Modelling Traffic Flow

Lily Beech, Helena Dawson, Robert Snell, Yulun Wu, Yi Xu

March 2024

Executive Summary

This report investigates numerous ways that traffic flow can be modelled and predicted. The results will be used to find the value of parameters, such as speed and density, that maximise the traffic flow. It looks at several models such as cellular automata models, car following models and a classical model. We compare these models to accurately investigate conditions that cause the most traffic congestion. For each model we set assumptions, then vary the parameters such as traffic density, speeds, and the conditions to cause congestion, to see the validity and limitations of the models being studied.

We found that for the cellular automata single lane open road model, a density of approximately 0.2 cars per site maximises the traffic flow whereas the two lane model gives a density of 0.1 cars per site which maximises traffic flow. For the car-following model a density of 50 cars per km is ideal for optimised traffic flow. We then obtained speed density relations in both models to use in the classical model and simulate how traffic moves for this last model. For the classical model, we then obtain a density of around 90 cars per km, which maximises the traffic flow.

We find that the model with the most realistic set of rules is the car-following model so these results should be the most accurate when recommending parameters to maximise the traffic flow.

Contents

1 Introduction

Traffic flow affects the lives of millions around the globe every day. The management of traffic flow is challenging but crucial for managing the safety of drivers and passengers . There are a range of influences, such as traffic lights, vehicle features and driver behaviour that all affect how traffic flows.

Mathematical modelling is a key part of understanding and predicting traffic flow. This report explores two microscopic models which are the cellular automaton model and the car-following model. The macroscopic model is the classical model.

The cellular automaton model splits the road into cells that can either be empty or occupied by a car. It uses a set of rules to update the car's position and velocity at discrete time steps. The car-following method uses an ordinary differential equation to describe a car's motion depending on the behaviour of the car ahead, for example, its velocity and position. By extending these rules to a line of cars and creating a system of coupled ODEs, this simulates traffic. Lastly, the classical model uses a macroscopic approach to predict traffic flow behaviour, by the use of partial differential equations for averaged quantities, including the traffic density and flow rate.

All three of these models are effective at predicting realistic traffic flow. Throughout this report we examine these models' applications in real-world scenarios by analysing and comparing results from each model, and how varying parameters can affect the models' outputs.

2 Cellular Automata Method

2.1 Introduction of the CA Method

This section introduces the Cellular Automaton (CA) method, an influential computational model for simulating complex systems, in this case traffic flow. Initially conceptualised by Stanislaw Ulam and John von Neumann in the 1940s [1] for modelling biological systems, CA has since expanded as a versatile tool across various disciplines, including computer science, physics, and biology.

In traffic flow simulation, the CA method models the road as a grid, with each cell representing a segment that may either be occupied by a vehicle or left empty. The state of each cell is updated at discrete time intervals based on predefined rules reflecting the conditions of adjacent cells. This methodology facilitates the examination of traffic patterns and the effects of individual vehicle interactions on overall traffic behavior.

This section will explore the principles underlying the CA models, starting with a basic single-lane model and examining both closed and open systems, and then expanding to more complex scenarios, including two-lane traffic models with symmetric and asymmetric rules. The exploration of these models aims to provide a comprehensive understanding of traffic dynamics and offer potential solutions for real-world traffic issues.

2.2 Single-Lane Model

2.2.1 Model

The single-lane model is described as a one-dimensional lattice of L sites, with either open or periodic boundary conditions applied. Each site can be occupied by a single vehicle or remain vacant. Vehicles are assigned integer velocities within a range from zero to a predefined maximum v_{max}.

During a simulation, the system updates through four synchronized steps for all vehicles [2]:

1. **Acceleration**: If a vehicle's current speed is below v_{max} and the gap ahead exceeds its speed plus one, it accelerates by one unit.
2. **Deceleration**: Vehicles reduce speed to one less than the gap to the next vehicle ahead, provided this gap is less than their current speed, ensuring a safe following distance and preventing collisions.
3. **Randomisation**: Each vehicle may decelerate by one unit with a fixed probability, simulating random braking due to various factors.

4. **Movement**: Vehicles advance according to their updated speeds, reflecting their motion on the road.

These steps collectively model the dynamics of traffic flow on a single lane, capturing key behaviors such as acceleration, braking, and random fluctuations in speed. Each step is crucial. Step 1 simulates the natural desire of drivers to move as fast as allowed (up to v_{max}) when there is sufficient space ahead. It reflects how drivers increase their speed when the road ahead is clear. Step 2 models safety measures taken by drivers to prevent accidents. If a vehicle is too close to the one ahead (closer than its current speed), it slows down to avoid a collision, maintaining a safe distance. Step 3 introduces variability and mimics real-life driving conditions, where vehicles might slow down due to factors such as weather, road conditions, or driver distraction. This step is of vital importance since without this randomness, every initial configuration of vehicles and their corresponding velocities reaches a stationary pattern very quickly. Step 4 represents the actual movement of vehicles. After adjusting their speeds through the first three steps, vehicles move forward according to their new velocities, simulating the traffic flow on the road.

2.2.2 Traffic in Closed System

Method

In this section, we consider traffic flow in a closed system, which means vehicles arriving at the end of the road will re-enter from the beginning of the road, creating a continuous circular driving pattern. This can be used to model things such as car races as the road goes back around. However, this is a single lane model so we do not consider overtaking. We define a constant system density as [2]

$$\rho = \frac{N}{L} = \frac{\text{Number of cars in the loop}}{\text{Number of sites in the loop}} \tag{1}$$

where essentially, the longer the road, the more sites it will have. However, this formula doesn't really work out in reality. Thus, we introduce another way to meausure densities in our simulation. The densities (occupancies) $\bar{\rho}^T$ can be calculated by averaging over a time period T at a fixed site i.

$$\bar{\rho}^T = \frac{1}{T}\sum_{t=t_0+1}^{t_0+T} n_i(t) \tag{2}$$

where $n_i(t)$ is 0 if the site i is empty and $n_i(t)$ is 1 if the site i is occupied. As T tends to infinity, $\bar{\rho}^T$ tends to ρ. The time-averaged traffic flow between site i and $i+1$ is defined as

$$\bar{q}^T = \frac{1}{T}\sum_{t=t_0+1}^{t_0+T} n_{i,i+1}(t) \tag{3}$$

where $n_{i,i+1}(t)$ is 1 if there are cars moving from site i to $i+1$.

Using these definitions, we simulate models of different densities in Python (all simulations in this section use the above definitions).

Simulation

Below in **Figure 1**, we have modelled position against time with each car, represented as a dot, travelling with a different velocity indicated by the grey scale. Cars travelling at a high velocities are darker in colour whereas cars travelling at low velocities are a lighter grey. For this simulation we have chosen a density of 0.3 cars per site. We further show in **Figure 2** how each individual car will move with each colour line representing a different car travelling through time.

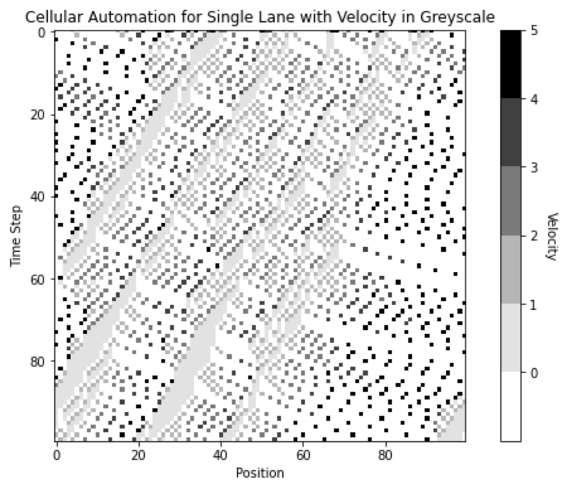


Figure 1: Velocity in Grey-scale

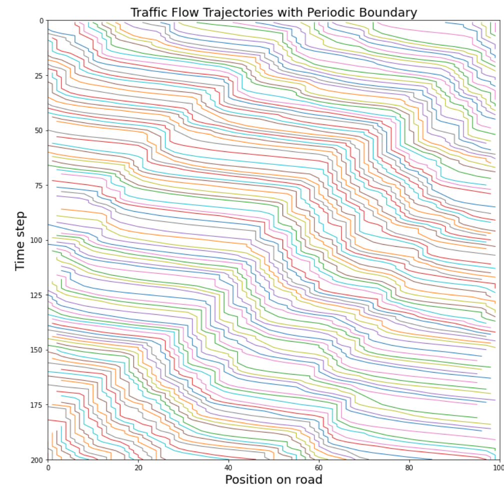


Figure 2: Trajectories of cars

As we can see from **Figure 1**, when there is a significant density of cars on the road, traffic jams start to form. The light coloured diagonal lines represent traffic jams due to cars ahead slowing down which creates a chain reaction, causing further cars to slow down. This results in a traffic jam. The diagram shows multiple traffic jams on the single lane model. This could be due to the fact that the cars can not overtake or change lanes if the car ahead slows down so congestion is more likely to form as these cars would have no choice but to also slow down. We investigate later in the report if the two lane model makes a difference to the amount of traffic jams that occur.

In **Figure 2** we see that when, the car furthest ahead starts to slow down the cars behind in turn also slow down and may even stop. Through time this will then cause a traffic jam. When cars come to a complete stop the trajectories face directly downwards. This is how we can identify when a traffic jam forms.

We now are interested in investigating the differences in traffic when there is a low density of cars compared to when there is a higher density of cars on the road. Below in **Figure 3** we model the traffic flow when the density is low at 0.1 cars per site and when the density is high at 0.7 cars per site.

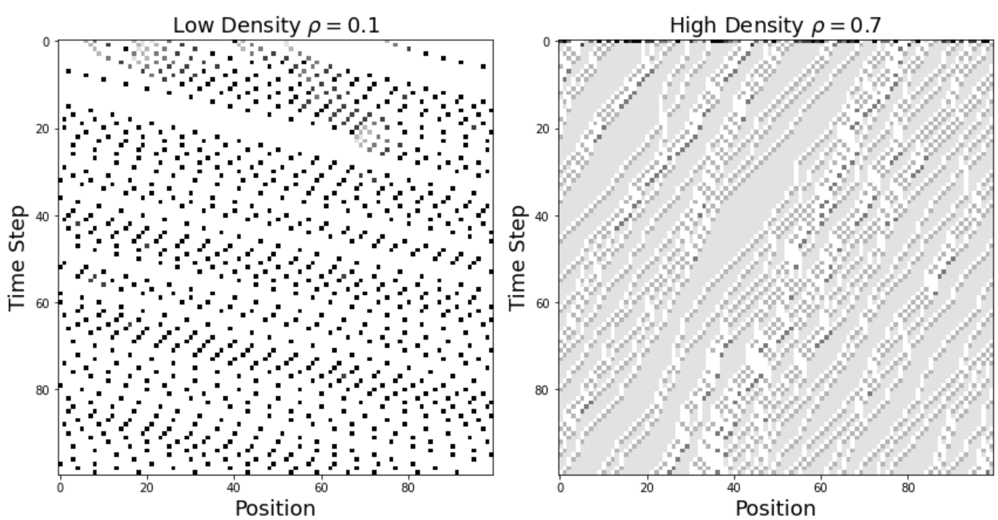


Figure 3: Traffic flow under low and high density

In **Figure 3** we observe that in the graph with density equal to 0.1 there is very little congestion with a small traffic jam between time step 0-20 and position 40-60. After this the cars involved in the traffic jam continue to move at mostly constant and higher speeds with a large enough headway to the car in front so that there is no need to slow down.

However in the graph with density set to 0.7 there is a large amount of congestion with many traffic jams forming. This suggests that when there are a lot of cars on the road, for example at peak travel time, there will be more congestion. This causes traffic to flow at a much slower pace which is reiterated by the graph as the cars are represented by light grey coloured dots, showing their slower velocities. This is therefore what we would expect in accordance to real life data [3] as in rush hour the high density of cars on the road causes more traffic frequent traffic jams.

Analysis

We now want to investigate at what traffic densities is the traffic flow maximised. Below in **Figure 4** we model density against traffic flow.

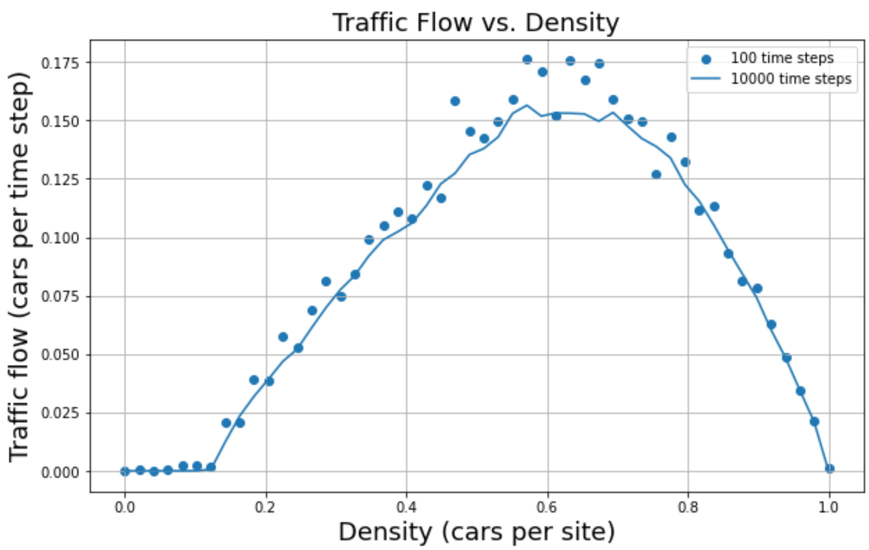


Figure 4: Traffic Flow vs. Density

Figure 4 above shows that initially, increasing the density will cause the traffic flow to also increase due to a larger number of cars passing through per time step. However when the density is significantly high traffic jams begin to form, as shown previously in **Figure 3**. As density continues increasing, more and more jams occur, creating congestion and hence slowing down the overall movement of traffic. This results in the traffic flow decreasing and is what causes the parabola seen above. In **Figure 4** we see that the traffic flow is maximised when the traffic density is at about 0.6 cars per site. This is an important result as we can see for single lane traffic, modellers should aim for the traffic density to average 0.6 cars per site and can investigate ways to reduce density of cars on the road if the density is larger than 0.6. Methods to do this could be maximising public transport options at peak times.

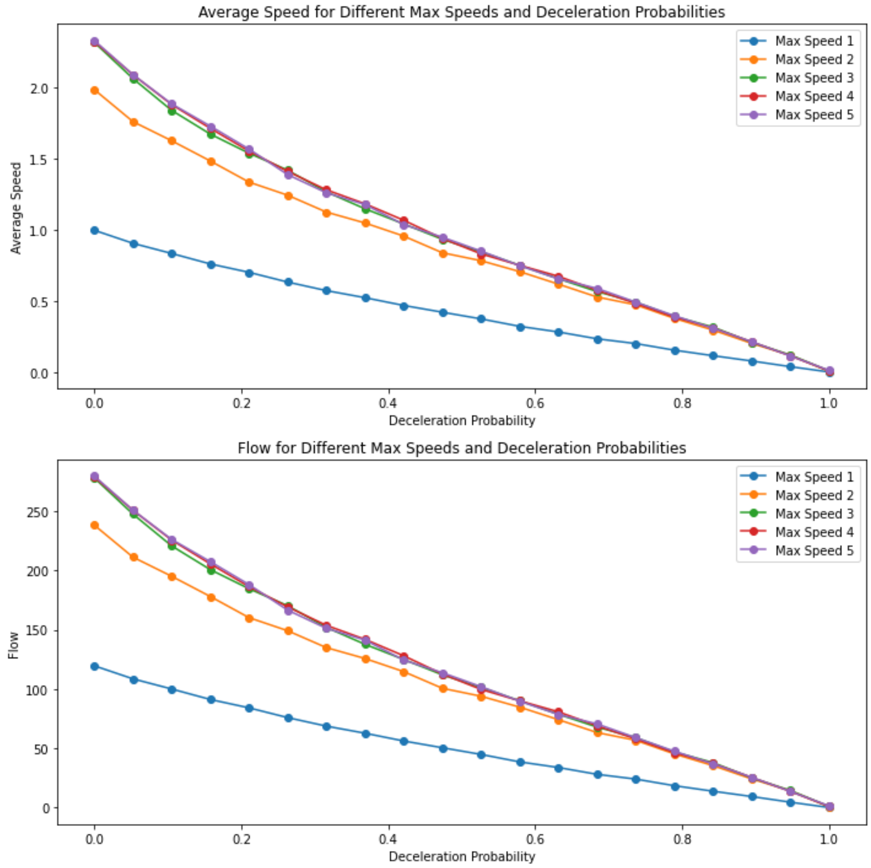


Figure 5: Analysis of V_{max} and DecProb

Figure 5 presents firstly the relationship between deceleration probability and average velocity and then deceleration probability vs flow rate, each under varying maximum speeds.

Average Speed Analysis:

- Increasing deceleration probability uniformly reduces average speed across all maximum speed settings, due to enhanced vehicle interactions and consequent speed reduction.
- Higher maximum speeds consistently result in greater average speeds, indicating that higher speed limits can enhance system efficiency.
- The impact of random deceleration on average speed is more significant at higher maximum speeds, as evidenced by the steeper descent in speed curves.

Flow Analysis:

- Traffic flow diminishes with rising deceleration probabilities, highlighting the negative impact of random braking on fluidity.
- While higher maximum speeds boost flow at low deceleration probabilities, the advantage diminishes with increasing random deceleration, suggesting a cap on efficiency gains under high-braking conditions.
- Notably, flow rates remain above zero when the deceleration rate is 1, indicating inherent system resilience, likely due to maintained vehicle spacing.

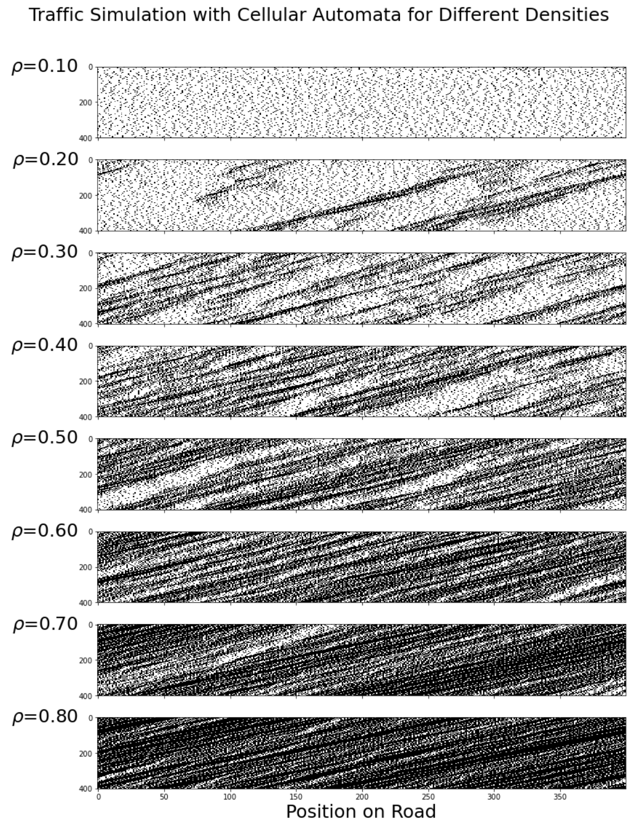


Figure 6: Periodic fluctuations and self-organizing phenomena

Figure 6 illustrates traffic flow simulations across different densities, revealing distinct behaviors. At low densities (equal to 0.1), vehicles are able to move freely because they are able to travel near or at maximum speed, almost unaffected by other vehicles. This is shown in the figure as lots of uniform and sparse black dots, which move downwards in a straight line, reflecting the steady advance of the vehicles. There is hardly any interaction between vehicles so congestion is highly unlikely to form.

At moderate densities (between 0.2 and 0.6), interactions between vehicles become more common, leading to the emergence of some fluctuations in traffic. As can be seen in the figure, some ripple-like or diagonal structures have formed, indicating a change in vehicle speed. The formation of vehicle convoys can be observed, which is also a manifestation of the self-organisation phenomenon. These convoys are characterized by groups of vehicles moving at similar speeds and maintaining consistent distances from one another.

At high densities (between 0.7 and 0.8), the congestion of traffic becomes more pronounced. This state is illustrated by denser black regions interspersed with white ripples, indicative of frequent stopping and starting motions of vehicles, known as "stopping waves." These patterns, resembling diagonal lines from the top right to the bottom left, reflect the stop-and-go behavior prevalent in high-density traffic conditions.

The simulation of the model corroborates periodic fluctuations and self-organisation phenomena in traffic flow. Periodic fluctuations, observed at moderate densities, denote the cyclical nature of traffic, where vehicles consistently accelerate and decelerate in response to the actions of preceding vehicles. The self-organising nature of traffic, evident in the spontaneous formation of congestion clusters and waves, occurs without any central control, stemming purely from local interactions among vehicles. This transition and associated patterns are critical to understanding real-world traffic dynamics and developing strategies to alleviate congestion.

2.2.3 Traffic in Open System

Method

The open system, also known as the bottleneck situation, is simulated by using different boundary conditions, where the updating rules of the model remain the same:

- When the leftmost site (site 1) is empty, add a car with speed 0 to that spot. The purpose of this step is to simulate the uninterrupted entry of vehicles into the road (e.g., from a saturated dual carriageway into a single lane).
- On the far right, the vehicles at the six rightmost sites are deleted, thus creating an open boundary. The purpose of this step is to recreate a situation where, at the end of the lane, vehicles enter a more open lane (into a four-lane road).

Simulation

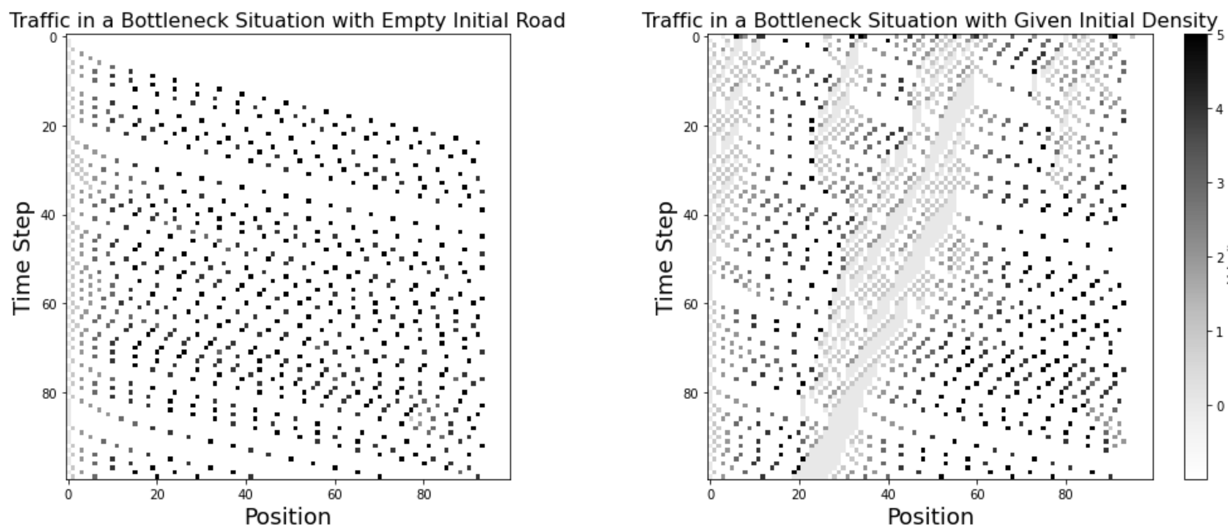


Figure 7: Visualisation with empty initial road and inital density

Analysis

As we can see from **Figure 7** The diagram on the left shows the bottleneck situation described above, with initial boundary conditions of a density of 0 cars per site. The diagram shows that there is little to know traffic with the cars continuously following each other with no persistent breaking.

On the right side, we start with a given initial density of 0.3 cars per site. In this situation we see that multiple traffic jams are forming. This most likely stems from the roads having a given initial density as the following cars will have to react to the cars in front, that could be going at a slow given velocity. We therefore note that there is a significant difference in traffic flow as the initial density is increased.

