Traffic Simulation

Vinícius Miranda*
College of Computational Sciences, Minerva Schools at KGI

February 16, 2018

Contents

*Replication code for this assignment is available at `https://gist.github.com/viniciusmss/e274117929d3c6b34e52f1d024ea5e3f`. A pdf version of a Jupyter notebook is also appended to this paper.

1 Introduction

Cellular automaton models provide a simple and insightful way of simulating traffic flow. With few rules and short programs, researchers can reproduce realistic phenomena such as backward-flowing congestion waves. I aim to replicate partially two seminal papers describing simulations of single- and double-lane traffic dynamics, respectively. Furthermore, I simulate a triple-lane road and describe whether n-lane roads should increase traffic flow.

This paper is organized as follows. In Section 2, I implement and analyze a single lane, variable speed traffic model based on the original research of Nagel and Schreckenberg (1992). Section 3 provides an extension to a two-lane, symmetric, uni-directional traffic model, partially replicating Rickert et al. (1996). Section 4 presents the results of several multi-lane simulations. Section 5 discusses these results and the shortcoming of the model, and section 6 points at directions for future work.

2 Single-Lane Model

The simulation models a single lane road as a one-dimension list of L sites with periodic boundaries. Sites may be empty or occupied by a car with integer velocity v between zero and v_{max}. Following the notation of Rickert et al. (1996), the index i represents the number of a vehicle, $x(i)$ its position, $v(i)$ its current velocity, $v_d(i)$ its maximum speed, $pred(i)$ the number of the vehicle behind it, and $gap(i) = x(pred(i)) - x(i) - 1$ the width of the gap to the preceding vehicle. Time is modeled as a discrete variable, and three rules apply to the vehicles in parallel at each time step. Finally, vehicles move according to their updated velocities. The update rules are:

1. **IF** $v(i) \neq v_d(i)$ **THEN** $v(i) = v(i) + 1$
2. **IF** $v(i) > gap(i)$ **THEN** $v(i) = gap(i)$
3. **IF** $v(i) > 0$ **AND** $rand < pd(i)$ **THEN** $v(i) = v(i) - 1$

The first rule accelerates a car if it has not reached its maximum velocity. The second rule ensures that collisions will not occur by slowing down if there's an obstacle ahead. The final rule randomly slows down a car to model suboptimal driver behavior, where rand is a pseudo-random number from 0 to 1, exclusive.

I follow the parametrization of Nagel and Schreckenberg (1992) and set $v_{max} = 5$ and $p_d = 0.5$. I analyze the simulations regarding two main variables. The traffic density is defined as

$$\rho = \frac{N}{L}, \tag{1}$$

where N and L refer to the number of cars and sites on the road, respectively. Given that cars do not leave or enter the simulation due to the periodic boundaries, ρ is constant over time. The traffic flow q is both time- and space-averaged and can be calculated as

$$q = \frac{1}{T}\frac{1}{L}\sum_{i}^{L}\sum_{t}^{T} v(i,t). \tag{2}$$

Figures 1 and 2 show the state of the simulation during ten time-steps for a traffic density of 0.03 and 0.10, respectively. Figure 3 shows the key diagram plotting density against traffic flow, showing a critical density point at $\rho = 0.08$. The data collection process worked as follows. For each traffic density in the range $[0.01, 0.02, ..., 0.78, 0.79]$, 10 independent simulations were run on a road of $L = 1000$ sites. For each simulation, 100 steps were executed before data collection started. The flow data was collected for the 1000 ensuing steps.

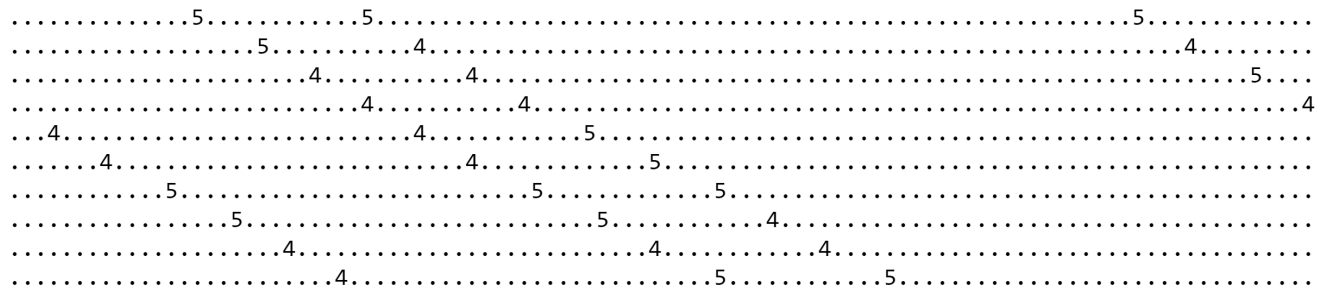


Fig. 1. 10 time-steps of a single-lane simulation. $\rho = 0.03$. Time flows from top to bottom.

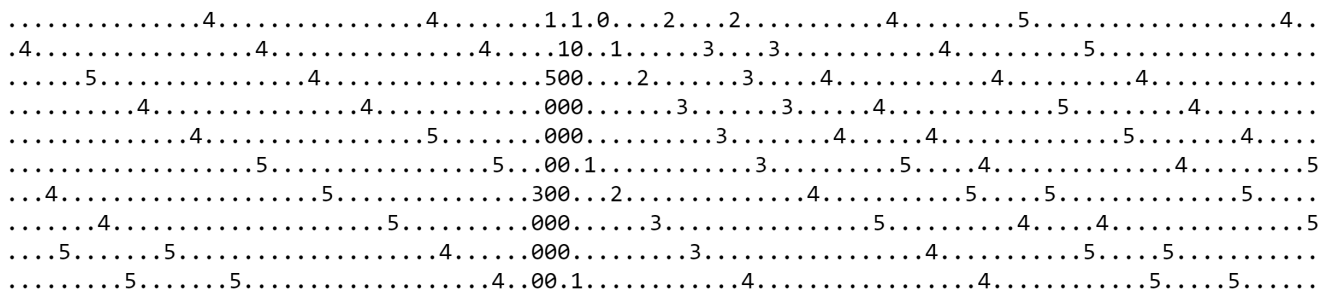


Fig. 2. 10 time-steps showing the spontaneous formation of a traffic jam. $\rho = 0.10$.

