Traffic Simulation

Anirudh Nair CS166

Fall 2019

Source to ipython notebook:https://colab.research.google.com/drive/1GE6sbx_AjiQDkt_xrEPf2wxEgF5PTJ_e

Introduction

Road traffic is a complex system which is integral to the life of many human beings. Having an efficient traffic system can help societies be more productive. Through this paper, I aim to model traffic on a single straight road and design several constraints around them. By simulating these constraints, I would try to bring the model as close to reality as possible and see what combinations of these constraints bring out the highest efficiency into the system.

Part 1: Single Lane Traffic Simulation

The first part of the paper aims to model and simulate traffic in a single lane. The model takes inspiration from the Nagel-Schreckenberg Single Lane model and applies the same rules as applied in their paper (Nagel & Schreckenberg, 1992).

We implement a one-dimensional array to mirror the actual road with each cell of the array being the space occupied by a single vehicle. The array has a pre-specified length and has a periodic boundary condition. A vehicle is represented by an integer which holds the value of the velocity of the vehicle (between 0 and a maximum specified velocity value) in the array. The model is updated for every-time step and during this update, certain rules are applied on each of the vehicles present in the array to model movement.

Code

We implement Nagel-Schreckenberg model as an object as shown below. The model requires the following constraints to be specified: Road Length, Density, Maximum Velocity and Noise Probability.

```
#defining the traffic simulation object
class trafficsimulator:

    #storing the pre-specified values for future use in the model
implementation
    def __init__(self, road_length, density, max_velocity, random_p):
        self.road_length = road_length
        self.density = density
        self.max_velocity = max_velocity
        self.random_p = random_p
        self.counter = 0
        self.initialize()

    #this funtion initializes a one-dimensional array filled with negative
values to model the real-world road
    #and fills it with vehicles at random locations with random speeds
based on the specified density
    def initialize(self):
      #initializing the array
        self.sim = -np.ones(self.road_length, dtype = int)
        #filling with vehicles
        for i in range(len(self.sim)):
            if random.random() <= self.density:
                self.sim[i] = random.randint(0,self.max_velocity)

    #this function updates the array to simulate the movement on the road
over one time step
    def update(self):
        #creating a temporary array to store the updated locations and
velocities of the vehicles
        next_sim = -np.ones(self.road_length, dtype = int)

        #finding the locations of different vehicles present in the array
and storing them under a seperate variable name
        car_indexes = np.argwhere(self.sim  > -1)
        number_of_cars = len(car_indexes)
```

```
        for i in range(number_of_cars):
            #storing the current vehicles's location, next vehicles's
location and current vehicle's velocity
            ci = car_indexes[i]
            ni = car_indexes[(i+1)%number_of_cars]
            v = self.sim[ci]
            #calculating the distance between the current vehicle and the
next one
            distance = (ni-ci)%self.road_length
            #accelerating the vehicle
            if v < self.max_velocity and distance >= v+1:
                v += 1
            #slowing down the vehicle
            if v > 0 and distance <= v:
                v = distance-1
            #slowing random vehicles down to add noise
            if v > 0 and random.random() <= self.random_p:
                v = v-1
            #updating the temporary array
            next_sim[(ci+v)% self.road_length] = v
            #counting the number of vehicles passing through the last cell
            if ci + v >= self.road_length:
                self.counter += 1
        #updating the simulation
        self.sim = next_sim
   #function to display the simulation for every time-step
    def display(self):
        print(''.join('-' if x == -1 else str(x) for x in self.sim))
```

Under the update function in the traffic simulator object, we can notice that the rules for the movement of the car as follows:

1. Acceleration: If the velocity of a car is less than the maximum specified velocity and the distance between that car and the next one is larger than velocity(+1), the velocity of that car is increased by one.
2. Slowing Down: If there is a vehicle in between a car's current cell and the target cell, then its velocity is decreased to one less than the distance between that car and the next one.

3. Randomization: For a pre-specified probability, a random car's velocity is decreased by one.
4. Car Motion: Each car advances by v cells where v is its velocity.

Visualizations

The traffic is visualized by displaying the array which is modelling the traffic. Thus, each number represents the presence of a vehicle and its speed, and a '-' represents empty space. The zeros represent a car that is not moving and thus a cluster of zeros would represent a traffic jam in the real world. We visualize traffic situations with different levels of density. The maximum velocity and random noise stay constant at 5 and 0.4 respectively.

