Dennis Kageni

Traffic Simulation Using Cellular Automata

Spring 2020

Introduction

In this report, I discuss my findings after modeling highway traffic using cellular automata based on previous research by Nagel & Schreckengerg (1992). I use the rules from Rickert et al. (1996) to simulate the behavior on two-lane and three-lane highways. The goal is to explore the differences that arise from varying highway conditions such as the number of lanes, car density and driver behavior. The traffic models that were examined are:

1. Single-Lane Traffic Model
2. Two-Lane Traffic Model
3. Three-Lane Traffic Model

Here are the model's base features:

- Numpy arrays with periodic boundary conditions are used to represent **lanes.** "Periodic boundary conditions" basically means it's a looped/ circular road where cars exiting the road on the right re-enter the space on the left.
- Integers >= 0 in the Numpy array represent **cars**
- A car's **velocity** is encoded in the integer value with 0 denoting a stationary car. For example, a velocity of 2 means that the car can move two cells per timestamp. Note that velocities are capped by the parameter max_speed

Assumptions

1. Car movement is unidirectional
2. The cars have variable speeds
3. Time is discretized into steps
4. At each time step, drivers are limited to a rule set and do not act outside these rules

Part 1. Traffic Jams on a Circular Road

I instantiate the Nagel-Schreckengerg single-lane traffic model with the following parameters:

1. length - number of cells in the Numpy array which represents the length of the road
2. car_density - the probability that a cell will instantiate with an integer value (car) inside it. When low, this represents having fewer cars on the road and vice versa
3. max_speed - the speed limit assigned to a given car
4. slow_down_prob - the probability for a car to randomly slow down. This parameter makes the system non-deterministic and helps model real-world traffic behavior.

At each time step, the following rule-set is observed in the order below:

1. Acceleration: If the car has not reached speed limit, accelerate until the car velocity equals the maximum speed
2. Deceleration: Check the distance to the car in front. If it is less than our current speed, slow down such that our velocity is equal to the distance to the car in front. This helps avoid a collision
3. Randomization: If the car is in motion, reduce its speed by 1 with a probability slow_down_prob. The randomization element helps simulate "human error" e.g hard brake, slow down without cause (because they are distracted). This is what results in the "shockwave phenomenon" that propagates backward in the entire system, where these result in small fluctuations in the distances between cars which breaks the free flow. Without the randomization step, it would mean that no human error would be observed and a flow breakdown would not occur
4. Car motion: Move each vehicle forward. The movement steps are equal to the velocity value

1.1 Model Building (Single-Lane)

```
# import required libraries
import numpy as np
import random
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()
%matplotlib inline

class TrafficSimulation():

    def __init__(self, length = 100, car_density = 0.3, slow_down_prob =
0.5, max_speed = 5):
        '''
        Create a new traffic simulation object. Cars are distributed
randomly
        along the road and start with random velocities.

        Inputs:

            length (int) The number of cells in the road. Default: 100.

            car_density (float) The fraction of cells that have a car on
them.
              Default: 0.3.

            slow_down_prob (float) The probability that a car will randomly
              slow down by 1 during an update step. Default: 0.5.

            max_speed (int) The maximum speed in car cells per update step.
              Default: 5.
        '''
        self.length = length # number of cells in the road
        self.car_density = car_density # fraction of cells that have a car
on them
        self.max_speed = max_speed # maximum speed in car cells per update
step
        self.slow_down_prob = slow_down_prob # probability that a car will
randomly slow down by 1 during an update step

        self.flow = 0 # Flow counter to track the number of cars exiting
```

```
the road on the left side after
                    # each time step.

        # Initialize lane 1; place cars on random index; initialize cars at
random velocities
        self.lane_one = -np.ones(self.length, dtype=int) # "-1" represents
an empty cell
        random_idx = np.random.choice(range(self.length),
size=int(round(car_density * self.length)), replace=False)
        self.lane_one[random_idx] = np.random.randint(0, self.max_speed +
1, size=len(random_idx))

        # Track the time steps and total number of cars that passed the
simulation boundary
        # to estimate average traffic flow.

        self.time_step = 0
        self.cumulative_traffic_flow = 0


    def step(self, display = True):
        """
        Method to update car velocities

        We implement the following rules
        - acceleration
        - deceleration
        - random slow down
        """

        # looping through cars in the lane
        for i in range(self.length):
            if self.lane_one[i] != -1: # if there is a car in the cell
                dist = 0
                while self.lane_one[(i + (dist + 1)) % self.length] == -1:
                        dist += 1
                # accelerate
                if dist > self.lane_one[i] and self.lane_one[i] <
self.max_speed:
                    self.lane_one[i] += 1
                # decelerate
                if dist < self.lane_one[i]:
```

```
                    self.lane_one[i] = dist
                # slow down at random
                if self.lane_one[i] > 0 and np.random.random() <
self.slow_down_prob:
                    self.lane_one[i] -= 1

        # Here, we loop through updated velocities and move the cars to
their new positions
        lane_one_next_state = -np.ones(self.length, dtype=int)
        for i in range(self.length):
            if self.lane_one[i] != -1:
                lane_one_next_state[(i + self.lane_one[i]) % self.length] =
self.lane_one[i]
        self.lane_one = lane_one_next_state


        self.time_step += 1 # update time-step
        #update traffic-flow
        for i in range(self.max_speed):
            if self.lane_one[i] > i:
                self.cumulative_traffic_flow += 1

        if display:
            self.display()

    def display(self):
        """
        Method to print our the current state of the simulation
        """
        print(''.join('.' if x == -1 else str(x) for x in self.lane_one))

sim = TrafficSimulation()
for i in range(25): # observe the next 25 time steps
    sim.step()
```

