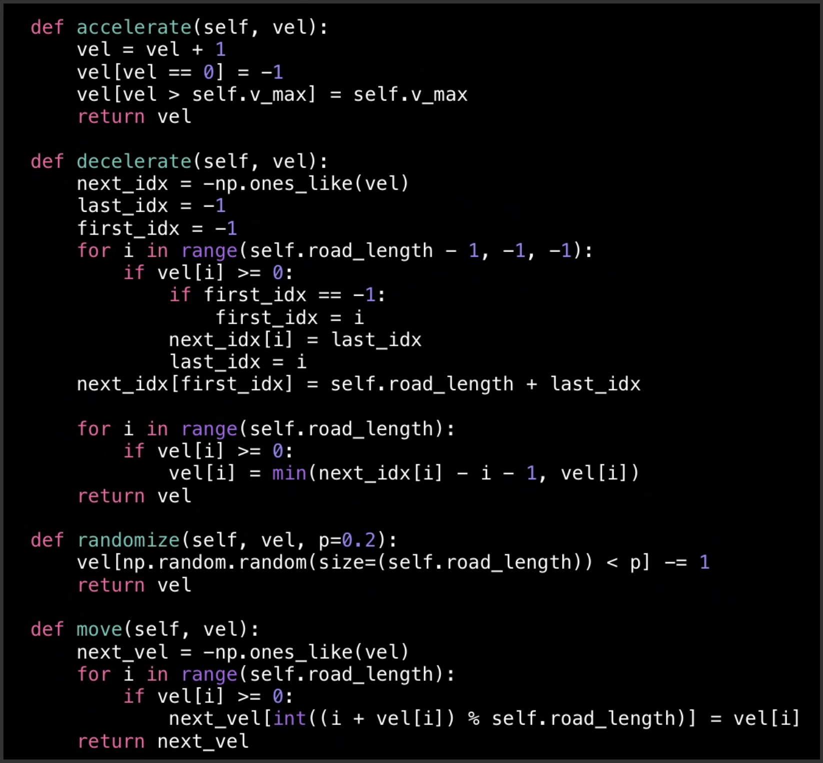
$$\forall i \quad v_i := \begin{cases} v_i - 1 & \text{with probability } p \\ v_i & \text{else} \end{cases}$$

4) Move

$$\forall i \quad v_{(i+v_i)} := v_i$$

Task 2:

1) Using a CA simulation for a situation like this makes sense as there are clear objects that interact with their neighbors. The four steps depicted above (with step 3 being optional) are all coded into the simulation as defined above and shown below:

