MATH351 Numerical Analysis Term Project

Final Report

Dongyun Kim[a], Gyumin Choi[b]

[a]**Student ID**: 20160082, **Major**: Industrial & Management Engineering, POSTECH
[b]**Student ID**: 20160595, **Major**: Mathematics, POSTECH

Abstract

Traffic flow problems are modeling problems to understand efficient movements and solve traffic congestion problems by mathematically modeling the flow of cars, vehicles, and people on the road. The purpose of this project is to model traffic flow problems using Burgers' Equation, and to solve the Inviscid Burgers' Equation numerically using Lax-Wendroff Method and Total Variation Diminishing Runge-Kutta 3rd order method (TVD-RK3) with WENO scheme.

Keywords: Traffic flow; Hyperbolic PDE; Burgers' Equation; Lax-Wendroff Method; ENO/WENO; TVD-RK3;

1 Background

In class, we have discussed about Burgers' equation, a famous differential equation that is known for showing a shock at a certain time, and simple methods to solve it numerically. It is known that Burgers' equation can be used to represent and simulate traffic flows and fluid mechanics. As a term project, we present various situations in a 1-dimensional traffic flow that resembles a one-way road traffic with cars and examinate how the flow changes along time and space. To express such solutions, we attempt to numerically solve the Burgers' equation using Lax-Wendroff Method and third order TVD Runge-Kutta using WENO scheme via Python programming.

2 Modeling

Consider the flow of cars on a road(x-axis). Let $\rho(x,t)$ be the density of cars at x, time t. Let $f(x,t)$ be the traffic flow. Since $\rho \geq 0$ has an upper bound, ρ_{max}, which is defined as the density when the road is packed with cars where each car is bumper to bumper. Thus, $0 \leq \rho \leq \rho_{max}$.

Overview To model traffic flow, we consider the following.

1. $0 \leq \rho \leq \rho_{max}$
2. A normal driver will control its speed under density; When the density is ρ_{max}, all drivers are stuck in full traffic jam so speed $v = 0$. When the density $\rho = 0$, then a driver can speed up as much as possible, up to v_{max}(speed limit).

Thus, flow f is a function of density $\rho(x,t)$ and speed v, while speed v is a function of density ρ.

We can represent v as a linear relation with ρ by considering the conditions **1, 2** above.

$$v(\rho) = -\frac{v_{max}}{\rho_{max}}\rho + v_{max} = v_{max}(1 - \frac{\rho}{\rho_{max}})$$

where $0 \leq \rho \leq \rho_{max}$.

Inviscid Burgers' equation Since cars are conserved, we use the continuity equation, where the flow is one-dimensional(the road is along x-axis) to construct the following.

$$\frac{\partial \rho}{\partial t} + \boldsymbol{\nabla} \cdot f = \frac{\partial \rho}{\partial t} + \frac{\partial f}{\partial x} = 0$$

Finally, the number of cars passing through x at time t, which is flow f, can be simply defined as $f = \rho v$. We now, can derive the following:

If we rescale the density by $\rho \leftarrow \frac{\rho}{\rho_{max}}$, we have $\rho_{max} = 1$ and $0 \leq \rho \leq 1$.

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x}(v_{max}\rho(1-\rho)) = 0,$$
$$\frac{\partial \rho}{\partial t} + v_{max}(1-2\rho)\frac{\partial \rho}{\partial x} = 0$$

We rescale $v \leftarrow \frac{v}{v_{max}}$, so $v_{max} = 1$. and use the transformation $u = 1-2\rho$. Then since $0 \leq \rho \leq 1$, $-1 \leq u \leq 1$, and we have the following.

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = 0$$

This is known as the *inviscid Burgers' equation*.

Viscid Burgers' equation We can also derive a different model by defining flow f differently.

A normal driver on the road will slow its speed when he/she expects an increase of traffic density(ρ) ahead. On the other hand, a driver will increase its speed when he/she expects a decrease of traffic density ahead.

This motivates us to modify the flow equation($f = \rho v$) as follows:

$$f = \rho v - \epsilon\frac{\partial \rho}{\partial x}$$

where ϵ is constant.

Then by the same procedure we rescale ρ and v and derive the following equation

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \epsilon\frac{\partial^2 u}{\partial x^2}$$

This is known as the *viscid Burgers' equation*.

3 Methodology

A discretized spatial form of solution $u(x,t)$ can be written as $u_j^n = u(x_j, t^n)$.
Then, the total variation(TV) for this discrete case is defined as

$$TV(u^n) = \sum_j |u_{j+1}^n - u_j^n|.$$

A numerical method is **total variation diminishing(TVD)** if $TV(u^{n+1}) \leq TV(u^n)$, i.e. total variation nonincreases as time t increases.