In **Figure 8** below we model initial density against traffic flow to find at which initial density the traffic flow is its highest and lowest.

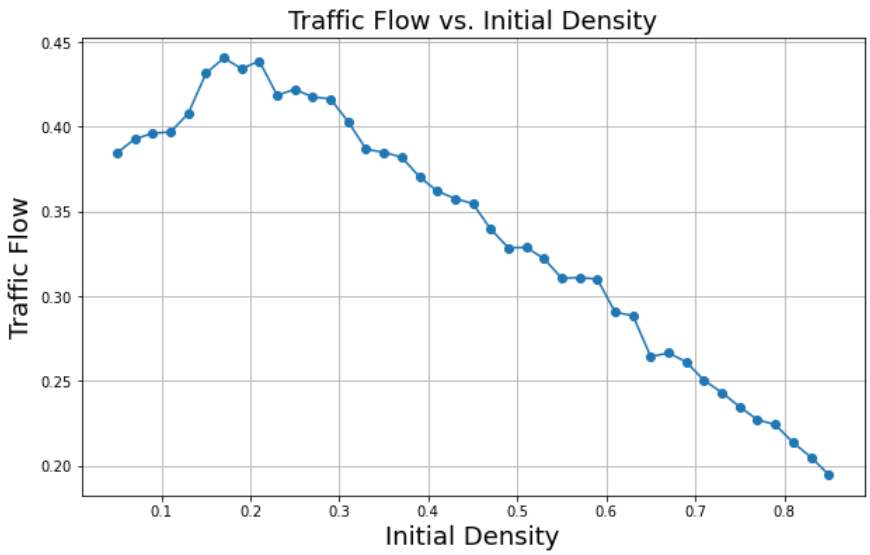


Figure 8: Traffic Flow vs. Initial Density

Figure 8 shows that traffic flow starts of by increasing as the initial density increases. This is because as the density of cars on the road increases more cars will progress along the road in a given time period. This trend increases until the initial density is at about 0.2 cars per site. This is the initial density where the traffic flow is at its maximum. As the density is then increased the traffic flow starts to linearly decrease. This is because there will start to be a larger number of cars per site meaning that if even one of those cars decreases their speed the traffic behind will also have to decrease their speed. This results in various amounts of traffic, which consequently reduces traffic flow. The more cars per site after density of 0.2 cars per site, the lower the traffic flow. At initial density of 1 car per site the flow would reduce to 0 as no car would be able to get through.

Traffic Light Situation

In this next section we want to see how traffic lights will effect the traffic flow. Traffic lights are used to help pedestrians cross a busy road and are also used to organise traffic at busy junctions to increase traffic flow from every side in the safest way. They first came about in 1868 in Parliament Square London [4]

We model traffic lights using the CA model in open system, by checking the relationship between the vehicle and the traffic light: First, check whether the vehicle will cross the traffic light if it continues to travel at the current speed.

When the traffic light is red, vehicles need to slow down or stop to avoid entering the intersection. By taking the minimum value of the distance from the current position of the vehicle to the traffic light and the movable distance in front of the vehicle, the movement of the vehicle in the next time step is limited to ensure that it will not drive into the location of the traffic light. That is to say, if the distance between the vehicle and the traffic light is less than the number of free spaces in front of it, the vehicle will stop at the traffic light instead of continuing forward.

When the traffic light is yellow, vehicles will try to slow down and stop unless they get too close to the light. If the vehicle's distance minus 1 is less than the maximum speed and the vehicle speed is greater than 1, the vehicle will be forced to stop. Otherwise, if vehicles are already very close to a traffic light (e.g., traveling at close to or equal to their current speed), they will continue through the yellow light.

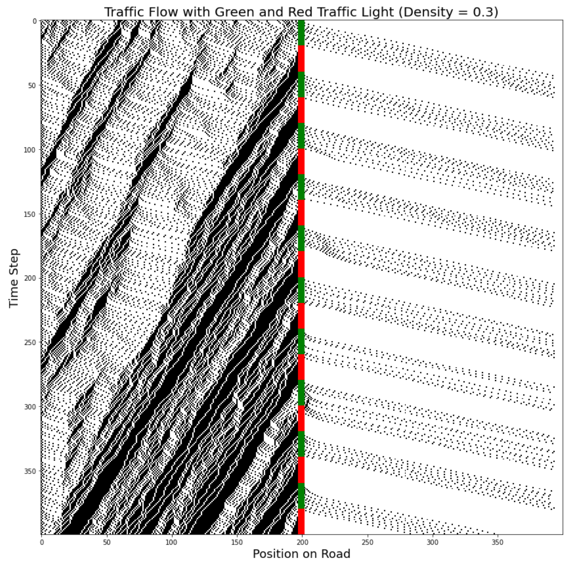


Figure 9: Traffic Light

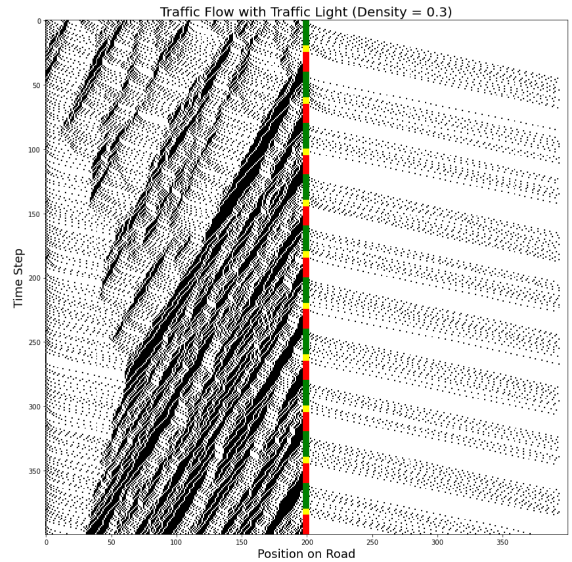


Figure 10: Traffic Light with yellow light

We first model traffic lights in **Figure 9** that don't include the amber light. We can see how traffic lights result in queues of traffic waiting for the light to turn green. This model can be used to plug in different traffic densities at different times of the day and calculate how long it will take cars to get to there destination. We can see that after the traffic lights there is no obstruction so the cars carry on increasing in velocity until the cars meet there maximum speed. At this point the traffic flow will be high.

The more accurate model in **Figure 10** includes amber lights which represents traffic lights used today. This is more accurate as it gives cars time to slow down and stop if the car is close to the traffic light. The amber traffic light creates slightly less dense queues with traffic jams spread further apart.

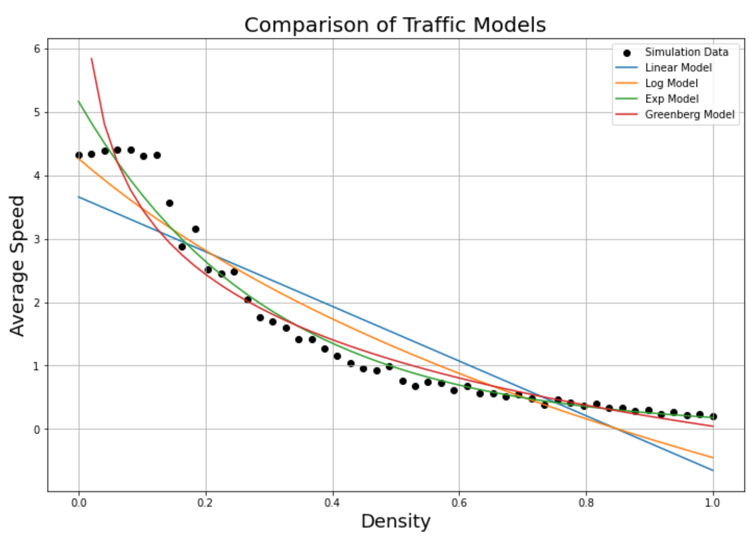


Figure 11: Relationship Between Average Speed and Density

Using our simulation data we analyse the relationship between the density of cars on the road and the average speed of the traffic. This led to four possible models that could be fitted to the data as shown in **Figure 11**. The equations for each of these fitted models are given below.

Linear model:

$$v = 3.64\left(1 - \frac{\rho}{0.85}\right) \tag{4}$$

Log model:

$$v = 4.25 - 4.25\log\left(\frac{\rho}{0.49} + 1\right) \tag{5}$$

Exp model:

$$v = 5.17\exp\left(-\frac{\rho}{0.30}\right) \tag{6}$$

Greenberg model:

$$v = 1.48\log\left(\frac{1.03}{\rho}\right) \tag{7}$$

Upon inspection of **Figure 11**, it appears that the Exponential and Greenberg models fit our simulated data points the most accurately while the linear and logarithmic models are far less accurate. These are the models we will use later in the report when exploring the classical model.

2.3 Two-Lane Model

2.3.1 Model

Single-lane models usually do not simulate realistic traffic flow very well, because in reality there will be different types of vehicles on the road with different maximum speeds and probabilities of slowing down. Having only one lane would result in fast vehicles having to follow slower vehicles, travelling at the maximum speed of the slower vehicles. Therefore, we introduce the two-lane model in the following section.

The two-lane model is composed of two previously defined parallel single-lane models with periodic boundary conditions and four additional rules that define vehicle exchanges between lanes. The update step is divided into two sub-steps:

- Determine the exchange of vehicles between two lanes according to the defined exchange rules. During the exchange, the vehicle can only move sideways to the other lane and not forward. Since the vehicle is generally not capable of purely lateral movements, this step only makes physical sense when considered together with the second sub-step, the update rule.
- Each lane on a two-lane road is updated independently according to the single-lane update rules. In this second sub-step, the resulting configuration from the first sub-step is used.

Define $gap(i)$ to denote the number of empty sites in front of the current lane, $gap_o(i)$ to denote the number of empty sites in front of the other lane, and $gap_{o,back}(i)$ to denote the number of empty sites behind the other lane. l, l_o and $l_{o,back}$ are parameters that indicate how far forward to look in this lane, how far forward to look in another lane, or how far backward to look in another lane, respectively. The lane exchange rules for the two-lane model are as follows [5]:

1. $gap(i) < l$
2. $gap_o(i) > l_o$
3. $gap_{o,back}(i) > l_{o,back}$
4. $rand() < p_{change}$

When all the above conditions are satisfied, vehicle i performs a lane exchange. For the subsequent analysis in this section, the model will be simulated by taking

$$\begin{aligned} l &= v+1, \\ l_o &= l, \\ l_{o,back} &= v_{max} = 5 \\ p_{change} &= 1. \end{aligned}$$

Symmetry is one of the important properties of the model. The set of rules defined for vehicle lane changing can be either symmetric or asymmetric. Symmetric models are very effective for theoretical analyses, while asymmetric models are closer to reality. In the subsequent section, the symmetric and asymmetric models will be analysed separately.

2.3.2 Simulation

Symmetric Version

In the symmetric version of the model, the vehicle stays in its original lane as long as there is no other vehicle ahead (i.e., $gap \geq v+1$). If there is a car ahead in the current lane (i.e., $gap < v+1$), the vehicle follows the lane exchange rules to check if it is possible to switch lanes, and if so, it does. After that, as long as there is no car in front of it, it will remain in this lane until the lane exchange rules are satisfied again.

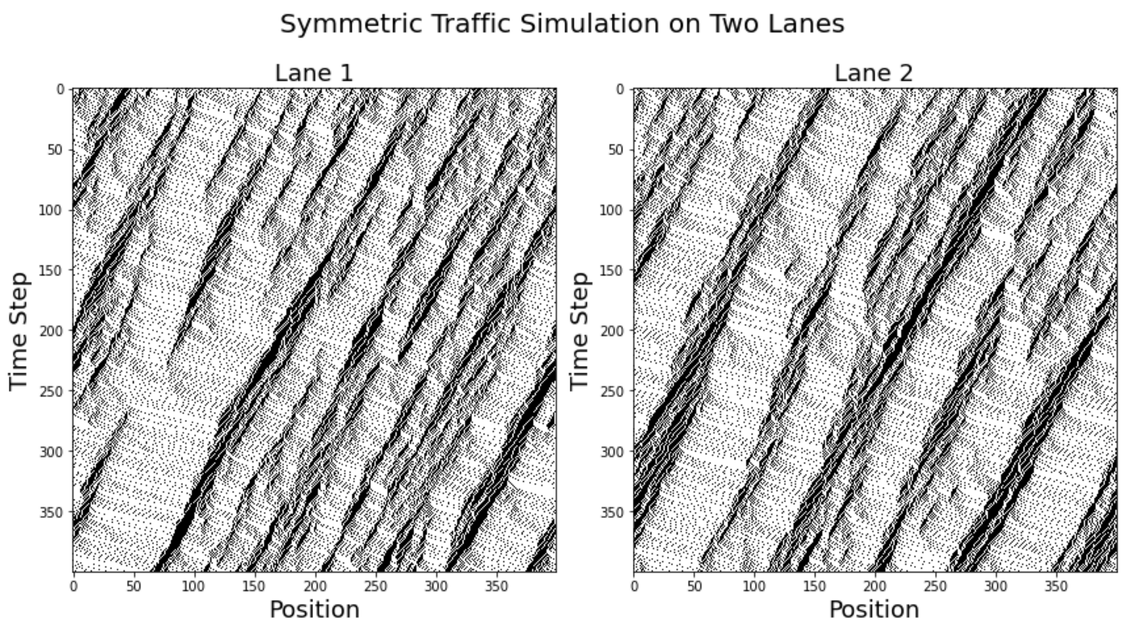


Figure 12: Simulation of symmetric model

Figure 12 is a symmetric CA simulation of a two-lane road with a density of 0.3. It can be observed that both lanes are utilised relatively equally and effectively, with no obvious over-congestion on one side and idleness on the other. This indicates that the symmetric lane changing rule promotes a balanced distribution of vehicles between the two lanes. Both lanes exhibit traffic fluctuations which can be seen as “ripple-like” structures that represent the formation of traffic jams. The fluctuations appear to be present in both lanes and are relatively symmetrical, reflecting similar vehicle behavior in both lanes. Vehicles can also be observed changing lanes from one lane to another. Because the model is symmetric, vehicles are relatively evenly distributed between the two lanes, and lane changing behavior is responsive to the speed and distance of the vehicle ahead as the vehicle attempts to find the fastest route.

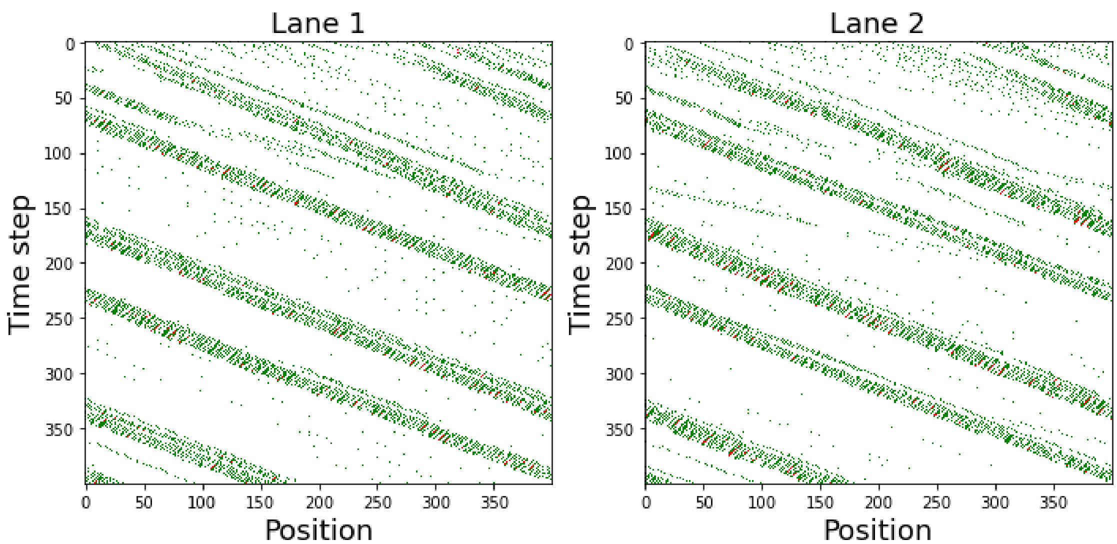


Figure 13: Simulation of road with different types of vehicles

Based on the two-lane model, we consider the situation where the road contains different types of vehicles. Suppose there are two types of vehicles on the road: cars and trucks. They have different maximum speeds and deceleration probabilities, as well as different initial vehicle densities. Additionally, vehicles take these different attributes into account when following lane changing rules.

The **Figure 13** shows the results of the simulation. We used two colors (red for cars and green for trucks) to differentiate between different types of vehicles. The dynamics of both types of vehicles at each time step can be observed from the figure. A line with a steeper slope usually indicates a faster car, while a line with a smaller slope indicates a slower truck. With an initial setting of low vehicle density, it can be observed that both cars and trucks move relatively freely, but as density increases, the traffic flow may become more congested, especially for slower trucks. Differences in speed between cars and trucks also result in different traffic flow patterns; for example, a car may slow down because of a truck in front of it.

Asymmetric Version

The symmetric model assumes that driving behavior is identical in both lanes and does not take into account the difference between fast and slow lanes, or the driver's preference to stay in one lane over the other. By introducing different rules for lane-exchanging, the asymmetric model can better simulate real-world driving behavior.

In the asymmetric version, the vehicle always tries to stay in Lane 1, regardless of the road conditions in Lane 2. The vehicle will not change lanes to Lane 2 unless the lane-exchange rules are met. After changing lanes to Lane 2, regardless of the distance between the vehicles ahead, as long as the conditions are met that there are many vacancies in front in Lane 1 and many vacancies behind in Lane 1, the vehicle will try to change back to Lane 1. This is representative of real life where the left lane is the default lane.

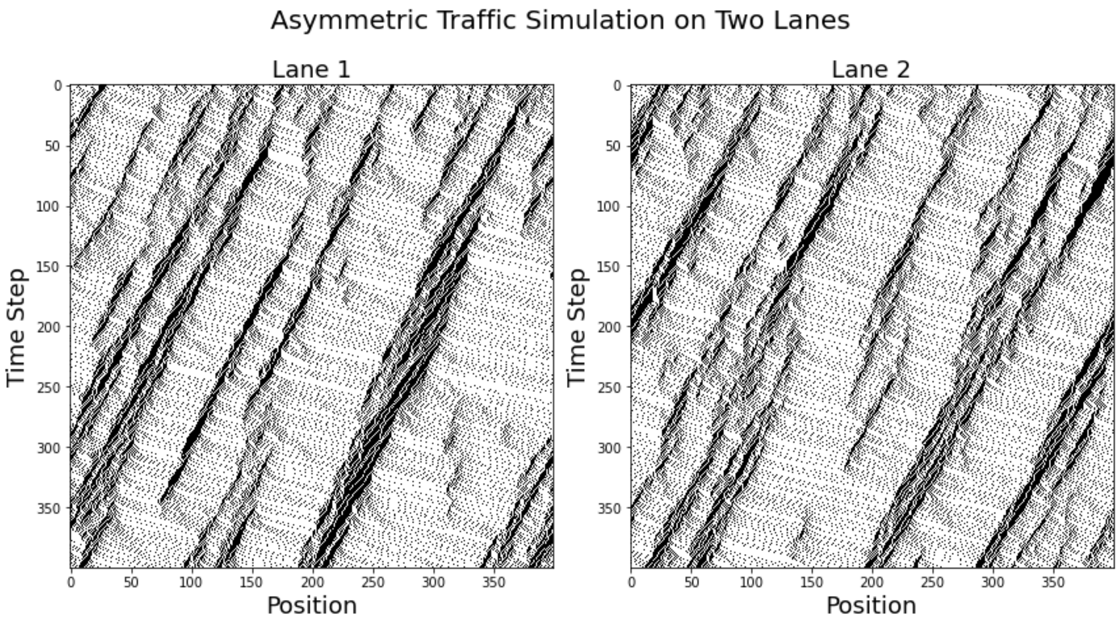


Figure 14: Simulation of asymmetric model

Figure 14 is a simulation of an asymmetric CA model at a density of 0.3, showing how vehicle preference for specific lanes affects traffic flow patterns.

In Lane 1 (left plot), it can be observed that the vehicle attempts to stay in this lane, which is consistent with the set rules of the asymmetric model. Lane 2 (right plot) shows that although there may be fewer vehicles in that lane, vehicles still tend to move back to Lane 1 and only move to Lane 2 when traffic conditions in Lane 1 force them to change lanes.

Both lanes show the formation of traffic waves, but Lane 1 exhibits a more pronounced traffic wave pattern, possibly due to higher vehicle density and vehicles slowing down and stopping more frequently. In comparison, the corrugation pattern in Lane 2 is relatively light, reflecting lower vehicle density and smoother traffic flow.

Vehicle lane changing behavior appears to be more purposeful in the asymmetric model. Vehicles will only change lanes from Lane 1 to Lane 2 when necessary, and will quickly return to Lane 1 when conditions permit. This causes Lane 2 to look empty during many periods of time.

Regarding traffic distribution, overall, Lane 1 is more crowded than Lane 2. This is consistent with the expectation of an asymmetric model, with vehicles preferring to use Lane 1. This phenomenon can be demonstrated more clearly in **Figure 15** shown below. Lane 2 serves as an auxiliary lane and is mainly used when Lane 1 is congested. When traffic conditions on Lane 1 improve, vehicles will quickly return to Lane 1, resulting in less traffic on Lane 2.

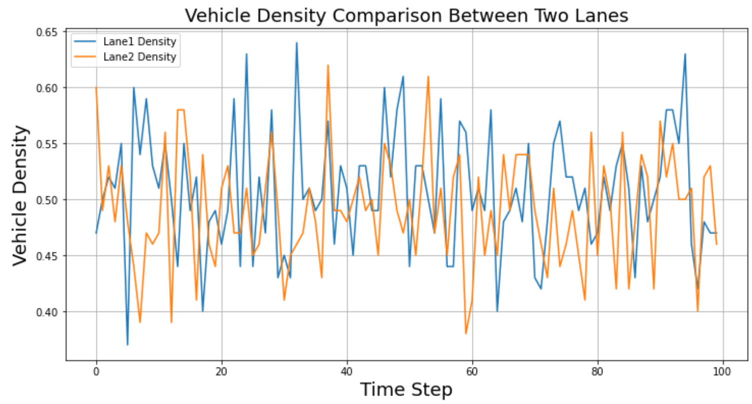


Figure 15: Density comparison between two lanes

2.3.3 Analysis

For the two-lane model, we define density and flow in a similar way to the single-lane model. We define the density as

$$\rho = \frac{N}{2L} = \frac{\text{Number of cars}}{2 \times \text{Number of sites}} \tag{8}$$

since there are two lanes. Traffic flow can be calculated by counting the number of vehicles passing through a certain interval within a specific time step.

$$\bar{q}^T = \frac{5}{T}\frac{1}{L}\sum_{i}^{L}\sum_{t}^{T/5} n_i(5t) \tag{9}$$

In our model, for each time step, we iterate through each vehicle in each lane and accumulate whether their position before and after the update crosses the first 1/10 of the road. Then, the average flow rate of the two-lane model is obtained by averaging the flows of the two lanes.

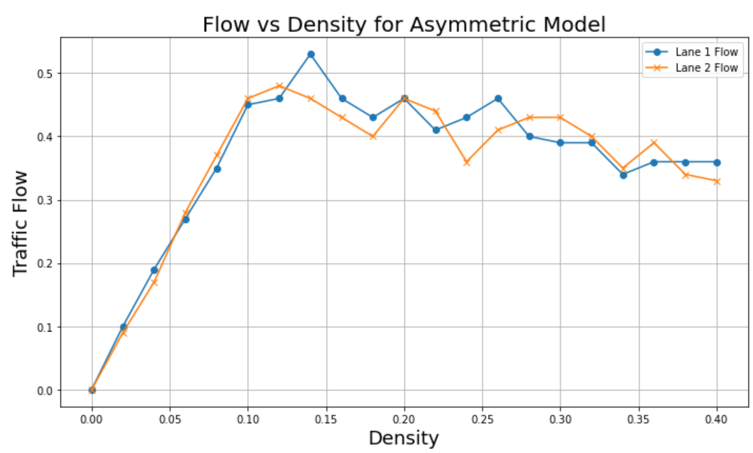


Figure 16: Traffic Flow vs. Density on two lanes

Figure 16 shows a comparison of traffic density versus flow on two lanes in an asymmetric model. The two lanes have similar traffic flow and density relationships and their graphs follow the same shape. In the lower density interval, the flow of both lanes increases with increasing density. The traffic volume of both lanes peaks near a certain density value and then starts to decrease. Once they start to decrease, more differences in flow start to arise between different the two lanes. In general, the flow of Lane 1 (assumed to be the preferred lane) is higher than that of Lane 2. This is because in the asymmetric model, drivers tend to choose Lane 1 where possible leading to more cars usually in this lane. As density continues to increase, both lanes show a downward trend in traffic volume, but Lane 1's decline is more significant. This shows that in high density situations, traffic congestion and mutual interference increase, and vehicles cannot maintain optimal speeds, resulting in a reduction in overall flow.