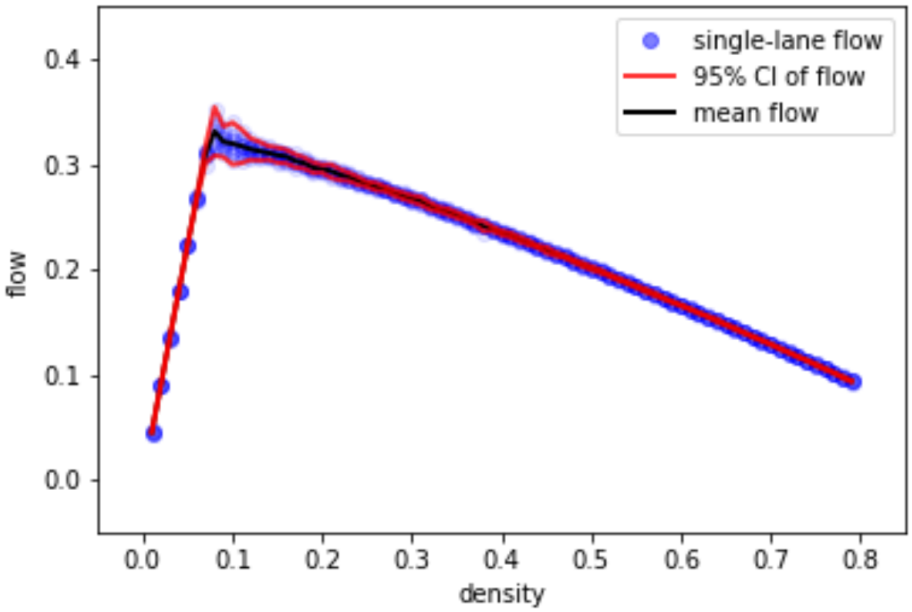


Fig. 3. Density vs. flow plot. Black lines shows the mean flow for 10 independent simulations for a given density. Red lines shows the 95% confidence interval for traffic flow. The critical point is reached at $\rho = 0.08$.

3 Double-Lane Model

Now, the model presented above is extended to support a two-lane road. Several simplifying assumptions are preserved, such as the homogeneity of the road, the equal maximum velocity of all cars, and the standard behavior of all drivers. The two-lane model introduces, however, a new decision problem of when cars should change lanes. The update rules of the model can be thus broken down into two sub-steps. During the first sub-step, cars decide whether or not they will change lanes. If they do, they move sideways to the other lane. In the second sub-step, cars update their velocities according to the rules set in Section 2 based on the resulting configuration of the previous sub-step. Notice that although the sideways motion of changing lanes is unrealistic, a full update step will result in a diagonal movement forward, a more plausible scenario.

There are three critical parameters in the two-lane model. Symmetry defines whether there is a preferential, fast, or default lane or whether cars should consider both lanes equally. The direction of causality establishes whether vehicles look behind them before making a decision. In the single lane model, cars only looked ahead of them, leading causality to move solely backward.

In the two-lane model, cars may consider whether they will block an approaching car on the other lane before making a decision. If they make this consideration, the model enables causality to travel both ways. This parameter represents a way or modeling driver behavior as good drivers will tend to be more considerate of others.

Finally, stochasticity represents the randomness in the simulation. In the single lane model, randomness was critical to model erratic driver behavior that can enable traffic jams on circular roads. In the expanded model, a deterministic update rule for lane change may create synchronized platoons of cars. These cars occupy the same lane and will see the other lane as preferable, leading to a collective lane-changing movement that is continuously reversed until a new vehicle interferes with this pattern. Rickert et al. (1996) modeled stochasticity by introducing a lane change probability similar to the slow down likelihood of Section 2. I depart from their choice and model stochasticity by running the lane-changing sub-step in sequence rather than in parallel. Notice that if a platoon exists, once the first car changes lanes, the destination lane should no longer be desirable to other vehicles. The update sequence is chosen at random, and the velocity update sub-step continues to be executed in parallel.

Therefore, cars change lanes if they satisfy all of the following update rules:

1. $gap(i) < l$,
2. $other\_lane(x(i)) = -1$,
3. $gap_o(i) > l_o$,
4. $gap_{o,back}(i) > l_{o,back}$.

Where $gap(i)$ is the same function described in Section 2, and gap_o and $gap_{o,back}$ return the forward and backward gap on the other lane, respectively. The $other\_lane(x(i))$ function assesses if the position of car i in the other lane is empty. l, l_o, and $l_{o,back}$ are parameters that describe how far a vehicle should look forward in its current lane, ahead on the other lane, and backward on the other lane, respectively. These rules are slightly different from Rickert et al. (1996) as I add the requirement of an empty sideway cell as a separate rule and disregard the probability of lane change they implement as cars' lanes are updated in order.

3.1 Simulation Setup

I simulate multi-lane roads with 1000 sites with periodic boundaries. Cars are initialized at random positions with their velocity equal to zero. The simulation runs for 100 initial runs so take initial random fluctuations die out, after which data is collected for 1000 additional runs. The flow is both space- and time-averaged, collected at every time step as described in Equation 2. This process is repeated ten times at each density point for the graphs of density vs. traffic flow. Finally, the simulation parameters are set as $l = v + 1$, $l_o = l$, $l_{o,back} = v_{max} = 5$, and lane changes are updated in parallel. Figures 4 and 5 show two different types of visualization for the double-lane simulation.

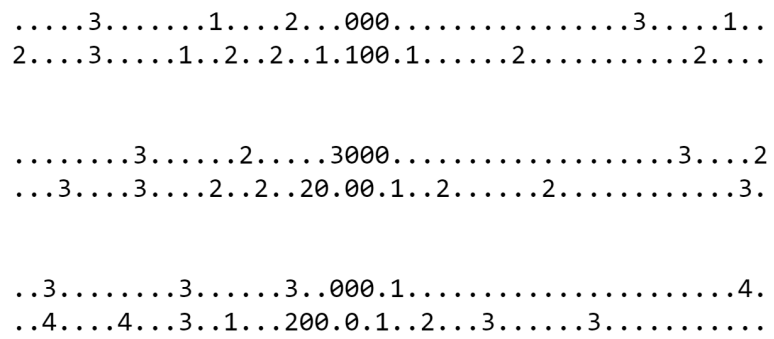


Fig. 4. Three time-steps of a double-lane simulation showing lanes side by side. $\rho = 0.20$. A Traffic jam forms on both sides of the road.