Recall the inviscid Burgers' equation we obtained from our traffic model.

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \epsilon\frac{\partial^2 u}{\partial x^2}$$

Notice that since we used transformation $u = 1 - 2\rho$, when we have no cars at all in the road, i.e. $\rho = 0$, the flow $f = \rho v = 0$ and $u = 1$. When the road is fulled with cars, i.e. $\rho = \rho_{max} = 1$, the flow $f = \rho v = 1 \times 0 = 0$ and $u = -1$. The flow is maximum to $f = 1/4$ when $\rho = 1/2$ and $u = 0$. This seems reasonable because in reality, the flow of cars in the road will be proportional to the speed of cars, while a density ρ too large will slow down the cars and cause a decrease in the flow.

TVD Runge-Kutta The Runge-Kutta method is a numerical method for approximating ODE problems, for example the inital value problem. Thus, in order to use this method for solving PDE, we need to first discretize the given PDE. (7) discussed a TVD Runge-Kutta type time discretization method. Using this, (2) presents the following TVD second order Runge-Kutta method, where $L(u)$ is a discrete approximate of $\mathcal{L}(u)$ of the equation $u_t = \mathcal{L}(u)$ we are considering.

$$\begin{cases} u^{(1)} = u^n + \Delta t L(u^n), \\ u^{n+1} = \frac{1}{2}u^n + \frac{1}{2}u^{(1)} + \frac{1}{2}\Delta t L(u^{(1)}). \end{cases} \tag{1}$$

Further details are available in Chapter 7.4.4. "TVD Runge-Kutta Methods Applied to Hyperbolic Conservation Laws" of Drikakis, D. and Rider, W. (2006)(1) and Chapter 3.5 "TVD Runge-Kutta" of Osher, S. and Fedkiw, R. (2006)(6). There are further issues to discuss, such as the convergence of the PDE when solving numerically by considering the CFL condition, but we shall not go further and use the schemes that are already proved to converge or experiment the convergence by Python programming ourselves.

Lax-Wendroff Method Another method presented in this paper is the Lax-Wendroff(Hellevik). This method is a numerical method for the solution of hyperboilic partial difference equations based on conservative methods. Applying for inviscid Burgers' Equation, we can write a hyperbolic partial differential equation on the following form:

$$\frac{\partial u}{\partial t} + \frac{\partial F}{\partial x} = 0 \tag{2}$$

where $F(u) = \frac{1}{2}u^2$.

The Lax-Wendroff method starts with Taylor approxiamtion of u_j^{n+1} :

$$u_j^{n+1} = u_j^n + \Delta t \left.\frac{\partial u}{\partial t}\right|_j^n + \frac{(\Delta t)^2}{2} \left.\frac{\partial^2 u}{\partial t^2}\right|_j^n + \cdots \tag{3}$$

For non-linear systems of hyperbolic PDE, the extension of the Lax-Wendroff Method, **Richtmyer two-step Lax-Wendroff method**, was suggested.

1. First step:

$$u_{j+1/2}^{n+1/2} = \frac{1}{2}(u_{j+1}^n + u_j^n) - \frac{\Delta t}{2\Delta x}(F(u_{j+1}^n) - F(u_j^n)) \tag{4}$$

$$u_{j-1/2}^{n+1/2} = \frac{1}{2}(u_j^n + u_{j-1}^n) - \frac{\Delta t}{2\Delta x}(F(u_j^n) - F(u_{j-1}^n)) \tag{5}$$

2. Second step:

$$u_j^{n+1} = u_j^n - \frac{\Delta t}{\Delta x}(F(u_{j+1/2}^{n+1/2}) - F(u_{j-1/2}^{n+1/2})) \tag{6}$$

From the partial differential equation 2, we get

$$\left.\frac{\partial u}{\partial t}\right|_j^n = -\left.\frac{\partial F}{\partial x}\right|_j^n \quad \text{and} \quad \left.\frac{\partial^2 u}{\partial t^2}\right|_j^n = -\left.\frac{\partial^2 F(u)}{\partial t \partial x}\right|_j^n = -\left.\frac{\partial(\frac{\partial F(u)}{\partial t})}{\partial x}\right|_j^n = -\left.\frac{\partial(\frac{\partial F(u)}{\partial u}\frac{\partial u}{\partial t})}{\partial x}\right|_j^n = \left.\frac{\partial(u\frac{\partial F}{\partial x})}{\partial x}\right|_j^n \tag{7}$$