1. Visualization 1
    a. Density: 0.2

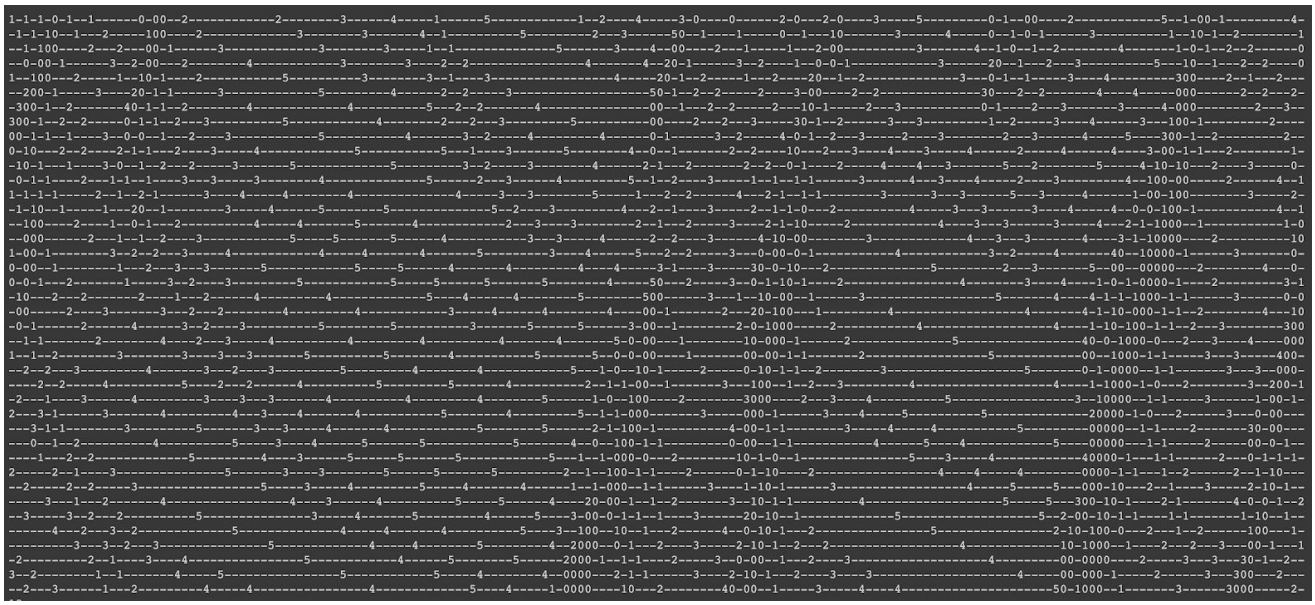

Fig. 1: Low Density Model

2. Visualization 2

    a. Density: 0.4

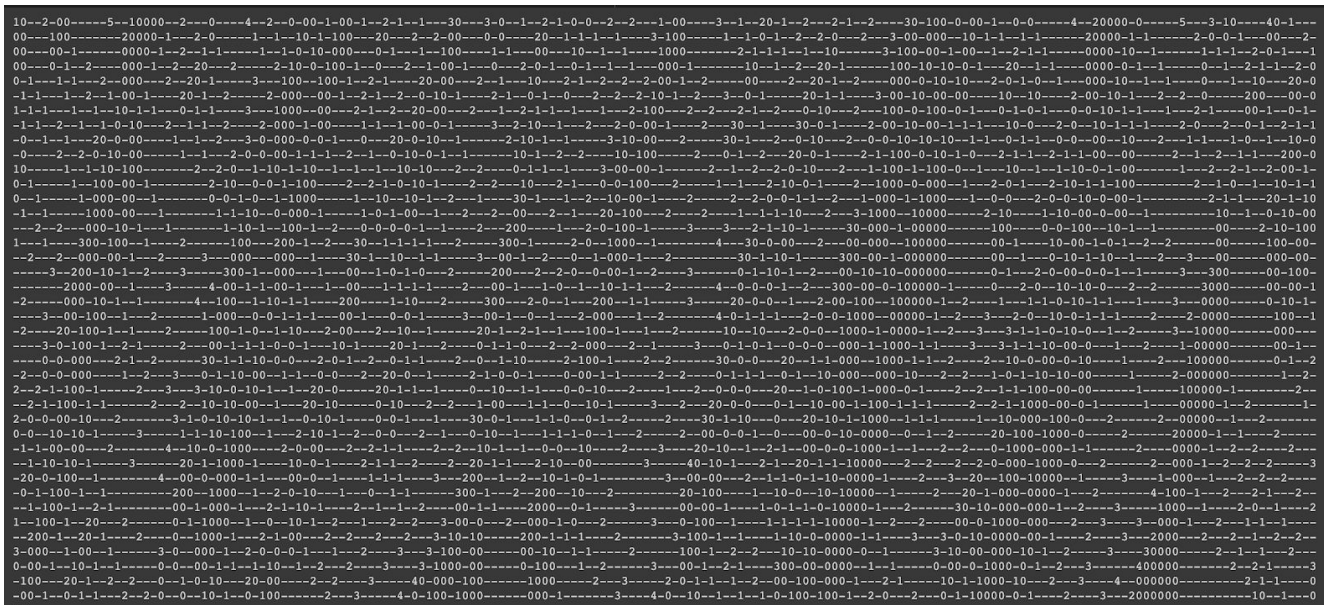

Fig. 2: Medium Density Model

3. Visualization 3

    a. Density: 0.6

Fig. 3: High Density Model

The clustering of zeroes in these different visualizations give us an idea how traffic jams can develop in our simulation over time. We can see that as density increases, we see an increase not just in the occurrences of these traffic jams, but also in the magnitude of these jams as higher densities have traffic jams affecting higher number of cars.

Flow vs Density

The following graph plots the rate of flow of the cars through one point in the array for different levels of density. The flow rate is calculated after 100 iterations of the model with a max velocity of 5, noise level of 0.2 and on a 100 cell 2-D array. The flow count is given by the counter variables calculated through the update function. Through this graph, we can see that the average flow rate initially increases upto a threshold (near density = 0.2 in our case) and gradually decreases and approaches zero as density increases.