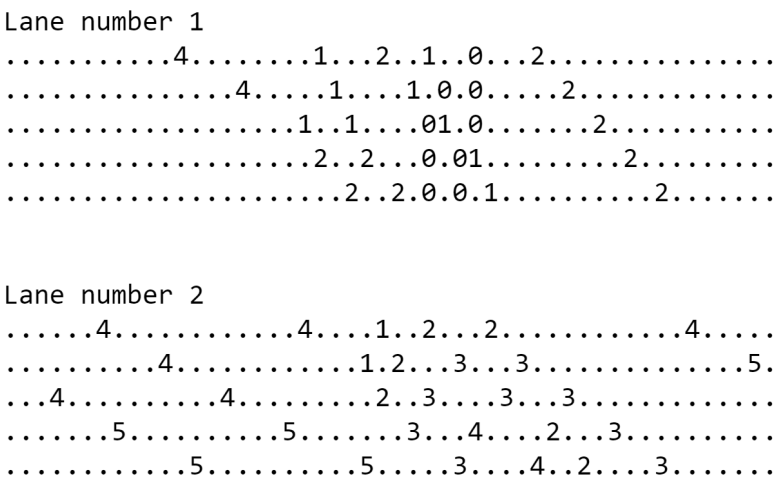


Fig. 5. Alternative visualization of three time-steps of a double-lane simulation. $\rho = 0.12$. Lanes are shown separately and, car velocities represent movement to-be-made rather than just-made.

4 Results

I report the following observations based on the results presented in figures 6, 7, and 8:

(i) The maximum flow of the double-lane road is higher than twice the equivalent for the single-lane road for densities at or past the critical point of $\rho = 0.08$ (Fig. 6). Therefore, the lane changing behavior seems to increase the total capacity of the road.

(ii) Both the single- and double-lane roads have similar flows for densities below their critical point, which is reached at about $\rho \approx 0.08$ (Fig. 6). Below this critical density, flow is laminar, and the lane changing behavior seems to be rarely triggered.

(iii) The difference between the maximum flows of the single- and double-lane roads appears to be of about 0.025 positions per site, per time-step (Fig. 6).

(iv) The triple-lane road shows only a marginal improvement in flow compared with its double-lane equivalent (Fig. 7). It may be the case that after lane changes become available for double-lane roads, additional lanes show a diminishing marginal utility beyond their carrying capacity.

(v) Indeed, for roads with four or more lanes, the flow per lane is the same as for the triple-lane road (Fig. 8). Also, single- and double-lane roads show some variability in their average flow, whereas roads with three or more lanes stability in a maximum flow of about 0.36 positions per site, per time-step.

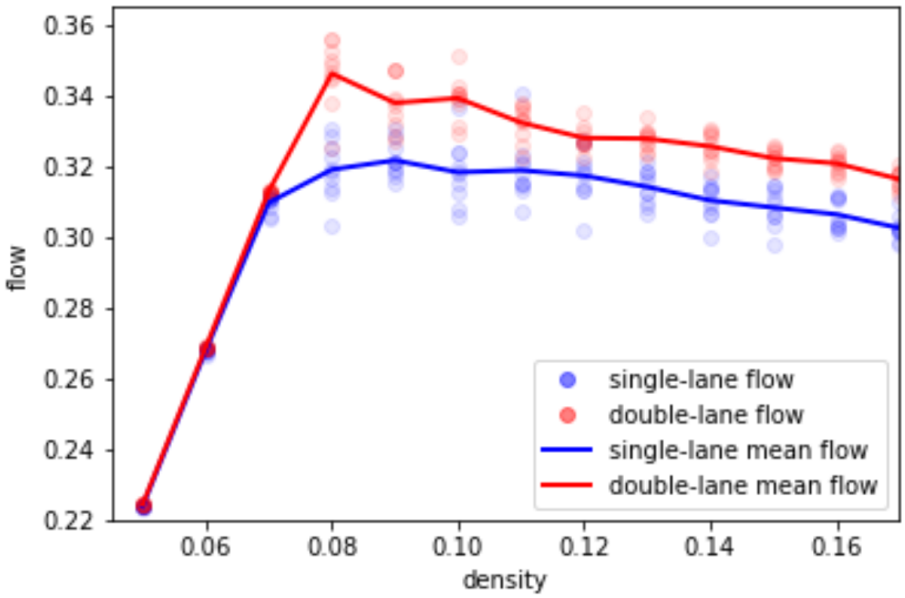


Fig. 6. Density vs. flow plot comparing single- and double-lane roads. Lines represent the average flow over ten independent simulations for a given density.

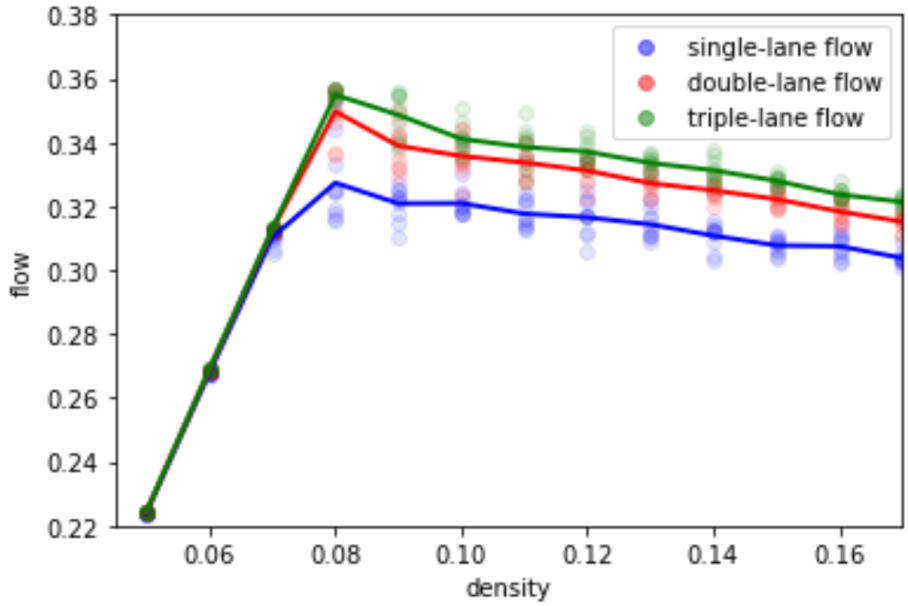


Fig. 7. Density vs. flow plot comparing single-, double-, and triple-lane roads.