Now, we can insert these terms into Taylor approximation 3, we get

$$u_j^{n+1} = u_j^n - \Delta t \frac{\partial F(u)}{\partial x} + \frac{(\Delta t)^2}{2} \frac{\partial (u \frac{\partial F}{\partial x})}{\partial x} \tag{8}$$

further, we get Lax-Wendroff method for the equation

$$u_j^{n+1} = u_j^n - \frac{\Delta t}{2\Delta x}(F_{j+1} - F_{j-1}) + \frac{\Delta t^2}{2\Delta x^2}\Big[u_{j+1/2}(F_{j+1} - F_j) - u_{j-1/2}(F_j - F_{j-1})\Big] \tag{9}$$

where $u_{j+1/2} = \frac{1}{2}(u_j^n + u_{j+1}^n)$, $u_{j-1/2} = \frac{1}{2}(u_{j-1}^n + u_j^n)$, which is simply done by averaging the neighboring values.

WENO-JS Method The Lax-Wendroff method described above is second-order accurate in both space and time. Therefore, Lax-Wendroff method may encounter oscillation problems when plotting the solution. We now discuss WENO(Weighted Essentially Non-Oscillatory) method, does not encounter such oscillation problems. WENO is used for numerical solutions of hyperbolic PDEs and is developed from ENO(Essentially Non-Oscillatory) methods. As the name "WENO(Weighted Essentially Non-Oscillatory)" shows, WENO is a numerical solution wthat is used when high accuracy is required around shocks and discontinuities. In particular, when solving the Burgers' equation, which is known for appearing a shock at a certain point, WENO methods will improve the numerical results that we earned from Lax-Wendroff Method. Of all the many WENO methods, we introduce WENO-JS scheme from Jiang, G.-S. and Shu, C.-W. (1996)(5).

Guo and Jung (2017) (3) improved WENO-JS by adopting infinitely smooth radial basis functions(RBF), which is a type of non-polynomial finite volume ENO/WENO method. In this report, we consider the original WENO-JS proposed by Jiang and Shu (1996)(5) for order 3, i.e. $r = 3$ (Figure 1). The coefficients $a_{k,l}^r, 0 \leq k, l-1$ are provided in **Table I** of Jiang and Shu (1996)(5) and will be used in the Python code.

TABLE I

Coefficients $a_{k,l}^r$

r	k	$l = 0$	$l = 1$	$l = 2$
3	0	1/3	−7/6	11/6
	1	−1/6	5/6	1/3
	2	1/3	5/6	−1/6

Figure 1: Table I for $r = 3$ (Jiang and Shu, 1996)

About our python code Our final python code uses WENO-JS scheme to discretize $\mathcal{L}(u)$ of the equation $u_t = \mathcal{L}(u)$ using the coefficients by WENO-JS provided in Table I of Jiang and Shu (1996)(5). Then, instead of the TVD second order Runge-Kutta method in Equation 1, we use the third order TVD Runge-Kutta which can be represented as the following.

$$\begin{cases} u^{(1)} = u^n + \Delta t L(u^n), \\ u^{(2)} = \frac{3}{4}u^n + \frac{1}{4}u^{(1)} + \frac{1}{4}\Delta t L(u^{(1)}), \\ u^{n+1} = \frac{1}{3}u^n + \frac{2}{3}u^{(2)} + \frac{2}{3}\Delta t L(u^{(2)}). \end{cases} \tag{10}$$

The next section discusses the results of Python programming when implementing Lax-Wendroff method and RK-3 with discretization of WENO-JS schemes. It turns out that the latter solves the inaccurate ocsillation issues that the former suffers. Note that we shall indicate WENO-JS as WENO for the rest of the report since there is no other WENO method that we are referring to.

4 Simulation Results

We simulated Burgers' Equation using Equation 6(Appendix A).

First of all, we simulated continuous initial condition case with $u(x, 0) = -\sin(\pi x)$, which were frequently used in class. With Lax-Wendroff method(Appendix B), we were able to get a better result than we get one method in class. Please see Figure 2.