1.2 Simulation Step (Single-Lane)

```
#_____________________________
# SINGLE LANE SIMULATION
#_____________________________

one_lane_traffic_flow = []
one_lane_traffic_density = []

for el in np.concatenate((
    np.arange(0.01, 0.079, 0.02), [0.08],
    np.arange(0.09, 0.109, 0.002),
    [0.11, 0.115, 0.12, 0.13, 0.14],
    np.arange(0.15, 1, 0.05))):
        one_lane_traffic_density.append(el)
        one_lane_traffic_flow.append([])
        for run in range(10):
            sim = TrafficSimulation(car_density = el)
            for i in range(50):
                sim.step(display=False)
            one_lane_traffic_flow[-1].append(sim.cumulative_traffic_flow /
sim.time_step)

#_____________________________________________________
# PLOTTING THE DENSITY OF THE SINGLE LANE SIMULATION
#_____________________________________________________

avg = np.mean(np.array(one_lane_traffic_flow), axis=1)
std_dev = np.std(np.array(one_lane_traffic_flow), axis=1)


plt.figure(figsize=(12, 6))
plt.plot(one_lane_traffic_density, one_lane_traffic_flow, 'r.', markersize
= 1)
plt.plot(one_lane_traffic_density, avg, 'k-')
plt.plot(one_lane_traffic_density, avg + 1.96*std_dev, 'k--')
plt.plot(one_lane_traffic_density, avg - 1.96*std_dev, 'k--')
plt.xlabel('Traffic Density')
plt.ylabel('Traffic Flow')
plt.title('Plot of Traffic Density vs Traffic Flow for a Single Lane
Model')
plt.show()
```

1.3 Model Outputs (Single-Lane)

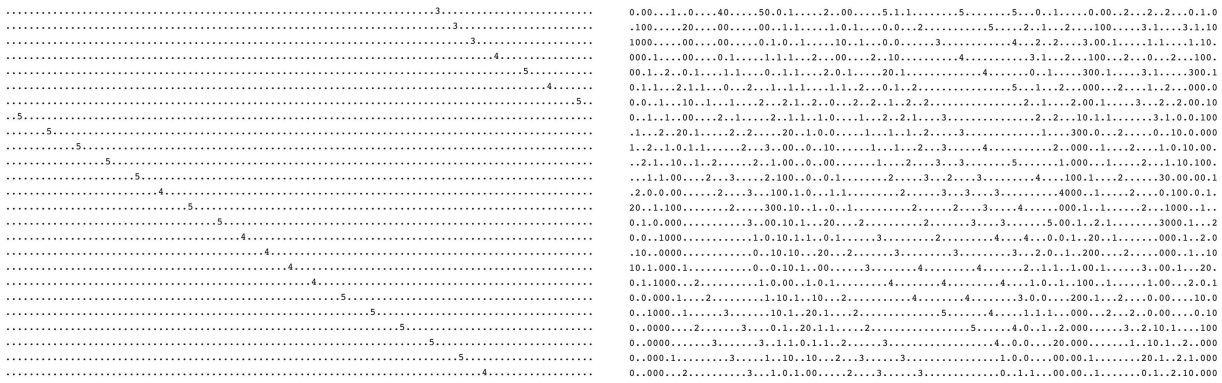

Figure 1. Cellular automata simulations showing the evolution of a single lane simulation over time. Parameters of the left: length = 100, car_density = 0.01, slow_down_prob = 0.5 , max_speed = 5. Parameters of the right: length = 100, car_density = 0.3, slow_down_prob = 0.5 , max_speed = 5.

In Figure 1, we can observe the effect of varying car densities. Intuitively, when controlling for other factors, more cars on the road results in the formation of traffic jams. Since this is a closed system, the traffic propagates backwards as cars keep flowing into the system.