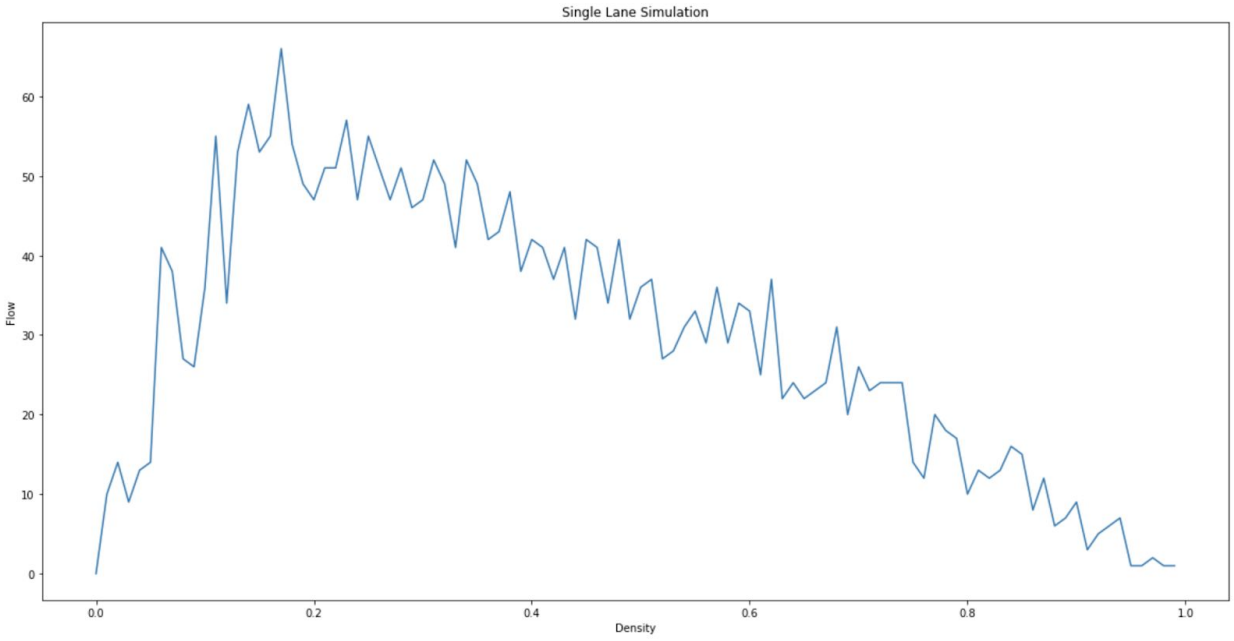

Fig. 4: Flow vs Density Graph for Single Lane model

Hence, from the real world perspective, this would mean that as the number of cars increases on the road, up to a certain point, the number of cars passing through a point would keep going up. Once the number of cars on the road surpasses that point, less and less cars would pass through that point and the probability of having traffic jams would increase as density increases. The density of cars on the road at this point can be called the threshold velocity

We can also plot the graph for different maximum velocities.

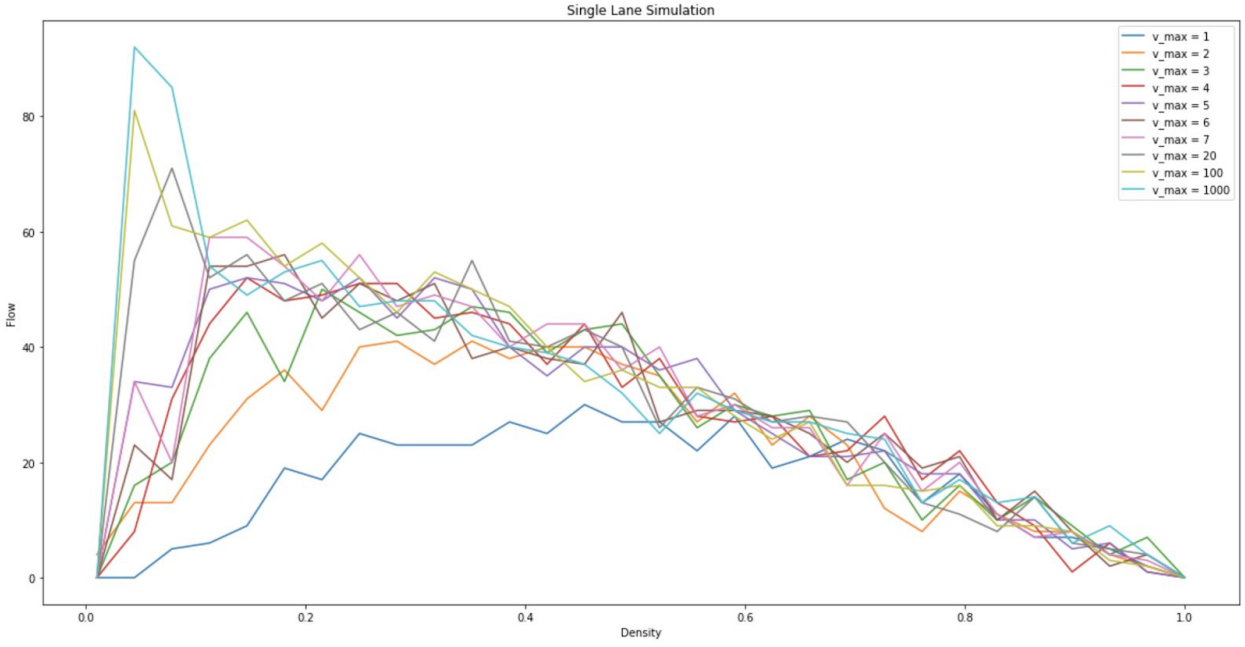

Fig. 5: Flow vs Density Graph for different maximum velocities.(Single Lane Model)

We can see how the threshold density changes for different maximum velocities. For higher maximum velocities, the threshold density tends to be lower, but the highest flow rate tends to be higher. If we calculate the area under the graph for each of these velocities, we can see which maximum velocity reacts best to change in densities and allows the most number of cars to pass through the system in a fixed amount of time.

The graph below plots the area under the graph for each of the maximum velocities plotted above.

