22.107 Term Paper
Traffic-Simulation

Ittinop(Pun) Dumnernchanvanit

Introduction:

Traffic flow, in mathematics and civil engineering is defined as the study of interactions between vehicles, drivers and infrastructures. In a network with no congestion, traffic stream depends on speed and concentration of vehicles. As density of cars on road reaches maximized flow rate and starts to exceed optimal density, the flow of traffic turns unstable and minor perturbation such as minor accident can results in congestion. Calculations for congested networks are far more complex than that of free-flow networks, therefore, requires more reliance on empirical studies and extrapolation of data.

Traffic phenomena are usually complex and nonlinear making it quite difficult to characterize accurately with specific equations. Therefore, it is advantageous to use microscopic traffic model which simulates single vehicle units. In contrast to macroscopic flow model that stores macro dynamic variables such as density, flow, mean speed, microscopic flow model stores microscopic properties such as position, direction, speed etc. This project's goal is to build a microscopic traffic simulation that can be used to aid us in understanding traffic flow, help find an optimal way to efficiently control traffic movement or give results that can be used for transportation forecasting. More specific examples problems that we can use this simulation to tackle includes problems such as, determining most efficient red-light/green-light pattern, determining how far from a bottle neck to place a warning sign, or determining whether all-way stop sign or red-light/green-light is suited for specific road system.

Background:

History

History of traffic flow theory dated back as far as 1920, Frank Knight produced an analysis of traffic equilibrium in order to support his argument for toll road. He argued that privately owned toll roads would set its tolls to reduce congestion to its efficient level (Now the theory in the analysis he proposed is known as Wardrop's Principle since it was later refined by Wardrop)

Almost a century has passed since, but there is still no consistent general theory that can be repeatedly applied for different real traffic flow systems. However, with the rising computer processing power, it is now possible to build microscopic traffic simulation that in the future, could be the solution to this unsolved problem.

Shockwave:

Shockwave is a traveling disturbance in the distribution of cars on roads or highways. Although it usually appears to be upstream in motion, it can also travels downstream or even stay as a stationary wave.

Platoon:

It was suggested that grouping vehicle into platoons can help increase the capacity of roads and highways. Grouping vehicles into platoons can potentially decrease distances between cars by using artificial intelligence. The artificial intelligence in all cars will be synchronized allowing many cars to accelerate or brake synchronously. Usually drivers will leave space in front of him to account for the delay in his reaction time after seeing the front car brake. However, if all cars move synchronously, there would be no need for such extra space. Hence the cars can be packed close together to increase road capacity.

Of course, this idea is still very far from actualization, and would need smart cars with artificial intelligence, capable of driving on its own, equipped. Nevertheless, it is quite interesting to see it simulated which is why it was added to the project.

Phenomena simulated in this project:

-Shockwave:

Simple shockwave on road was demonstrated in the video attached. The shockwave tally video is also attached. Gaussian car representation for tally had been inserted into the code to make the graphs look better.

-Platoon:

Platoon (defined previously) is simulated in this project in order to give a clear picture of how implemented platoon might look and to determine how much more effective could it be compare to non-platoon system.

-Road Block:

In this project we will also model road block and examine how placing warning sign before road block affects traffic flow through it.

-Cutting in Front:

The behavior of drivers cutting in front in highway is also modeled. For these cases, driver looks to another lane to decide whether it is worth to cut in front judging from distance and velocity of the front car in his target lane. Then he looks back to see if the car in the back in that lane is far back enough for him to cut in front without crashing. (For more detailed explanation, see Code Explanation section, Car Class, React() function)

-Intersections:

Intersection will also be simulated in this project. We will use the intersection to obtain data comparing stop sign and red light in different situations and determine the effectiveness of each.

Note that there is one significant simplification that has been made in creating intersection in this project which is that there is no left turn. It was neglected because it would add quite a lot of unnecessary complexity which is not preferred given the time constrain.

Left turn could have been simulated without clogging the problem by building another lane to handle the vehicles turning left and opening the green light only for them at some point. Alternatively, the vehicles that are waiting to turn left when there is no car during straight green light can be modeled. This can be done by having the driver check if the area of intersection and area leading up to intersection is clear of any incoming cars and proceed in turning when that happens.

Assumptions:

This model explicitly assumes that drivers do not look at their rearview mirrors except when they are trying to change lane. In other words, drivers do not look at their rearview mirrors when adjusting his car's speed. This should reflects real world since majority of drivers will always look front most of the time and will slow down or speed up depending on what the front car is doing.

It was assumed that the driver adjusts his speed according to his car's distance to front car. The formula used for speed is the one that had been described in class where speed depends on density:

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

In order to relate the density to distance we assumed that density is:

$$\rho \approx \frac{1}{d} \qquad \rho_{max} \approx \frac{1}{d_{min}}$$

In the figure below it is intuitive that the density would be the total number of cars divided by distance D. This ratio should also be approximately equals to the small 1/d, which is how we arrived at the above approximation.