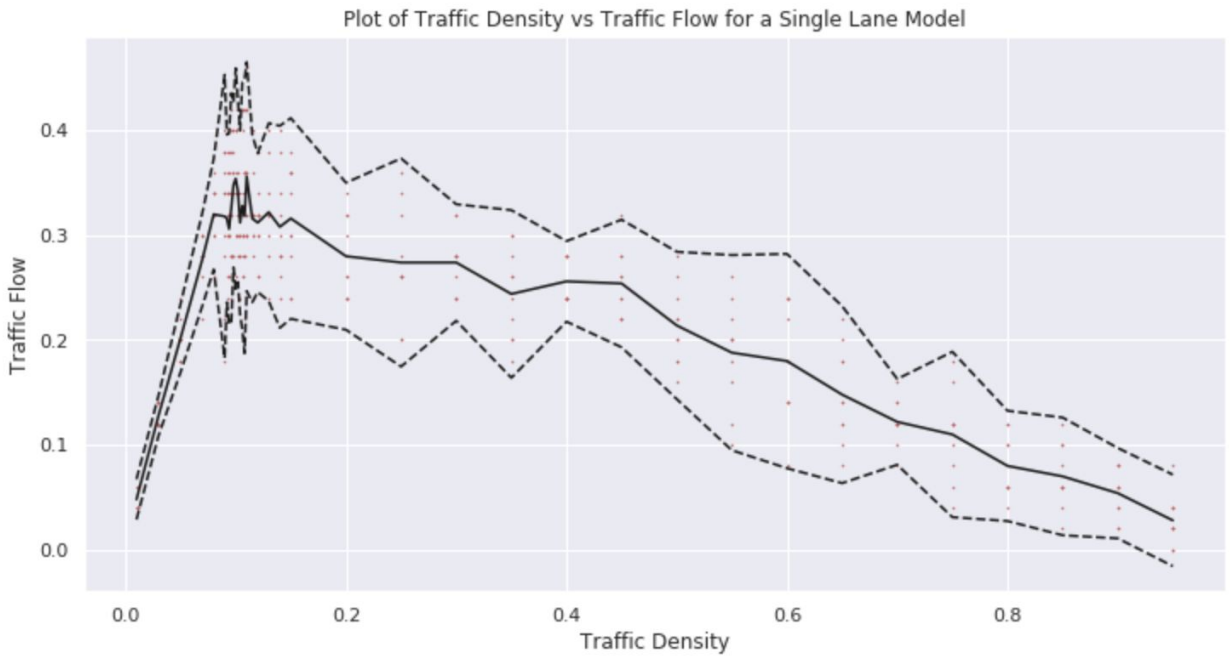

Figure 2. Density-Flow plot for the single lane model. The black line is the average flow over all simulations. The dashed lines are the 5% and 95% percentiles flow values to visualize the confidence interval[1]

Here, we observe a peak around the traffic density value of 0.1. We can interpret this as for a single lane model with 100 cells, maximum flow peaks when the system has roughly 10 cars. However, this estimate is highly variable and uncertain given the wide range of the upper and lower bounded confidence intervals. Although this can be corrected by running more simulations, the current implementation makes it extremely computationally expensive

[1] #confidenceintervals - presented 95% confidence intervals in the Density-Flow plot capture the variability in the simulation runs. These help us communicate the inherent uncertainty when using the mean as summary statistic

Part 2. Multi-Lane Highways

This is based on Ricket et al. (1996) extension of the Nagel-Schreckenberg model where he simulates lane changing behavior in two-lane highways. The key difference now is that when a car is no longer able to accelerate in its current lane, it is able to switch lanes given that there is enough forward and backward space in the lane it intends to switch to.

The model is instantiated with the following parameters:

1. length - number of cells in the Numpy array which represents the length of the road
2. car_density - the probability that a cell will instantiate with an integer value (car) inside it. When low, this represents having fewer cars on the road and vice versa
3. max_speed - the speed limit assigned to a given car
4. slow_down_prob - the probability for a car to randomly slow down. This parameter makes the system non-deterministic and helps model real-world traffic behavior.
5. switch_lanes_prob - if a lane switch is possible, this is the probability that the car will switch lanes

The updated rule-set is as follows:

1. Acceleration in the current lane: If the car has not reached speed limit, accelerate until the car velocity equals the maximum speed
2. Deceleration in the current lane: Check the distance to the car in front. If it is less than our current speed, slow down such that our velocity is equal to the distance to the car in front. This helps avoid a collision

3. Lane switch condition: When there isn't enough front space in the current lane, check if there is enough front space to accelerate in the lane you want to switch to (this is equal to current car velocity + 1). Also check that the available back space is greater than the maximum speed to make it impossible for any oncoming car to rear-end you. When both conditions are satisfied, switch lanes with probability switch_lanes_prob
4. Randomization: If the car is in motion, reduce its speed by 1 with a probability slow_down_prob
5. Car Motion: Move each vehicle forward. The movement steps are equal to the velocity value