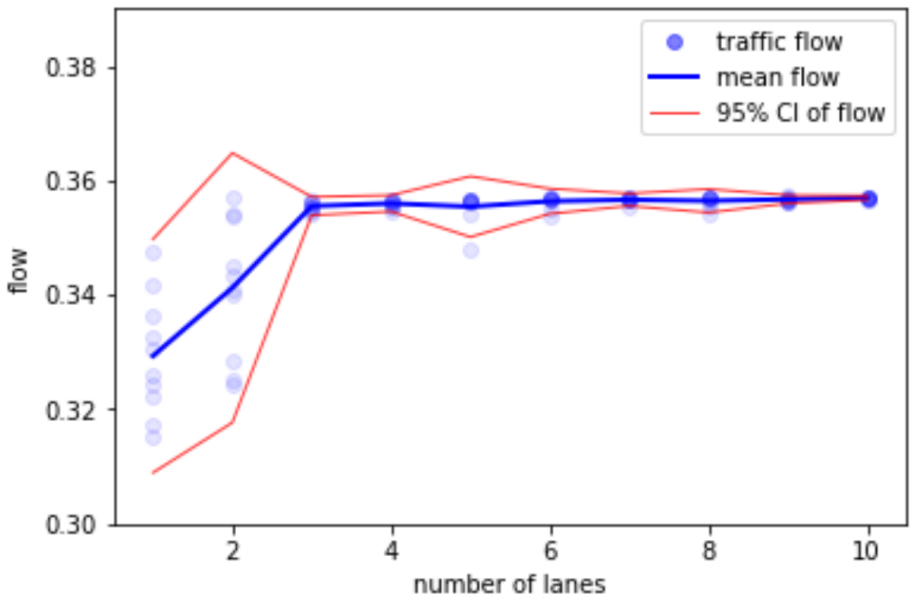


Fig. 8. Number of lanes vs. flow plot comparing different lane sizes for a set density of $\rho = 0.08$. The blue line shows the average flow over ten independent simulations whereas the red lines represent the 95% confidence interval for flow.

5 Discussion

My results are congruent with those presented by Rickert et al. (1996). The implementation of a second lane has a beneficial effect beyond its carrying capacity as cars are able to change lanes when necessary. Furthermore, I extend on their original results to find that this increased utility is marginal for roads with more than three lanes. The reason behind this effect may be that cars in middle lanes have too few opportunities to change their lanes given that they are surround by two lanes. If this explanation is correct, cars in the middle of the road will tend to stick to their path and be unable to escape traffic jams.

I also note that this model is likely not applicable to the traffic in Hyderabad. In Hyderabad, drivers tend to disregard lanes altogether, a cornerstone assumption and structure of the model. Furthermore, the high percentage of motorcycles and three-wheelers disrupts the convention I adopted that a site can only hold one vehicle at a time. Finally, motorcycles can bypass short traffic jams by traveling through the intersections of cars or on sidewalks, a phenomena hard to emulate in this model.

6 Future Work

Two primary directions for future work aim to test different choice of parameters or to model more chaotic traffic systems. Concerning the selection of parameters, researchers may wish to investigate the effect of different maximum velocities, lane asymmetries, and in-order lane updates. In the single-lane model, the modeling of cars with different velocities resulted in the trivial scenario where the flow is bounded by slower cars. In multi-lane models where cars can change lanes, different maximum velocities may produce more interesting effects (e.g., decreasing the point of critical density). Also, future research may introduce intermediary asymmetries that deviate from the model where cars return a default lane when it is available. These intermediary asymmetries may be modeled at the driver-level (e.g., drivers may have a preferred lane) or at the lane-level (e.g., well-kept lanes will tend to be preferred over those with holes). Finally, while I modeled stochasticity through in-order lane updates, all the results presented above model updates in parallel. The reason is that Rickert et al. (1996) model their parameter of lane-change probability when concerned with the number of lane changes, a variable I do not investigate. Researchers may wish to compare the effect of these different methodological choices.

Future research may also focus on modeling more chaotic traffic flows, as seen in Hyderabad. The current model is mostly unsuited for this purpose. A necessary development would be to move away from the assumption of homogeneity of vehicles. Therefore, an object-oriented implementation could model car attributes such as the number of wheels or maximum velocity to better approximate the diversity of vehicles in Hyderabad. Furthermore, the implementation of the road will also gain in complexity to model phenomena such as the percolation of motorcycles through traffic jams and the competition among two- and three-wheelers for spots in the road irrespective of lanes.

Therefore, the cellular automaton model of traffic flow can produce several insights but is restricted to rule-bound and homogeneous systems that are incongruent with the reality of developing countries like India.

References

Nagel, K., & Schreckenberg, M. (1992). A cellular automaton model for freeway traffic. *Journal de Physique I, 2*(12), 2221-2229.

Rickert, M., Nagel, K., Schreckenberg, M., & Latour, A. (1996). Two Lane Traffic Simulations using Cellular Automata. *Physica A: Statistical Mechanics and its Applications, 231*(4), 534-550.

Part 1: Traffic jams on a circular road

We work through this model in class in sessions 4.1 and 4.2. You should use the pre-class work and discussions and problem solving sessions in class to complete most of this part of the assignment. In the next part we extend this model to more realistic situations.

For Part 1, implement the single lane, variable speed traffic model described in this paper: Nagel, K., Schreckenberg, M. (1992). A cellular automaton model for freeway traffic. Journal de Physique I, 2(12), 2221–2229.

- Write well-structured and well-documented Python code to implement the model described in the paper.
- Visualize the state of this model over time, to show how traffic jams can appear when traffic density is high enough.
- Analyze how the overall average traffic flow rate varies with traffic density and present your results in your report.