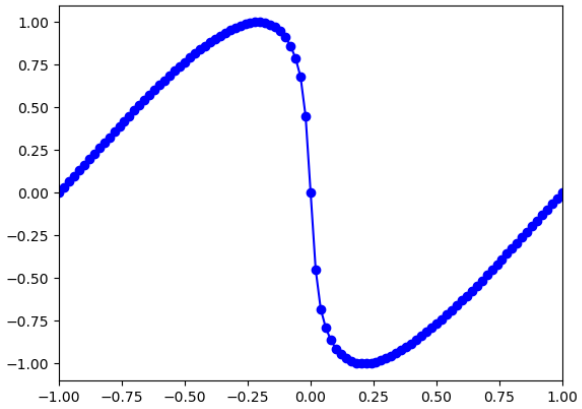
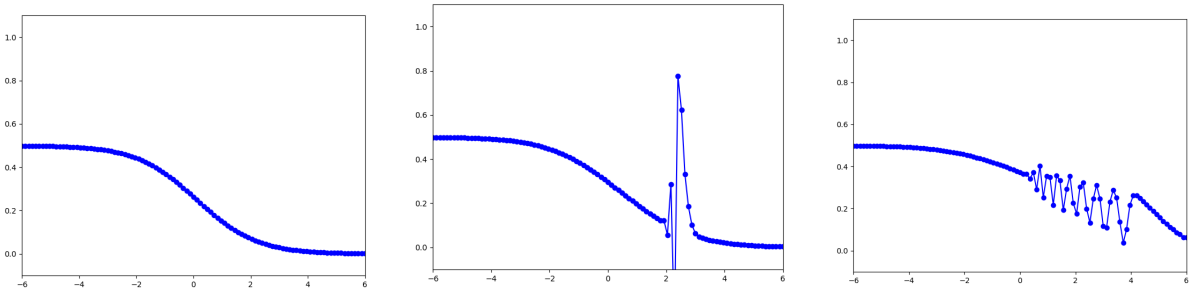


Figure 2: Burgers' Equation with continuous IC(Lax-Wendroff)

However, assuming traffic flow, the initial condition for discontinuous initial condition cases resulted in an error in which all values disappeared, caused by unknown errors such as overflow(Appendix C). On the other hand, when using the code in class, shock appeared in three places, but the values of the graph were not lost and plotted well(Appendix D). It turns out that there was error in code(Appendix C) when dealing with the boundary condition. After fixing the boundary as `u[0] = u[1], u[N] = u[N-1]`, our method worked well without disappearing phenomenon. After solving this problem, as we transformed the density of cars with $u = 1 - 2\rho$, we need to transform again to see the traffic flow intuitively.

4.1 Scenario 1 - Lax-Wendroff Method

Initial condition is $1 - 2\rho = u = sigmoid(x) = \frac{1}{1+e^{-x}}$, which represented the scenario that traffic is congested before $x < 0$, and relatively less congested after $x > 0$. Meanwhile, the traffic is accidentally stucked at certain section, say $72 \leq x \leq 74$. This scenario was simulated as shown in Figure 3 below. Now, we plotted $x - \rho$ graph. Please see the code in Appendix E.



(a) Initial (b) Accidentally Stuck (c) final

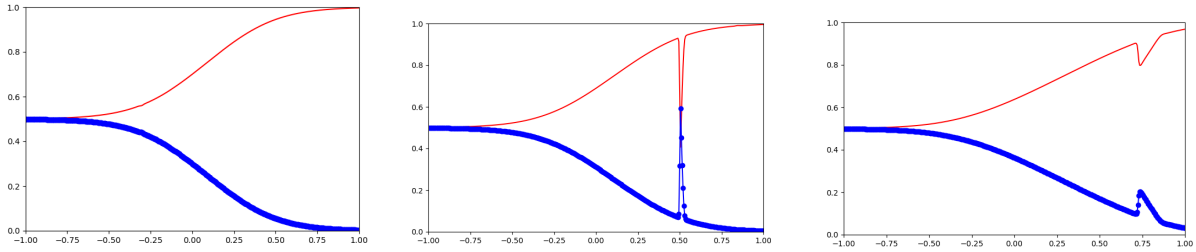
Figure 3: Traffic Flow simulation with sigmoid IC and accidentally stuck

However, because the Lax-Wendroff method was second-order accruate in both space and time, oscillation was very severe in the congested area. Also, it has not been successfully explained physically. Therefore, we simulated the same scenario using the WENO method discussed above.

4.2 Scenario 1 - WENO Method

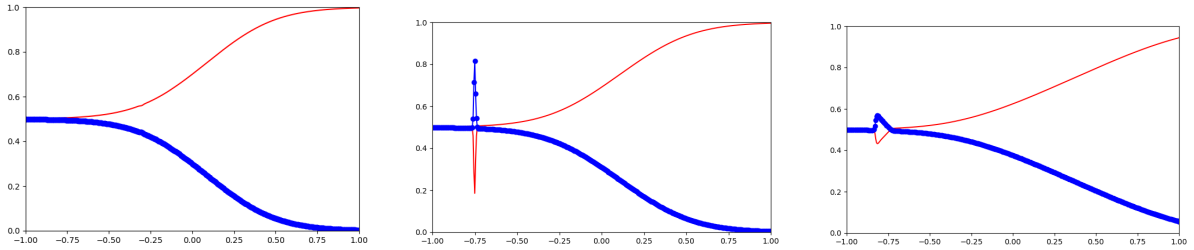
To fix the range of x to $[-1, 1]$, we changed the sigmoid function to $\frac{1}{1+e^{-6x}}$ which is initial condition, and simulated a sudden congestion at $150 \leq x \leq 200$, with $N = 400$. Here, the blue line represents the density of cars, ρ, and the red line represents the speed of cars, v. The higher the density of cars, the lower the speed, and the lower the density of cars, the faster the speed. We used python code(Appendix F, G) to simulate scenarios with WENO method, while the code was provided by Prof. Jung. Other scenarios were implemented using variations of the code in the **Appendix**.