2.1 Model Building (Two Lanes)

```
class TwoLaneTrafficSimulation():
    def __init__(self, length = 100, car_density = 0.3, max_speed = 5,
slow_down_prob = 0.5, switch_lane_prob = 1):
        """
        Traffic simulation object for Two Lanes. Identical to the
`TrafficSimulation` class above
        with the only exception being we have added a new variable
switch_lane_prob

        Inputs:

            length (int) The number of cells in the road. Default: 100.

            car_density (float) The fraction of cells that have a car on
them.
              Default: 0.3.

            slow_down_prob (float) The probability that a car will randomly
              slow down by 1 during an update step. Default: 0.5.

            max_speed (int) The maximum speed in car cells per update step.
              Default: 5.

            switch_lane_prob (int) The probability of a car switching lanes
when
```

```
                conditions allow.
                Default: 1
        """

        self.length = length
        self.car_density = car_density
        self.max_speed = max_speed
        self.slow_down_prob = slow_down_prob
        self.switch_lane_prob = switch_lane_prob # The probability of
switching lanes

        # Initialize lane 1; place cars on random index; initialize cars at
random velocities
        self.lane_one = -np.ones(self.length, dtype=int) # "-1" represents
an empty cell
        random_idx = np.random.choice(range(self.length),
size=int(round((car_density * self.length))), replace=False)
        self.lane_one[random_idx] = np.random.randint(0, self.max_speed +
1, size=len(random_idx))

        # Initialize lane 2; place cars on random index; initialize cars at
random velocities
        self.lane_two = -np.ones(self.length, dtype=int) # "-1" represents
an empty cell
        random_idx = np.random.choice(range(self.length),
size=int(round(car_density * self.length)), replace=False)
        self.lane_two[random_idx] = np.random.randint(0, self.max_speed +
1, size=len(random_idx))

        '''Track the time steps and total number of cars that passed the
simulation boundary
        to estimate average traffic flow.'''
        self.time_step = 0
        self.cumulative_traffic_flow = 0


    def step(self, display = True):

        self.orig_lane_one = self.lane_one.copy()
        self.orig_lane_two = self.lane_two.copy()

        for i in range(self.length):
```

```
            # encoding lane changing behavior rule set for lane one
            if self.lane_one[i] != -1: # if there's a car in the cell

                front_dist_lane_one = 0
                front_dist_lane_two = 0
                back_dist_lane_two = 0

                while self.lane_one[(i + (front_dist_lane_one + 1)) %
self.length] == -1:
                    front_dist_lane_one += 1
                while self.lane_two[(i + (front_dist_lane_two + 1)) %
self.length] == -1:
                    front_dist_lane_two += 1
                while self.lane_two[(i - (back_dist_lane_two + 1)) %
self.length] == -1:
                    back_dist_lane_two +=1

                if self.lane_one[i] == self.max_speed:
                    if front_dist_lane_one < self.lane_one[i] and
self.lane_two[i] == -1 and front_dist_lane_two >= self.lane_one[i] and
back_dist_lane_two >= self.max_speed and np.random.random() <
self.switch_lane_prob:
                        self.lane_two[i] = self.lane_one[i]
                        self.lane_one[i] = -1

                elif self.lane_one[i] < self.max_speed and self.lane_one[i]
> 0:
                    if front_dist_lane_one < self.lane_one[i]+1 and
self.lane_two[i] == -1 and front_dist_lane_two >= self.lane_one[i]+1 and
back_dist_lane_two >= self.max_speed and np.random.random() <
self.switch_lane_prob:
                        self.lane_two[i] = self.lane_one[i]
                        self.lane_one[i] = -1

                elif self.lane_one[i] == 0:
                    if front_dist_lane_one < self.lane_one[i]+1 and
self.lane_two[i] == -1 and front_dist_lane_two >= self.lane_one[i]+1 and
back_dist_lane_two >= self.max_speed and np.random.random() <
self.switch_lane_prob:
                        self.lane_two[i] = self.lane_one[i]
                        self.lane_one[i] = -1
```

```
            # encoding lane changing behavior rule set for lane two
            if self.orig_lane_two[i] != -1:

                front_dist_lane_two = 0
                front_dist_lane_one = 0
                back_dist_lane_one = 0

                while self.orig_lane_two[(i + (front_dist_lane_two + 1)) %
self.length] == -1:
                    front_dist_lane_two += 1
                while self.lane_one[(i + (front_dist_lane_one + 1)) %
self.length] == -1:
                    front_dist_lane_one += 1
                while self.lane_one[(i - (back_dist_lane_one + 1)) %
self.length] == -1:
                    back_dist_lane_one += 1

                if self.orig_lane_two[i] == self.max_speed:
                    if front_dist_lane_two < self.orig_lane_two[i] and
self.lane_one[i] == -1 and front_dist_lane_one >= self.orig_lane_two[i] and
back_dist_lane_one >= self.max_speed and np.random.random() <
self.switch_lane_prob:
                        self.lane_one[i] = self.orig_lane_two[i]
                        self.lane_two[i] = -1

                elif self.orig_lane_two[i] < self.max_speed and
self.orig_lane_two[i] > 0:
                    if front_dist_lane_two < self.orig_lane_two[i]+1 and
self.lane_one[i] == -1 and front_dist_lane_one >= self.orig_lane_two[i]+1
and back_dist_lane_one >= self.max_speed and np.random.random() <
self.switch_lane_prob:
                        self.lane_one[i] = self.orig_lane_two[i]
                        self.lane_two[i] = -1

                elif self.orig_lane_two[i] == 0:
                    if front_dist_lane_two < self.orig_lane_two[i]+1 and
self.lane_one[i] == -1 and front_dist_lane_one >= self.orig_lane_two[i]+1
and back_dist_lane_one >= self.max_speed and np.random.random() <
self.switch_lane_prob:
                        self.lane_one[i] = self.orig_lane_two[i]
```

```
                    self.lane_two[i] = -1

            # speed change rules for lane one
            if self.lane_one[i] != -1:
                dist = 0
                while self.lane_one[(i + (dist + 1)) % self.length] == -1:
                    dist += 1
                if dist > self.lane_one[i] and self.lane_one[i] <
self.max_speed:
                    self.lane_one[i] += 1
                if dist < self.lane_one[i]:
                    self.lane_one[i] = dist
                if self.lane_one[i] > 0 and np.random.random() <
self.slow_down_prob:
                    self.lane_one[i] -= 1

            # speed change rules for lane one
            if self.lane_two[i] != -1:
                dist_other = 0
                while self.lane_two[(i + (dist_other + 1)) % self.length]
== -1:
                    dist_other += 1
                if dist_other > self.lane_two[i] and self.lane_two[i] <
self.max_speed:
                    self.lane_two[i] += 1
                if dist_other < self.lane_two[i]:
                    self.lane_two[i] = dist_other
                if self.lane_two[i] > 0 and np.random.random() <
self.slow_down_prob:
                    self.lane_two[i] -= 1

        # the car motion rule
        lane_one_next_state = -np.ones(self.length, dtype=int)
        for i in range(self.length):
            if self.lane_one[i] != -1:
                lane_one_next_state[(i + self.lane_one[i]) % self.length] =
self.lane_one[i]
        self.lane_one = lane_one_next_state

        lane_two_next_state = -np.ones(self.length, dtype=int)
        for i in range(self.length):
            if self.lane_two[i] != -1:
```

```
                lane_two_next_state[(i + self.lane_two[i]) % self.length] =
self.lane_two[i]
        self.lane_two = lane_two_next_state
        # update time and traffic flows
        self.time_step += 1
        for i in range(self.max_speed):
            if self.lane_one[i] > i:
                self.cumulative_traffic_flow += 1
            if self.lane_two[i] > i:
                self.cumulative_traffic_flow += 1


        '''Display the function'''
        if display:
            self.display()


    def display(self):
        print(''.join('.' if x == -1 else str(x) for x in self.lane_one))
        print(''.join('.' if x == -1 else str(x) for x in self.lane_two))
        print('')

sim = TwoLaneTrafficSimulation()
sim.display()
for i in range(30):
    sim.step()
```