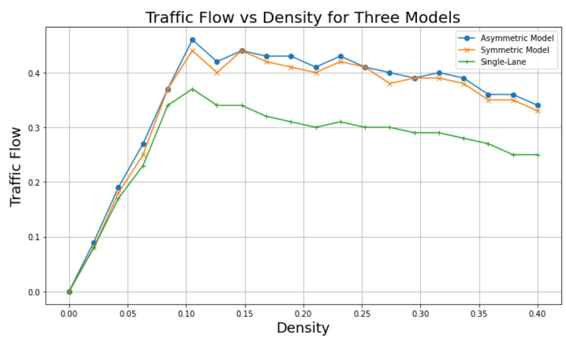


Figure 17: Comparison on three models

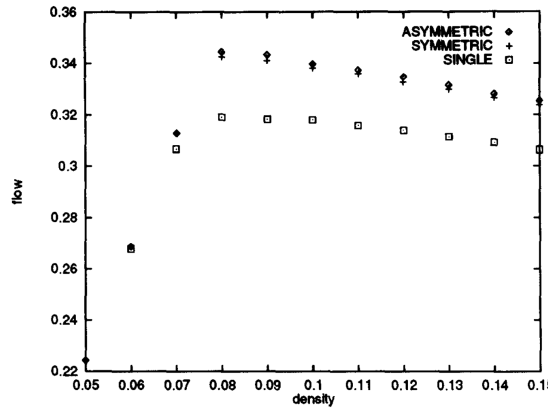


Figure 18: Results from Rickert et al 1996

In **Figure 17** , we compare the three previously built models. By setting the same road length (400) and time step (400), the density-flow relationship diagram of the single-lane model (closed system), symmetrical two-lane and asymmetric two-lane model can be drawn. Comparing these with the simulation results from Rickert et al. (1996) [5], we can observe that the characteristics of the two graphs are similar. Both show that the single lane model differs most from the asymmetric and symmetric models which are much more similar. This is due to the obvious fact that the average traffic flow over two lanes will be higher than for one lane. The traffic flow peaks at a density of around 0.1 in our simulation, compared to a very similar density of approximately 0.08 from the results from Rickert et al (1996). This justifies the accuracy of the CA model as our results correspond well to results from real life data.

In the observed simulation results, the traffic flow versus density for symmetric and asymmetric models exhibits a noteworthy enhancement compared to the single-lane model. This suggests that implementing a lane-changing mechanism can indeed improve overall traffic conditions by allowing vehicles to bypass slower-moving cars, leading to an increase in the total traffic flow.

The three models exhibit a pronounced peak, suggesting an optimal density around 0.1 vehicles per site at which traffic flow is maximized. This phenomenon aligns with typical traffic flow theory, where an increase in density leads to higher flow rates until reaching a critical density, beyond which interactions among vehicles result in reduced flow rates due to congestion.

The symmetric and asymmetric models display very similar flow patterns across the range of densities. This could indicate that the additional flexibility provided by asymmetric lane changing does not significantly alter the overall capacity of the road compared to the symmetric model where lane changes are governed purely by vehicle spacing and speed.

For higher densities (approximately higher than 0.1), the flow rates for all models begin to decline, yet the rate of decline appears more gradual for the symmetric and asymmetric models compared to the single-lane model. This could be attributed to the additional maneuvering space provided by the extra lane, which helps in partially mitigating the congestion effects seen at high densities.

3 Car-Following Model

3.1 Introduction to the Model

In this section we are investigating car-following models, with specific focus on Gipps' model [6]. This method simulates traffic flow by using dynamical information from the car directly ahead. Specifically, it uses the speed and position of the leader car in combination with the reaction time of the driver in the following car to reproduce real world traffic scenarios. This method is often preferred to other models considered throughout this report because, unlike the other models, this one takes into consideration variations in driving styles and characteristics of drivers and the impact that this will have on traffic flow.

The car-following model considers both 'normal driving', such as driving along a motorway and 'emergency' situations, for example when a leader car suddenly brakes and the following cars consequently need to stop as quickly and safely as possible, whilst always maintaining a safe distance to the car ahead. To model different driving styles, including fast and slow drivers we assign varying maximum velocities. Aspects such as recklessness can also be modelled which is simulated by a driver following another driver at a much closer distance than others would or by a driver accelerating and braking at higher rates. Further characteristics may be investigated by varying the parameters of this model which we discuss later through this section.

3.2 Set Up

To set up this model, it is assumed that the driver following a leader vehicle adjusts their speed so that they can come to a safe stop if the leading vehicle were to suddenly brake to a halt. We assumed a maximum desired speed of 70mph (the speed limit of a motorway) [7] and that this speed will never be exceeded. We include a safety parameter, θ, for a possible additional time delay before reacting to the change of speed of the car in front. This gives us a total safety reaction time of $\tau + \theta$.

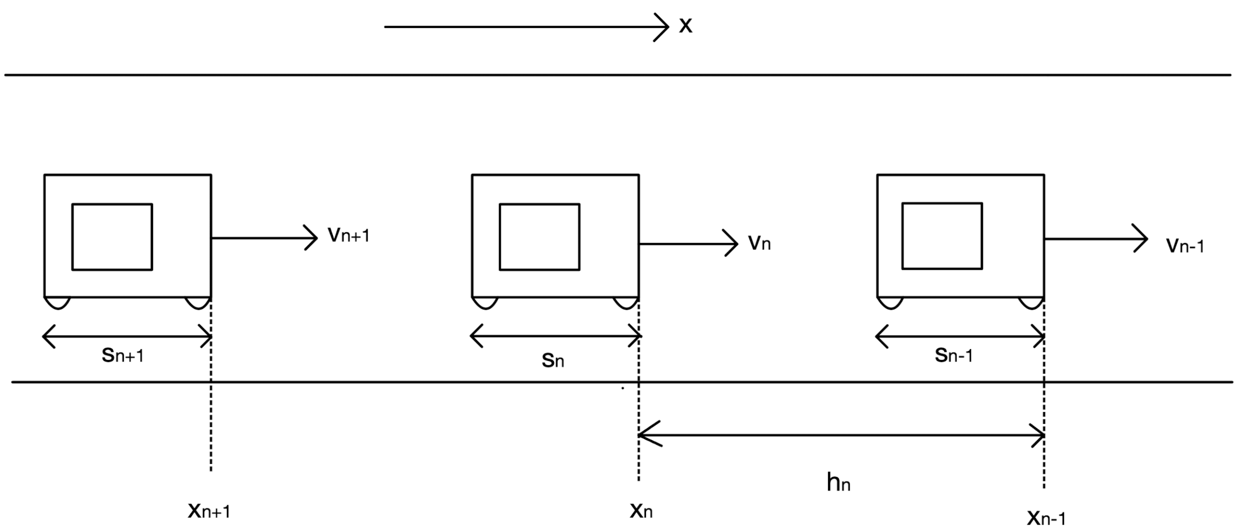


Figure 19: Diagram for Car-Following Model

Figure 19 above is a diagram of three cars on a single lane road to help visualise the concept of the car-following model. Focusing on the leader car, car $n-1$, and its follower car, car n, we state that the velocity of the follower car, v_n, is dependent on the velocity of a leader car, v_{n-1}. x represents the displacement of the cars, s is the length of each car and h_n is the headway of car n (the distance between the front of the leader car and the front of the following car). Extending this main idea to the third car and then all cars following that is how the car-following model is set up.

3.3 Simple Equations

To begin, we derive simplified equations for the car-following model and we write the minimum distance that must be maintained between cars at all times as

$$d_{min} = x_n(t) - x_{n-1}(t). \tag{10}$$

Then we define the acceleration of the following car as a function that depends on a reaction time, τ, the speed of the car itself and also the speed of the car in front at time t.

$$\ddot{x}_n(t+\tau) = \alpha(\dot{x}_{n-1}(t) - \dot{x}_n(t))$$

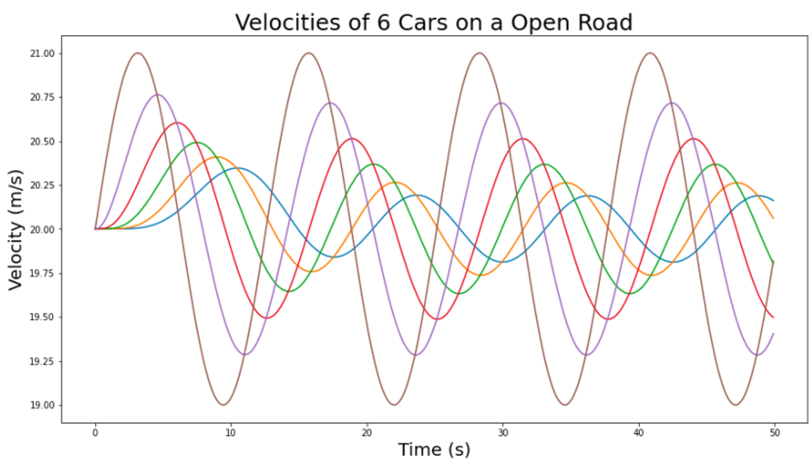


Figure 20: $\alpha = 0.5$

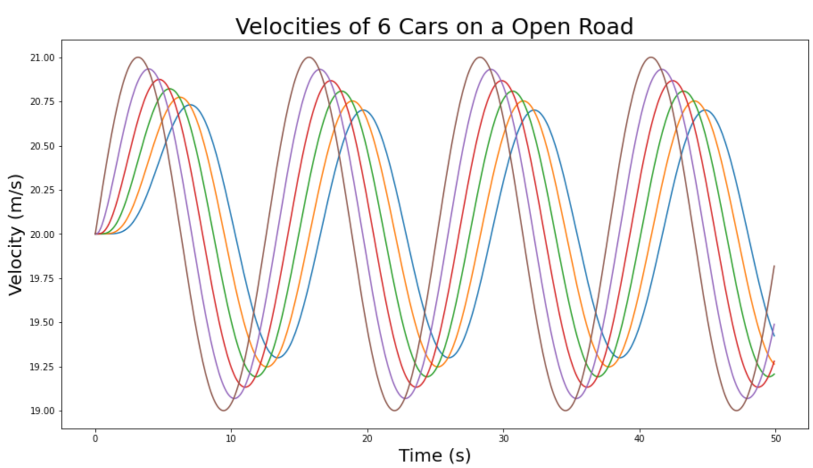


Figure 21: $\alpha = 1.2$

Figure 20 shows how the velocities of a stream of 6 cars change. All cars shares the same length, headway and initial speed as this is a very simplified model and $\alpha = 0.5$. From the graph we conclude that the velocity of each car oscillates periodically in response to the oscillations of the first car's velocity, suggesting repeated patterns of acceleration and deceleration.

In **Figure 21**, $\alpha = 1.2$ to show what happens for larger values of α. Comparing to **Figure 20**, we see that as α increases, the following cars tend to be more sensitive to the leading car changes in velocity and they ajust their speeds much quicker.

However, this is an incredibly simplified description of a complex phenomenon. This doesn't take into account the characteristics of the driver in front so we use this purely as motivation to then derive the complex equations that we use in our simulations.

3.4 Derivation of Complex Equations

To begin the derivation, we use the assumption that the velocity of a car is the minimum of its free velocity, v_{free}, and its safe velocity, v_{safe}, where v_{free} is the speed a car travels when there is a large distance between itself and the car in front, providing a big headway. This happens when the road is not congested. We have modelled this as a function of the speed of the car at time t, acceleration, a, the driver's reaction time, τ, and a limited factor, the maximum velocity v_{max}. v_{safe} is the speed of the car when the road is congested, meaning headway's are smaller and there are factors that limit the car from travelling at its maximum speed whilst also being able to maintain a safe distance to the car ahead. Hence, the safe speed is considered as the worse case scenario.

$$v(t+\tau) = min(v_{free}, v_{safe}) \tag{11}$$

$$v_{free} = min\left(v(t) + a\tau(1 - \frac{v(t)}{v_{max}}), v_{max}\right) \tag{12}$$

The safe speed requires a minimum gap s_0 to be kept at all time between cars. Now, looking at how the following cars react to the change in behaviour of the car in front we derive the equation for v_{safe}. We use a diagram to show the distances travelled and velocities for a lead car (car $n-1$) and a follower car (car n) over time.

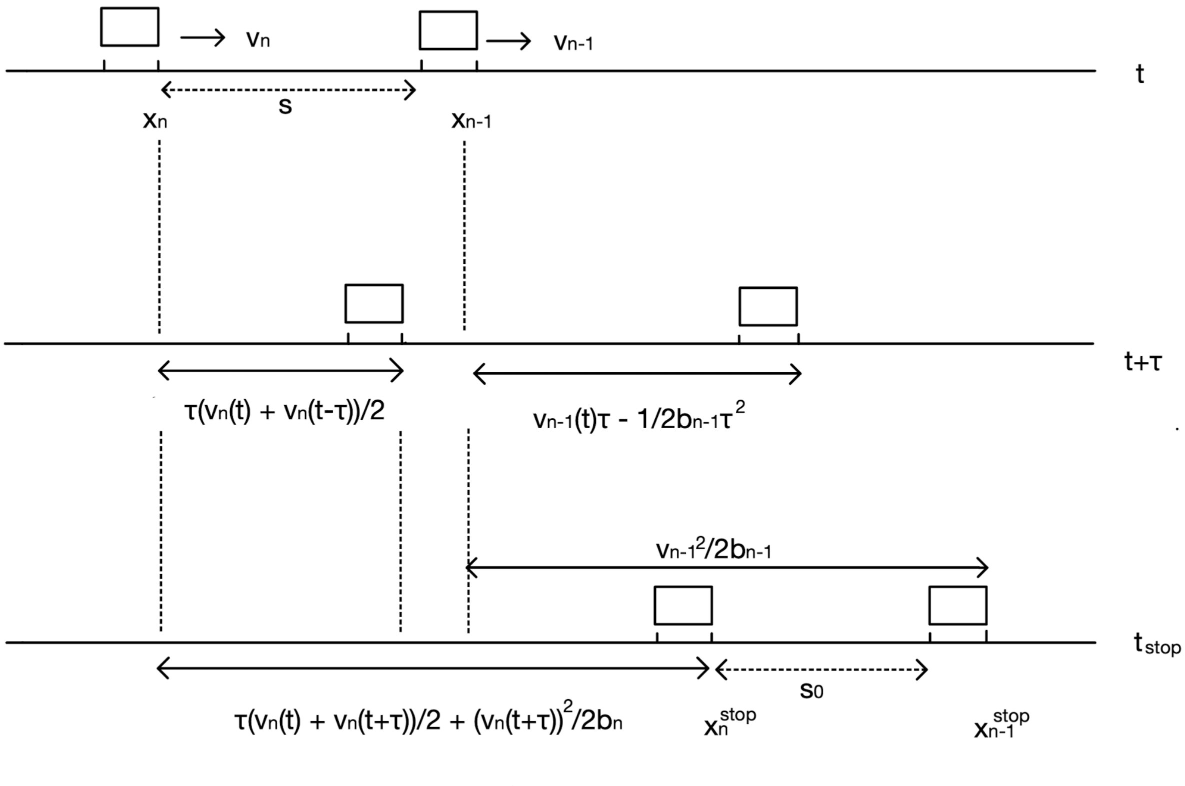


Figure 22: Breaking of the leading car

Figure 22 shows initial velocities of v_n and v_{n-1} for the following car and leader car respectively. The leader car brakes at time t with a deceleration of b_{n-1}. The following car continues to maintain its initial speed for a reaction time τ and then also brakes with a deceleration of b_n at the end of the reaction period, time $t+\tau$. They both eventually come to a complete stop, at time t_{stop}, whilst always maintaining the safe distance s_0 between them. The diagram illustrates the distance travelled by each car at every step of this process.

The total stopping distances of the lead car and follower car are respectively given by

$$x_{n-1}^{stop} - x_{n-1} = \frac{v_{n-1}^2}{2b_{n-1}} \tag{13}$$

$$x_n^{stop} - x_n = \frac{\tau}{2}(v_n(t) + v_n(t+\tau)) + \theta v_n(t+\tau) + \frac{v_n(t+\tau)^2}{2b_n} \tag{14}$$

where the additional $\theta v_n(t+\tau)$ has been included to account for an extra safety reaction time parameter, θ. We use these to calculate the minimum gap to be kept at all times, s_0, and then rearrange to obtain a quadratic for $v_n(t+\tau)$ which is the velocity of follower car at time $t+\tau$, where $s = x_{n-1} - x_n$.

$$s_0 = s_{stop} = s + (x_{n-1}^{stop} - x_{n-1}) - (x_n^{stop} - x_n) \tag{15}$$

$$s - s_0 + \frac{v_{n-1}^2}{2b_{n-1}} - \left(\frac{\tau}{2}(v_n(t) + v_n(t+\tau)) + \theta v_n(t+\tau) + \frac{v_n(t+\tau)^2}{2b_n}\right) = 0 \tag{16}$$

$$v_n(t+\tau) = -b_n\left(\frac{\tau}{2}+\theta\right) + \sqrt{b_n^2\left(\frac{\tau}{2}+\theta\right)^2 - 2b_n\left(-s+s_0-\frac{v_{n-1}(t)^2}{2\hat{b}}+\frac{\tau}{2}v_n(t)\right)} \tag{17}$$

$\hat{b}$ is the assumed braking rate of the driver in front, as this cannot be predicted by the driver behind. This value must satisfy the inequality $\hat{b} \geq b_n$ to ensure that no collisions occur.

Finally, we write the velocity for car n as a minimum of the free velocity and the velocity when it is limited by the behaviour of the leader car.

$$v_n(t+\tau) = min\left(v_n(t) + a\tau(1-\frac{v_n(t)}{v_{max}}), -b_n\left(\frac{\tau}{2}+\theta\right) + \sqrt{b_n^2\left(\frac{\tau}{2}+\theta\right)^2 - 2b_n\left(-s+s_0-\frac{v_{n-1}(t)^2}{2\hat{b}}+\frac{\tau}{2}v_n(t)\right)}\right) \tag{18}$$

The speed is determined by the minimum of these two arguments. When there is a sufficiently large headway its speed is given by the first argument and the car is free to accelerate towards its maximum speed. Where there is a smaller headway, the speed is determined by the second argument which causes the driver to decelerate to adjust their velocity depending on the behaviour of the leader car.

3.5 Analysis

3.5.1 Open System

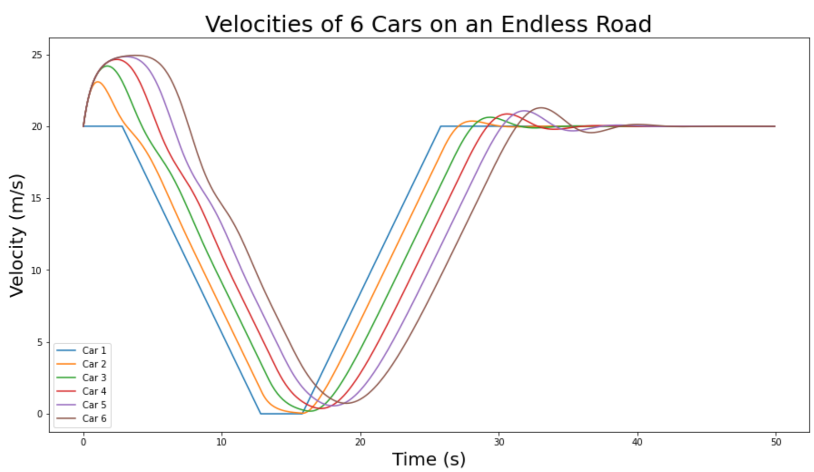
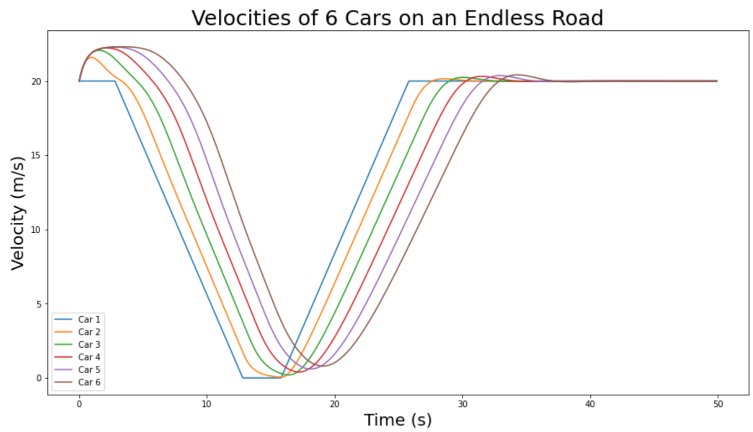


Figure 23: Velocities When the Leader Brakes

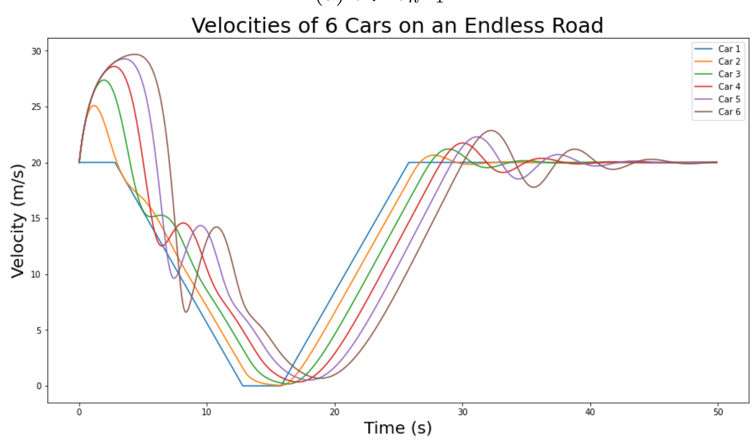
We create a simple simulation using our derived equations to illustrate how traffic on an endless road (hence an open system) would react when a driver sharply brakes. We demonstrate how the velocities of 6 cars on an endless single lane road change when the lead car decelerates to a stop, as seen above in **Figure 23**. In this figure all 6 cars have the same initial velocity and are spaced evenly with plenty of headway. Initially, we observe the following cars increasing their velocities relative to the car in front to close the gaps between their own car and the car that they are directly following. Then car 1, the leader car, brakes to a complete stop. We observe how the following cars change their velocities after a short reaction time by also braking in turn to avoid colliding with the car in front. Car 1 then remains at a halt for 3 seconds before accelerating again to a constant velocity and the following cars do the same due to the headway in front of them increasing so it becomes safe to increase velocity again. Eventually they all return back to the same velocity and the traffic flows. This mimics the behaviour of real traffic in the scenario of a driver braking suddenly. This is a very simple model to begin with as all driver's are assumed to accelerate and decelerate at the same rates and they also all have the same reaction time. In real life, this would not hold true as each individual driver has their own unique style of driving.

3.5.2 Assumed Braking Rate

The value of $\hat{b}$ represents the assumed braking rate of the car in front, which is then used to determine the following car's velocity. In real life this is uncertain so now we look at how varying $\hat{b}$ affects disturbances to the traffic flow. This is shown in the figure underneath.