It was assumed that there was a temporary congestion due to a sudden obstacle, and the pattern of progress was different depending on where this congestion occurred. When congestion occurred in a low-density area, the congestion progressed forward due to the fast movement of surrounding cars(Figure 4). On the other hand, in case of congestion in dense areas, congestion gradually progressed backward.(Figure 5). However, in both cases, it was observed that the congestion was resolved relatively quickly because the congestion was at a narrow point.



(a) Initial (b) Sudden congestion (c) Congestion proceeds forward

Figure 4: Scenario 1 with WENO method(low density area)



(a) Initial (b) Sudden congestion (c) Congestion proceeds backward

Figure 5: Scenario 1 with WENO method(high density area)

4.3 Scenario 2

In Scenario 2, the range of the congested area was increased, and the simulation was conducted assuming a congested situation from the beginning. Similar to the situation in which an obstacle suddenly appeared in a narrow area, the congestion was resolved as the congestion section progressed backwards(Figure 6).

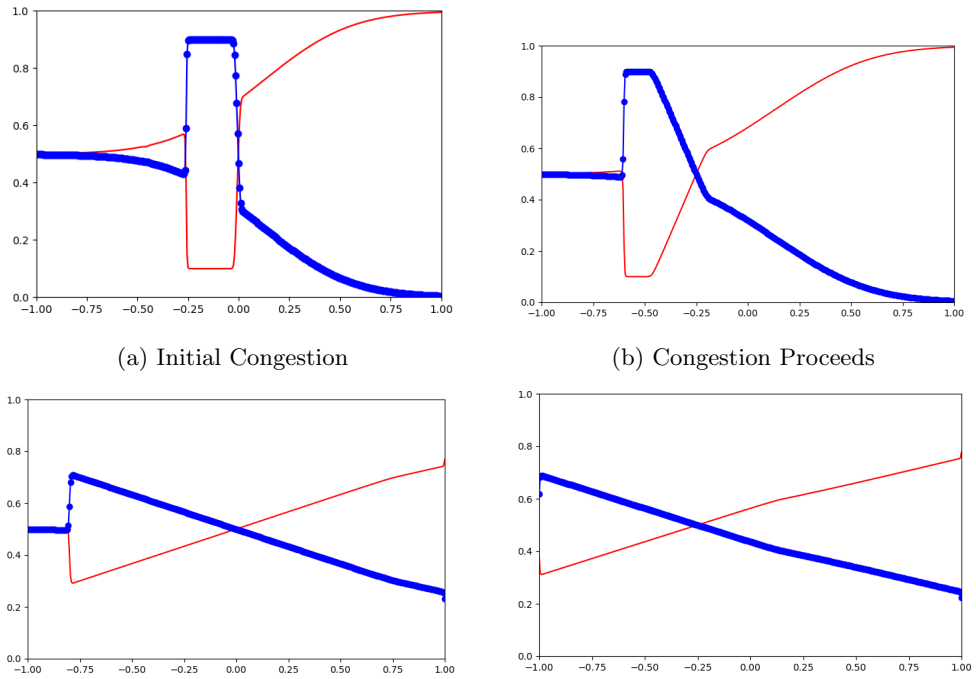
4.4 Scenario 3

In Scenario 3, the sigmoid function, which is the initial condition so far, has been changed into $\frac{2}{1+e^{6x}}$. It has been transformed to be symmetric on the y-axis, so congestion zone was set to the range of $x \geq 0$, and simulated assuming that the congestion has intensified. As the simulation progressed, we were able to observe Shock($x = 0$), a characteristic of Burgers' Equation, and subsequently observed that the congestion was resolved(Figure 7).

4.5 Scenario 4

We modeled a circular road by fixing the first and last boundaries equally, and setting initial condition function to $u(x, 0) = \cos(\pi x)$. It is unknown whether because we artificially fixed the boundary condition or because the method of handling the boundary of the WENO method is not appropriate, but a strange hook-shaped shape appears in the beginning. However, shock congestion is soon observed($x \approx -0.55$), and a physical phenomenon

assuming a normally circular situation appears. At the end, it can be seen that the sudden stagnation almost disappears(Figure 8).