In [264]:

```
import pylab as py
import scipy
import random
import matplotlib
from matplotlib import pyplot as plt
%matplotlib inline

class TrafficSimulation:

    def __init__(self, road_length=100, traffic_density=0.1,
maximum_velocity=5,
                 slow_probability=0.5, parallel_lane_update=True,lane_chang
_prob=1, n_lanes=1):
        '''
        Creates a new traffic simulation object. Cars are distributed rando
mly
        along the road and start with velocity equal to zero.

        Inputs:

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

            traffic_density (float) The fraction of cells that have a car o
n them.
              Default: 0.1.

            maximum_velocity (int) The maximum speed in car cells per updat
e step.
              Default: 5.

            slow_probability (float) The probability that a car will random
ly
              slow down by 1 during an update step. Default: 0.5.
```

```
            parallel_lane_update (bool) Whether the lane-changing step shou
ld be
              executed in parallel. The lane_change_prob input can only be
1
              if parallel_change is set to false. Default: True.

            lane_change_prob (float) The probability that a car will change
              lanes when all other conditions are satisfied. Default: 1.

            n_lanes (int) The number of lanes in the simulation. For n_lane
s > 1,
              lane changing behavior is enabled automatically. Default: 1.

        '''

        self.length = road_length
        self.density = traffic_density
        self.max = maximum_velocity
        self.slow_prob = slow_probability
        self.parallel_lane_update = parallel_lane_update
        self.lane_change_prob = lane_change_prob
        self.n_lanes = n_lanes
        self.time_step = 0

        self.traffic_flow = [] # Flow tracker
        self.current_state = -scipy.ones((self.n_lanes, self.length), dtype
=int)

        # For each lane, create cars with velocity equal to 0.
        for this_lane in range(self.n_lanes):
            random_indices = scipy.random.choice(
                range(self.length),
                size=int(round(self.density * self.length)),
                replace=False)
            self.current_state[this_lane][random_indices] = 0

    def get_gap(self, lane, cell, front=True):
        '''
        Calculates the gap between a given cell and a neighboring car.
        The gap is 0 if there is a car in the immediate neihborhood.

        Inputs:

          lane (int) The lane in which the gap should be calculated.

          cell (int) The cell relative to which  the gap should be calculat
ed.

          front (bool) Whether the gap should be calculated looking in
            front of the car (cell). If false, the gap is calculated
            looking back. Default: True.

        '''

        gap = 0

        # gap ahead
        while front and self.current_state[lane][(cell + gap + 1) % self.len
gth] == -1:
            gap += 1
```

```
        # gap behind
        while not front and self.current_state[lane][(cell + gap - 1) % self
.length] == -1:
            gap -= 1

        if gap < 0:
            return -gap
        else:
            return gap

    def change_lanes(self):
        '''
        Assesses whether cars should lanes. It evalutes several conditions
        which should all be satisfied before a car changes lanes. Cars may
        change lanes in parallel or in random order. Notice that if cars
        are update in parallel, the order has no effect. The handling
        of lane change for multiple-lane roads is executed from the
        left-most to the right-most lane.
        '''

        if not self.parallel_lane_update:
            if self.lane_change_prob < 1:
                self.lane_change_prob = 1
                print "When cars change lanes in order, they will always
change lanes when possible."
                print "Lane change probability reset to 1."

        lane_changing_state = -scipy.ones((self.n_lanes, self.length),
dtype=int)

        # Array storing all lane combinations that allow cars to change lan
es.
        pairs = [[lane, lane+1] for lane in range(self.n_lanes-1)]

        for lane_pair in pairs: # For each legal lane pair

            # Decide the order in which cars assess whether to change
lanes.
            lane_changing_order = scipy.random.choice(
                    range(self.length),
                    size=int(self.length),
                    replace=False)

            # For nth iterations (n > 1), guarantee that the temporary arra
y
            # is empty so cars will not be replicated.
            lane_changing_state[min(lane_pair)] =
-scipy.ones((self.length), dtype=int)

            for lane in lane_pair:                              # For each lan
                for cell in lane_changing_order:                # For each cel
                    if self.current_state[lane][cell] > -1:  # If this cell
is a car

                        # Is there a car ahead? i.e., gap(i) < l = v + 1?
                        this_lane_gap = self.get_gap(lane, cell)
                        is_car_ahead = this_lane_gap < self.current_state[la
ne][cell] + 1

                        # Is the same position at the other lane empty?
```

```
                        # Is the same position at the other lane empty?
                        is_other_lane_empty = self.current_state[(max(lane_p
air) - lane) + min(lane_pair)][cell] == -1

                        # Is the other lane free looking ahead? i.e., gap_o
i) > l_o = l?
                        other_lane_gap = \
                        self.get_gap((max(lane_pair) - lane) + min(lane_pai
), cell)
                        is_other_lane_free_ahead = other_lane_gap > self.cu
rent_state[lane][cell] + 1

                        # Is the other lane free loking behind? i.e., gap_b
ck(i) > l_back = v_max?
                        other_lane_gap_behind = \
                        self.get_gap((max(lane_pair) - lane) + min(lane_pai
), cell, front=False)
                        is_other_lane_free_behind = other_lane_gap_behind >
self.max

                        # Should the car change lanes if possible?
                        can_change = random.random() < self.lane_change_prob

                        # Flag if change is possible
                        change_flag = is_car_ahead and is_other_lane_empty a
nd \
                        is_other_lane_free_ahead and
is_other_lane_free_behind and can_change

                        if change_flag:

                            # Car changes position to the other lane in the
pair.
                            lane_changing_state[(max(lane_pair) - lane) + mi
n(lane_pair)][cell] = \
                            self.current_state[lane][cell]

                            # If in-order update, erase the car.
                            if not self.parallel_lane_update:
                                self.current_state[lane][cell] = -1
                        else:
                            lane_changing_state[lane][cell] =
self.current_state[lane][cell]

            # After each lane pair, transcribe cars from the temporary
            # to the attribute array.
            for lane in lane_pair:
                self.current_state[lane] = lane_changing_state[lane]

    def update_velocity(self):
        '''
        Update velocities of all cars. Assesses whether cars should acceler
ate,
        slow down due to another car, or slow down randomly.
        '''

        update_velocity_state = -scipy.ones((self.n_lanes, self.length), dt
ype=int)

        for lane in range(self.n_lanes):
            for cell in range(self.length):
```

```
            if self.current_state[lane][cell] > -1:

                # Calculate distance to the next car
                gap = self.get_gap(lane, cell)

                # Acceleration
                if self.current_state[lane][cell] < self.max:
                    self.current_state[lane][cell] += 1

                # Slowing Down
                if self.current_state[lane][cell] > gap:
                    self.current_state[lane][cell] = gap

                # Randomization
                if self.current_state[lane][cell] > 0 and random.random(
) < self.slow_prob:
                    self.current_state[lane][cell] -= 1

    def move(self):
        '''
        Moves all cars according to their velocities.
        '''

        next_state = -scipy.ones((self.n_lanes, self.length), dtype=int)
        for lane in range(self.n_lanes):
            for cell in range(self.length):
                if self.current_state[lane][cell] > -1:
                    next_state[lane][(cell + self.current_state[lane][cell]
% self.length] = self.current_state[lane][cell]
        self.current_state = next_state

    def update(self, measure_flow=True):
        '''
        Updates the state of the simulation by checking whether cars
        should change lanes and then updates their velocities. Cars move
        according to their updated velocities. Finally, measures the
        time and space averaged flow at the given step.

        Inputs:

          measure_flow (bool) Whether the flow should be measured.
            This paratemer serves for internal control and cannot
            be changed by the user. On the first 100 iterations of
            the simulation, the flow is not measured. Defaults: True.

        '''
        if self.n_lanes > 1: self.change_lanes()
        self.update_velocity()
        self.move()

        if measure_flow:
            self.time_step += 1
            total_movement = sum([car if car > 0 else 0 for lane in self.cur
rent_state for car in lane])
            self.traffic_flow.append(total_movement/float(self.n_lanes*self
.length))

    def draw(self):
        '''
        Draws the current state of the simulation.
```

```
        '''
        for lane in self.current_state:
            print(''.join('.' if car == -1 else str(car) for car in lane))
        if self.n_lanes > 1: print "\n"

    def graph_separate_lane(self, steps):
        '''
        Draws lanes separately after the end of a simulation.

        Inputs:

          steps (int) How many steps should be shown in the
            graph. The steps shown are always the last ones.
            Default: 10.

        '''

        for lane in range(self.n_lanes):
            print "Lane number %d" % (lane+1)
            for step in range(-steps, 0):
                print(''.join('.' if car == -1 else str(car) for car in self
.lane_history[lane][step]))
            print "\n"

    def store_lane_state(self):
        '''
        Stores the state of each lane's history. This method
        enables the drawing of lanes separately.
        '''
        for lane in range(self.n_lanes):
            self.lane_history[lane].append(self.current_state[lane])

    def simulate(self, burns = 100, iterations=1000, draw=True,
draw_separate_lanes=False, steps=10):
        '''
        Executes the simulation for a given number of steps.

        Inputs:

          burns (int) The initial amount of time steps executed
            before flow measurement starts. Default: 100.

          iterations (int) The number of times steps that should
            be executed after 100 initial steps. Default: 1000.

          draw (bool) Whether the simulation should be shown
            at each time step. Default: True.

          draw_separate_lanes (bool) Whether different lanes should
            be drawn separately. If false, the figure will show
            both lanes at each step. If true, n figures will be
            generated, showing n lanes separately. Default: False.

          steps (int) How many steps should be shown in the
            graph for separate lanes. The argument is ignored if
            draw_separate_lanes is false. Steps shown are always
            the last ones. Default: 10.

        '''
```

```
        # Burn initial iterations:
        for i in range(burns):
            self.update(measure_flow=False)

        # To draw lanes side by side.
        if not draw_separate_lanes:
            for i in range(iterations):
                self.update()
                if draw: self.draw()

        # To draw lanes separately.
        else:
            self.lane_history = [[] for lane in range(self.n_lanes)]
            for i in range(iterations):
                self.update()
                self.store_lane_state()
            if steps > iterations:
                print "The simulation cannot graph more steps than
iterations."
                if iterations < 10:
                    print "Setting steps to: %d." % (iterations)
                    steps = iterations
                else:
                    print "Setting steps to default: 10."
                    steps = 10
            if draw: self.graph_separate_lane(steps)
```