(a) $\hat{b} > b_{n-1}$



(b) $\hat{b} < b_{n-1}$

Figure 24: Varying $\hat{b}$

In **Figure 24** we see that when $\hat{b}$ is larger than b_{n-1}, meaning that the car in front braked at a slower rate than anticipated, then the disturbances to traffic are damped. This is because the follower car has overestimated how fast the leader car will decelerate so they respond faster to the leading car's changes in velocity. This dampens disturbances because the driver can quickly adapt to any fluctuations in the leading car's velocity, leading to more smooth traffic flow. However if $\hat{b}$ is smaller than b_{n-1}, where the car in front has braked at a higher rate than expected, the disturbances are amplified. This results from follower cars adjusting slower to the leader car's changes in velocity. As they have underestimated the rate at which the leader car will break, the followers then take longer to adapt their own velocity. This is what gives larger disturbances and oscillations in the traffic flow. Therefore, increasing $\hat{b}$ causes quicker adjustments and hence less fluctuations whereas decreasing $\hat{b}$ makes drivers adjust slower and amplifies fluctuations in traffic flow.

3.5.3 Closed System

In this section, we first demonstrate how the displacements of multiple vehicles travelling on a closed circular road change over time. All vehicles were evenly spaced on a 1000 meter long single lane road and shared the same initial speed of 20 meters per second.

We consider a scenario where all vehicles have the same length but different accelerations and there are no traffic lights involved. The parameters used are as follows:

a_n	sampled from a normal distribution $N(1.7, 0.3^2)$ m/sec^2.
b_n	equated to $2.0a_n$.
S_n	6 m.
V_n	sampled from a normal population. $N(20, 3.1^2)$ m/sec^2.
τ	2/3 second.
θ	1/2 τ.
$\hat{b}$	minimum of 3.0 and $(b_n - 3.0)/2$ m/sec^2.

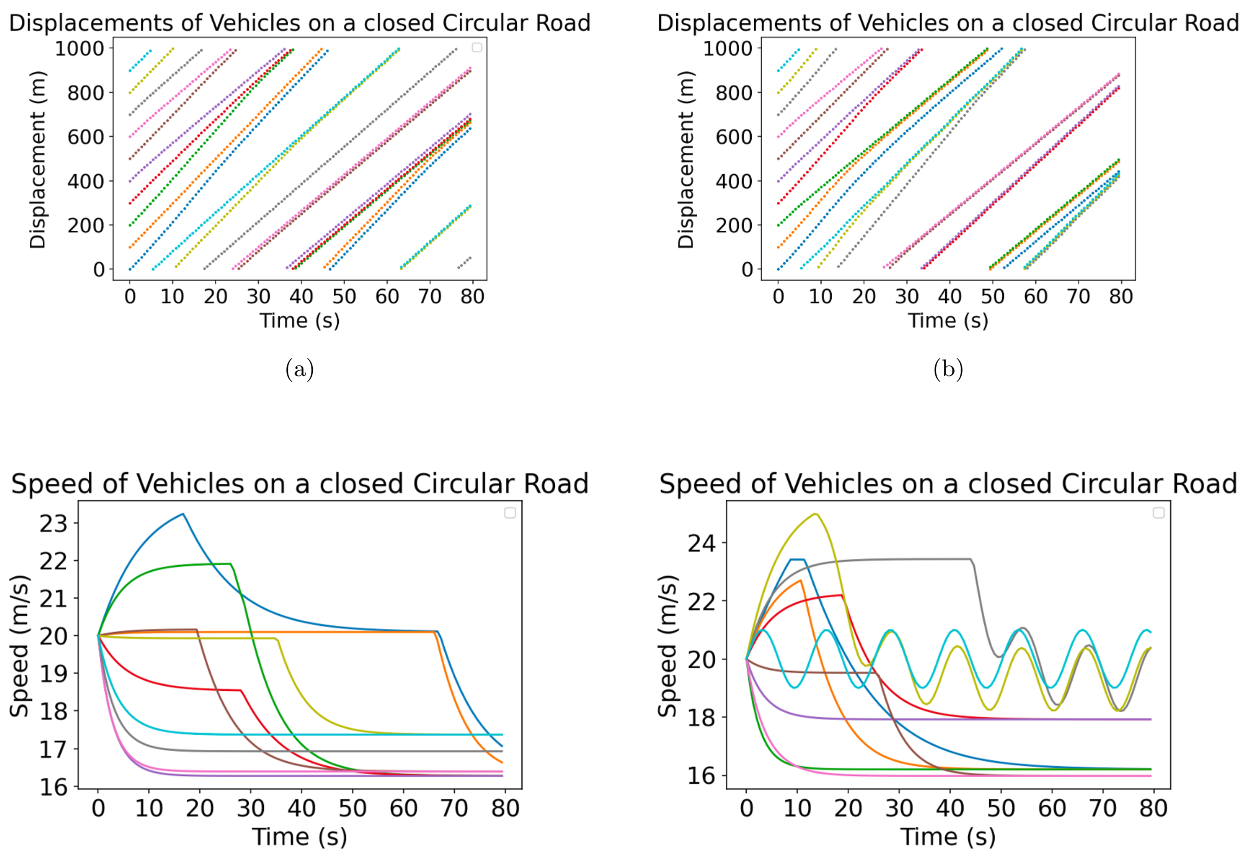
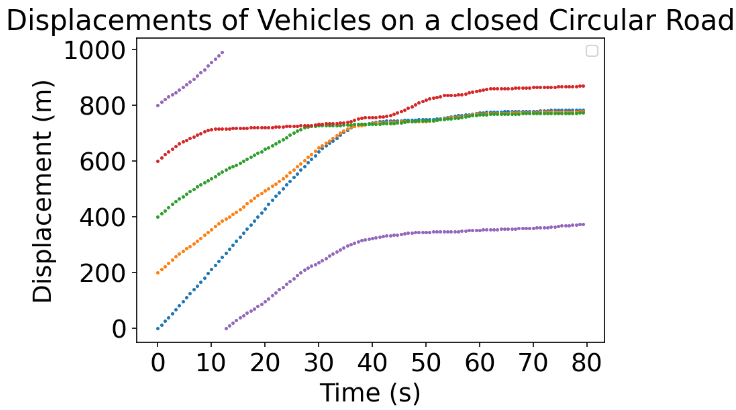
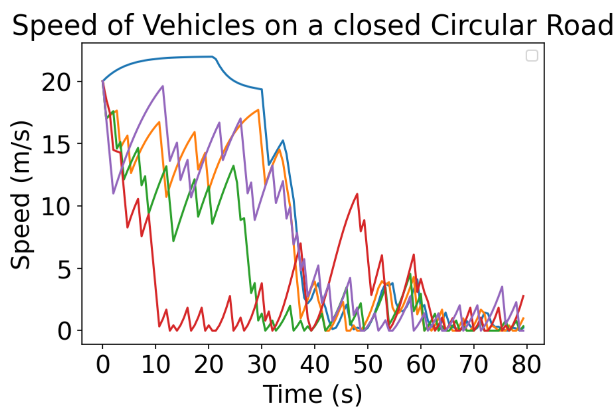


Figure 25: Complex model outcome

Figure 25 shows the displacements and speeds of 20 vehicles, each marked by a different colour, using the complex model. Graph (a) and (c) are the result of the complex model without any fixed leading car speed function. Graph (b) and (d) show the outcome when the leading car's speed function is $v(t) = \sin(0.5t) + v(0)$. We can conclude that the speed of some cars oscillate periodically in response to the oscillations of the first car's velocity, suggesting repeated patterns of acceleration and deceleration. The reason that some of the cars' velocities do not oscillate is from the definition of the Complex Model, as each cars has its own unique desired max speed V_n which is randomly distributed.



(a)



(b)

Figure 26: Model outcome with randomised breaking

Furthermore, we consider randomised braking for each driver. **Figure 26** shows model outcome of 5 vehicles on a closed circular road.

Also, we can conclude from **Figure 25** that the speed of all cars plateaus to constant values as time increases. While **Figure 26** may not result in a constant speed, generally the velocities appear to tend to 0.

Finally, we look into establishing a relationship between the average speed and density of cars which is displayed beneath in **Figure 27**.

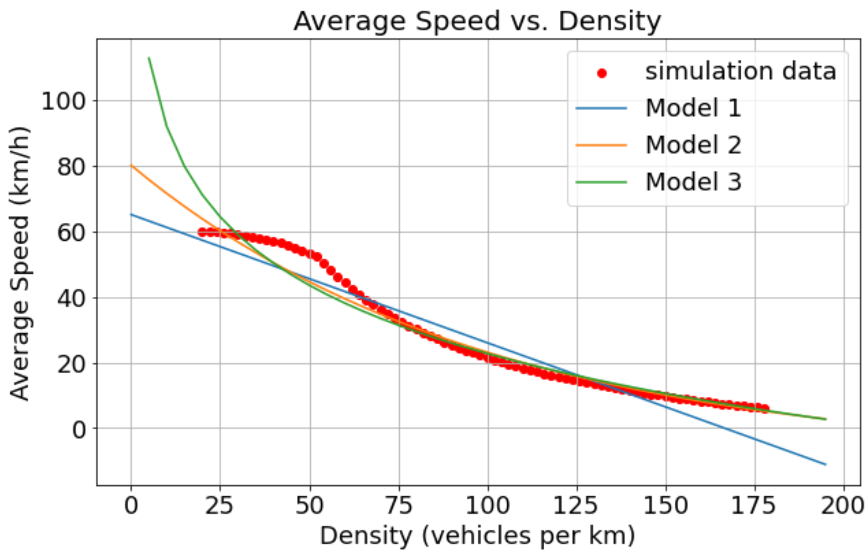


Figure 27: Average Speed vs Density

The red dots represent points of data from our simulations which we fit three different models to, each marked by a different colour. These are defined by the following equations.

Model 1:

$$v = -0.39\rho + 65 \tag{19}$$

Model 2:

$$v = 90 \exp\left(\frac{\rho}{-100}\right) - 10 \tag{20}$$

Model 3:

$$v = 30 \log\left(\frac{1.03}{\rho}\right) + 160 \tag{21}$$

It can be seen from **Figure 27** that the exponential (orange) and logarithmic (green) models fit the data far more accurately than the linear model (blue). This is consistent with the results found earlier in this report from CA model where a similar exponential and logarithmic function were found to fit the data points well. Again, these will be used in the later part of this report where the classical method utilizes a velocity-density equation to model traffic flow.

In the next section we further explore the relationship between traffic density and traffic flow and we plot the graph below.

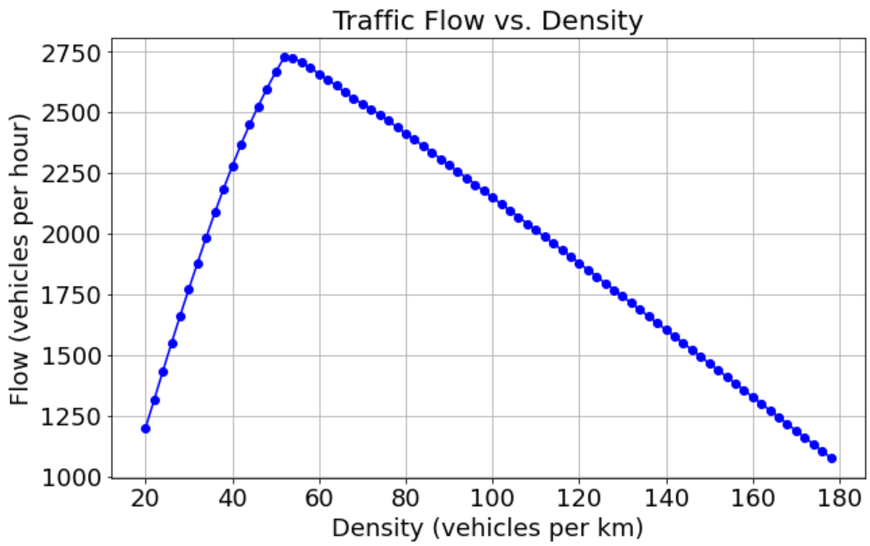


Figure 28: Traffic Flow vs Density

Figure 28 shows that at low traffic densities the flow is also low. Initially, as traffic density increases, the flow also increases while there is still enough room on the road for cars to move freely with very little interactions with the car ahead, allowing them to drive close to or at the maximum speed. However at sufficient densities, the traffic flow begins to decrease as the road becomes congested. This congestion causes drivers to reduce their speed, starting a chain effect and forming traffic jams which then decreases the traffic flow. Therefore this model accurately simulates how real life traffic behaves and the affect of density on flow. This plot has a similar shape to the plot of flow against density made earlier in this report using the Cellular Automaton model, which reiterates that it is an accurate model of traffic flow.

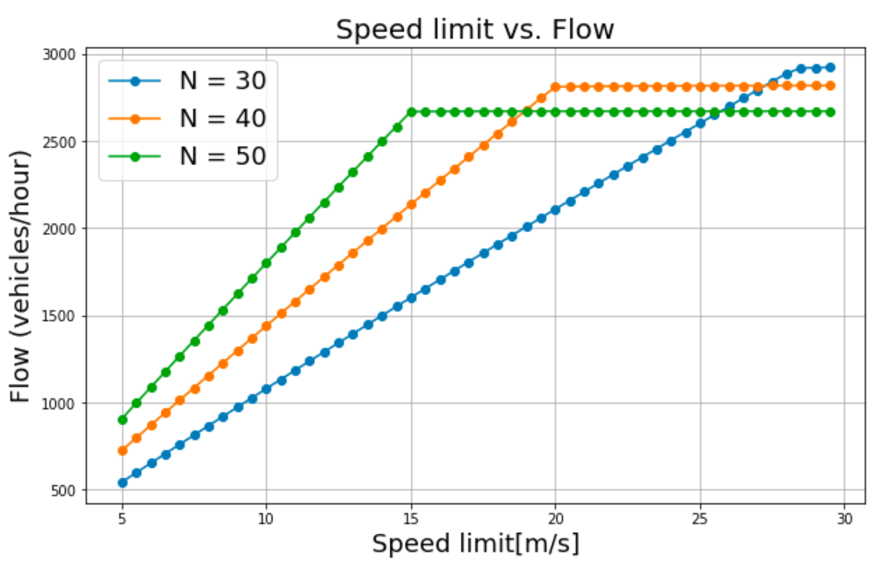


Figure 29: Speed limit vs. Flow

To research the impact of variable speed limits, we set the maximum speed that a driver wishes to drive at between a range of just under 5 m/s to 30 m/s. **Figure 29** shows the relationship between the speed limit and traffic flow with different densities: 30, 40 and 50 cars on a 1km circular road.

All three lines show a near-linear positive trend initially, which suggests that as the speed limit increases, the flow also increases. However, each line reaches a point where it plateaus, indicating that further increasing the speed limit does not result in an increased flow of traffic. This is a typical pattern when approaching a system's capacity or a maximum efficiency point.

Additionally, the line with a lower density (the blue line) plateaus later and at a higher flow. This implies that a decrease in number of cars or density is associated with higher performance or capacity in the system being measured until it reaches its limit.

Also, the levels at which all three lines plateau appear to be very close to one another, which might suggest a ceiling effect where increasing density above a certain point yields no further significant gains.

4 Classical Model: PDE Model

4.1 Introduction and Formulation

The continuum PDE model follows the classical theory approach to traffic flow modelling, largely following the work of Howison [8] and core concepts used in the Lighthill-Whitham-Richards (LWR) Model [9] . This is a macroscopic model that treats traffic flow as a fluid rather than individual cars, as seen in previous methods. Here, we use the results from both our cellular automata and car-following models and apply them to the macroscopic approach.

Although this model is the most simplistic of the three models investigated, it provides useful insight in assessing the effectiveness of our other models. Derived speed-density relationships from the other models are used to obtain kinematic relationships which show how these quantities change and predict flow behavior over time. In this section, we explain why these models are interdependent.

Firstly, we define our motives and the theory behind generating this model. The movement of traffic is described as the evolution of traffic density in both space and time.

We treat the cars as a continuum and assume that no cars leave or join the road (it is a closed system). This is a single-lane model so there is no overtaking. Suppose that x measures the distance along the road (km) and $\rho(x,t)$ is the density (cars per km) and $v(x,t)$ is the speed [8]. We note that the conservation law of cars is given by the following equation

$$\frac{\partial \rho}{\partial t} + \frac{\partial(\rho v)}{\partial x} = 0 \tag{22}$$

where x is distance along the road, $\rho(x,t)$ density of cars, $v(x,t)$ is speed and $q = \rho v$ is the flux/flow rate of cars.

We use the relationship below for the wave speed

$$c(\rho) = v(\rho) + \rho v'(\rho) \tag{23}$$

[8] and applying this to the previous equation gives

$$\frac{\partial(\rho v)}{\partial x} = (v(\rho) + \rho v'(\rho))\frac{\partial(\rho)}{\partial x}. \tag{24}$$

Therefore, the conservation of cars can be given as a nonlinear first-order wave equation.

$$\frac{\partial(\rho)}{\partial t} + c(\rho)\frac{\partial(\rho)}{\partial x} \tag{25}$$

This is the equation we use to model traffic flow. Now we introduce three different applications of our model, all investigating a different speed-density relationship: classic PDE, cellular automata and the car following model. To achieve the most realistic traffic flow behavior, we carefully selected speed-density relationships for both the cellular automata and car-following models we developed. These relationships were chosen to closely match the average speed observed in the simulation data. It's important to note that multiple models were considered for the CA and car-following models, using the simulation data and trends in **Figure 11** and **Figure 27** to provide a more relevant analysis.

4.2 Analysis

The use of the wave equation allows us to examine the emergence of congestion and shockwaves, by establishing a relationship between speed and velocity to determine traffic evolution. As traffic density increases, the wave equation predicts a decrease in traffic velocity, leading to the formation of regions with traffic jams or congestion. These congested regions create shockwaves that propagate backwards through the traffic flow. The wave equation also describes how disturbances in traffic density propagate along the roadway. A sudden change in traffic conditions, such as a bottleneck or an abrupt slowdown, results in the formation of a disturbance that travels upstream. This disturbance causes fluctuations in traffic density and velocity. This is especially useful in analysing the PDE model with the CA model implemented as bottleneck situations were investigated in **Section 2.2.3**. The model formed above ultimately allows us to assess the stability of the systems formulated and which factors lead to irregularities and trends in our simulations.

4.2.1 Linear Speed-Density Relationship

Firstly, we look at the general case of the partial differential equation continuum situation. The choice of the traffic flow model parameters and the speed-density relationship can impact the stability and behavior of the solution, as we'll see further below. Here we model traffic flow in a continuous manner, without discrete vehicles, in contrast to the other models.

The general most simplistic speed-density relationship is defined as follows

$$v = v_{max}\left(1 - \frac{\rho}{\rho_{max}}\right). \tag{26}$$

Figure 30 shows the traffic evolution in the general simple PDE continuum system.

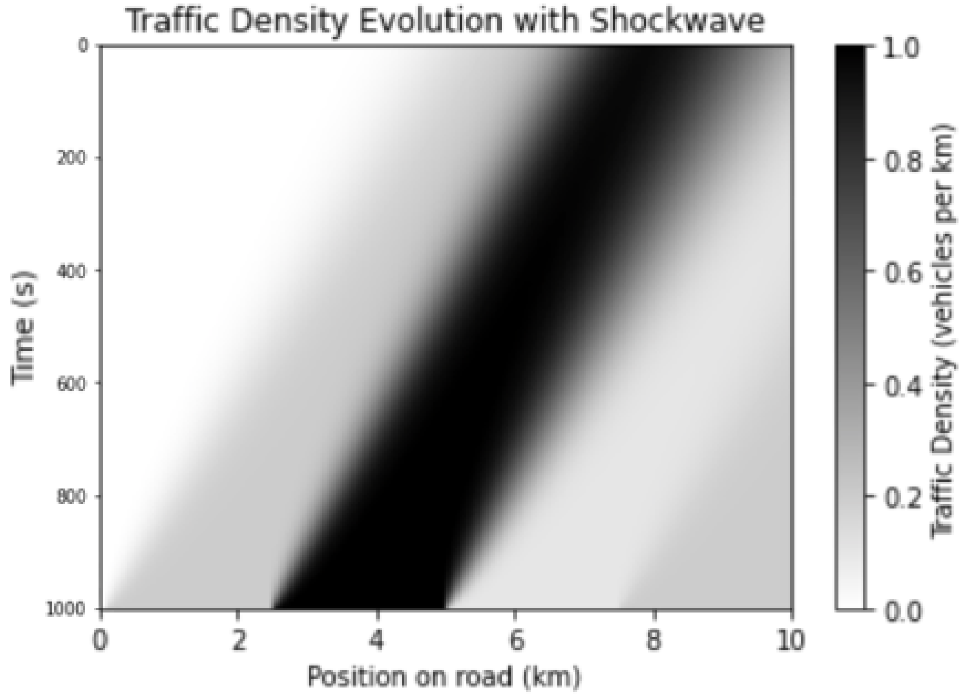


Figure 30: Traffic Flow vs Density: General PDE case

From **Figure 30**, we can see that, put simply, diagonal lines are created as a result of the traffic becoming more concentrated as time goes on. This propagation comes as a consequence of simulating **equation 22** into the model, where in this most simple case, the diagonal lines depict the movement of the initially less congested area. Traffic enters at time 0s and as it reaches the end of the closed loop, it re-enters the system at time 0, eventually reaching the traffic jam at 4km.

The model assumes that the system starts as slightly dispersed traffic where vehicles can travel more freely and condenses as vehicles catch up with denser traffic, which is what we'd expect from this basic model. This reflects the impact of vehicles at the front accelerating and creating space for those behind. This is why the traffic waves travel backwards and are most dense at the bottom left of the graph.

To summarise this result, on this closed loop, the simulation of a linear speed-density relationship shows a linear evolution of density change over the 1000 second period. In this system, there are no sudden changes in density (shockwaves) which is likely with a simplified system. Although on a small scale with limited assumptions, this model may accurately reflect the evolution of traffic flow, realistically this is too simple and not feasible in contemporary road systems where unexpected changes change the stability of density changes, which requires more non-linear complexities to be added.

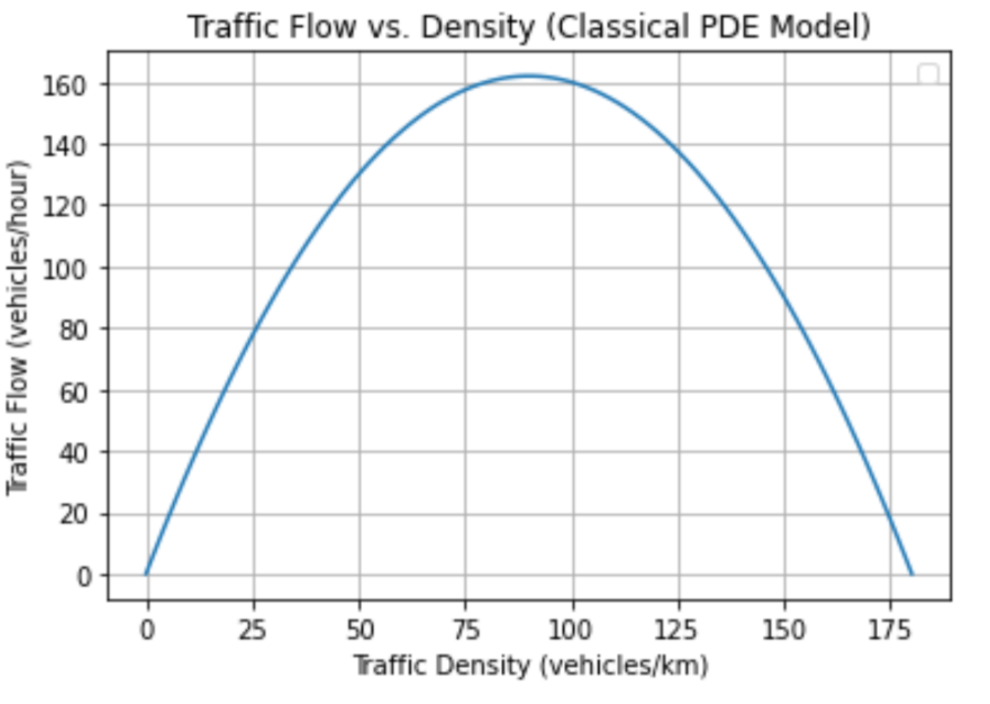


Figure 31: Traffic Flow vs Density for the General PDE case

Figure 31 shows a plot of traffic flow versus traffic density for the Classical PDE Model. As we can see this is a much simpler graph compared to the traffic flow versus density graphs for the CA and Car-Following models. Due to the linear nature of the initial speed-density relationship applied, we see a parabolic curve with maximum flow at 90 vehicles per km. In line with the LWR Model [9] this produces the curve we'd expect, due to the lack of disturbances in the system that cause shockwaves. This implies the conservation of cars is maintained, showing trends in the relationship from **Equation (24)**.