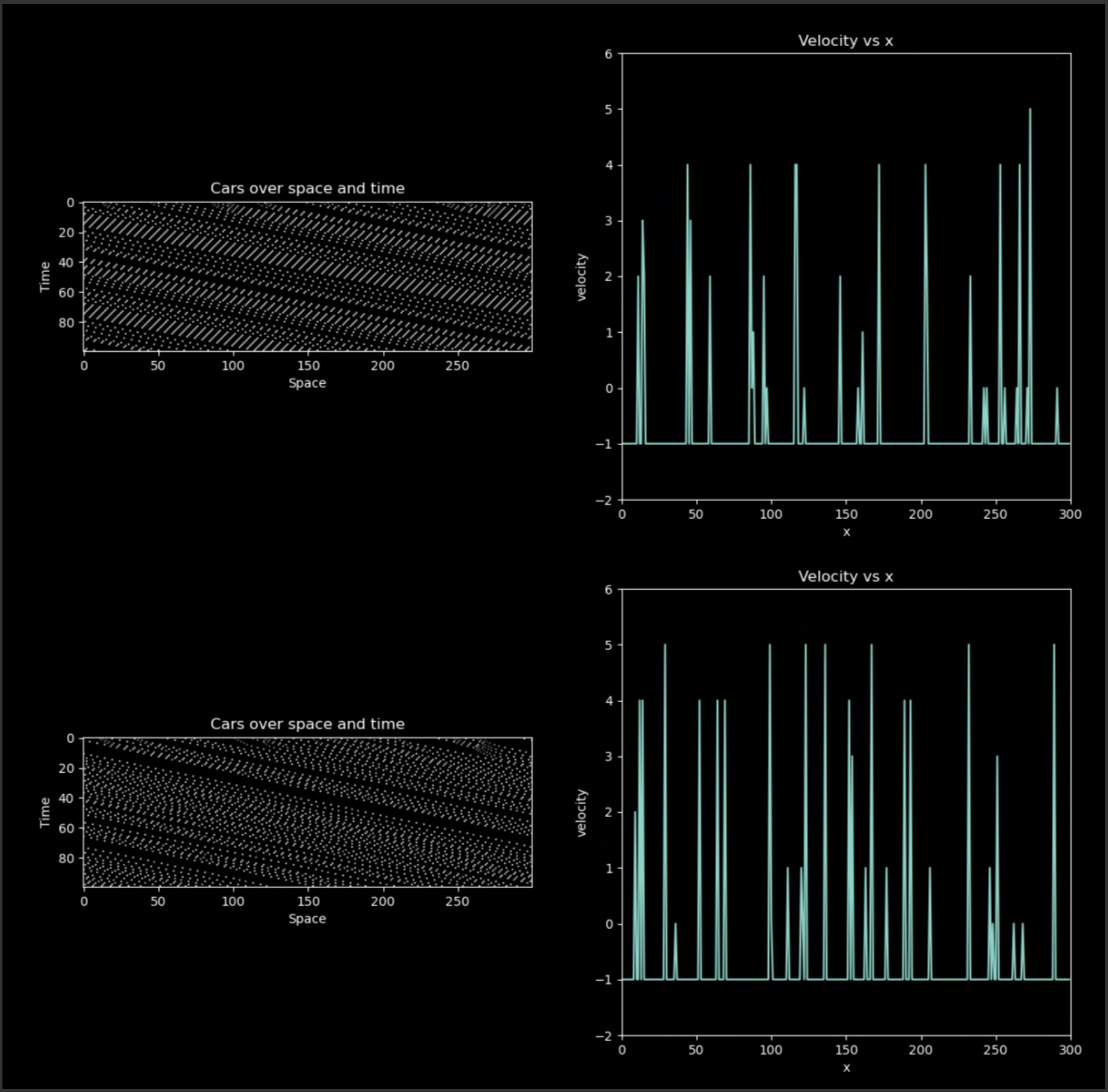


```
def accelerate(self, vel):
    vel = vel + 1
    vel[vel == 0] = -1
    vel[vel > self.v_max] = self.v_max
    return vel

def decelerate(self, vel):
    next_idx = -np.ones_like(vel)
    last_idx = -1
    first_idx = -1
    for i in range(self.road_length - 1, -1, -1):
        if vel[i] >= 0:
            if first_idx == -1:
                first_idx = i
            next_idx[i] = last_idx
            last_idx = i
    next_idx[first_idx] = self.road_length + last_idx

    for i in range(self.road_length):
        if vel[i] >= 0:
            vel[i] = min(next_idx[i] - i - 1, vel[i])
    return vel

def randomize(self, vel, p=0.2):
    vel[np.random.random(size=(self.road_length)) < p] -= 1
    return vel

def move(self, vel):
    next_vel = -np.ones_like(vel)
    for i in range(self.road_length):
        if vel[i] >= 0:
            next_vel[int((i + vel[i]) % self.road_length)] = vel[i]
    return next_vel
```

Each of these functions take in the velocity vector and then return the new velocities vector after the steps is taken. This process is then iterated on and run every time step.

Since the method described above seems like a plausible method to simulate traffic and the code to the left follows that method, this implementation's validity seems likely.

2) Below is the initial conditions (randomized initial velocities) of the simulation. On the right is the density heatmap over space and time. To see an animation of the left graph moving in real time through time space in the solution, run the code and it should use a matplotlib animation to show it. The top two graphs do not include the randomization step while the bottom two include it:



3) These results are more similar to the results from 1.2 than 1.1 since the traffic does seem to actually flow past local jams. The local jams do seem to dissipate over a little bit of time in both the randomized version and normal version of the CA models. Without the randomization step, the CA model quickly converges to a relatively stable traffic flow in which the cars are spread out enough to keep the flow of traffic moving. However, with the randomization step, there is a chance that cars decelerate for no reason, which can cause local traffic jams, which get dissipated over time quickly after they are formed. These results are pretty reasonable and match up to the expectation. Further, it also seems to have features of real traffic flow.