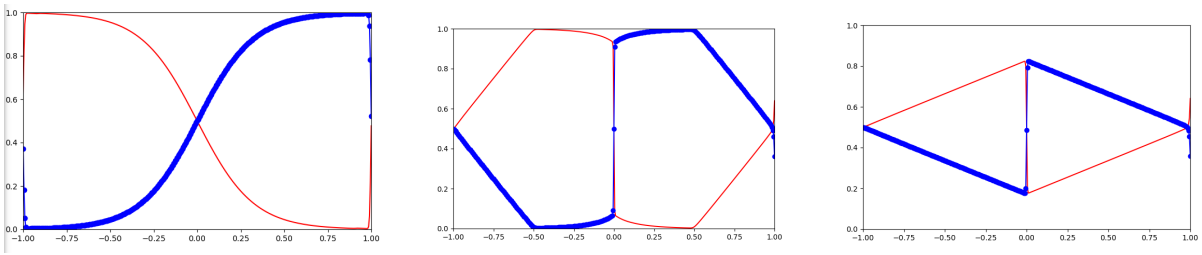
(a) Initial Congestion

(b) Congestion Proceeds

(c) Congestion resolves

(d) Final

Figure 6: Scenario 2 with WENO method

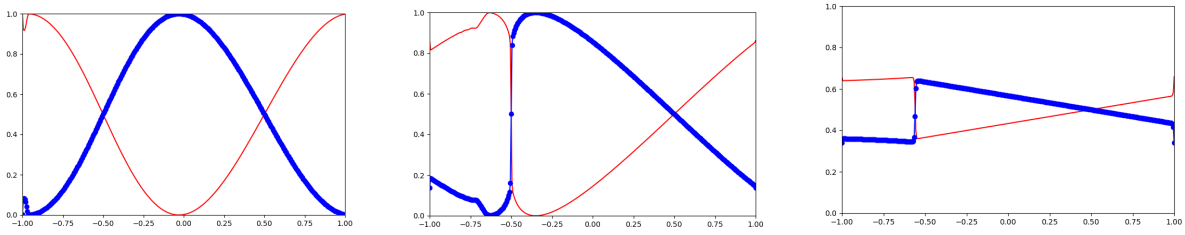


(a) Initial

(b) Proceeding

(c) Final

Figure 7: Scenario 3 with WENO method



(a) Initial

(b) Proceeding

(c) Final

Figure 8: Scenario 4 with WENO method(Circular Road)

5 Discussion

In this paper, we mathematically modeled the traffic flow, and converted it to Burgers Equation, and simulated it in Python programming through a numerical method, such as Lax-Wendroff and RK-3 with WENO. During our simulation, various scenarios were simulated, while a sigmoid function was used as an initial condition to simulate a situation in which congestion occurs due to an obstacle in the middle, and a situation that starts from a stagnant state, etc. In the case of a situation where there was a sudden congestion in the middle, the direction of the proceeding congestion was different depending on whether the congestion occurred in a place with high or low density of surrounding cars. When simulating the case where the congestion gets worse (Scenario 3), it can be observed that the simulation is similar to the real situation, such as the shock appears and then the congestion is gradually resolved. It was also possible to simulate a round road by fixing the boundary. However, by forcibly fixing the boundary, or by the method of handling the boundary of the WENO method, little anomaly around the boundary occurred which is difficult to explain. We should be able to solve these boundary problems in the future. The results and ideas we have earned can be applied in real-life traffic simulations and predict the locations of traffic congestion for a certain time.

Acknowledgments The authors thank Professor Jung, J. H for introducing his paper (3) and a better method to numerically solve our traffic flow PDE and kindly advising throughout this project.

References

Drikakis, D. and Rider, W. (2006). *High-resolution methods for incompressible and low-speed flows*. Springer Science & Business Media.

Gottlieb, S. and Shu, C.-W. (1998). Total variation diminishing runge-kutta schemes. *Mathematics of computation*, 67(221):73–85.

Guo, J. and Jung, J.-H. (2017). Radial basis function eno and weno finite difference methods based on the optimization of shape parameters. *Journal of Scientific Computing*, 70(2):551–575.

Hellevik, L. R. Numerical methods for engineers.

Jiang, G.-S. and Shu, C.-W. (1996). Efficient implementation of weighted eno schemes. *Journal of computational physics*, 126(1):202–228.

Osher, S. and Fedkiw, R. (2006). *Level set methods and dynamic implicit surfaces*, volume 153. Springer Science & Business Media.

Shu, C.-W. and Osher, S. (1988). Efficient implementation of essentially non-oscillatory shock-capturing schemes. *Journal of Computational Physics*, 77(2):439–471.