4.2.2 Implementing CA Model

Secondly, we apply the results from the Cellular Automata model to explore how our PDE simulation is altered. When fitting the speed-density relationships derived from the cellular automata and car following models, the PDE simulation should, in principle, reproduce similar traffic flow patterns. We investigate whether these applications verify that the continuum model (PDE) aligns with the behavior predicted by discrete models.

To model how drivers are affected by other drivers around them, a relationship between the speed of cars at a point and their density is established. In the general PDE model, **Equation (6)** was used to define the speed-density relationship. For the cellular automata model, a different non-linear speed-density relationship is obtained from the simulations earlier examined in this report. Using this, we can formulate the above equation as

$$v(\rho) = 5.17exp\left(-\frac{\rho}{0.30}\right). \tag{27}$$

We know this is accurate compared to real-life traffic situations because higher densities of traffic tend to have a lower average speed than lower densities and the equation above is a decreasing function of ρ and it starts from a maximum when $\rho = 0$ and then tends to zero as ρ increases. We then can state the flux of the cars as

$$q = \rho v = 5.17\rho exp\left(-\frac{\rho}{0.30}\right). \tag{28}$$

Looking at the cellular automata models produced above, a smooth initial condition is applied and the speed-density relationship is incorporated into the PDE model we've formulated. Simulating this in Python, the below result has been produced.

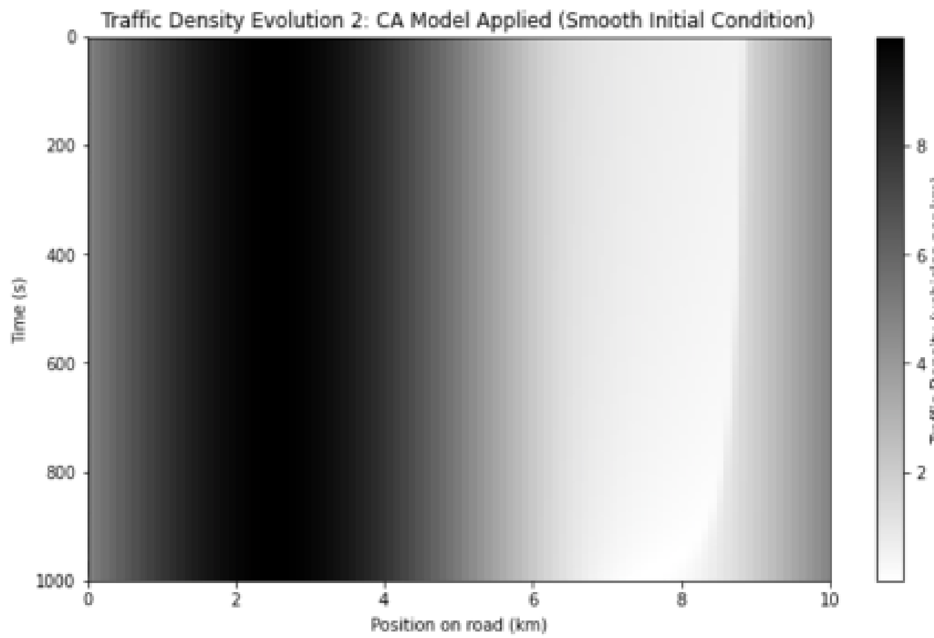


Figure 32: Traffic Flow vs Density: Cellular Automata Case

Figure 32 shows a much different distribution to the previous one, with the condensing of traffic being much less than the last. Notably, starting at around 9km, there's a gradual parabolic increase of traffic density as vehicles decelerate in anticipation of the upcoming high-density traffic flow around 2km in the closed loop. Here, it's clear that traffic jams form, where cars then have to wait for a substantial period (2km) in congestion before they can accelerate and move more freely in the open, low-density part of the loop.

The formation of waves should propagate backwards which is what makes the behavior of the simulation interesting after 4km. From literature focusing on shockwaves [10], the these high-density areas are formed by local disturbances such as sudden braking or traffic signal changes, which has been investigated in the Cellular Automata section. The anticipation of this congestion (driver behaviour) is investigated in this previous section, with assumptions on braking probabilities. As these factors are difficult to implement into the PDE model, this affects the wave formation in the PDE model simulation. The post-4km region exhibits a diffusion of density that deviates from the expected diagonal wave propagation. As seen in section 2, we'd expect diagonal traffic waves to form as time increases however, this speed density-relationship exhibits diffusion in density that is the same through time at this point around 4km in the loop.

Further investigation into the causes of this deviation suggests the need for additional complexity in the cellular automata model to better capture more realistic traffic evolution in this PDE model.

4.2.3 Implementing the Car-Following Model

Thirdly, we implement **equation (20)** into this PDE simulation, to investigate how the car-following approach performs.

By integrating this car-following speed-density relationship into the PDE model, we're introducing a more detailed consideration of how individual vehicle behaviors impact the overall traffic dynamics. The non-linear nature of the speed-density relationship reflects, slightly differently to the previous models, features of traffic such as the emergence of shockwaves and traffic jams.

The PDE model, with the incorporated car-following speed-density relationship, allows us to investigate and observe how microscopic interactions among vehicles contribute to macroscopic traffic patterns. It provides a bridge between the individual-level behavior captured by the car-following model and the aggregate traffic flow described by the PDE model.

A possible reason for the result in **Figure 26** is the discrepancy between the PDE model and CA model. The difference in approach (CA is discrete). There is a possibility that, while we can extract a speed-density relationship from a CA model, it might not be directly transferable to our PDE model. The continuous nature of the PDE might require adjustments to the relationship to accurately represent wave propagation in future research. The gradual build-up of traffic from 8km supports our theory around traffic congestion however the ambiguous traffic dispersion at 4km highlights this need for further adjustment.

Recalling from the car-following model, we have the nonlinear speed-density relationship

$$v = 90 \exp\left(\frac{\rho}{-100}\right) - 10. \tag{29}$$

Implementing this equation into our model and simulating in Python, gives us **Figure 33** below.

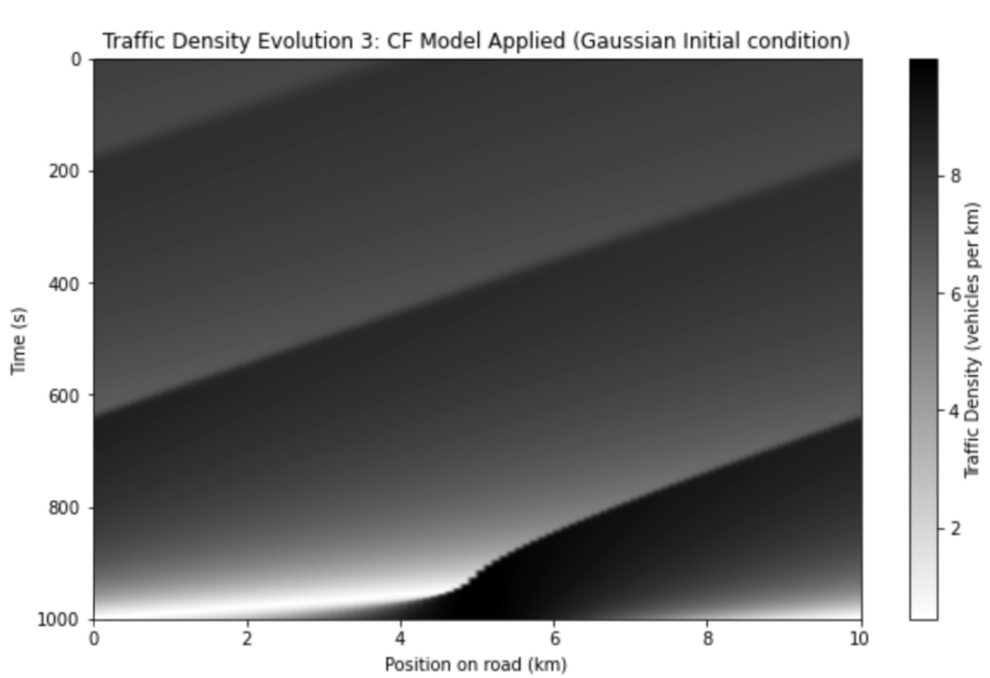


Figure 33: Traffic Flow vs Density: Car Following Model Case

Figure 33 shows a completely contrasting distribution to that of the previous two simpler simulations. Initially, we observe a distinct diagonal wave from 0-180 seconds, representing the gradual backward movement of this wave as traffic slows down in anticipation of the high-density area around 4km. This process repeats after approximately 240 seconds, where we see a full wave of this nature.

The non-linear nature of the car-following equation used becomes apparent towards the latter end of the time scale. Near 900 seconds in the simulation, an extreme traffic jam at 5km occurs. After the initial waves we've mentioned, the system destabilises after the build-up of traffic starting at around 700 seconds. Here we see the least dense section of this result (bottom left of the graph), where essentially the cars in the system 'catch up' and stop at this traffic jam. This sudden change in density represents the shockwave created in the non-linear system.

Taking into account the max speed, acceleration probabilities, the following distance and the reaction time gives a much deeper understanding of the results behind the simulation. The anticipation of traffic is much more apparent in this model, which would support our previous investigations in section 3, as these further complications in the factors inputted into the model are more likely to yield an accurate representation of traffic flow. This implies the usefulness of the result in simulating this into the PDE model, whereas when the CA model was implemented, a much more ambiguous result followed.

To summarise, the three simulations provide some context into each model type and how the continuum model bodes with the others. As we can see a simplistic, but useful result was obtained from **Figure 26**, where there is a fairly linear evolution of density throughout the simulation, with a much more condensed traffic system towards the latter stages of the loop. **Figure 27** gave us a much more vague result. The build-up of traffic was as we expected however the subsequent decrease in traffic density was found to not realistically represent this model. Our final result in **Figure 28**, gave a much more realistic simulation of traffic flow in this case, with there being both linear and non-linear elements of the model, with the formation of traffic waves (linear) and shockwaves (non-linear) towards the latter end of the loop.

5 Conclusion

In this investigation, we looked into three distinct models: the Cellular Automata models (CA), the Car-Following Model and the Classical Continuum Model (PDE) to analyse traffic flow and explore the differences and advantages/disadvantages of each model. By analysing these models both numerically and qualitatively, we have looked at the benefits of each modeling approach and how they interact with each other, as seen when applying them in Section 4.

Firstly, in the cellular automata model section, we use grid-based simulation methods to carefully analyse traffic flow dynamics. The adaptability of the CA approach is demonstrated through a variety of scenarios. The single-lane closed system demonstrates the formation of traffic congestion as vehicle density increases. We find that for this model the density of 0.6 cars per site maximises the traffic flow. In closed systems, we observe cyclic patterns of congestion, illustrating how congestion evolves and disappears over time. We extend the simulation to open systems and identify bottleneck effects and their impact on traffic. For the open system we find that a density of 0.2 cars per site maximises the traffic flow. We then incorporated traffic lights into the CA framework allows us to accurately model how traffic lights affect traffic flow . In a two-lane model, we explore the impact of lane changing rules on traffic flow by comparing symmetric and asymmetric configurations. Our findings highlight the complexity introduced by additional lanes and the delicate balance between vehicle distribution and flow efficiency. By comparing with simulation results in the literature, our model exhibits similar density-flow relationships, demonstrating the accuracy and reliability of the simulation. We find that a density of 0.1 cars per site maximises traffic flow. The CA model provides a powerful platform for understanding traffic patterns, not only deepening our understanding of traffic flows, but also laying the foundation for developing strategies to alleviate congestion and improve traffic management in practice.

Secondly, from the Car-Following Model we deduced that adjusting parameters, such as whether a driver overestimates or underestimates breaking rates, leads to contrasting fluctuations in traffic flow. Results showed that overestimating the rate at which the driver in front will brake causes minimal disturbances whereas an underestimate leads to amplified fluctuations and more erratic traffic flow. On a closed system, results showed a non-linear relationship between traffic flow and density, indicating that traffic flow is optimised at an approximate density of 50 vehicles per km. This can be used to maximise the efficiency of road traffic and ensure the highest flow possible.

Thirdly, the classical approach explored how the PDE model in traffic flow simulations captured the continuous evolution of traffic density over space and time, both in a linear and non-linear fashion. Using partial derivative simulations and relationships between speed, density, and the flow of vehicles we modeled how congestion builds up and dissipates in each model we've investigated. Like the CA and Car-Following Models, our aim is to achieve maximum flow of which we achieved this at approximately 90 vehicles per km for the traffic density. The classical approach gave further insight into the concepts behind the CA and car-following models, however this model lacks complexity. It doesn't accurately express a realistic traffic flow model with more factors included, such as the traffic light systems investigated in the CA model.

For these models, we conclude that the model with the most complex and realistic rules is the car-following model. This can accurately predict traffic flow and should be recommended to traffic engineers to provide the most accurate predictions. Although this model should be the most accurate it is more complicated to add extra lanes and traffic lights due to the complexity of the model, meaning we struggled to compare the more complex models that we were able to achieve in the cellular automata models. Therefore, if we were required to look at a multiple lane road the result would be more easily simulated by the cellular automata model.

6 Further Considerations

To further our investigation into traffic flow models, interactions at intersections could be added to the simulations for the cellular automata model. This could be done by designing additional rules for vehicles yielding, stopping and turning. Environmental factors could also be taken into account, including weather conditions like rain or snow or obstructions like debris on the road. This is more realistic and would show how external factors affect driving behaviour and the flow of traffic.

The car-following model could be extended to multiple lanes. This would simulate traffic more accurately as the interactions between cars in neighbouring lanes would be considered, rather than the driver only responding to the car in front on the same lane. Stability analysis of the models to examine how small perturbations affect a uniform flow of traffic can be carried out. This would investigate if perturbations decay, meaning the uniform flow is stable, or if they grow.

For the classical model the effects of accidents or incidents could be incorporated to show disruptions to the traffic flow and congestion. A more complicated version of the model could be explored to investigate systems with more lanes, different driver behaviours and randomized events. With further investigations in the CA and car-following models, useful research could be carried out using all relevant models to optimize traffic flow systems used.

Smart motorways aim to prevent traffic jams using variable speed limits. They try to keep traffic moving even if it has to move at a lower speed to ensure driver safety and reduce congestion on the road. With this in mind, dynamic speed limits could be introduced into all the models we discussed in the report where the speed limit is adjusted based on the traffic density or road conditions or other factors.

7 References

[1] B.Vescio C.Cosentino and F.Amato. Cellular automata. doi:https://doi.org/10.1007/978-1-4419-9863-7_989., 2013. Accessed: 2024-03-01.

[2] K.Nagel and M.Schreckenberg. A cellular automaton model for freeway traffic. https://hal.science/jpa-00246697/document, 1992. Accessed: 2024-03-02.

[3] Wikipedia. Rush hour. https://en.wikipedia.org/wiki/Rush_hour: :text=Normally Accessed: 2024-03-03.

[4] L.Wagner. A brief history of traffic lights. https://www.inclusivecitymaker.com/1868-2019-a-brief-history-of-traffic-lights/: :text=December Accessed: 2024-03-10.

[5] M.Schreckenberg A.Latour M.Rickert, K.Nagel. Two lane traffic simulations using cellular automata. https://www.sciencedirect.com/science/article/pii/0378437195004424, 1995. Accessed: 2024-03-08.

[6] P.G.Gipps. A behavioural car-following model for computer simulation. https://doi.org/10.1016/0191-2615(81)90037-0, 1981. Accessed: 2024-02-20.

[7] B.Custard. Uk speed limit guide 2023 – all you need to know. https://www.motorpoint.co.uk/guides/speed-limit-guide, 2023. Accessed: 2024-03-01.

[8] Sam Howison. Practical applied mathematics modelling, analysis, approximation. https://people.maths.ox.ac.uk/fowler/courses/tech/sdh.pdf, 2004. Accessed: 2024-03-06.

[9] A.Kesting M.Treiber. Lighthill-whitham-richards model. https://link.springer.com/chapter/10.1007/978-3-642-32460-4_8citeas, 2013. Accessed: 2024-03-12.

[10] L.Wagner. Linear and non-linear waves. https://ebookcentral.proquest.com/lib/nottingham/reader.action?docID=70820 1999. Accessed: 2024-03-12.

8 Appendix

Appendices

8.1 Code for Sing-Lane CA Model

```
#%%Setting of Single lane Model
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

# Set model parameters
road_length = 100 # Length of the road
max_speed = 5 # vmax
deceleration_prob = 0.3 # randomization
steps = 100 # steps

# Update function
def update_road(road, max_speed, deceleration_prob):
    new_road = -np.ones_like(road)
    for i, speed in enumerate(road):
        if speed >= 0:
            # Calculate distance to the next car
            distance = 1
            while road[(i + distance) % road_length] == -1 and distance <= max_speed:
                distance += 1

            # Acceleration
            if speed < max_speed and distance > speed + 1:
                speed += 1
```

```
            # Slowing down
            speed = min(speed, distance - 1)

            # Randomization
            if speed > 0 and np.random.rand() < deceleration_prob:
                speed -= 1

            # Car motion
            new_road[(i + speed) % road_length] = speed
    return new_road

# Initialize road function
def initialize_road(density):
    road = -np.ones(road_length, dtype=int) # -1 represents no car
    initial_cars = np.random.choice(range(road_length), size=int(density * road_length), replace=False)
    road[initial_cars] = np.random.randint(0, max_speed + 1, size=len(initial_cars))
    return road

# Simulate traffic function
def simulate_traffic(road, steps, max_speed, deceleration_prob):
    road_states = []
    for _ in range(steps):
        road_states.append(road.copy())
        road = update_road(road, max_speed, deceleration_prob)
    return road_states

#%%Plot Single lane with closed system
# Define specific density
car_density = 0.3

# Initialize road with specific density
road = initialize_road(car_density)

# Simulate traffic
road_states_speed = simulate_traffic(road, steps, max_speed, deceleration_prob)

# Define Grey color map
cmap_greyscale = plt.cm.get_cmap('Greys', max_speed + 1)

# Plot
fig, ax = plt.subplots(figsize=(10, 6))
ax.set_xlabel("Position")
ax.set_ylabel("Time Step")
img = ax.imshow(road_states_speed, cmap=cmap_greyscale, interpolation="nearest", animated=True, vmin=-1, vmax=max_speed)
ax.set_title("Cellular Automata for Single Lane with Velocity in Greyscale")
colorbar = plt.colorbar(img, ticks=range(max_speed + 1), label='Velocity')
colorbar.set_label('Velocity', rotation=270, labelpad=15)

# Update plot function
def update_anim_greyscale(i):
    if i == 0:
```

```python
        return img,
    img.set_array(road_states_speed[:i])
    return img,

ani_greyscale = animation.FuncAnimation(fig, update_anim_greyscale, frames=steps,
    interval=50, blit=True)

plt.show()

#%%Low and High Density
# Low and high density settings
low_density = 0.1
high_density = 0.7

# Initialize road for low and high density
road_low_density = initialize_road(low_density)
road_high_density = initialize_road(high_density)

# Simulate low and high density traffic
road_states_low = simulate_traffic(road_low_density, steps, max_speed, deceleration_prob)
road_states_high = simulate_traffic(road_high_density, steps, max_speed,
    deceleration_prob)

# Create subplots for low and high density traffic
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))

# Low density traffic
img1 = ax1.imshow(road_states_low, cmap=cmap_greyscale, interpolation="nearest",
    vmin=-1, vmax=max_speed)
ax1.set_title("Low Density $\\rho = 0.1$", fontsize=18)
ax1.set_xlabel("Position", fontsize=18)
ax1.set_ylabel("Time Step", fontsize=18)

# High density traffic
img2 = ax2.imshow(road_states_high, cmap=cmap_greyscale, interpolation="nearest",
    vmin=-1, vmax=max_speed)
ax2.set_title("High Density $\\rho = 0.7$", fontsize=18)
ax2.set_xlabel("Position", fontsize=18)
ax2.set_ylabel("Time Step", fontsize=18)

plt.tight_layout()
plt.show()


#%%Draw Car Trajectories
car_density=0.3
steps=200

def update_road_with_wrap(road, car_ids, max_speed, deceleration_prob):
    new_road = -np.ones_like(road)
    new_car_ids = -np.ones_like(road)
    car_moves = {} # Dictionary to track the moves

    for i, speed in enumerate(road):
```

```
        if speed >= 0:
            distance = 1
            while road[(i + distance) % road_length] == -1 and distance <= max_speed:
                distance += 1
            if speed < max_speed and distance > speed + 1:
                speed += 1
            speed = min(speed, distance - 1)
            if speed > 0 and np.random.rand() < deceleration_prob:
                speed -= 1
            new_position = (i + speed) % road_length
            new_road[new_position] = speed
            new_car_ids[new_position] = car_ids[i]
            car_moves[car_ids[i]] = (new_position, speed)
    return new_road, new_car_ids, car_moves


def initialize_road_with_ids(density):
    road = -np.ones(road_length, dtype=int)
    car_ids = -np.ones(road_length, dtype=int)
    initial_positions = np.random.choice(range(road_length), size=int(density *
        road_length), replace=False)
    initial_ids = np.arange(len(initial_positions))
    road[initial_positions] = np.random.randint(0, max_speed + 1,
        size=len(initial_positions))
    car_ids[initial_positions] = initial_ids
    return road, car_ids


road, car_ids = initialize_road_with_ids(car_density)

car_trajectories = {car_id: [] for car_id in car_ids if car_id != -1}

for time_step in range(1, steps + 1):
    road, car_ids, car_moves = update_road_with_wrap(road, car_ids, max_speed,
        deceleration_prob)
    for car_id, (new_position, speed) in car_moves.items():
        if car_trajectories[car_id] and (new_position < car_trajectories[car_id][-1][1])
            and speed > 0:
            car_trajectories[car_id].append(None)
        car_trajectories[car_id].append((time_step, new_position))

plt.figure(figsize=(12, 12))

for car_id, trajectory in car_trajectories.items():
    segments = []
    current_segment = []
    for point in trajectory:
        if point is None:
            if current_segment:
                segments.append(current_segment)
                current_segment = []
        else:
            current_segment.append(point)
    if current_segment: # Add the last segment
```

```
        segments.append(current_segment)

    for segment in segments:
        if segment:
            times, positions = zip(*segment)
            plt.plot(positions, times, lw=1)

plt.xlabel('Position on road', fontsize=18)
plt.ylabel('Time step', fontsize=18)
plt.title('Traffic Flow Trajectories with Periodic Boundary', fontsize=18)
plt.xlim(0, road_length)
plt.ylim(0, steps)
plt.gca().invert_yaxis() # Invert the y-axis so that time increases downwards
plt.show()


#%%Traffic flow vs density
def calculate_flow(road_states, T):
    road_length = len(road_states[0])
    flow = np.zeros(road_length)

    # Calculate flow
    for t in range(T):
        for i in range(road_length):
            if road_states[t][i] > 0 and ((i + road_states[t][i]) % road_length) == (i +
                1) % road_length:
                flow[i] += 1

    flow /= T

    return np.mean(flow)

densities = np.linspace(0, 1, 50)
flow_10_steps = []
flow_1000_steps = []

for density in densities:
    # Initialize road with specific density
    road = initialize_road(density)

    # Simulate traffic for 10 and 1000 time steps
    road_states_10 = simulate_traffic(road, 100, max_speed, deceleration_prob)
    flow_10 = calculate_flow(road_states_10, 100)
    flow_10_steps.append(flow_10)

    road_states_1000 = simulate_traffic(road, 10000, max_speed, deceleration_prob)
    flow_1000 = calculate_flow(road_states_1000, 10000)
    flow_1000_steps.append(flow_1000)

# Traffic flow vs. Density
plt.figure(figsize=(10, 6))
plt.scatter(densities, flow_10_steps, label='100 time steps')
plt.plot(densities, flow_1000_steps, label='10000 time steps')
plt.xlabel('Density (cars per site)', fontsize=18)
```

```
plt.ylabel('Traffic flow (cars per time step)', fontsize=18)
plt.title('Traffic Flow vs. Density', fontsize=18)
plt.legend()
plt.grid(True)
plt.show()

#%%Traffic flow under different densities
import numpy as np
import matplotlib.pyplot as plt

# Set model parameters
road_length = 400 # Length of the road
max_speed = 5 # Maximum speed
deceleration_prob = 0.3 # Probability of random deceleration
densities = np.linspace(0.1, 0.8, 8) # Range of densities to simulate
steps = 400 # Number of simulation steps

# Update function for the traffic model
def update_road(road, max_speed, deceleration_prob):
    new_road = -np.ones_like(road)
    for i, speed in enumerate(road):
        if speed >= 0:
            distance = 1
            while road[(i + distance) % road_length] == -1 and distance <= max_speed:
                distance += 1

            # Acceleration
            if speed < max_speed:
                speed += 1

            # Slowing down due to other cars
            speed = min(speed, distance - 1)

            # Random deceleration
            if speed > 0 and np.random.rand() < deceleration_prob:
                speed -= 1

            # Car movement
            new_road[(i + speed) % road_length] = speed
    return new_road

# Initialize road function
def initialize_road(road_length, density):
    road = -np.ones(road_length, dtype=int)
    filled_cells = np.random.choice(road_length, size=int(density * road_length),
        replace=False)
    road[filled_cells] = 0 # Change here: set occupied cells to 0 (will be black)
    return road

# Simulate traffic for different densities and visualize
fig, axes = plt.subplots(len(densities), 1, figsize=(12, 2 * len(densities)),
    sharex=True)

for ax, density in zip(axes, densities):
```

```python
    road = initialize_road(road_length, density)
    road_states = [road.copy()]
    for _ in range(steps):
        road = update_road(road, max_speed, deceleration_prob)
        road_states.append(road.copy())

    # Convert road states for visualization: 1 for empty, 0 for occupied
    road_states_visual = np.where(np.array(road_states) == -1, 1, 0)

    # Visualization
    ax.imshow(road_states_visual, cmap='gray', interpolation='nearest', aspect='auto')
    ax.set_ylabel(f'Density={density:.2f}', fontsize=18)
    ax.set_yticks([])

axes[-1].set_xlabel('Position on Road', fontsize=18)
plt.suptitle('Traffic Simulation with Cellular Automata for Different Densities',
    fontsize=20)
plt.tight_layout(rect=[0, 0, 1, 0.96]) # Adjust layout to not overlap the title
plt.show()

#%%Bottleneck
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

# Model parameters
road_length = 100 # Length of the road
max_speed = 5 # Maximum speed
deceleration_prob = 0.3 # Probability of random deceleration
steps = 100 # Number of simulation steps
delete_sites = 6 # Number of sites to delete cars at the right side

# Initialize road with specific density
def initialize_road_with_density(road_length, density=None):
    road = -np.ones(road_length, dtype=int) # -1 represents no car
    if density is not None:
        num_cars = int(density * road_length)
        positions = np.random.choice(road_length, size=num_cars, replace=False)
        road[positions] = np.random.randint(0, max_speed + 1, size=num_cars)
    return road

# Update function with open boundary conditions
def update_road_bottleneck(road, max_speed, deceleration_prob, delete_sites):
    new_road = -np.ones_like(road)
    if road[0] == -1:
        road[0] = 0 # Occupying with a car of velocity 0 if the leftmost site is empty

    for i, speed in enumerate(road):
        if speed >= 0:
            distance = 1
            while road[(i + distance) % road_length] == -1 and distance <= max_speed:
                distance += 1

            if speed < max_speed and distance > speed + 1:
```

```
                speed += 1

            speed = min(speed, distance - 1)

            if speed > 0 and np.random.rand() < deceleration_prob:
                speed -= 1

            new_position = (i + speed) % road_length
            if new_position < road_length - delete_sites:
                new_road[new_position] = speed

    new_road[-delete_sites:] = -np.ones(delete_sites)
    return new_road

# Simulate traffic
def simulate_traffic(road_length, density, steps, max_speed, deceleration_prob, delete_sites):
    road = initialize_road_with_density(road_length, density)
    road_states = []
    for _ in range(steps):
        road_states.append(road.copy())
        road = update_road_bottleneck(road, max_speed, deceleration_prob, delete_sites)
    return road_states

# Visualization in 1*2 layout
fig, axs = plt.subplots(1, 2, figsize=(15, 6))

# Simulation without initial density
road_states_bottleneck = simulate_traffic(road_length, None, steps, max_speed, deceleration_prob, delete_sites)
img1 = axs[0].imshow(road_states_bottleneck, cmap='Greys', interpolation="nearest", animated=True, vmin=-1, vmax=max_speed)
axs[0].set_title("Traffic in a Bottleneck Situation with Empty Initial Road", fontsize=16)
axs[0].set_xlabel("Position", fontsize=18)
axs[0].set_ylabel("Time Step", fontsize=18)

# Simulation with specific initial density
car_density = 0.3 # Initial density of cars
road_states_bottleneck_density = simulate_traffic(road_length, car_density, steps, max_speed, deceleration_prob, delete_sites)
img2 = axs[1].imshow(road_states_bottleneck_density, cmap='Greys', interpolation="nearest", animated=True, vmin=-1, vmax=max_speed)
axs[1].set_title("Traffic in a Bottleneck Situation with Given Initial Density", fontsize=16)
axs[1].set_xlabel("Position", fontsize=18)
axs[1].set_ylabel("Time Step", fontsize=18)

# Adjust colorbar to be shared by subplots
plt.colorbar(img2, ax=axs[1], ticks=range(max_speed + 1), label='Velocity')

plt.tight_layout()
plt.show()
```

```python
#%%Traffic flow vs. Density

# Set model parameters
road_length = 500
max_speed = 5
deceleration_prob = 0.3
steps = 500
delete_sites = 6

# Modified update_road_bottleneck function to return the number of moves
def update_road_bottleneck_and_count_moves(road, max_speed, deceleration_prob,
    delete_sites):
    new_road = -np.ones_like(road)
    moves = 0 # Count of total moves in this step
    if road[0] == -1:
        road[0] = 0

    for i, speed in enumerate(road):
        if speed >= 0:
            distance = 1
            while road[(i + distance) % road_length] == -1 and distance <= max_speed:
                distance += 1

            if speed < max_speed and distance > speed + 1:
                speed += 1

            speed = min(speed, distance - 1)

            if speed > 0 and np.random.rand() < deceleration_prob:
                speed -= 1

            new_position = (i + speed) % road_length
            if new_position < road_length - delete_sites:
                new_road[new_position] = speed
                moves += speed # Add speed to moves as it represents the distance moved

    new_road[-delete_sites:] = -np.ones(delete_sites)
    return new_road, moves

# Modified simulate_traffic function to calculate flow based on total moves
def simulate_traffic_and_calculate_flow_based_on_moves(road, steps, max_speed,
    deceleration_prob, delete_sites):
    total_moves = 0 # Total moves for all cars
    for _ in range(steps):
        road, moves = update_road_bottleneck_and_count_moves(road, max_speed,
            deceleration_prob, delete_sites)
        total_moves += moves
    flow = total_moves / (road_length * steps) # Average flow based on total moves
    return flow

# Calculate flow for different densities
densities = np.linspace(0.05, 0.85, 41)
flows = []