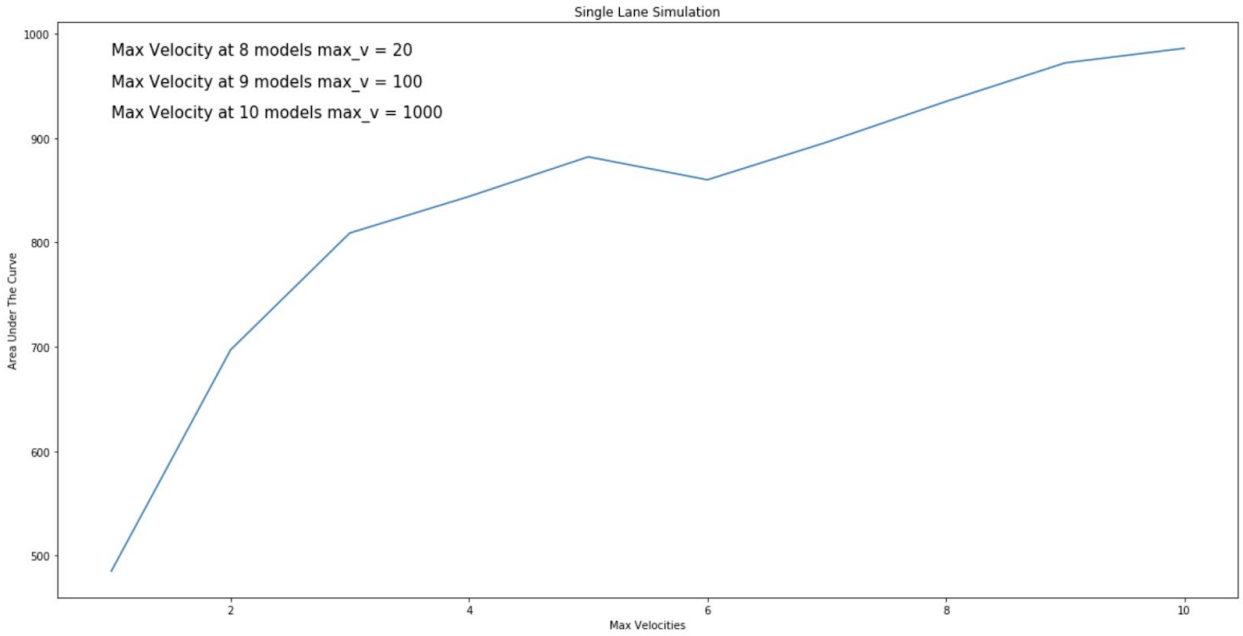

Fig. 6: Area under the flow vs density curve for different maximum velocities.(Single Lane

Through this graph, we can see that for higher maximum velocities, the flow volume over different densities tends to be higher.

Part 2: Double Lane Traffic Simulation

This part of the paper aims to model and simulate traffic in a double lane. The model takes inspiration from the Ricket-Nagel-Schreckenberg-Latour Two Lane model and applies the same rules as applied in their paper (Ricket, Nagel, Schreckenberg & Latour, 2019).

The basics of this model are exactly like the basics of the first model discussed in the paper . However, in this case, we implement two one-dimensional arrays to mirror the two lanes of the road. Along with Road Length, Density, Maximum Velocity and Noise Probability we also factor in a switching probability, which basically enumerates the probability of the car switching from one lane to the other.