In [3]:

```
# Figure 1
print "Figure 1. Traffic density set to 0.03.\n"
sim = TrafficSimulation(traffic_density=0.03, n_lanes = 1)
sim.simulate(iterations=10)

# Figure 2
print "\nFigure 2. Traffic density set to 0.10.\n"
sim = TrafficSimulation(traffic_density=0.1, n_lanes = 1)
sim.simulate(iterations=10)
```

Figure 1. Traffic density set to 0.03.

```
......5.................................................4.................
.................5..
.4........4.....................................................5.............
.....................
......5........5.....................................................5.........
.....................
...........5.......4.....................................................4.....
.....................
...............4........5.....................................................4.
.....................
...................4.........5.....................................................
.....................
.........................5.........5.....................................
...5.................
.............................4.........4.....................................
.......4.............
.................................5........4.................................
...........4.........
```

```
.....................................4.........5.............................
.................5....
```

Figure 2. Traffic density set to 0.10.

```
...............0.00......2........4...0..1.....3............4..............
.....................
.................100.........3.......2..1..1.......3.............4..........
...5..................
.................00.1............4....1...2..2.........4.............4......
.......4.............
.................00..1..............3..1...1...2............5.............5.
...........4.........
.................0.1...2.............1...2...2....3..............5..........
.................5....
5.................1.1....2............1....2....3.....4................5.....
..4...................
.....5.............1..2.....3...........2.....3...3.......5................5
.......5.............
.........4..........2..2......3...........3......4....4........5...........
...........4.........
..............4.......1...2........4...........4.....3......5.......4.......
....5...........5....
5.................4....1.....3..........5..........4.....4.......5......4...
.........4............
```

In [8]:

```
import numpy as np
import pandas as pd

def graph(numRuns = 10, densities=[.01*x for x in range(1,80)], lane_sizes=[
1,2],
          iterations=1000, road_length=1000, xlim=[-.05,0.85], ylim=[-.05,0
45]):
    '''
    Generates density vs. flow plots according to the given parameters.

    Inputs:

        numRuns (int) Number of independent simulations for each density.
          Default: 10.

        densities (list) An array containing all the densities at each
          different roads should be compared. Default: [0.01, 0.02, ..., 0.
79]

        lane_sizes (list) An array containing the number of lanes in the
          roads to be compared. Default: [1,2]

        iterations (int) The number of time steps in a given simulation for
          which flow data should be collected.

        road_length (int) How many sites each lane of the simulated roads
          should contain.

        xlim, ylim (list, list) Two lists of two values definied the limits
          of the x and y axes in the graph to be generated.

    '''
```