Appendix A Lax-Wendroff Method

```
from numpy import *
%pylab

def F(u):
    return 0.5*u**2

def Lax_W_Two_Step(u):
    """method that solves u(n+1), for the scalar conservation equation with source term:
        du/dt + dF/dx = 0,
        where F = 0.5u^2 for the burger equation
        with use of the Two-step Lax-Wendroff scheme

        Args:
            u(array): an array containing the previous solution of u, u(n).
        Returns:
            u[1:-1](array): the solution of the interior nodes for the next timestep, u(n+1).
    """
    ujm = u[:-2].copy() #u(j-1)
    uj = u[1:-1].copy() #u(j)
    ujp = u[2:].copy() #u(j+1)
    up_m = 0.5*(ujm + uj) - 0.5*(dt/dx)*(F(uj)-F(ujm))
    up_p = 0.5*(uj + ujp) - 0.5*(dt/dx)*(F(ujp)-F(uj))

    u[1:-1] = uj -(dt/dx)*(F(up_p) - F(up_m))
    return u[1:-1]
```

Appendix B Burger's Equation with IC $= -sin(\pi x)$

```
## Inviscid Burger's Equation(Lax_W_Two_Step)

N = 100
x = linspace(-1, 1, N+1)
u = - sin(pi*x)

dx=2/N
dt=0.1*dx
time=0

for k in range(150):
    time = time + dt
    u[1:-1] = Lax_W_Two_Step(u)
    u[0] = 0.5*(u[1]**2-u[N-1]**2)/dx
    u[N] = u[0]
    clf()
    plot(x, u, '-bo')
    axis((-1, 1, -1.1, 1.1))
    pause(0.1)
```

Appendix C Burger's Equation with Discontinuous IC (Error in Boundary Condition)

```
# Traffic Flow(Discontinuous, Lax_W_Two_Step)

N = 100
x = linspace(-1, 1, N+1)
u = x.copy()
u[(u >= 0)] = 0.8
u[(u < 0)] = 0

dx=2/N
```

```
dt=0.1*dx
time=0

for k in range(150):
    time = time + dt
    u[1:-1] = Lax_W_Two_Step(u)
    u[0] = 0.5*(u[1]**2-u[N-1]**2)/dx
    u[N] = u[0]
    clf()
    plot(x, u, '-bo')
    axis((-1, 1, -1.1, 1.1))
    pause(0.1)
```

Appendix D Burger's Equation with Discontinuous IC (In-class Manner)

```
# Traffic Flow (Discontinuous, In class)

N=100
x=linspace(-1, 1, N+1)
u = x.copy()
u[(u >= 0)] = 0.8
u[(u < 0)] = 0


dudx=u*0

h=2/N
dt=0.1*h
time=0

for k in range(150):
    time = time + dt
    for i in range(1, N):
        dudx[i] = 0.5*(u[i+1]**2-u[i-1]**2)/2/h
    dudx[0] = 0.5*(u[1]**2-u[N-1]**2)/2/h
    dudx[N] = dudx[0]
    u = u - dt*dudx
    clf()
    plot(x, u, '-bo')
    axis((-1, 1, -1.1, 1.1))
    pause(0.1)
```

Appendix E Traffic Flow Scenario 1

```
# Traffic Flow(Scenario 1)

N = 100
x = linspace(-6, 6, N+1)
u = 1/(1+exp(-x))
rho = (1 - u) / 2

dx=2/N
dt=0.1*dx
time=0

for k in range(350):
    time = time + dt
    u[1:-1] = Lax_W_Two_Step(u)
    u[0] = u[1]
    u[N] = u[N-1]
    if k == 50:
```

```
        u[70:72] = -0.5
    clf()
    rho = (1 - u) / 2
    plot(x, rho, '-bo')
    axis((-6, 6, -0.1, 1.1))
    pause(0.1)
```