Code

```
#defining the traffic simulation object
class ml_trafficsimulator:

    #storing the pre-specified values for future use in the model
implementation
    def __init__(self, road_length, density, max_velocity, random_p, switch_p):
        self.road_length = road_length
        self.density = density
        self.max_velocity = max_velocity
        self.random_p = random_p
        self.switch_p = switch_p
        self.counter = 0
        self.initialize()

    #this funtion initializes two one-dimensional arrays filled with negative
values to model the real-world road
    #and fills it with vehicles at random locations with random speeds based on
the specified density
    def initialize(self):
        #initializing the arrays
        self.sim_L1 = -np.ones(self.road_length, dtype = int)
        self.sim_L2= -np.ones(self.road_length, dtype = int)
        #filling with vehicles
        for i in range(2*self.road_length):
            if random.random() <= self.density:
              if i < self.road_length:
                self.sim_L1[i] = random.randint(0,self.max_velocity)
              else:
                self.sim_L2[i-self.road_length] =
random.randint(0,self.max_velocity)


    #this function updates the array to simulate the movement on the road over
```

```
one time step
    def update(self):
        global rear
        global
        #creating temporary arrays to store the updated locations and
velocities of the vehicles
        next_sim_L1 = -np.ones(self.road_length, dtype = int)
        next_sim_L2 = -np.ones(self.road_length, dtype = int)

        #finding the locations of different vehicles present in both the arrays
and storing them under seperate variable names
        car_indexes_L1 = np.argwhere(self.sim_L1  > -1)
        car_indexes_L2 = np.argwhere(self.sim_L2  > -1)
        number_of_cars_L1 = len(car_indexes_L1)
        number_of_cars_L2 = len(car_indexes_L2)
        dynamic_index_L1 = car_indexes_L1
        dynamic_index_L2 = car_indexes_L2

        #this for loop checks if the cars in Lane 1 need to switch lanes
        for i in range(number_of_cars_L1):
            #storing the current vehicles's location, next vehicles's location
and current vehicle's velocity
            ci = car_indexes_L1[i]
            ni1 = car_indexes_L1[(i+1)%number_of_cars_L1]
            v = self.sim_L1[ci]
            #breaks out if there is enough space in the front for the car to
move
            if ((ni1-ci)%self.road_length)-1 >= v+1:
              break
             #breaks out if there is a car in the same position on the other
lane
            if self.sim_L2[ci] >= 0 or (ni1-ci)%self.road_length > v:
              break
            #checks if there is a car ahead in the other lane and adjust the
speed of the car based on its location if it were to switch
            for i in range(1, self.max_velocity):
              if self.sim_L2[(ci + i)%self.road_length] >= 0:
                front = i - 1
                break
              else:
                front = self.max_velocity-1
            #checks if there is a car in the other lane that can potentially
crash if the car was to switch lanes
            for i in range(1,self.max_velocity):
```

```
            if self.sim_L2[(ci - i)%self.road_length] >= 0:
              rear = False
            else:
              rear = True
          #given the conditions, the lane is changed the indexes are updated
          if front >= v and rear == True and random.random() <=
self.switch_p:
            curr_index = np.argwhere(dynamic_index_L1 == ci)
            update_index_L1 = np.delete(dynamic_index_L1, curr_index)
            dynamic_index_L1 = update_index_L1

            dynamic_index_L2 = dynamic_index_L2.flatten()
            insert = np.searchsorted(dynamic_index_L2, ci[0])
            dynamic_index_L2 = np.insert(dynamic_index_L2, insert, ci)

        #this for loop checks if the cars in Lane 2 need to switch lanes with
same mechanisms as the previous function
        for i in range(number_of_cars_L2):
            ci = car_indexes_L2[i]
            ni2 = car_indexes_L2[(i+1)%number_of_cars_L2]
            v = self.sim_L2[ci]

            if ((ni2-ci)%self.road_length) >= v+1:
              break

            if self.sim_L1[ci] >= 0:
              break

            for i in range(1, self.max_velocity):
              if self.sim_L1[(ci + i)%self.road_length] >= 0:
                front = i - 1
                break
              else:
                front = self.max_velocity-1

            for i in range(1,self.max_velocity):
              if self.sim_L1[(ci - i)%self.road_length] >= 0:
                rear = False
              else:
                real = True

            if front >= v and rear == True and random.random() <=
self.switch_p:
              curr_index = np.argwhere(dynamic_index_L2 == ci)
```

```
            update_index_L2 = np.delete(dynamic_index_L2, curr_index)
            dynamic_index_L2 = update_index_L2

            dynamic_index_L1 = dynamic_index_L1.flatten()
            insert = np.searchsorted(dynamic_index_L1, ci[0])
            dynamic_index_L1 = np.insert(dynamic_index_L1, insert, ci)

        #updating the locations of the vehicles in lane 1
        for i in range(len(dynamic_index_L1)):
          #storing the current vehicles's location, next vehicles's location
and current vehicle's velocity
          ci = dynamic_index_L1[i]
          ni = dynamic_index_L1[(i+1)%len(dynamic_index_L1)]
          v = self.sim_L1[ci]
          #calculating the distance between the current vehicle and the next
one
          distance = (ni-ci)%self.road_length
          #accelerating the vehicle
          if v < self.max_velocity and distance >= v:
              v += 1
          #slowing down the vehicle
          if v > 0 and distance <= v:
              v = distance-1
          #slowing random vehicles down to add noise
          if v > 0 and random.random() <= self.random_p:
              v = v-1
          #updating the temporary array
          next_sim_L1[(ci+v)% self.road_length] = v
          #counting the number of vehicles passing through the last cell
          if ci + v >= self.road_length:
              self.counter += 1

        #updating the locations of the vehicles in lane 2 with same mechanisms
as the previous function
        for i in range(len(dynamic_index_L2)):
          ci = dynamic_index_L2[i]
          ni = dynamic_index_L2[(i+1)%len(dynamic_index_L2)]
          v = self.sim_L2[ci]

          distance = (ni-ci)%self.road_length
          if v < self.max_velocity and distance >= v:
              v += 1

          if v > 0 and distance <= v:
```

```
            v = distance-1

        if v > 0 and random.random() <= self.random_p:
            v = v-1

        next_sim_L2[(ci+v)% self.road_length] = v

        if ci + v >= self.road_length:
            self.counter += 1

    #updating the simulation
    self.sim_L1 = next_sim_L1
    self.sim_L2 = next_sim_L2

#function to display the simulation for every time-step
 def display(self):
    print(''.join('-' if x == -1 else str(x) for x in self.sim_L1))
    print(''.join('.' if x == -1 else str(x) for x in self.sim_L2))
```

Under the update function in the multi lane traffic simulator object, we can notice that there are some additional rules for the movement of the car. These rules arise from the fact that there are two lanes in this simulation, and thus we need to model the lane changing behavior of the car. The function first considers the cars in lane 1 and checks if they need to switch. This process consists of the following actions:

1. Front: If the distance between the current car and the next one is larger than its velocity(+1), then there is no requirement to change lanes and the function breaks.
2. Adjacent Cell: If there is a car in the same cell of the adjacent lane, then it's not possible for the car to switch lanes and the function breaks.
3. Adjacent Lane Front: If the distance between the current car and the next one in the other lane is larger than its velocity, then the velocity of the car is decreased to one lower than the other car's velocity.

4. Adjacent Lane Rear: The cells behind the current cell in the adjacent lane are checked to see if there is a car that can potentially crash into the current car after the lane switch.

Based on the above four conditions, the car switches lanes. The same process is carried out for lane 2. Each of the lanes are then updated under the same rules from part 1.

Visualizations

We visualize traffic situations with different levels of density. The maximum velocity, random noise and switching probability stay constant at 5, 0.2 and 0.8 respectively. The "-" represent empty spaces in lane 1 while "." represent empty spaces in lane 2.