```

```python
for density in densities:
    road = initialize_road_with_density(road_length, density)
    flow = simulate_traffic_and_calculate_flow_based_on_moves(road, steps, max_speed,
        deceleration_prob, delete_sites)
    flows.append(flow)

# Plotting
plt.figure(figsize=(10, 6))
plt.plot(densities, flows, marker='o')
plt.title("Traffic Flow vs. Initial Density", fontsize=18)
plt.xlabel("Initial Density", fontsize=18)
plt.ylabel("Traffic Flow", fontsize=18)
plt.grid(True)
plt.show()



#%%Traffic Light with yellow light
import numpy as np
import matplotlib.pyplot as plt

# Model parameters
road_length = 400 # Length of the road
max_speed = 5 # Maximum speed
deceleration_prob = 0.3 # Probability of random deceleration
steps = 400 # Number of simulation steps
delete_sites = 6 # Sites to delete cars at the right side
traffic_light_cycle = 40 # Total cycle length (green + yellow + red)
green_light_duration = 20 # Green light duration
yellow_light_duration = 5 # Yellow light duration
traffic_light_position = road_length // 2 # Position of the traffic light

# Initialize road with specific density, only on the left side of the traffic light
def initialize_road_with_density(road_length, density=None):
    road = -np.ones(road_length, dtype=int) # -1 represents no car
    if density is not None:
        # Calculate the number of cars based on the density and the length of the road
            before the traffic light
        num_cars = int(density * traffic_light_position) # Only populate left side of the
            road
        positions = np.random.choice(range(traffic_light_position), size=num_cars,
            replace=False) # Choose positions only before the traffic light
        road[positions] = 0 # Initialize cars with velocity 0
    return road


# Determine traffic light status
def get_traffic_light_status(step):
    cycle_position = step % traffic_light_cycle
    if cycle_position < green_light_duration:
        return 'green'
    elif cycle_position < green_light_duration + yellow_light_duration:
        return 'yellow'
    else:
```

```
        return 'red'

# Update function with open boundary conditions and a traffic light
def update_road_bottleneck(road, max_speed, deceleration_prob, delete_sites, step):
    new_road = -np.ones_like(road)
    traffic_light_status = get_traffic_light_status(step)

    if road[0] == -1:
        road[0] = 0 # Occupying with a car of velocity 0 if the leftmost site is empty

    for i, speed in enumerate(road):
        if speed >= 0:
            distance = 1
            while road[(i + distance) % road_length] == -1 and distance <= max_speed:
                distance += 1

            if i + distance > traffic_light_position and i < traffic_light_position:
                if traffic_light_status == 'red':
                    distance = min(distance, traffic_light_position - i)
                elif traffic_light_status == 'yellow':
                    # Try to stop if possible, else pass the yellow light if too close
                    if distance - 1 < max_speed and speed > 1:
                        speed = 0

            if speed < max_speed and distance > speed + 1:
                speed += 1

            speed = min(speed, distance - 1)

            if speed > 0 and np.random.rand() < deceleration_prob:
                speed -= 1

            new_position = (i + speed) % road_length
            if new_position < road_length - delete_sites:
                new_road[new_position] = speed

    new_road[-delete_sites:] = -np.ones(delete_sites) # Remove cars at the end
    return new_road

# Simulate traffic with a traffic light
def simulate_traffic(road_length, density, steps, max_speed, deceleration_prob,
    delete_sites):
    road = initialize_road_with_density(road_length, density)
    road_states = []
    for step in range(steps):
        road_states.append(road.copy())
        road = update_road_bottleneck(road, max_speed, deceleration_prob, delete_sites,
            step)
    return road_states

density = 0.3 # Example density
road_states_bottleneck = simulate_traffic(road_length, density, steps, max_speed,
    deceleration_prob, delete_sites)
```

```
# Convert road states for visualization: 1 for empty, 0 for occupied
road_states_visual = np.where(np.array(road_states_bottleneck) == -1, 1, 0)

# Create figure and plot
plt.figure(figsize=(12, 12))
ax = plt.gca() # Get current axes
im = ax.imshow(road_states_visual, cmap='gray', interpolation="nearest", aspect='auto')

# Plot traffic light status
for step in range(steps):
    traffic_light_status = get_traffic_light_status(step)
    if traffic_light_status == 'green':
        ax.axhline(y=step, color='green', xmin=0.495, xmax=0.505, linewidth=2)
    elif traffic_light_status == 'yellow':
        ax.axhline(y=step, color='yellow', xmin=0.495, xmax=0.505, linewidth=2)
    elif traffic_light_status == 'red':
        ax.axhline(y=step, color='red', xmin=0.495, xmax=0.505, linewidth=2)

# Setting the title, labels and colorbar
plt.title(f"Traffic Flow with Traffic Light (Density = {density})", fontsize=20)
plt.xlabel("Position on Road", fontsize=18)
plt.ylabel("Time Step", fontsize=18)

plt.tight_layout()
plt.show()

#%%Traffic Light(Only Green and Red)
import numpy as np
import matplotlib.pyplot as plt

# Model parameters
road_length = 400 # Length of the road
max_speed = 5 # Maximum speed
deceleration_prob = 0.3 # Probability of random deceleration
steps = 400 # Number of simulation steps
delete_sites = 6 # Sites to delete cars at the right side
traffic_light_cycle = 40 # Total cycle length (green + yellow + red)
green_light_duration = 20 # Green light duration
traffic_light_position = road_length // 2 # Position of the traffic light

# Initialize road with specific density, only on the left side of the traffic light
def initialize_road_with_density(road_length, density=None):
    road = -np.ones(road_length, dtype=int) # -1 represents no car
    if density is not None:
        # Calculate the number of cars based on the density and the length of the road
            before the traffic light
        num_cars = int(density * traffic_light_position) # Only populate left side of the
            road
        positions = np.random.choice(range(traffic_light_position), size=num_cars,
            replace=False) # Choose positions only before the traffic light
        road[positions] = 0 # Initialize cars with velocity 0
    return road


```

```python
# Determine traffic light status
def get_traffic_light_status(step):
    cycle_position = step % traffic_light_cycle
    if cycle_position < green_light_duration:
        return 'green'
    else:
        return 'red'

# Update function with open boundary conditions and a traffic light
def update_road_bottleneck(road, max_speed, deceleration_prob, delete_sites, step):
    new_road = -np.ones_like(road)
    traffic_light_status = get_traffic_light_status(step)

    if road[0] == -1:
        road[0] = 0 # Occupying with a car of velocity 0 if the leftmost site is empty

    for i, speed in enumerate(road):
        if speed >= 0:
            distance = 1
            while road[(i + distance) % road_length] == -1 and distance <= max_speed:
                distance += 1

            if i + distance > traffic_light_position and i < traffic_light_position:
                if traffic_light_status == 'red':
                    distance = min(distance, traffic_light_position - i)


            if speed < max_speed and distance > speed + 1:
                speed += 1

            speed = min(speed, distance - 1)

            if speed > 0 and np.random.rand() < deceleration_prob:
                speed -= 1

            new_position = (i + speed) % road_length
            if new_position < road_length - delete_sites:
                new_road[new_position] = speed

    new_road[-delete_sites:] = -np.ones(delete_sites) # Remove cars at the end
    return new_road

# Simulate traffic with a traffic light
def simulate_traffic(road_length, density, steps, max_speed, deceleration_prob,
    delete_sites):
    road = initialize_road_with_density(road_length, density)
    road_states = []
    for step in range(steps):
        road_states.append(road.copy())
        road = update_road_bottleneck(road, max_speed, deceleration_prob, delete_sites,
            step)
    return road_states

density = 0.3 # Example density
```

```
road_states_bottleneck = simulate_traffic(road_length, density, steps, max_speed,
    deceleration_prob, delete_sites)

# Convert road states for visualization: 1 for empty, 0 for occupied
road_states_visual = np.where(np.array(road_states_bottleneck) == -1, 1, 0)

# Create figure and plot
plt.figure(figsize=(12, 12))
ax = plt.gca() # Get current axes
im = ax.imshow(road_states_visual, cmap='gray', interpolation="nearest", aspect='auto')

# Plot traffic light status
for step in range(steps):
    traffic_light_status = get_traffic_light_status(step)
    if traffic_light_status == 'green':
        ax.axhline(y=step, color='green', xmin=0.495, xmax=0.505, linewidth=2)
    elif traffic_light_status == 'red':
        ax.axhline(y=step, color='red', xmin=0.495, xmax=0.505, linewidth=2)

# Setting the title, labels and colorbar
plt.title(f"Traffic Flow with Green and Red Traffic Light (Density = {density})",
    fontsize=20)
plt.xlabel("Position on Road", fontsize=18)
plt.ylabel("Time Step", fontsize=18)

plt.tight_layout()
plt.show()

#%%
from scipy.optimize import curve_fit
import numpy as np
import matplotlib.pyplot as plt

# Linear
def linear_model(density, vmax, rho_max):
    return vmax * (1 - density / rho_max)

# Log
def log_model(density, vmax, rho_max):
    return vmax - vmax * np.log(density / rho_max + 1)

# Exp
def exp_model(density, vmax, rho_max):
    return vmax * np.exp(-density / rho_max)

# Greenberg
def greenberg_model(density, vmax, rho_max):
    return vmax * np.log(rho_max / density)


road_length = 400
max_speed = 5
deceleration_prob = 0.3
steps = 400
```

```python
delete_sites = 6

def update_road_bottleneck(road, max_speed, deceleration_prob, delete_sites):
    new_road = -np.ones_like(road)
    if road[0] == -1:
        road[0] = np.random.randint(0, max_speed)

    for i, speed in enumerate(road):
        if speed >= 0:
            distance = 1
            while road[(i + distance) % road_length] == -1 and distance <= max_speed:
                distance += 1

            if speed < max_speed and distance > speed + 1:
                speed += 1

            speed = min(speed, distance - 1)

            if speed > 0 and np.random.rand() < deceleration_prob:
                speed -= 1

            new_position = (i + speed) % road_length
            if new_position < road_length - delete_sites:
                new_road[new_position] = speed

    new_road[-delete_sites:] = -np.ones(delete_sites)
    return new_road

def simulate_traffic(density):
    road = -np.ones(road_length, dtype=int)
    for _ in range(int(density * road_length)):
        while True:
            pos = np.random.randint(road_length)
            if road[pos] == -1:
                road[pos] = np.random.randint(0, max_speed + 1)
                break

    speeds = []
    for _ in range(steps):
        road = update_road_bottleneck(road, max_speed, deceleration_prob, delete_sites)
        speeds.append(np.mean(road[road >= 0]))

    return np.nanmean(speeds)

densities = np.linspace(0, 1, 50)
average_speeds = []

for density in densities:
    average_speed = simulate_traffic(density)
    average_speeds.append(average_speed)

params_linear, _ = curve_fit(linear_model, densities, average_speeds, p0=[max_speed, 1])
params_log, _ = curve_fit(log_model, densities, average_speeds, p0=[max_speed, 1],
    bounds=(0, np.inf))
```

```
params_exp, _ = curve_fit(exp_model, densities, average_speeds, p0=[max_speed, 1],
    bounds=(0, np.inf))

positive_densities = densities[densities > 0]
positive_average_speeds = np.array(average_speeds)[densities > 0]

params_greenberg, _ = curve_fit(greenberg_model, positive_densities,
    positive_average_speeds, p0=[max_speed, 1], bounds=(0, np.inf))

params_linear, params_log, params_exp, params_greenberg

#%%Plot

plt.figure(figsize=(12, 8))

plt.scatter(densities, average_speeds, color='black', label='Simulation Data')

predicted_speeds_linear = linear_model(densities, *params_linear)
plt.plot(densities, predicted_speeds_linear, label='Linear Model')

predicted_speeds_log = log_model(densities, *params_log)
plt.plot(densities, predicted_speeds_log, label='Log Model')

predicted_speeds_exp = exp_model(densities, *params_exp)
plt.plot(densities, predicted_speeds_exp, label='Exp Model')

predicted_speeds_greenberg = greenberg_model(positive_densities, *params_greenberg)
plt.plot(positive_densities, predicted_speeds_greenberg, label='Greenberg Model')

plt.title('Comparison of Traffic Models', fontsize=20)
plt.xlabel('Density', fontsize=18)
plt.ylabel('Average Speed', fontsize=18)
plt.legend()
plt.grid(True)
plt.show()
```