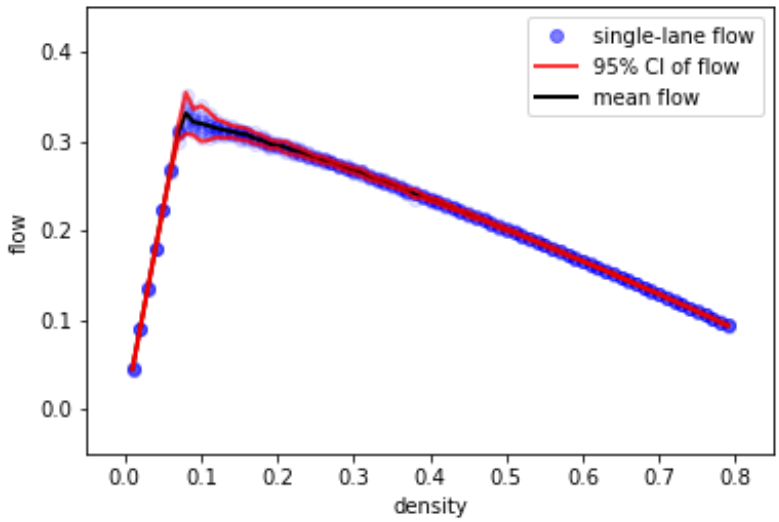
```
    flow_rates = [pd.DataFrame(index=range(numRuns),columns = densities) fo
r _ in lane_sizes]
    colors = {1:'b', 2:'r', 3:'g'}
    descriptions = {1: 'single-lane flow', 2: 'double-lane flow', 3: 'triple
-lane flow'}

    # Simulate higher traffic density to see traffic jams forming
    for index in range(len(lane_sizes)):
        for density in densities: # For each density
            temp_flows = []
            for run in range(numRuns): # For each simulation
                sim = TrafficSimulation(traffic_density = density,
                                        road_length = road_length, n_lanes =
lane_sizes[index])

                # Simulation runs for 200 iterations with 100 initial
burns.
                sim.simulate(draw=False, iterations=iterations)
                temp_flows.append(np.mean(sim.traffic_flow))

            flow_rates[index][density] = temp_flows

    flow_metrics = [flow_rates[i].describe() for i in
range(len(flow_rates))]
    flow_means = [flow_metrics[i].loc['mean'].values for i in range(len(flow
_rates))]
    if len(lane_sizes)==1:
        flow_upper = [flow_means[i] + 1.96*flow_metrics[i].loc['std'].values
for i in range(len(flow_rates))]
        flow_lower = [flow_means[i] - 1.96*flow_metrics[i].loc['std'].values
for i in range(len(flow_rates))]

    plt.figure()
    toys, means = [], []
    for lane in lane_sizes:
        for density in densities:
            plt.scatter([density for i in range(numRuns)], flow_rates[lane_s
izes.index(lane)][density],
                        c=colors[lane],edgecolor=colors[lane],alpha=.1)
        temp_toy, = plt.plot([],[], color=colors[lane], marker='o', label=de
scriptions[lane], alpha=.5, linestyle="")
        toys.append(temp_toy)
        if len(lane_sizes) == 1:
            temp_mean, =
plt.plot(densities,flow_means[lane_sizes.index(lane)],
                                    c='k',lw=2, label="mean flow")
            CI, = plt.plot(densities,flow_upper[lane_sizes.index(lane)],
                           c='r', lw=2,alpha = .8, label="95% CI of flow")
            plt.plot(densities,flow_lower[lane_sizes.index(lane)],
                     c='r', lw=2, alpha = .8)
            means.append(CI)
        else:
            temp_mean, =
plt.plot(densities,flow_means[lane_sizes.index(lane)],
                                    c=colors[lane],lw=2, label=descriptions[la
e][:-4] + "mean flow")
        means.append(temp_mean)
    if len(toys) < 3:
        handles = toys + means
```

```
    else:
        handles = toys
    plt.legend(handles = handles,facecolor="white")
    plt.xlabel("density")
    plt.ylabel("flow")
    plt.xlim(*xlim)
    plt.ylim(*ylim)
    plt.show()
```

In [9]:

```
print "Figure 3. Traffic density vs. traffic flow.\n"
graph(lane_sizes=[1])
```

```
Figure 3. Traffic density vs. traffic flow.
```



Part 2: Multi-lane highways

Implement the 2-lane, symmetric, uni-directional, variable speed model in this paper: Rickert, M., et al. (1996). Two Lane Traffic Simulations using Cellular Automata. Physica A: Statistical Mechanics and its Applications, 231(4), 534–550.

We start looking at this model in class and you will have an opportunity there to share ideas and discuss problems you encounter. You will need complete this part of the assignment on your own though.

- A description, in your report, of how this model works. What are the assumptions, parameters, and update rules of the model? Do not just copy and paste from the paper. Explain the cellular automaton in your own words and as clearly as possible.
- Write well-structured and well-documented Python code to implement the model described in the paper.
- Visualize the state of this model over time, to show the typical traffic patterns that can emerge. Your results will again depend on traffic density.
- Analyze how much more traffic can flow through a multi-lane road, compare to a single lane road, at the same traffic density. Present your results in your report.