4. Visualization 1
    a. Density: 0.2

Fig. 7: Low Density Model

5. Visualization 2

    a. Density: 0.4

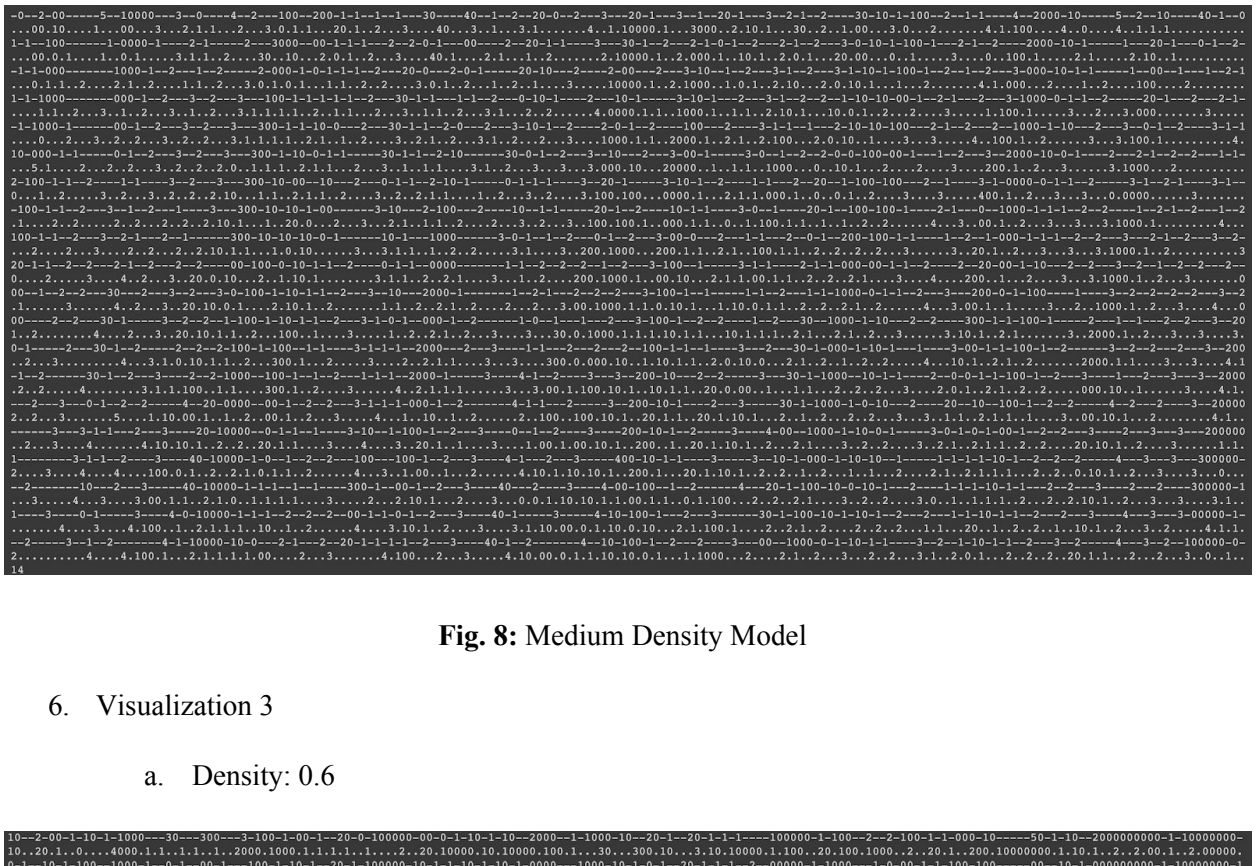

Fig. 8: Medium Density Model

6. Visualization 3

    a. Density: 0.6

Fig. 9: High Density Model

Similar to the results in part 1, we see that as density increases, there is an increase not just in the occurrences of these traffic jams, but also in the magnitude of these jams as higher densities have traffic jams affecting higher number of cars. However, when comparing the magnitudes of the traffic jams in Part 1 to the traffic jams in Part 2, we see that the jams have much lesser magnitude in Part 2 for the same density, showcasing how having 2 lanes is able to increase the efficiency of traffic for the same density.

Flow vs Density

The following graph plots the rate of flow of the cars through a point in each of the array for different levels of density. The flow rate is calculated after 100 iterations of the model with a max velocity of 5, noise level of 0.2 and on two 100 cell 2-D arrays. The flow count is given by the counter variables calculated through the update function. Similar to part 1, we see that the average flow rate increases upto the threshold (near density = 0.2 in our case) and gradually decreases and approaches zero as density increases.

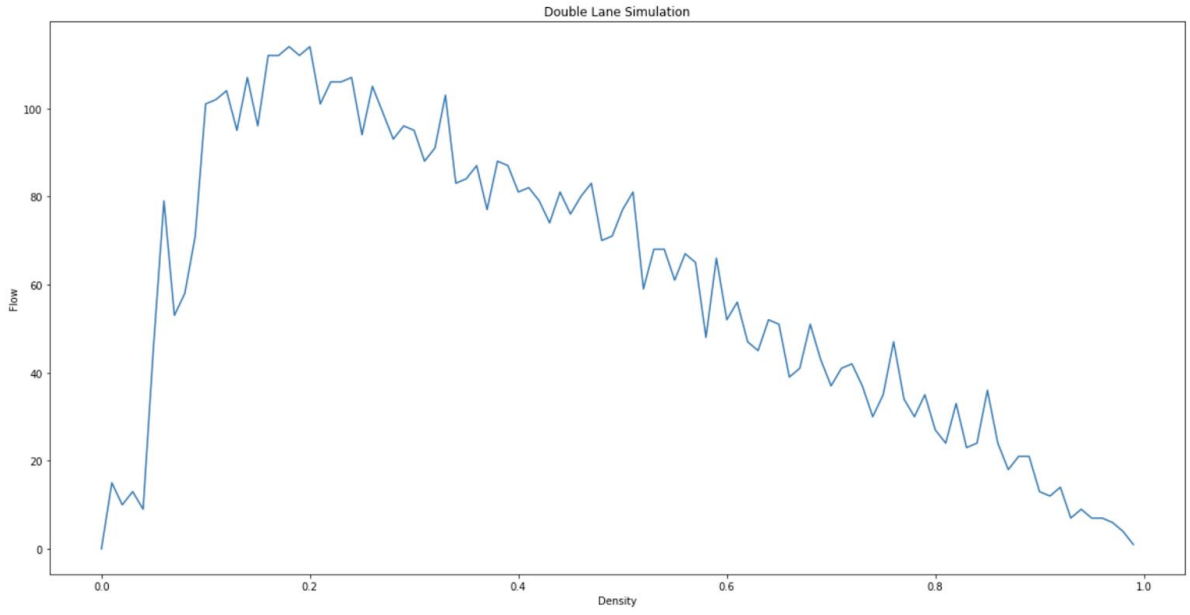

Fig. 10: Flow vs Density Graph for Two Lane model

If we compare the flow rates of the first model to the flow rates of the second model over different densities, we get the following plot.