2.2 Simulation Step (Two Lanes)

```
#_____________________________
# TWO LANE SIMULATION
#_____________________________
two_lane_traffic_flow = []
two_lane_traffic_density = []
for el in np.concatenate((
    np.arange(0.01, 0.079, 0.02), [0.08],
    np.arange(0.09, 0.109, 0.002),
    [0.11, 0.115, 0.12, 0.13, 0.14],
    np.arange(0.15, 1, 0.05)
)):
    two_lane_traffic_density.append(el)
    two_lane_traffic_flow.append([])
```

```
    for run in range(10):
        sim = TwoLaneTrafficSimulation(car_density=el)
        for i in range(50):
            sim.step(display=False)
        two_lane_traffic_flow[-1].append(sim.cumulative_traffic_flow /
sim.time_step)


#_____________________________________________________
# PLOTTING THE DENSITY OF THE TWO LANE SIMULATION
#_____________________________________________________

avg_twolane = np.mean(np.array(two_lane_traffic_flow), axis=1)
std_dev_twolane = np.std(np.array(two_lane_traffic_flow), axis=1)
plt.figure(figsize=(18, 6))
plt.subplot(1,2,1)
plt.plot(one_lane_traffic_density, one_lane_traffic_flow, 'r.', markersize
= 1)
plt.plot(one_lane_traffic_density, avg, 'k-')
plt.plot(one_lane_traffic_density, avg + 1.96*std_dev, 'k--')
plt.plot(one_lane_traffic_density, avg - 1.96*std_dev, 'k--')
plt.xlabel('Traffic Density')
plt.ylabel('Traffic Flow')
plt.title('Plot of Traffic Density vs Traffic Flow for a One Lane Model')

plt.subplot(1,2,2)
plt.plot(two_lane_traffic_density, two_lane_traffic_flow, 'r.',
markersize=1)
plt.plot(two_lane_traffic_density, avg_twolane, 'k-')
plt.plot(two_lane_traffic_density, avg_twolane + 1.96*std_dev_twolane,
'k--')
plt.plot(two_lane_traffic_density, avg_twolane - 1.96*std_dev_twolane,
'k--')
plt.xlabel('Traffic Density')
plt.ylabel('Traffic Flow')
plt.title('Plot of Traffic Density vs Traffic Flow for a Two Lane Model')
plt.show()
```