8.2 Code for Two-Lane CA Model

```
#%%Two Lanes Models
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

# Set model parameters
road_length = 400 # Length of the road
max_speed = 5 # vmax
deceleration_prob = 0.3 # Randomization
steps = 400 # Number of simulation steps
car_density = 0.3 # Density of cars
l_back = 5 # l_o_back
P_change = 1 # Probability of changing lanes

# Update function for a single lane
```

```python
def update_road_single_lane(road, max_speed, deceleration_prob):
    new_road = -np.ones_like(road)
    for i, speed in enumerate(road):
        if speed >= 0:
            distance = 1
            while road[(i + distance) % road_length] == -1 and distance <= max_speed:
                distance += 1

            if speed < max_speed and distance > speed + 1:
                speed += 1

            speed = min(speed, distance - 1)

            if speed > 0 and np.random.rand() < deceleration_prob:
                speed -= 1

            new_road[(i + speed) % road_length] = speed
    return new_road

# Initialize road function for two lanes
def initialize_road_two_lanes(density):
    roads = [-np.ones(road_length, dtype=int) for _ in range(2)] # Two lanes
    for road in roads:
        initial_cars = np.random.choice(range(road_length), size=int(density * road_length), replace=False)
        road[initial_cars] = np.random.randint(0, max_speed + 1, size=len(initial_cars))
    return roads

# Calculate gaps
def calculate_gaps(road, position):
    gap = 1
    while road[(position + gap) % road_length] == -1 and gap <= max_speed:
        gap += 1
    return gap

# Calculate backward gaps
def calculate_backward_gaps(road, position):
    gap_back = 1
    while road[(position - gap_back) % road_length] == -1 and gap_back <= l_back:
        gap_back += 1
    return gap_back

# Check and perform lane changes
def check_and_perform_lane_changes(roads, l, l_o, l_back, P_change):
    new_roads = [road.copy() for road in roads] # Copy roads to avoid in-place modification
    for lane in range(2):
        other_lane = 1 - lane
        for i, speed in enumerate(roads[lane]):
            if speed >= 0:
                gap = calculate_gaps(roads[lane], i)
                gap_o = calculate_gaps(roads[other_lane], i)
                gap_o_back = calculate_backward_gaps(roads[other_lane], i) # Use the corrected backward gap calculation
```

```python

                if gap < l and gap_o > l_o and gap_o_back > l_back and np.random.rand() < P_change:
                    # Move car to the other lane
                    new_roads[other_lane][i] = roads[lane][i]
                    new_roads[lane][i] = -1
    return new_roads

# Simulate traffic for two lanes
def simulate_traffic_two_lanes(roads, steps, max_speed, deceleration_prob, l, l_o, l_back, P_change):
    road_states = [[], []]
    for _ in range(steps):
        roads = check_and_perform_lane_changes(roads, max_speed + 1, max_speed + 1, l_back, P_change)
        for lane in range(2):
            road_states[lane].append(roads[lane].copy())
            roads[lane] = update_road_single_lane(roads[lane], max_speed, deceleration_prob)
    return road_states

# Initialize two lanes with specific density
roads = initialize_road_two_lanes(car_density)

#%%Greyscale_velocity

# Simulate traffic
road_states_speed = simulate_traffic_two_lanes(roads, steps, max_speed, deceleration_prob, max_speed + 1, max_speed + 1, l_back, P_change)

# Plotting
fig, axs = plt.subplots(1, 2, figsize=(12, 10), sharex=True)
cmap_greyscale = plt.cm.get_cmap('Greys', max_speed + 1)

for i, ax in enumerate(axs):
    ax.set_xlabel("Position")
    ax.set_ylabel("Time Step")
    img = ax.imshow(road_states_speed[i], cmap=cmap_greyscale, interpolation="nearest", animated=True, vmin=-1, vmax=max_speed)
    ax.set_title(f"Lane {i+1} with Velocity in Greyscale")

plt.tight_layout()
plt.show()


#%%Black_white Visulaization

# Simulate traffic
road_states_speed = simulate_traffic_two_lanes(roads, steps, max_speed, deceleration_prob, max_speed + 1, max_speed + 1, l_back, P_change)

# Plotting with new requirements and adding a big title
fig, axs = plt.subplots(1, 2, figsize=(12, 7)) # Adjusted for side-by-side subplots (1*2 layout)
```

```python
cmap_binary = plt.cm.get_cmap('binary') # Using binary colormap for black and white representation

for i, ax in enumerate(axs):
    ax.set_xlabel("Position", fontsize=18)
    ax.set_ylabel("Time Step", fontsize=18)
    # Convert road states to binary for black and white representation
    binary_road_states = np.array(road_states_speed[i]) >= 0
    img = ax.imshow(binary_road_states, cmap=cmap_binary, interpolation="nearest", animated=True)
    ax.set_title(f"Lane {i+1}", fontsize=18)

plt.tight_layout()
fig.suptitle("Symmetric Traffic Simulation on Two Lanes", fontsize=20) # Adding a big title to the whole figure
plt.show()

#%%Asymmetric Model
# Set model parameters
road_length = 400 # Length of the road
max_speed = 5 # vmax
deceleration_prob = 0.3 # Randomization
steps = 400 # Number of simulation steps
car_density = 0.25 # Density of cars
l_back = 5 # l_o_back
P_change = 1 # Probability of changing lanes

# def check_and_perform_lane_changes_asymmetric(roads, l, l_o, l_back, P_change, asymmetric=False):
#    for lane in range(2):
#        other_lane = 1 - lane
#        for i, speed in enumerate(roads[lane]):
#            if speed >= 0:
#                gap = calculate_gaps(roads[lane], i)
#                gap_o = calculate_gaps(roads[other_lane], i)
#                gap_o_back = calculate_backward_gaps(roads[other_lane], i)


#                if lane == 0 and gap < l and gap_o > l_o and gap_o_back > l_back and np.random.rand() < P_change:
#                    roads[other_lane][i] = speed
#                    roads[lane][i] = -1
#                elif lane == 1 and gap_o_back > l_back:
#                    roads[other_lane][i] = speed
#                    roads[lane][i] = -1
#    return roads


def check_and_perform_lane_changes_asymmetric(roads, l, l_o, l_back, P_change):
    for lane in range(2):
        other_lane = 1 - lane
        for i, speed in enumerate(roads[lane]):
            if speed >= 0:
                gap = 1
```

```
                while roads[lane][(i + gap) % road_length] == -1 and gap <= max_speed:
                    gap += 1

                gap_o = 1
                while roads[other_lane][(i + gap_o) % road_length] == -1 and gap_o <= max_speed:
                    gap_o += 1

                gap_o_back = 1
                while roads[other_lane][(i - gap_o_back) % road_length] == -1 and gap_o_back <= l_back:
                    gap_o_back += 1

                if lane == 0 and gap < l and gap_o > l_o and gap_o_back > l_back and np.random.rand() < P_change:
                    roads[other_lane][i] = speed #2
                    roads[lane][i] = -1

                elif lane == 1 and gap_o > l and gap_o_back > l_back and np.random.rand() < P_change:
                    roads[other_lane][i] = speed # 1
                    roads[lane][i] = -1

    return roads


# Use the asymmetric lane change function in the simulation
def simulate_traffic_two_lanes_asymmetric(roads, steps, max_speed, deceleration_prob, l, l_o, l_back, P_change):
    road_states = [[], []]
    for _ in range(steps):
        roads = check_and_perform_lane_changes_asymmetric(roads, max_speed + 1, max_speed + 1, l_back, P_change)
        for lane in range(2):
            road_states[lane].append(roads[lane].copy())
            roads[lane] = update_road_single_lane(roads[lane], max_speed, deceleration_prob)
    return road_states

# Initialize two lanes with specific density
roads = initialize_road_two_lanes(car_density)

# Simulate traffic using the asymmetric model
road_states_speed_asymmetric = simulate_traffic_two_lanes_asymmetric(roads, steps, max_speed, deceleration_prob, max_speed + 1, max_speed + 1, l_back, P_change)

# Plotting remains the same as before
# Plotting with new requirements and adding a big title for asymmetric model
fig, axs = plt.subplots(1, 2, figsize=(12, 7)) # Adjusted for side-by-side subplots (1*2 layout)
cmap_binary = plt.cm.get_cmap('binary') # Using binary colormap for black and white representation

for i, ax in enumerate(axs):
```

```
    ax.set_xlabel("Position", fontsize=18)
    ax.set_ylabel("Time Step", fontsize=18)
    # Convert road states to binary for black and white representation
    binary_road_states = np.array(road_states_speed_asymmetric[i]) >= 0
    img = ax.imshow(binary_road_states, cmap=cmap_binary, interpolation="nearest", animated=True)
    ax.set_title(f"Lane {i+1}", fontsize=18)

plt.tight_layout()
fig.suptitle("Asymmetric Traffic Simulation on Two Lanes", fontsize=20) # Adding a big title to the whole figure
plt.show()

#%%Density between left and right

steps = 100
road_length = 100
road_states_speed_asymmetric = [
    np.random.randint(0, 2, size=(steps, road_length)),
    np.random.randint(0, 2, size=(steps, road_length))
]

density_lane0 = road_states_speed_asymmetric[0].sum(axis=1) / road_length
density_lane1 = road_states_speed_asymmetric[1].sum(axis=1) / road_length

time_steps = np.arange(steps)
plt.figure(figsize=(12, 6))
plt.plot(time_steps, density_lane0, label='Lane1 Density')
plt.plot(time_steps, density_lane1, label='Lane2 Density')
plt.xlabel('Time Step', fontsize=18)
plt.ylabel('Vehicle Density', fontsize=18)
plt.title('Vehicle Density Comparison Between Two Lanes', fontsize=18)
plt.legend()
plt.grid(True)
plt.show()



#%%TrafficFlow vs. Density between left and right

import numpy as np
import matplotlib.pyplot as plt

# Calculate gaps
def calculate_gaps(road, position):
    gap = 1
    while road[(position + gap) % road_length] == -1 and gap <= max_speed:
        gap += 1
    return gap

# Calculate backward gaps
def calculate_backward_gaps(road, position):
    gap_back = 1
    while road[(position - gap_back) % road_length] == -1 and gap_back <= l_back:
```

```
        gap_back += 1
    return gap_back

def initialize_road_two_lanes(density):
    roads = [-np.ones(road_length, dtype=int) for _ in range(2)] # Two lanes
    for road in roads:
        initial_cars = np.random.choice(range(road_length), size=int(density * road_length), replace=False)
        road[initial_cars] = np.random.randint(0, max_speed + 1, size=len(initial_cars))
    return roads

def update_road_single_lane(road, max_speed, deceleration_prob):
    new_road = -np.ones_like(road)
    for i, speed in enumerate(road):
        if speed >= 0:
            distance = 1
            while road[(i + distance) % road_length] == -1 and distance <= max_speed:
                distance += 1
            if speed < max_speed and distance > speed + 1:
                speed += 1
            speed = min(speed, distance - 1)
            if speed > 0 and np.random.rand() < deceleration_prob:
                speed -= 1
            new_road[(i + speed) % road_length] = speed
    return new_road

def check_and_perform_lane_changes_asymmetric(roads, l, l_o, l_back, P_change):
    for lane in range(2):
        other_lane = 1 - lane
        for i, speed in enumerate(roads[lane]):
            if speed >= 0:
                gap = 1
                while roads[lane][(i + gap) % road_length] == -1 and gap <= max_speed:
                    gap += 1

                gap_o = 1
                while roads[other_lane][(i + gap_o) % road_length] == -1 and gap_o <= max_speed:
                    gap_o += 1

                gap_o_back = 1
                while roads[other_lane][(i - gap_o_back) % road_length] == -1 and gap_o_back <= l_back:
                    gap_o_back += 1

                if lane == 0 and gap < l and gap_o > l_o and gap_o_back > l_back and np.random.rand() < P_change:
                    roads[other_lane][i] = speed
                    roads[lane][i] = -1

                elif lane == 1 and gap_o > l and gap_o_back > l_back and np.random.rand() < P_change:
                    roads[other_lane][i] = speed
                    roads[lane][i] = -1
```

```

    return roads

road_length = 300 # Length of the road
max_speed = 5 # vmax
deceleration_prob = 0.3 # Randomization
steps = 300 # Number of simulation steps
l_back = 5 # l_o_back
P_change = 1 # Probability of changing lanes


def simulate_traffic_flow_density(roads, steps, max_speed, deceleration_prob, l, l_o,
    l_back, P_change):
    road_flows = [0, 0]
    road_speeds = [[], []]

    count_start = 0
    count_end = road_length // 10

    for _ in range(steps):
        roads_before = [road.copy() for road in roads]
        roads = check_and_perform_lane_changes_asymmetric(roads, l, l_o, l_back, P_change)

        for lane in range(2):
            road = update_road_single_lane(roads[lane], max_speed, deceleration_prob)
            roads[lane] = road

            for i, speed in enumerate(roads_before[lane]):
                if speed >= 0:
                    new_position = (i + speed) % road_length
                    if count_start <= new_position <= count_end and not (count_start <= i
                        <= count_end):
                        road_flows[lane] += 1

            car_speeds = road[road >= 0]
            avg_speed = np.mean(car_speeds) if len(car_speeds) > 0 else 0
            road_speeds[lane].append(avg_speed)

    avg_speeds = [np.mean(speed) for speed in road_speeds]
    return road_flows, avg_speeds



densities = np.linspace(0, 0.4, 21)
flows_lane1 = []
flows_lane2 = []
speeds_lane1 = []
speeds_lane2 = []

for density in densities:
    roads = initialize_road_two_lanes(density)
    avg_flows, _ = simulate_traffic_flow_density(roads, steps, max_speed,
        deceleration_prob, max_speed + 1, max_speed + 1, l_back, P_change)
    flows_lane1.append(round(avg_flows[0] / 300, 2))
```

```python
    flows_lane2.append(round(avg_flows[1] / 300, 2))


plt.figure(figsize=(10, 6))

plt.plot(densities, flows_lane1, label='Lane 1 Flow', marker='o')
plt.plot(densities, flows_lane2, label='Lane 2 Flow', marker='x')
plt.xlabel('Density', fontsize=18)
plt.ylabel('Traffic Flow', fontsize=18)
plt.title('Flow vs Density for Asymmetric Model', fontsize=20)
plt.legend()
plt.grid(True)


plt.tight_layout()
plt.show()

#%%Traffic flow vs. density for three models

def update_road(road, max_speed, deceleration_prob):
    new_road = -np.ones_like(road)
    for i, speed in enumerate(road):
        if speed >= 0:
            # Calculate distance to the next car
            distance = 1
            while road[(i + distance) % road_length] == -1 and distance <= max_speed:
                distance += 1

            # Acceleration
            if speed < max_speed and distance > speed + 1:
                speed += 1

            # Slowing down
            speed = min(speed, distance - 1)

            # Randomization
            if speed > 0 and np.random.rand() < deceleration_prob:
                speed -= 1

            # Car motion
            new_road[(i + speed) % road_length] = speed
    return new_road

# Initialize road function
def initialize_road(density):
    road = -np.ones(road_length, dtype=int) # -1 represents no car
    initial_cars = np.random.choice(range(road_length), size=int(density * road_length),
        replace=False)
    road[initial_cars] = np.random.randint(0, max_speed + 1, size=len(initial_cars))
    return road

# Simulate traffic function
def simulate_traffic(road, steps, max_speed, deceleration_prob):
    road_states = []
```

```python
    for _ in range(steps):
        road_states.append(road.copy())
        road = update_road(road, max_speed, deceleration_prob)
    return road_states


def calculate_traffic_flow(road_states):
    flow = []
    for state in road_states:
        cars = np.array(state) >= 0
        if cars.sum() > 0:
            avg_speed = np.mean(np.array(state)[cars])
            flow.append(cars.sum() * avg_speed / road_length)
        else:
            flow.append(0)
    return np.mean(flow)

densities = np.linspace(0, 1, 20)
flows_single_lane = []
flows_symmetric = []
flows_asymmetric = []

for density in densities:
    road = initialize_road(density)
    road_states = simulate_traffic(road, steps, max_speed, deceleration_prob)
    flows_single_lane.append(calculate_traffic_flow(road_states))

    roads = initialize_road_two_lanes(density)
    road_states_symmetric = simulate_traffic_two_lanes(roads, steps, max_speed,
        deceleration_prob, max_speed+1, max_speed+1, l_back, P_change)
    flow_symmetric = sum([calculate_traffic_flow(states) for states in
        road_states_symmetric])
    flows_symmetric.append(flow_symmetric / 2)

    road_states_asymmetric = simulate_traffic_two_lanes_asymmetric(roads, steps,
        max_speed, deceleration_prob, max_speed+1, max_speed+1, l_back, P_change)
    flow_asymmetric = sum([calculate_traffic_flow(states) for states in
        road_states_asymmetric])
    flows_asymmetric.append(flow_asymmetric / 2)

plt.figure(figsize=(10, 6))
plt.plot(densities, flows_single_lane, label="Single Lane", marker="o")
plt.plot(densities, flows_symmetric, label="Symmetric Two Lanes", marker="s")
plt.plot(densities, flows_asymmetric, label="Asymmetric Two Lanes", marker="^")
plt.xlabel("Car Density")
plt.ylabel("Traffic Flow")
plt.title("Traffic Flow vs. Car Density")
plt.legend()
plt.grid(True)
plt.show()



#%%Different Types of cars
```

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors as mcolors

# Set model parameters
road_length = 400 # Length of the road
max_speed_car = 5 # Max speed for cars
max_speed_truck = 3 # Max speed for trucks
deceleration_prob_car = 0.3 # Deceleration probability for cars
deceleration_prob_truck = 0.5 # Deceleration probability for trucks
steps = 400 # Number of simulation steps
car_density = 0.1 # Density of cars
l_back = 5 # Checking distance backwards
P_change = 0.5 # Probability of changing lanes

# Define vehicle types
vehicle_types = {
    'car': {'max_speed': max_speed_car, 'deceleration_prob': deceleration_prob_car,
        'color': 2}, # Red for cars
    'truck': {'max_speed': max_speed_truck, 'deceleration_prob':
        deceleration_prob_truck, 'color': 1} # Green for trucks
}

# Define colors
color_map = mcolors.ListedColormap(['white', 'green', 'red']) # White for empty, green
    for trucks, red for cars

# Initialize road for two lanes with heterogeneous traffic
def initialize_road_two_lanes(density, vehicle_distribution):
    roads = [-np.ones(road_length, dtype=int) for _ in range(2)] # Two lanes
    for road in roads:
        num_vehicles = int(density * road_length)
        vehicle_choices = np.random.choice(['car', 'truck'], size=num_vehicles,
            p=[vehicle_distribution['car'], vehicle_distribution['truck']])
        positions = np.random.choice(range(road_length), size=num_vehicles, replace=False)
        for pos, v_type in zip(positions, vehicle_choices):
            road[pos] = vehicle_types[v_type]['color'] # Assign a color code to represent
                different vehicle types
    return roads

# Update function for a single lane considering heterogeneous traffic
def update_road(road, vehicle_types):
    new_road = -np.ones_like(road)
    for i, vehicle_code in enumerate(road):
        if vehicle_code >= 0: # Check if there is a vehicle
            for v_type, v_info in vehicle_types.items():
                if v_info['color'] == vehicle_code:
                    max_speed = v_info['max_speed']
                    deceleration_prob = v_info['deceleration_prob']
                    break

            distance = 1
            while road[(i + distance) % road_length] == -1 and distance <= max_speed:
                distance += 1
```

```

            speed = min(max_speed, distance - 1)
            if np.random.rand() < deceleration_prob:
                speed = max(0, speed - 1) # Decelerate with certain probability

            new_road[(i + speed) % road_length] = vehicle_code
    return new_road

# Other functions like calculate_gaps, calculate_backward_gaps,
    check_and_perform_lane_changes remain unchanged

# Simulate traffic for two lanes with symmetric rules
def simulate_traffic_two_lanes(roads, steps, vehicle_types, l_back, P_change):
    road_states = [[], []]
    for _ in range(steps):
        roads = check_and_perform_lane_changes(roads, max_speed_car + 1, max_speed_car +
            1, l_back, P_change)
        for lane in range(2):
            road_states[lane].append(np.copy(roads[lane]))
            roads[lane] = update_road(roads[lane], vehicle_types)
    return road_states

# Main simulation setup
vehicle_distribution = {'car': 0.7, 'truck': 0.3} # 70% cars, 30% trucks
roads = initialize_road_two_lanes(car_density, vehicle_distribution)
road_states = simulate_traffic_two_lanes(roads, steps, vehicle_types, l_back, P_change)

# Visualization
fig, axs = plt.subplots(1, 2, figsize=(10, 5))
for i, ax in enumerate(axs):
    car_presence = np.array(road_states[i]) # Convert to numpy array for easier
        manipulation
    ax.imshow(car_presence, cmap=color_map, aspect='auto')
    ax.set_title(f'Lane {i + 1}')
    ax.set_xlabel('Position')
    ax.set_ylabel('Time step')
plt.tight_layout()
plt.show()
```

8.3 Code for Classical Model

```
#General PDE Model

import numpy as np
import matplotlib.pyplot as plt

# Parameters
L = 10.0 # Length of the road
T = 5.0  # Simulation time
Nx = 100  # Number of spatial grid points
Nt = 1000  # Number of time steps
V_max = 1.0 # Maximum free-flow speed
k = 0.1     # Density parameter
```

```python

# Discretization
dx = L / Nx
dt = T / Nt
x = np.linspace(0, L, Nx)
t = np.linspace(0, T, Nt)

# Initial condition
rho0 = np.ones(Nx) * 0.2
rho0[int(Nx / 4):int(Nx / 2)] = 1.0

# Numerical solution using finite difference
rho = np.zeros((Nt, Nx))
rho[0, :] = rho0

for n in range(1, Nt):
    for i in range(1, Nx):
        rho[n, i] = rho[n-1, i] - dt/dx * (rho[n-1, i] * (V_max - k * rho[n-1, i]) - rho[n-1, i-1] * (V_max - k * rho[n-1, i-1]))

# Plotting the results
plt.imshow(rho, extent=[0, L, 0, T], origin='lower', aspect='auto', cmap='gray_r')
plt.colorbar(label='Traffic Density (vehicles per km)')
plt.xlabel('Position on road (km)')
plt.ylabel('Time (s)')
plt.title('Traffic Density Evolution 1: General PDE Model')
plt.gca().invert_yaxis()
plt.show()

##############################################

#Flow versus density

import numpy as np
import matplotlib.pyplot as plt

# Parameters
rho_max = 180.0 / 1000 # Maximum density (180 vehicles per km)
v_max = 1.0 # Maximum velocity (units not specified, assumed m/s here)

# Conversion factor: seconds per hour
seconds_per_hour = 3600

# Traffic density values
rho_values = np.linspace(0, rho_max, 100)

# Calculate corresponding traffic flow values (example function)
Q_values = rho_values * v_max * (1 - rho_values / rho_max)

# Convert traffic flow to vehicles per hour
Q_values_per_hour = Q_values * seconds_per_hour

# Plot the traffic flow vs. density
plt.plot(rho_values * 1000, Q_values_per_hour)
```

```
plt.xlabel('Traffic Density (vehicles/km)')
plt.ylabel('Traffic Flow (vehicles/hour)')
plt.title('Traffic Flow vs. Density (Classical PDE Model)')
plt.legend()
plt.grid(True)
plt.show()
plt.show()

###############################################

#Cellula automatum into PDE Model

import numpy as np
import matplotlib.pyplot as plt

# Parameters
L = 10.0 # Length of the road
num_points = 100 # Number of spatial points
num_steps = 200 # Number of time steps
dt = 0.01 # Time step
dx = L / num_points # Spatial step
rho_max = 10.0 # Maximum density

# Smooth initial condition
x_values = np.linspace(0, L, num_points)
rho_initial = rho_max / 2.0 * (1 + np.sin(2 * np.pi * x_values / L))

# Initialize density array
rho = np.zeros((num_steps, num_points))

# Set initial condition
rho[0, :] = rho_initial

# Implement PDE simulation with the second speed-density relationship
for n in range(0, num_steps - 1):
    for i in range(0, num_points):
        rho_n = rho[n, i]
        rho_left = rho[n, (i - 1 + num_points) % num_points] # Apply periodic boundary
            conditions

        # Calculate speed using the second relationship
        v = 5.17 * np.exp(-rho_n / 0.30)

        # Update density using LWR PDE with the modified speed-density relationship
        rho[n + 1, i] = rho_n - dt / dx * (rho_n * v - rho_left * v)

# Plot the results
x_values = np.linspace(0, L, num_points)
t_values = np.arange(0, num_steps) * dt

plt.figure(figsize=(10, 6))
plt.pcolormesh(x_values, t_values, rho, shading='auto', cmap='gray_r')
plt.colorbar(label='Traffic Density (vehicles per km)')
plt.xlabel('Position on road (km)')
```

```python
plt.ylabel('Time (s)')
plt.title('Traffic Density Evolution 2: CA Model Applied (Smooth Initial Condition)')
plt.gca().invert_yaxis()
plt.show()

##################


#Car-following into PDE Model

import numpy as np
import matplotlib.pyplot as plt

# Parameters
L = 10.0 # Length of the road
num_points = 100 # Number of spatial points
num_steps = 1000 # Number of time steps (increase this for a larger time scale)
dt = 0.1 # Time step (adjust accordingly for the desired time scale)
dx = L / num_points # Spatial step
rho_max = 10.0 # Maximum density

# Smooth initial condition (Gaussian distribution)
x_values = np.linspace(0, L, num_points)
rho_initial = rho_max * np.exp(-0.5 * ((x_values - L / 2) / 2.0)**2) # Gaussian
    distribution

# Initialize density array
rho = np.zeros((num_steps, num_points))

# Set initial condition
rho[0, :] = rho_initial

# Implement PDE simulation with a different speed-density relationship
for n in range(0, num_steps - 1):
    for i in range(0, num_points):
        rho_n = rho[n, i]
        rho_left = rho[n, (i - 1 + num_points) % num_points] # Apply periodic boundary
            conditions

        # Different speed-density relationship
        v = 1.0 - rho_n / rho_max

        # Update density using LWR PDE with the modified speed-density relationship
        rho[n + 1, i] = rho_n - dt / dx * (rho_n * v - rho_left * v)

# Plot the results
x_values = np.linspace(0, L, num_points)
t_values = np.arange(0, num_steps) * dt

plt.figure(figsize=(10, 6))
plt.imshow(rho, extent=[0, L, 0, num_steps * dt], aspect='auto', cmap='gray_r')
plt.colorbar(label='Traffic Density (vehicles per km)')
plt.xlabel('Position on road (km)')
plt.ylabel('Time (s)')
```