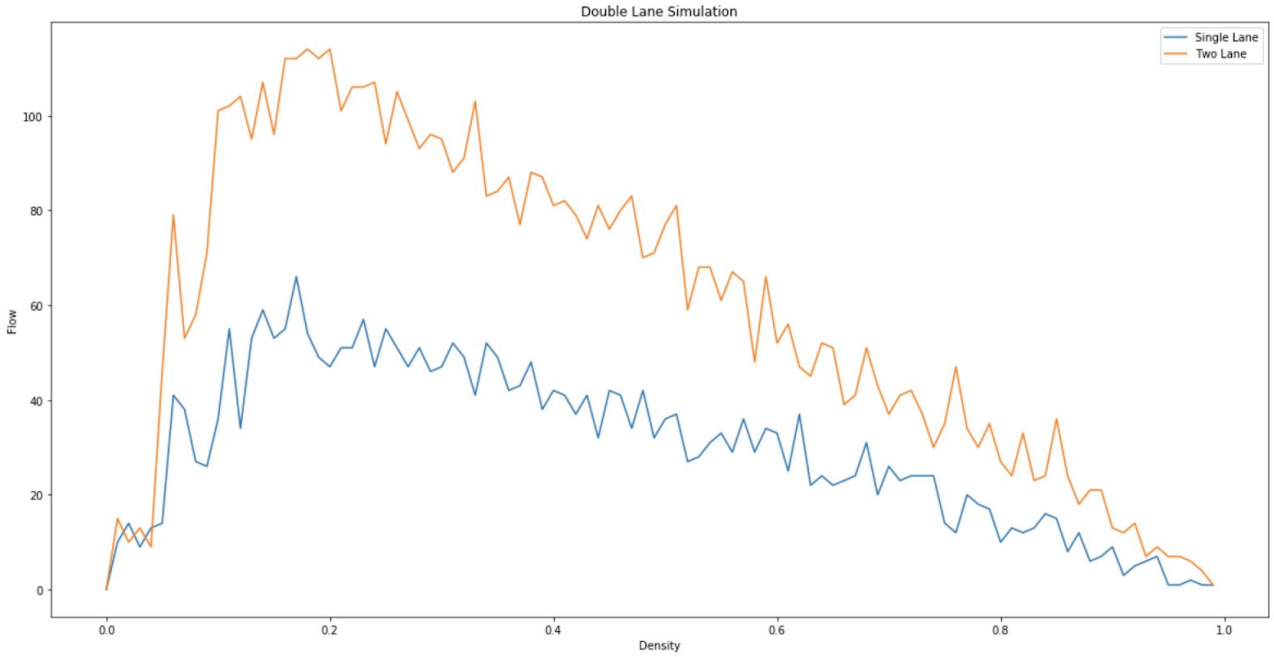

Fig. 10: Flow vs Density Graph for both models

Through the graph, we see that the general shape of the graph remains the same and the main difference in the graphs comes from the fact that the two lane model incorporates twice the number of cars as compared to the single lane model. We also see that the two lane model is able to hold and maintain a high flow rate around the threshold density as compared to the single lane model which touches it's highest flow rate and then immediately comes down. We can also plot the graph for different maximum velocities.

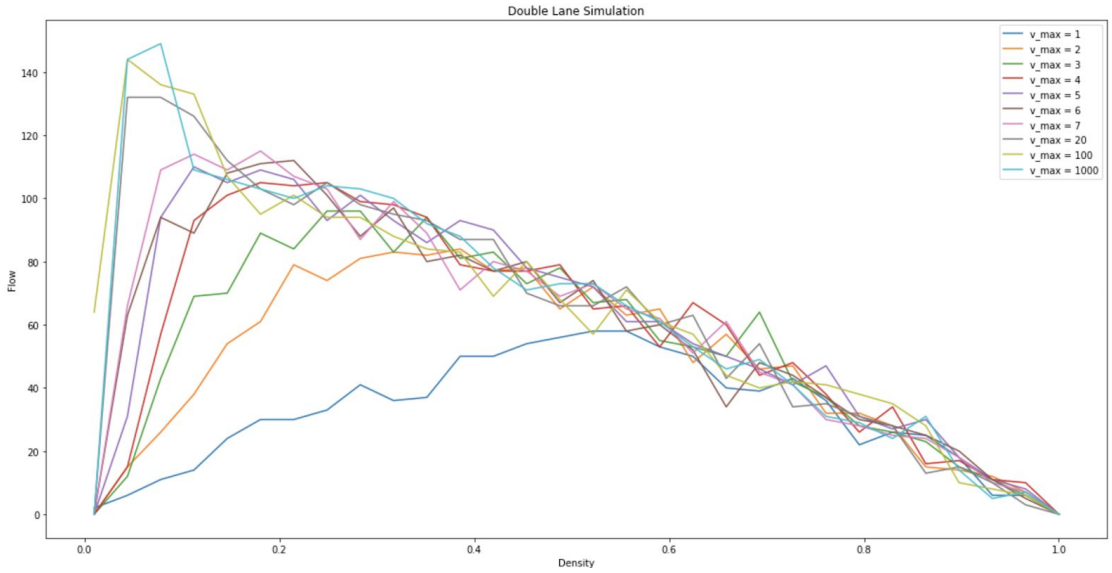

Fig. 11: Flow vs Density Graph for different maximum velocities. (Two Lane Model)

Through the graph in Fig 11, we see how the threshold density in this case, similar to part 1, changes for different maximum velocities and is lower for higher maximum velocities. However, in this case, the maximum flow rate is larger in relative scale as compared to part 1.