2.3 Model Outputs (Two Lanes)

```
...2........0..1...1....2..000.1......2..2.0000.....10.1...........3......5.....3..1000.....2.000...
.1....2.......3.0.1.....30...1.1..10..0.1...1000...00..0...1.........3........4.00.10...2.....2.00..

......3......1..1...1....1.000...2......2.10000.....00..1.............4.......5..20000.....0.000...
...2....2.....0..1.1....0.1...1.1.0.1..1.1..000.1..00..0....1...........3.....0.00.00......3..0.00..

..........4....2..2..1....1000.....2....0.00000.....0.1..1.................4.....20000.1.....100.1..
.....2.....3...1..1..2...1..2..1.10..1..1..200.1.1.0.1..1....1.............4.0.00.0.1......1..10.1.

..............4..2..2..2..0000........3..100000.....0...2..2...................500000..1....000..1.
........3.....3.1..1...2...2.1..100...1..1.000.0.0..1.1...2....2.............10.0.10...2......200..1

2..............1..1..1...20000..........2000000.....0.....2..2.................200000.1..1...00.1...
1..........3..0..1..1....2.0..1.000.....2.1000..1.1.0..1.....3....3...........0..100.....2....000...

...3.............2.1..1..0000.1.........000000.1.....1......2...3..............00000.1..2.1..0.1..2.
..2..........2.1...2.1...0..1.0.00.1.....10000...10..1..1........4....4.......0..000.......2..000...
```

Figure 3. Cellular automata simulation showing the evolution of a double lane simulation over time.

Figure 4. Comparing the Density-Flow plots for the single lane model and double lane model

2.3 Model Outputs (Three Lanes)

```
0..1..2.....2.1......2..2...0.1.......3......4....000.0...2...2......0...1.....0.0...2....10.0.00..0
1.......3.0..10............10.00000.........3.1...2...2...3.......3..1000.1......3...3...10.000...2.
000....1.....3........3...10..1..1.00......2......00..1..2.1.....3....2.1.1...20.0..1..2.....2...000

.1..1...2....1..2..........30...2.........4.....3.00.10.....2....3....1....2....10.....2..0.10.00..0
.1......0.0..0.1.......2...0.10000.1.........0...2...2....3...3......20000..1.........3..2.0.1000...0.
00.1....1.......3........300....2.100.........3...00....20..1.......3..1.1..2.0.1.1..1....3.....3000

1..2..2...2....2...3.......00......3..........4.0.0.10.1......2.....3..1......3.00......1.0.0.10.1..
...2.....10..0...2........30.00000..1.......0.....2...2....2....3...00000...1.........1.0.0.000.1...1
00...2...1.........3.....000....0.00.1..........2.00....00....2.......2.1..20..10...2..2.....3..0000

..2..2...3..2....2.....4.0.0.1........3........1.10.00..1........3....2.1.....0.0.1.....0.0..100..1.
1.....3..00...1....2......0.100000...1......0.......2...2.....3...2.00000....1.........10.0.000...2..
00.....2..1...........3...0.1....100...2.........100....0.1.....2......1.1.0.1.00....1....3....20000

...1...2...2...3...2...0..10..1...........4....0.0.10.1..1...........4.1..2...0.0..1.....10..000...1
..2....1.0.1...1.....2....0.000000.....2....0.........2....3....2..100000......2......00.0.00.1...1.
0.1......2.1............2..1..2..000.....2.......00.1...0..1......2....0..10..10.1....1......3.00000
```

Figure 5. Cellular automata simulation showing the evolution of a triple lane simulation over time.