```
plt.title('Traffic Density Evolution 3: CF Model Applied (Gaussian Initial condition)')
plt.gca().invert_yaxis()
plt.show()
```

8.4 Code for Simple Car-Following Model

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# from 3.1_gipps.py

### Closed system

# driver model


#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


#%%%%%%%%%%%%%%%%%%%%%%%%################################
import numpy as np
import matplotlib.pyplot as plt

#Function of simulation with Acceleration and Deceleration in Matrix

# Function to update bn based on the given condition
def b_hat_DA(b_n):
    m = b_n.copy()
    m[::]= -3
    max_array = [min(a, b) for a, b in zip(m, (b_n - 3)/2)]
    return max_array

def calculate_new_speed_DA(v_n, v_n_lead, s_n, a_n, b_n, tau, x_n, x_n_lead, b_h, Vn, theta):
    vp1 = v_n + 2.5*a_n*tau*(1- v_n/Vn)*((0.025+v_n/Vn)**0.5)
    vp2 = b_n*(tau/2 + theta) + np.sqrt((b_n*(tau/2+theta))**2 - b_n*((2*(x_n_lead + s_n - x_n)% road_length) - v_n*tau - (v_n_lead**2)/b_h))
    return min(vp1, vp2)


# Simulation loop
def simulation_loop_DA( positions, speeds, target_speeds, time_steps, N, road_length, a_n, b_n, s_n, tau, theta,leadcar_speed_f):
    for t in range(1, time_steps):
        for i in range(N):
            if i == N-1:
                speeds[t, i] = leadcar_speed_f[t-1]
                positions[t, i] = (positions[t-1, i] + speeds[t, i] * tau-0.5*np.cos(0.5*tau)) % road_length
            else:
            # The lead vehicle is the one in front of the current vehicle
                lead_vehicle_index = (i + 1) % N
                v_n = speeds[t-1, i]
```

```
                v_n_lead = speeds[t-1, lead_vehicle_index]
                x_n = positions[t-1, i]
                x_n_lead = positions[t-1, lead_vehicle_index]
                b_h = b_hat_DA(b_n)
                Vn = target_speeds[i]
                speeds[t, i] = calculate_new_speed_DA(v_n, v_n_lead, s_n[i], a_n[i], b_n[i], tau, x_n, x_n_lead, b_h[i], Vn, theta)
                positions[t, i] = (positions[t-1, i] + (speeds[t, i]+v_n) * tau/2) % road_length
    return positions, speeds

# Function of simulation with an and bn fixed number.

# Function to update bn based on the given condition
def b_hat(b_n):
    return min(-3, (b_n - 3) / 2)

def calculate_new_speed(v_n, v_n_lead, s_n, a_n, b_n, tau, x_n, x_n_lead, b_h, Vn):
    vp1 = v_n + 2.5*a_n*tau*(1- v_n/Vn)*(0.025+v_n/Vn)**0.5
    vp2 = b_n*tau + np.sqrt((b_n*tau)**2 - b_n*((2*(x_n_lead - s_n - x_n)% road_length) - v_n*tau-v_n_lead**2/b_h))
    return min(vp1, vp2)


# Simulation loop
def simulation_loop( positions, speeds, target_speeds, time_steps, N, road_length, a_n, b_n, s_n, tau):
    for t in range(1, time_steps):
        for i in range(N):
            # The lead vehicle is the one in front of the current vehicle
            lead_vehicle_index = (i + 1) % N
            v_n = speeds[t-1, i]
            v_n_lead = speeds[t-1, lead_vehicle_index]
            x_n = positions[t-1, i]
            x_n_lead = positions[t-1, lead_vehicle_index]
            b_h = b_hat(b_n)
            Vn = target_speeds[i]
            speeds[t, i] = calculate_new_speed(v_n, v_n_lead, s_n, a_n, b_n, tau, x_n, x_n_lead, b_h, Vn)
            positions[t, i] = (positions[t-1, i] + speeds[t, i] * tau) % road_length
    return positions, speeds



def density_flow_DA(road_length, initial_speed, s_n, tau, total_time, time_steps, Nmax, vehicle_counts, a_n, b_n, target_speeds):
    mean_speeds = []
    flows = []
    for N in vehicle_counts:
        time_m = np.arange(0, total_time, tau)

        # Initialize arrays to store positions and speeds of vehicles
        positions = np.zeros((time_steps, N))
        speeds = np.zeros((time_steps, N))
```

```
        target_speeds = np.random.normal(20, 3.2, N)

        # Set initial conditions
        positions[0, :] = initial_positions
        speeds[0, :] = initial_speed

        leadcar_speed_f = np.sin(0.5*time_m[1:]) + speeds[0,-1]
        speeds[1:,-1] = leadcar_speed_f
        leadcar_position_f = speeds[0,-1]*time_m[1:] + positions[0,-1] -2*np.cos(0.5*time_m[1:])
        positions[1:,-1] = leadcar_position_f% road_length


        # Simulation loop for this scenario
        positions, speeds = simulation_loop_DA(positions, speeds, target_speeds, time_steps, N, road_length, a_n[:N], b_n[:N], s_n, tau, theta,leadcar_speed_f)
        # Calculate average speed and flow for this scenario
        average_speed_km_h = np.mean(speeds) * 3.6 # Convert average speed to km/h
        mean_speeds.append(average_speed_km_h)
        density = N / road_length_km # Density = Number of vehicles / Road length
        flow = average_speed_km_h * density # Flow = Density * Average Speed
        flows.append(flow)
    return flows, mean_speeds



def speed_flow(Vmax, speed_step, initial_speed, N, road_length, a_n, b_n, s_n, tau,leadcar_speed_f ):
    flows = []
    for i in np.arange(speed_step):
        target_speeds_now = Vmax[i] * np.ones(N)

        time_m = np.arange(0, total_time, tau)

        positions = np.zeros((time_steps, N))
        speeds = np.zeros((time_steps, N))
        # Set initial conditions
        positions[0, :] = np.linspace(0, road_length, N, endpoint=False)
        speeds[0, :] = initial_speed

        leadcar_speed_f = np.sin(0.5*time_m[1:]) + speeds[0,-1]
        speeds[1:,-1] = leadcar_speed_f
        leadcar_position_f = speeds[0,-1]*time_m[1:] + positions[0,-1] -2*np.cos(0.5*time_m[1:])
        positions[1:,-1] = leadcar_position_f% road_length



        # Simulation loop for this scenario
        positions, speeds = simulation_loop_DA(positions, speeds, target_speeds_now, time_steps, N, road_length, a_n, b_n, s_n, tau, theta,leadcar_speed_f)
        # Calculate average speed and flow for this scenario
        average_speed_km_h = np.mean(speeds) * 3.6 # Convert average speed to km/h

```

```
        density = N / road_length_km # Density = Number of vehicles / Road length
        flow = average_speed_km_h * density # Flow = Density * Average Speed
        flows.append(flow)
    return flows


#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

#fontsize

fontsize_title = 20
fontsize_label = 18


# Given parameters
N = 10 # Number of vehicles
road_length = 1000.0 # Length of circular road in meters
initial_speed = 20 # Initial speed of all vehicles in m/s\

a_n = np.random.normal(1.7, 0.3, N)
b_n = -2.0 * a_n # Most severe braking in m/s^2
s_n = np.ones(N)*6 # Effective size of vehicle in meters
tau = 2/3 # Reaction time in seconds
theta = tau/2 #safety time parameter

# Initial positions of the vehicles equally spaced on the road
initial_positions = np.linspace(0, road_length, N, endpoint=False)

# Time setup
total_time = 80 # Total time of simulation in seconds
time_steps = int(total_time / tau)
time_m = np.arange(0, total_time, tau)

# Initialize arrays to store positions and speeds of vehicles
positions = np.zeros((time_steps, N))
speeds = np.zeros((time_steps, N))
target_speeds = np.random.normal(20, 3.2, N)

# Set initial conditions
positions[0, :] = initial_positions
speeds[0, :] = initial_speed

leadcar_speed_f = np.sin(0.5*time_m[1:]) + speeds[0,-1]
speeds[1:,-1] = leadcar_speed_f
leadcar_position_f = speeds[0,-1]*time_m[1:] + positions[0,-1] -2*np.cos(0.5*time_m[1:])
positions[1:,-1] = leadcar_position_f% road_length

#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
# 1
# Plotting the displacements of vehicles against time

positions, speeds = simulation_loop_DA(positions, speeds, target_speeds, time_steps, N,
    road_length, a_n, b_n, s_n, tau, theta,leadcar_speed_f)

```

```python
plt.figure(dpi=150)

for i in range(N):
    plt.scatter(np.arange(0, total_time, tau), positions[:, i],s=(2.0)) #, label=f'Vehicle {i+1}')


plt.xlabel('Time (s)', fontsize=fontsize_label)
plt.ylabel('Displacement (m)', fontsize=fontsize_label)
plt.title('Displacements of Vehicles on a closed Circular Road', fontsize=fontsize_title)
plt.yticks(fontsize=fontsize_label)
plt.xticks(fontsize=fontsize_label)
plt.show()

#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
# plot velocity of each vehicle

plt.figure(dpi=150)
for i in range(N):
    plt.plot(np.arange(0, total_time, tau), speeds[:, i])

plt.xlabel('Time (s)', fontsize=fontsize_label)
plt.ylabel('Speed (m/s)', fontsize=fontsize_label)
plt.title('Speed of Vehicles on a closed Circular Road', fontsize=fontsize_title)
plt.yticks(fontsize=fontsize_label)
plt.xticks(fontsize=fontsize_label)
plt.legend()



#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

#plot flow and density

road_length = 1000.0 # Length of circular road in meters
initial_speed = 20 # Initial speed of all vehicles in m/s\
tau = 2/3 # Reaction time in seconds
theta = tau/2 #safety time parameter


# Time setup
total_time = 100 # Total time of simulation in seconds
time_steps = int(total_time / tau)

road_length_km = road_length / 1000 # Convert road length to kilometers for density calculation

initial_positions = np.linspace(0, road_length, N, endpoint=False)

# Time setup
total_time = 80 # Total time of simulation in seconds
time_steps = int(total_time / tau)
time_m = np.arange(0, total_time, tau)

```

```python
# Initialize arrays to store positions and speeds of vehicles
positions = np.zeros((time_steps, N))
speeds = np.zeros((time_steps, N))
target_speeds = np.random.normal(20, 3.2, N)

# Set initial conditions
positions[0, :] = initial_positions
speeds[0, :] = initial_speed

leadcar_speed_f = np.sin(0.5*time_m[1:]) + speeds[0,-1]
speeds[1:,-1] = leadcar_speed_f
leadcar_position_f = speeds[0,-1]*time_m[1:] + positions[0,-1] -2*np.cos(0.5*time_m[1:])
positions[1:,-1] = leadcar_position_f% road_length

#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
# 2
# plot density against flow, single lane
initial_speed = 20
Nmax = 163
vehicle_counts = np.arange(1, Nmax, 2) # Different vehicle counts to simulate different
    densities
densities = vehicle_counts / road_length_km # Calculate densities for each vehicle count

a_n = np.random.normal(1.7, 0.3, Nmax)
b_n = -2.0 * a_n
target_speeds = np.random.normal(20, 10, Nmax)

s_n = np.random.normal(6.5,0.3,Nmax)

a_n4 = np.random.normal(1.7, 0.3, Nmax)
b_n4 = -2.0 * a_n4

flows4, mean_speeds4 = density_flow_DA(road_length, initial_speed, s_n, tau, total_time,
    time_steps, Nmax, vehicle_counts, a_n4, b_n4, target_speeds)

plt.figure(figsize=(10, 6))
plt.plot(densities, flows4, marker='o', linestyle='-')
plt.xlabel('Density (vehicles per km)', fontsize=fontsize_label)
plt.ylabel('Flow (vehicles per hour)', fontsize=fontsize_label)
plt.title('Traffic Flow vs. Density', fontsize=fontsize_title)
plt.grid(True)
#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

# 3 plot density against average speed

plt.figure(figsize=(10, 6))
plt.scatter(densities, mean_speeds4, marker='o', linestyle='-', linewidth = 1)
plt.xlabel('Density (vehicles per km)', fontsize=fontsize_label)
plt.ylabel('Average Speed (km/h)', fontsize=fontsize_label)
plt.title('Average Speed vs. Density', fontsize=fontsize_title)
plt.grid(True)


#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```python
#
# plot density against flow, different Sn
Nmax = 160
vehicle_counts = np.arange(1, Nmax, 1) # Different vehicle counts to simulate different
    densities
densities = vehicle_counts / road_length_km # Calculate densities for each vehicle count

a_n = np.random.normal(1.7, 0.3, Nmax)
b_n = -2.0 * a_n
target_speeds = np.random.normal(20, 3.1, Nmax)

s_n1 = np.random.normal(6.,0.3,Nmax)
s_n2 = np.random.normal(5.,0.3,Nmax)

flows1, mean_speeds1 = density_flow_DA(road_length, initial_speed, s_n1, tau,
    total_time, time_steps, Nmax, vehicle_counts, a_n, b_n, target_speeds)
#flows2, mean_speeds2 = density_flow_DA(road_length, initial_speed, s_n2, tau,
    total_time, time_steps, Nmax, vehicle_counts, a_n, b_n, target_speeds)

plt.figure(figsize=(10, 6))
plt.plot(densities, flows1, marker='o', linestyle='-')
#plt.plot(densities, flows2, marker='o', linestyle='-')
plt.xlabel('Density (vehicles per km)', fontsize=fontsize_label)
plt.ylabel('Flow (vehicles per hour)', fontsize=fontsize_label)
plt.title('Traffic Flow vs. Density', fontsize=fontsize_title)



#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#plot different driver types flow and density

Nmax = 160
#s_n = np.random.normal(6.,0.3,Nmax)
s_n = np.ones(Nmax)*6
road_length_km = road_length / 1000 # Convert road length to kilometers for density
    calculation

vehicle_counts = np.arange(1, Nmax, 1) # Different vehicle counts to simulate different
    densities

initial_positions = np.linspace(0, road_length, Nmax, endpoint=False)


# Store results
densities = vehicle_counts / road_length_km # Calculate densities for each vehicle count
target_speeds = np.random.normal(20, 3.1, Nmax)



#cautious
a_n1 = np.random.normal(1.5, 0.1, Nmax)
b_n1 = -2.0 * a_n1


```

```python
#mix
a_n2 = np.random.normal(1.7, 0.3, Nmax)
b_n2 = -2.0 * a_n2



#aggressive
a_n3 = np.random.normal(1.9, 0.1, Nmax)
b_n3 = -2.0 * a_n3


flows1, mean_speeds1 = density_flow_DA(road_length, initial_speed, s_n, tau, total_time,
    time_steps, Nmax, vehicle_counts, a_n1, b_n1, target_speeds)
flows2, mean_speeds2 = density_flow_DA(road_length, initial_speed, s_n, tau, total_time,
    time_steps, Nmax, vehicle_counts, a_n2, b_n2, target_speeds)
flows3, mean_speeds3 = density_flow_DA(road_length, initial_speed, s_n, tau, total_time,
    time_steps, Nmax, vehicle_counts, a_n3, b_n3, target_speeds)

plt.figure(figsize=(10, 6))
plt.plot(densities, flows1, marker='o', linestyle='-')
plt.plot(densities, flows2, marker='o', linestyle='-')
plt.plot(densities, flows3, marker='o', linestyle='-')

plt.xlabel('Density (vehicles per km)', fontsize=fontsize_label)
plt.ylabel('Flow (vehicles per hour)', fontsize=fontsize_label)
plt.title('Traffic Flow vs. Density', fontsize=fontsize_title)
plt.legend(['Cautious', 'Mix', 'Aggressive'], fontsize=fontsize_label)
plt.grid(True)

#%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#plot mean speed and density
plt.figure(figsize=(10, 6))
plt.scatter(densities, mean_speeds1, marker='o', linestyle='-', linewidth = 1)
plt.scatter(densities, mean_speeds2, marker='v', linestyle='-', linewidth = 1)
plt.scatter(densities, mean_speeds3, marker='s', linestyle='-', linewidth = 0.5)
plt.xlabel('Density (vehicles per km)', fontsize=fontsize_label)
plt.ylabel('Average Speed (km/h)', fontsize=fontsize_label)
plt.title('Average Speed vs. Density', fontsize=fontsize_title)
plt.legend(['Cautious', 'Mix', 'Aggressive'], fontsize=fontsize_label)
plt.grid(True)





#%%%%%%

# Plot density against flow
plt.figure(figsize=(10, 6))
plt.plot(densities, flows, marker='o', linestyle='-', color='blue')
plt.xlabel('Density (vehicles per km)', fontsize=fontsize_label)
plt.ylabel('Flow (vehicles per hour)', fontsize=fontsize_label)
plt.title('Traffic Flow vs. Density', fontsize=fontsize_title)
plt.yticks(fontsize=fontsize_label)
```

```python
plt.xticks(fontsize=fontsize_label)
plt.grid(True)
plt.show()

#%%
def f1(x):
    return -0.39*x+65

def f2(x):
    return 90*np.exp(-0.01*x)-10

def f3(x):
    return 30*np.log(1.03/x)+ 160



#



plt.figure(figsize=(10, 6))
plt.scatter(densities, mean_speeds, marker='o', linestyle='-', color='red')
plt.plot(np.arange(0, 200, 5), f1(np.arange(0, 200, 5)))
plt.plot(np.arange(0, 200, 5), f2(np.arange(0, 200, 5)))
plt.plot(np.arange(0, 200, 5), f3(np.arange(0, 200, 5)))
#plt.plot(np.arange(0, 200, 5), f4(np.arange(0, 200, 5)))
plt.legend(['simulation data','Model 1', 'Model 2', 'Model 3'], fontsize=fontsize_label)
plt.xlabel('Density (vehicles per km)', fontsize=fontsize_label)
plt.ylabel('Average Speed (km/h)', fontsize=fontsize_label)
plt.title('Average Speed vs. Density', fontsize=fontsize_title)
plt.yticks(fontsize=fontsize_label)
plt.xticks(fontsize=fontsize_label)
plt.grid(True)
plt.show()

# %%
#plot the speed of the 3rd vehicles in N = 60


N = 60 # Number of vehicles
road_length = 1000.0 # Length of circular road in meters
initial_speed = 20.0 # Initial speed of all vehicles in m/s\

a_n = np.random.normal(1.7, 0.3, N)
b_n = -2.0 * a_n # Most severe braking in m/s^2
s_n = 5.0 # Effective size of vehicle in meters
tau = 2/3 # Reaction time in seconds

# Initial positions of the vehicles equally spaced on the road
initial_positions = np.linspace(0, road_length, N, endpoint=False)

# Time setup
total_time = 100 # Total time of simulation in seconds
time_steps = int(total_time / tau)
```

```
# Initialize arrays to store positions and speeds of vehicles
positions = np.zeros((time_steps, N))
speeds = np.zeros((time_steps, N))
target_speeds = np.random.normal(20, 3.2, N)

# Set initial conditions
positions[0, :] = initial_positions
speeds[0, :] = initial_speed

positions, speeds = simulation_loop_DA_b(positions, speeds, target_speeds, time_steps,
    N, road_length, a_n, b_n, s_n, tau, theta,leadcar_speed_f)
plt.figure(dpi=150)
plt.plot(np.arange(0, total_time, tau), speeds[:, 2], label='Vehicle 3')
plt.xlabel('Time (s)', fontsize=fontsize_label)
plt.ylabel('Speed (m/s)', fontsize=fontsize_label)
plt.title('Speed of Vehicle 3', fontsize=fontsize_title)
plt.yticks(fontsize=fontsize_label)
plt.xticks(fontsize=fontsize_label)
plt.legend()


# %%
#plot displacement of the 3rd vehicles in N = 60
plt.figure(dpi=150)
for i in range(N):
    plt.scatter(np.arange(0, total_time, tau), positions[:, i],s=(2.0)) #,
        label=f'Vehicle {i+1}')

plt.xlabel('Time (s)', fontsize=fontsize_label)
plt.ylabel('Displacement (m)', fontsize=fontsize_label)
plt.title('Displacement of Vehicle 3', fontsize=fontsize_title)
plt.yticks(fontsize=fontsize_label)
plt.xticks(fontsize=fontsize_label)
plt.legend()



#%%%%%%%%%%%%%%%%%%%%%%%

#mix
Nmax = 60
a_n4 = np.random.normal(1.7, 0.4, Nmax)
b_n4 = -2.0 * a_n4
initial_speed = np.random.normal(20,5,Nmax)
vehicle_counts = np.arange(1, Nmax, 1)
target_speeds = np.random.normal(20,3,Nmax)
flows4, mean_speeds4 = density_flow_DA_b(road_length, initial_speed, s_n, tau,
    total_time, time_steps, Nmax, vehicle_counts, a_n4, b_n4, target_speeds)

plt.figure(figsize=(10, 6))
plt.plot(densities, flows4, marker='o', linestyle='-')
plt.xlabel('Density (vehicles per km)', fontsize=fontsize_label)
plt.ylabel('Flow (vehicles per hour)', fontsize=fontsize_label)
```

```python
plt.title('Traffic Flow vs. Density', fontsize=fontsize_title)
plt.grid(True)


#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Vmax = np.arange(1, 21, 1)
speed_step = len(Vmax)
initial_speed = 20.0 # Initial speed of all vehicles in m/s\
N = 20
road_length = 1000.0 # Length of circular road in meters
road_length_km = road_length / 1000 # Convert road length to kilometers for density
    calculation

a_n = np.random.normal(1.7, 0.3, N)
b_n = -2.0 * a_n # Most severe braking in m/s^2
s_n = np.ones(N)*6 # Effective size of vehicle in meters
tau = 2/3 # Reaction time in seconds
flows = speed_flow(Vmax, speed_step, initial_speed, N, road_length, a_n, b_n, s_n,
    tau,leadcar_speed_f)

plt.figure(figsize=(10, 6))
plt.plot(Vmax, flows, marker='o', linestyle='-')
plt.xlabel('Maximum Speed (km/h)', fontsize=fontsize_label)
plt.ylabel('Flow (vehicles per hour)', fontsize=fontsize_label)
plt.title('Traffic Flow vs. Maximum Speed', fontsize=fontsize_title)
plt.grid(True)

# %%
```

8.5 Code for Complex Car-Following Model

```python
import numpy as np
import matplotlib.pyplot as plt

a = 2 # acceleration
b = 3 # deceleration
L = 5 # distance in metres
sn_minus_1 = 4 # length of leading car
v = 20 # initial velocity
tau = 2/3 # time delay
theta = tau/2 # safety time parameter
b_hat = 3 # assumed braking rate of car in front

# initial conditions
xn_minus_1_t = L
vn_minus_1_t = v
xn_t = 0
vn_t = v

total_time = 50
dt = 0.1
timesteps = np.arange(0, total_time, dt)
```

```python

positions = [xn_t]
velocities = [vn_t]

for t in timesteps[1:]:
    vn_t_plus_tau = max(0, -b*(tau/2+theta) + np.sqrt(b**2 * ((tau/2+theta)**2) - 2*b*
        (-xn_minus_1_t+sn_minus_1+xn_t + tau*vn_t/2 - (vn_minus_1_t**2)/(2*b_hat) )))

    # update position and velocity for the following car
    vn_t = vn_t_plus_tau
    xn_t = xn_t + vn_t * dt

    positions.append(xn_t)
    velocities.append(vn_t)

# constants for the simulation
number_of_cars = 6
road_length = 150 # road length in metres

# initial conditions
initial_positions = np.linspace(road_length - L, 0, num=number_of_cars)
initial_velocities = np.full(number_of_cars, v)

car_positions = np.zeros((len(timesteps), number_of_cars))
car_velocities = np.zeros((len(timesteps), number_of_cars))

car_positions[0,:] = initial_positions
car_velocities[0,:] = initial_velocities

braking_position = 200
acceleration_after_braking = a
leader_car_stopped = False
leader_car_accelerating = False
leader_stopped_time = 0

# simulation for each car
for t_index, t in enumerate(timesteps[1:], start=1):
    for car_index in range(number_of_cars):
        if car_index == 0:
            if not leader_car_stopped:
                if car_positions[t_index-1, car_index] >= braking_position:
                    car_velocities[t_index, car_index] = max(0, car_velocities[t_index-1,
                        car_index] - a * dt) # Linear deceleration
                    if car_velocities[t_index, car_index] == 0:
                        leader_car_stopped = True
                        leader_stopped_time = t
                else:
                    car_velocities[t_index, car_index] = v
            else:
                if not leader_car_accelerating:
                    if t >= leader_stopped_time + 3:
                        leader_car_accelerating = True
                    else:
                        car_velocities[t_index, car_index] = 0
```

```python
                if leader_car_accelerating:
                    car_velocities[t_index, car_index] = min(v, car_velocities[t_index-1, car_index] + acceleration_after_braking * dt)
            car_positions[t_index, car_index] = car_positions[t_index-1, car_index] + car_velocities[t_index, car_index] * dt
        else:
            xn_minus_1_t = car_positions[t_index-1, car_index-1]
            vn_minus_1_t = car_velocities[t_index-1, car_index]
            xn_t = car_positions[t_index-1, car_index]
            vn_t = car_velocities[t_index-1, car_index]

            vn_t_plus_tau = max(0, -b*(tau/2+theta) + np.sqrt(b**2 * ((tau/2+theta)**2) - 2*b* (-xn_minus_1_t+sn_minus_1+xn_t + tau*vn_t/2 - (vn_minus_1_t**2)/(2*b_hat) )))
            car_velocities[t_index, car_index] = vn_t_plus_tau
            car_positions[t_index, car_index] = xn_t + vn_t * dt

# plot velocities of each car over time
plt.figure(figsize=(15, 8))

for car_index in range(number_of_cars):
    plt.plot(timesteps, car_velocities[:, car_index], label=f"Car {car_index+1}")

plt.title("Velocities of 6 Cars on an Endless Road", fontsize=25)
plt.xlabel("Time (s)", fontsize=20)
plt.ylabel("Velocity (m/s)", fontsize=20)
plt.legend()
plt.show()
```