Comparing the area under the curve for the different maximum velocities, we get the following graph, showing that even in the case of two lanes, for higher maximum velocities, the flow volume over different densities tends to be higher.

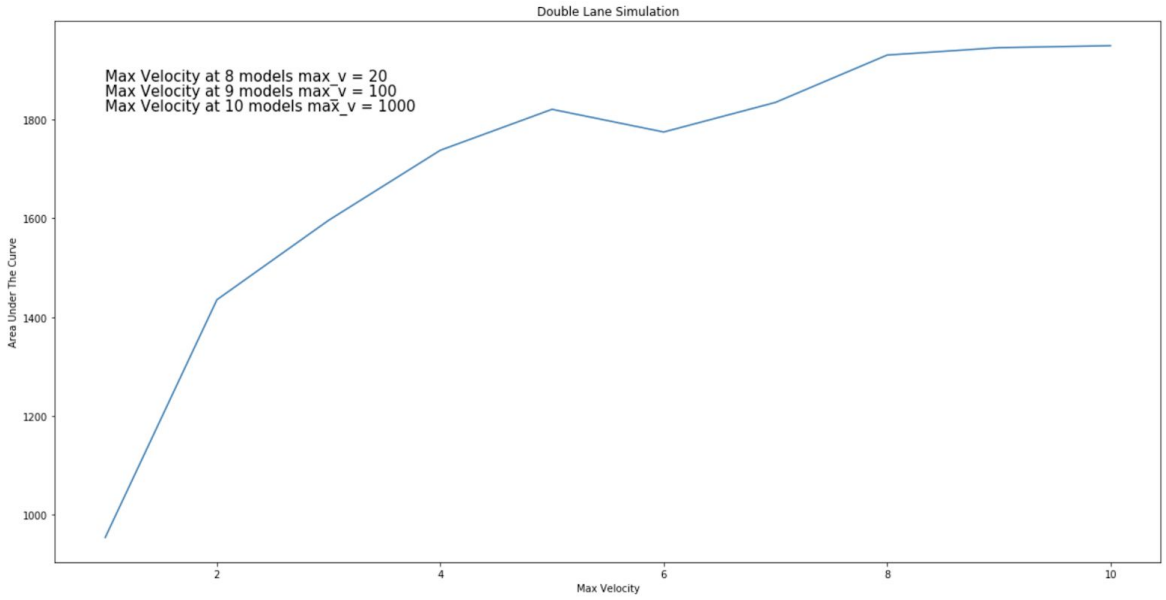

Fig. 12: Area under the flow vs density curve for different maximum velocities.(Two Lane Model)

Key questions

1. **Question 1**

   a. **How much more traffic can flow through a 2-lane road compared to a 1-lane road at the same traffic density?**

   Through figure 10, we can see the difference in traffic flow between the one lane simulation and the two lane simulation for every density value from zero to one. It is pretty clear from the graph that the two lane system can accommodate a higher flow of traffic through a point for a given density as compared to the single lane model. The difference in flow rate between the two models keeps increasing upto the threshold density after which it starts decreasing. Thus, the highest flow rate difference occurs around the threshold density wherein the two lane model flow rate is close to twice the flow rate of the single lane model.

What about roads with more than 2 lanes? Either model 3 or more lanes and report on your results, or predict how much larger the traffic flow through a 3+ lane road will be compared to a 1-lane road at the same traffic density.

As we increase the number of lanes in the simulation, the flow rate for a given density would also increase. This would occur because with the addition of new lanes, the choices that a given car would have to switch to another lane would increase, and thus the model would have a higher probability of cars being dispersed, driving down the probability of traffic jams and increasing flow rate. In the case of a one lane simulation, the car has no choice but to stop and wait till the car in front moves, but a car in the middle of a three lane simulation has the choice of going either left or right, and thus has a higher probability of not stopping.

b. **How applicable is this model to traffic in Berlin? (If you are not in Berlin, comment on how applicable the model is to traffic in your city.) Write a short paragraph and motivate why the model is suited to traffic in your city (or why it is not).**

The Autobahn is the federal controlled-access highway system in Germany. These highways are known all over the world for their system of having no speed limits. Through these simulations, I was able to understand how having higher speed limits can increase the overall efficiency of roads as they allow for a larger area under the curve for the flow vs density curve . The logical reasoning for this would be that:

1. The density on the autobahn roads increases from 0 to a maximum value which changes day to day depending on different circumstances but on most days only reaches 0.7 (approximated by my visits to the nearby autobahn stretch on a weekly basis).

2. Up to the threshold density, higher speed limits allow for higher flow at their respective density rates.

Because of 1 and 2, the total efficiency of the road remains higher at higher speed limits. The model allowed me to make this hypothesis because it's constraints largely resemble the characteristics of German traffic whose drivers have very disciplined behavior.

Future Work

There are several interesting metrics that can be calculated and compared from just the one and two lane simulations. For example, by adding a function to simulate a speed bump on one of the cells by reducing the speed of all cars passing through that cell to 0, we can measure the effect it has on the overall efficiency of the system. It could be possible that such a speed hump prevents vehicles from creating a traffic jam and increases overall efficiency. We could also change the structure of the model to implement intersections, roundabouts and other interesting road architectures to find the one with the best overall efficiency.

Bibliography

1. Nagel, K., & Schreckenberg, M. (1992). A Cellular Automaton Model for Freeway Traffic. Retrieved 7 November 2019, from https://course-resources.minerva.kgi.edu/uploaded_files/mke/YpqvNV/nagel-schreckenberg.pdf
2. Ricket, M., Nagel, K., Schreckenberg, M., & Latour, A. (2019). Two Lane Simulations using Cellular Automata. Retrieved 7 November 2019, from https://course-resources.minerva.kgi.edu/uploaded_files/mke/00100888-7879/rickert-et-al.pdf