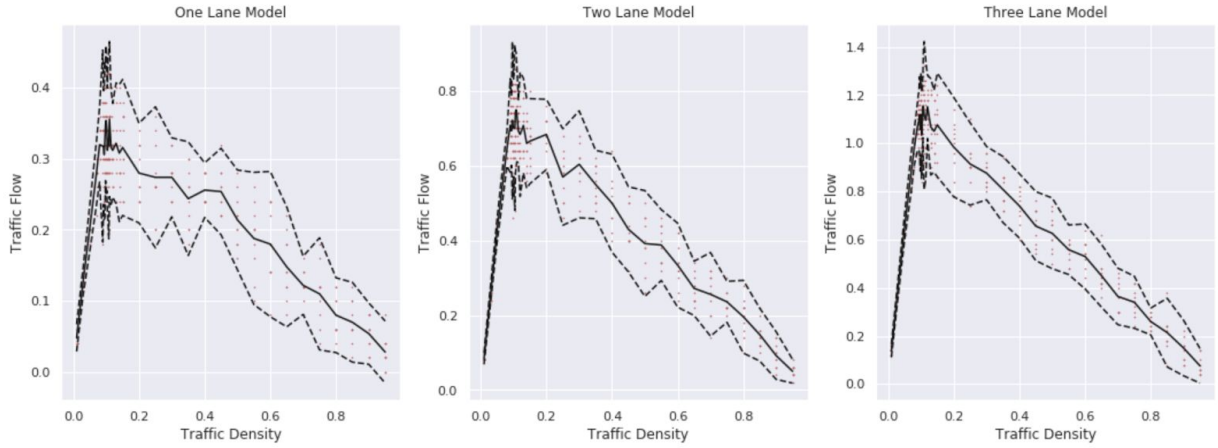

Figure 6. Comparing the Density-Flow plots for the single lane, double lane and triple lane models

Part 3. Discussion[2]

The double-lane model has double the flow of a single lane model. What's interesting, however, is that the traffic density for the double lane model still peaks at around 0.1. Therefore, it raises the question of what's the actual effect of adding a more lanes. Figure 7 shows how the Density-Flow plot of a three-lane model compares to the one-lane and two-lane models. We see that adding an extra lane results in the greatest increase in traffic flow only around the optimal density point. So essentially, adding lanes may only encourage the growth of the system with little or no impact to the resulting efficiency.

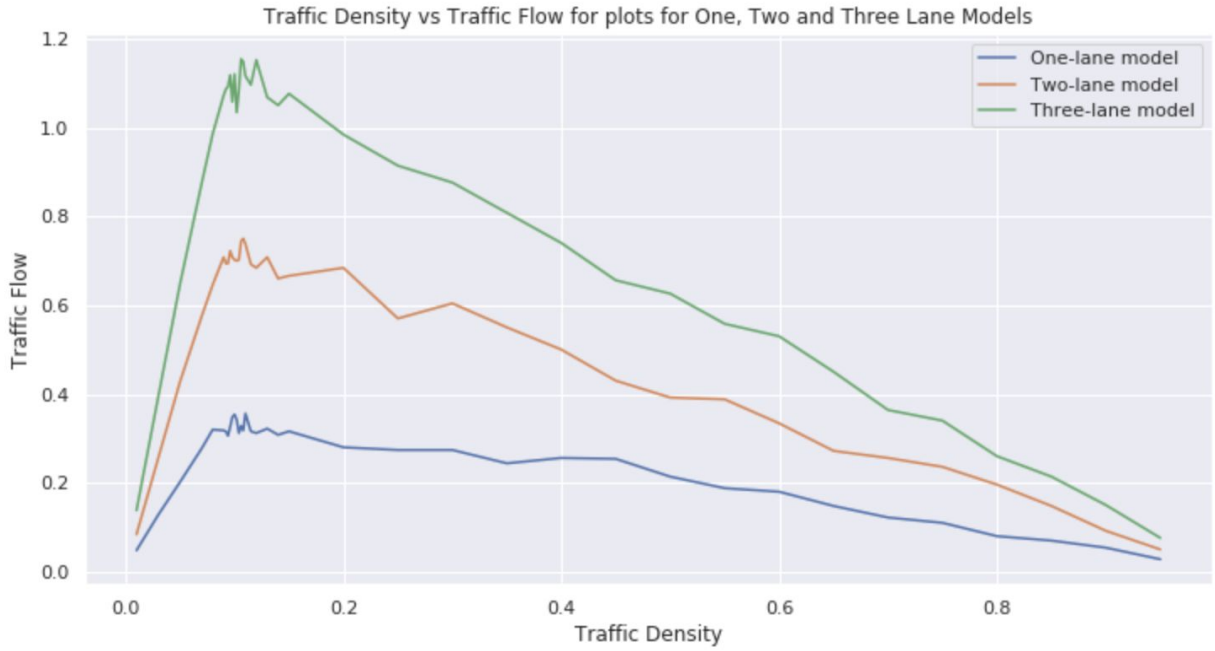

Figure 7. Comparing the effect of increasing lanes in a highway system

[2] #systemdymanics - the critical point for the traffic simulation for all three models is at a traffic density of approximately 0.1. The Density-Flow plots also help us observe the simulations' sensitivity with varying traffic flows and traffic densities.