Appendix F Traffic Flow Scenario 1 - WENO Method

```
%matplotlib qt

plt.figure(figsize=(8, 5))

rho = (1 - 1/(1+exp(-6*x)))/2
ub = 1 - 2*rho

time = 0
index = 0
for i in range(0, 10*int(time_step) + 1):
    y0 = ub

    # RK 1
    vm, _ = WENO3(vb_plus, N)
    _, vp = WENO3(vb_minus, N)

    fminus = vm[1:N + 2] + vp[2:N + 3]
    fplus = vm[0:N + 1] + vp[1:N + 2]

    flux = -(fminus - fplus) / dx
    y1 = y0 + dt * flux
    ub = y1
    vb_plus[3:N + 4] = 1/2 * (1/2 * ub ** 2 + ub)
    vb_minus[3:N + 4] = 1/2 * (1/2 * ub ** 2 - ub)

    # RK 2
    vm, _ = WENO3(vb_plus, N)
    _, vp = WENO3(vb_minus, N)

    fminus = vm[1:N + 2] + vp[2:N + 3]
    fplus = vm[0:N + 1] + vp[1:N + 2]

    flux = -(fminus - fplus) / dx
    y1 = 3/4 * y0 + 1/4 * (ub + dt * flux)
    ub = y1
    vb_plus[3:N + 4] = 1/2 * (1/2 * ub ** 2 + ub)
    vb_minus[3:N + 4] = 1/2 * (1/2 * ub ** 2 - ub)

    # RK 3
    vm, _ = WENO3(vb_plus, N)
    _, vp = WENO3(vb_minus, N)

    fminus = vm[1:N + 2] + vp[2:N + 3]
    fplus = vm[0:N + 1] + vp[1:N + 2]

    flux = -(fminus - fplus) / dx
    y1 = 1/3 * y0 + 2/3 * (ub + dt * flux)
    ub = y1
    vb_plus[3:N + 4] = 1/2 * (1/2 * ub ** 2 + ub)
    vb_minus[3:N + 4] = 1/2 * (1/2 * ub ** 2 - ub)


    time += dt
    clf()
```

```
    rho = (1 - ub)/2
    vel = 1 - rho
    index = index + 1
    if index % 5 == 0:
        plt.plot(x, vel, '-r', x, rho,'-ob')
        plt.axis((-1,1,0,1))
        pause(0.1)
    if index == 100:
        rho[300:302] = 0.9
        ub = 1 - 2*rho
```

Appendix G WENO3

```
def WENO3(vb, N):
    """
    WENO-JS scheme with k=3

    references
    ----------
    [Jingyang Guo and Jae-Hun Jung, 2016](https://arxiv.org/abs/1602.00183)
    """
    d0 = 3/10
    d1 = 3/5
    d2 = 1/10
    d0_tilda = d2
    d1_tilda = d1
    d2_tilda = d0
    epsilon = 1e-6

    vm = np.zeros(shape=N + 3)
    vp = np.zeros(shape=N + 3)

    for i in range(N + 3):
        vm0 = (1/3) * vb[i+2] + (5/6) * vb[i+3] + (-1/6) * vb[i+4]
        vm1 = (-1/6) * vb[i+1] + (5/6) * vb[i+2] + (1/3) * vb[i+3]
        vm2 = (1/3)  * vb[i] + (-7/6) * vb[i+1] + (11/6) * vb[i+2]

        vp0 = (11/6) * vb[i+2] + (-7/6) * vb[i+3] + (1/3) * vb[i+4]
        vp1 = (1/3) * vb[i+1] + (5/6) * vb[i+2] + (-1/6) * vb[i+3]
        vp2 = (-1/6) * vb[i] + (5/6) * vb[i+1] + (1/3) * vb[i+2]

        beta0 = 13/12 * (vb[i+2] - 2 * vb[i+3] + vb[i+4]) ** 2
        + 1/4 * (3 * vb[i+2] - 4 * vb[i+3] + vb[i+4]) ** 2
        beta1 = 13/12 * (vb[i+1] - 2 * vb[i+2] + vb[i+3]) ** 2
        + 1/4 * (vb[i+1] - vb[i+3]) ** 2
        beta2 = 13/12 * (vb[i] - 2 * vb[i+1] + vb[i+2]) ** 2
        + 1/4 * (vb[i] - 4 * vb[i+1] + 3 * vb[i+2]) ** 2

        alpha0 = d0 / (beta0 + epsilon) ** 2
        alpha1 = d1 / (beta1 + epsilon) ** 2
        alpha2 = d2 / (beta2 + epsilon) ** 2
        alpha = alpha0 + alpha1 + alpha2

        alpha0_tilda = d0_tilda / (beta0 + epsilon) ** 2
        alpha1_tilda = d1_tilda / (beta1 + epsilon) ** 2
        alpha2_tilda = d2_tilda / (beta2 + epsilon) ** 2
        alpha_tilda = alpha0_tilda + alpha1_tilda + alpha2_tilda


        w0 = alpha0 / alpha
        w1 = alpha1 / alpha
        w2 = alpha2 / alpha
```

```
        w0_tilda = alpha0_tilda / alpha_tilda
        w1_tilda = alpha1_tilda / alpha_tilda
        w2_tilda = alpha2_tilda / alpha_tilda

        vm[i] = w0 * vm0 + w1 * vm1 + w2 * vm2
        vp[i] = w0_tilda * vp0 + w1_tilda * vp1 + w2_tilda * vp2

    return vm, vp
```