In [103]:

```
print "Figure 4. Two lane model with traffic density set to 0.2 \n"
```

```
print "Figure 4. Two-lane model with traffic density set to 0.2.\n"
sim = TrafficSimulation(road_length = 50, traffic_density= 0.2, n_lanes = 2
)
sim.simulate(iterations=5, burns = 10)
```

Figure 4. Two-lane model with traffic density set to 0.2.

```
.....3.......1....2...000.................3.....1..
2....3.....1..2..2..1.100.1......2..........2....


........3......2.....3000...................3....2
...3....3....2..2..20.00.1..2......2............3.


..3........3......3..000.1.....................4.
..4....4...3..1...200.0.1..2...3......3..........


2.....4........4...1.00.1.1.....................
.....3...2...2..2.000..1..2..2.....4.....3........


..2.......4......2.0.00..1.1....................
........3..2...20.000...1..1...2........5....4....
```

In [265]:

```
print "Figure 5. Two-lane model with traffic density set to 0.12.\n"
sim = TrafficSimulation(road_length = 50, traffic_density=0.12, n_lanes = 2
)
sim.simulate(iterations=5, draw_separate_lanes=True, steps=10)
```

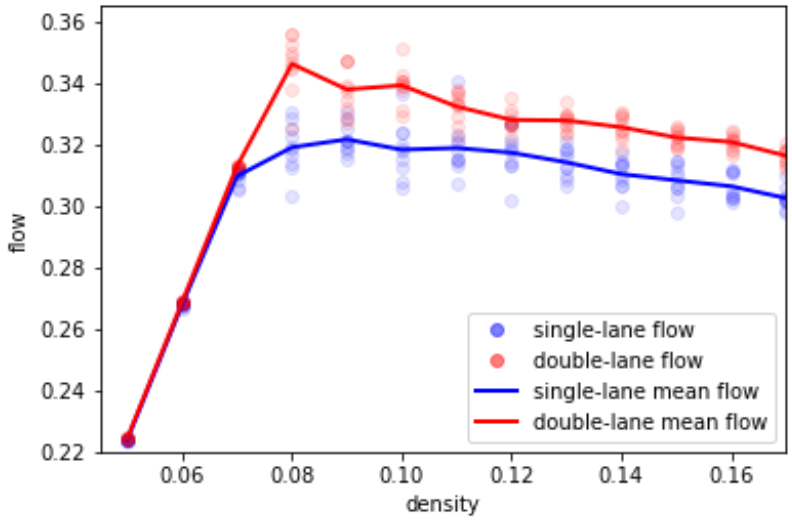
Figure 5. Two-lane model with traffic density set to 0.15.

```
The simulation cannot graph more steps than iterations.
Setting steps to: 5.
Lane number 1
...........4........1...2..1..0...2.............
..............4.....1....1.0.0.....2............
...................1..1....01.0.......2..........
...................2..2...0.01.........2........
.....................2..2.0.0.1..........2.......


Lane number 2
......4............4....1..2...2............4.....
..........4............1.2...3...3.............5.
...4..........4.........2..3....3...3............
.......5..........5.......3...4....2...3.........
............5..........5.....3....4..2....3.......
```

In [15]:

```
print "Figure 6. Single-lane vs. multi-lane.\n"
graph(lane_sizes=[1,2], densities=[.01*x for x in range(5,18)],xlim=[.045,0.
17], ylim=[0.22,0.365])
```

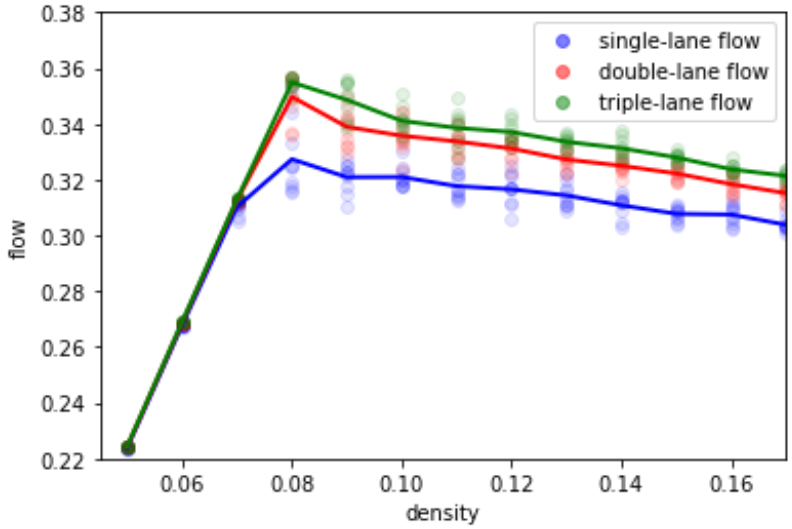
Figure 6. Single-lane vs. multi-lane.



In [19]:

```
print "Figure 7. Single-lane vs. double-lane vs. triple-lane.\n"
graph(lane_sizes=[1,2,3], densities=[.01*x for x in range(5,18)], xlim=[.045
,0.17], ylim=[0.22,0.38])
```

Figure 7. Single-lane vs. double-lane vs. triple-lane.



In [21]:

```
print "Figure 8. Traffic flow for n-lane road."

numRuns = 10
road_sizes = range(1, 11)
flow_rates = pd.DataFrame(index=range(numRuns),columns = road_sizes)
flow_metrics, flow_means = {}, {}

for size in road_sizes: # For each road
    temp_flow = []
    for run in range(numRuns): # For each simulation
        sim = TrafficSimulation(traffic_density=0.08, road_length = 1000,
n_lanes=size)
        sim.simulate(draw=False, iterations=1000)
        temp_flow.append(np.mean(sim.traffic_flow))
```

```
        temp_flow.append(np.mean(sim.traffic_flow))

    flow_rates[size] = temp_flow


flow_metrics = flow_rates.describe()
flow_means = flow_metrics.loc['mean'].values
flow_upper = flow_means + 1.96*flow_metrics.loc['std'].values
flow_lower = flow_means - 1.96*flow_metrics.loc['std'].values

plt.figure()
for size in road_sizes:
    plt.scatter([size for i in range(numRuns)], flow_rates[size],c='b',edge
color='b',alpha=.1)

blue_toy, = plt.plot([],[], color='b', marker='o', label='traffic flow',
alpha=.5, linestyle="")
means, = plt.plot(road_sizes,flow_means,c='b',lw=2, label="mean flow")
CI, = plt.plot(road_sizes,flow_upper, c='r', lw=1,alpha = .8, label="95% CI
of flow")
plt.plot(road_sizes,flow_lower, c='r', lw=1, alpha = .8)
plt.legend(handles=[blue_toy, means, CI],facecolor="white")
plt.xlabel("number of lanes")
plt.ylabel("flow")
plt.ylim(.30, .39)
plt.xlim(0.5, 10.5)
plt.show()
```

Figure 8. Traffic flow for n-lane road.