Bad Driver Behavior

The most common example of bad driver behavior would be drivers who slow down randomly often, perhaps because they are distracted by their phones or by an attractive person walking along the sidewalk. This can be modeled by simply increasing the slow_down_prob parameter to 0.9. Both the cellular automata simulation (Figure 8) and Density-Plots (Figure 9) show the negative effect of bad-driving where there's a prevalence of traffic jams and the traffic flow rate reduces.

```
.0...0.1....0..0.000.00...0..10........10.00.....10.000......0....00...........0..1...1.....0...0.00
..100.....0..0.00.......0..1..0..1......0.0..000....1......1....1...1.........0.0....1..1......1.0.0
0...1........0.0..000.......0..0......00.........0.00..0..1...0.0.000..........000...000.0......1.00

.0....1.1...0..0.000.00....1.00........00.00.....00.000......0....00...........0...1...1....0...0.00
..000.....0..0.00.......0...1.0...1......10..00.1....1......1....1...1.........0.0.....1..1.....0.0.0
0....1.......0.0..000.......0...1.....00.........0.00..0...1..0.0.000..........000...000.0......0.00

.0.....1.1..0..0.000.00....0.00........00.00.....00.00.1.....0....00...........1...1...1...0....100
..000.....0..0.00.......0...0..1....2....00..00..1....1.......2...1...1.......0..1.....1...2....10.0
0.....1.......10..000.......0.....2...00.........0.00..0....1.0.0.000..........000...000.0......0.00
```

Figure 8. Cellular automata simulation showing the evolution of a triple lane simulation with bad driving behavior over time.

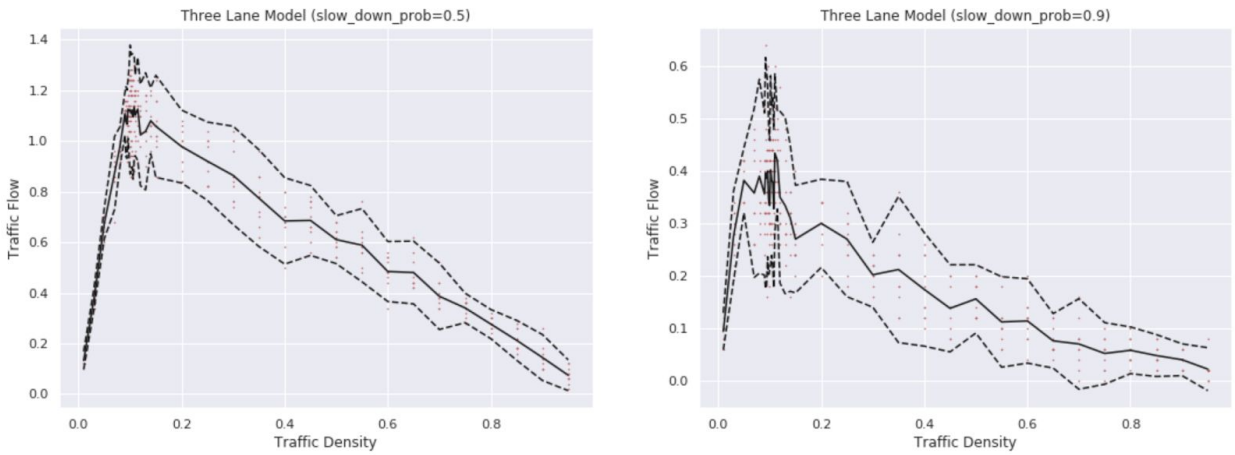

Figure 9. Comparing the Density-Flow plots for the three lane model with slow_down_prob = 0.5 and three lane model with slow_down_prob = 0.9

Applicability of the Model to Traffic in Buenos Aires

The unidirectional, straight, multi-lane model without any intersection or traffic lights is too simplistic to properly model traffic in Buenos Aires as a whole. However, the city center happens to have some unidirectional, multilane, mostly straight that are mostly straight roads. Should we fail to account for the presence of traffic lights and intersections, then our model may be better suited to make highly-localized inferences (for distances not more that a few blocks).

Future Work

The following are suggested modifications to make the model more realistic:

- Add multidirectionality to the model to reflect two-way traffic as seen in most roads
- Add road features such as traffic lights, intersections, roundabouts, exit-lanes, speed limits, among others.
- We can have a model tailored to study the effect of public transportation on highway design e.g having dedicated bus/taxi lanes, dropoff/pick-up zones
- Implement lane switching rules; e.g keep left
- Add road conditions (presence of potholes, slipperiness, bendy roads, among others)