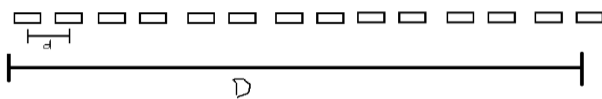

Figure 1: simple figure used to demonstrate the 1/d approximation.

For acceleration, the following formula is used:

$$\max acceleration \cdot \frac{(current\,desired\,speed - speed)}{(\max speed - speed)}$$

Angular movement variable is used in move() function to turn the car. In this code, the angular_movement variable represents the change in angle in one time step. Turn radius is used to simplify turning of the car so we will not have to worry about turning speed. (If we turn with fixed angular speed, slow vehicle will turn around with very little radius and fast vehicle will turn around with huge radius; this will not be consistent with what we want)

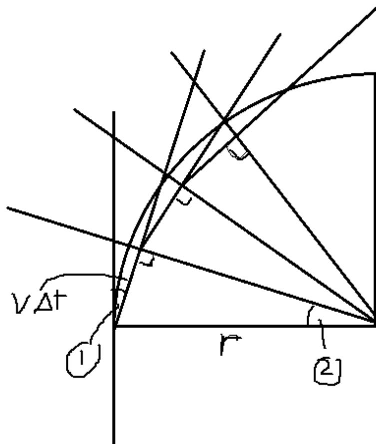

Figure 2: Figure representing how a vehicle will move during time steps in a turn. The vehicle will come in at the part of quarter circle close to angle (1).

From the figure above, we need to find the angle (1) which will be the amount of angular movement in one time step. From geometry, angle (1) should be equal to angle (2). Angle two can be found by calculating arcsin of v*Δt / r. "v*Δt" is just speed multiplied by amount of time in each time step, or in other words, change in position in each time step. So angular movement can be obtained from turn radius and speed with the following formula:

$$angular\,movement\;\;during\,\Delta t = \arcsin\left(\frac{speed \cdot \Delta t}{turn\,radius}\right)$$

In an intersection, turn radius is calculated by subtracting the distance in x or y from current land end to another lane's position. If lane is going in x direction and another lane is going in y direction, then radius would be position of lane y (which is in x-coordinate) minus the current lane end and vice versa for y direction.

After we have finished running, the C++ core code outputs a file that contains information regarding each car's position and direction etc. After placing the car at a position for display we have to turn it to the direction we want. This is done using rotational matrix:

$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

Since this rotational matrix will only works around the (0,0) coordinate, we first move the car so its center align with the (0,0) coordinate and then apply rotational matrix to rotate the car. After that, we bring it back to old position and display it.

Code Explanation:

The code for this simulation is divided into three main classes and main() function that uses objects created using these three classes to build the system that we want to simulate.

Note that straightforward things without ambiguous name are excluded. Ambiguous and potentially confusing variables will be explained in more details than the comments in the code.

Code index (PDF):

Car_code.pdf: Car.h, Car class

Creator_code.pdf: Creator.h, Creator class

Lane_code.pdf: Lane.h, Lane class

Main_*: main function for different systems according to the name

Matlab_car: Build movie of traffic flow from text file created by c++ program. The one shown is for stop sign but it could easily be used for other cases.

Matlab_tally: Build movie of graph of car density versus position for making shockwave graph movie.

Car class:

Car object contains information regarding each car on the road.

Variables:

x-position, y-position, speed, and acceleration (acc) are as the name implies.

start_timer :

This variable keeps the starting time (time in model) right when the car enters the system. This will be used to subtract from the time at which this car leaves the system to get the average final time.

number_of_cuts:

This variable keeps track of the number of times that this car cut in front of other cars.

try_change_left:

This Boolean variable is included for system of road block where we compare road block with warning sign and no warning sign. The reason this variable is separate from variable changing_lane_left is because when changing lane left becomes true, the car will try to push itself into the left lane right away even if it has to slow down or stop. However, for the case where there is warning sign before road block, the car might not want to push into the adjacent lane right away. Because of this, we put in another Boolean variable try_change_left to represent just this exact state of mind of the driver. When try_change_left is turned on, the driver will keep looking furiously to its adjacent lane without lowering its speed and push in if it can (if push in is decided, the driver's change_lane_left becomes true).

platoon_mode:

"platoon_mode" is a Boolean variable that allows turning on and off of platoon mode.

direction:

The angle representing the direction the car is facing.

angular_movement:

This is the change in direction angle on car in one time step.

turn radius:

This is the radius at which the car will circle around when turning, regardless of its speed.

alive:

This Boolean allow the main code to recognize that the car already exited the system and should be removed and tallied accordingly.

max_acceleration, max_brake

These two variables put cap on the maximum acceleration and deceleration the car can perform. Maximum acceleration in real world should be the limitation of the car regarding how powerful the engine is (and driver's personality).

desired_speed

This variable has a name that might be interpreted otherwise, but it is the speed that the driver would like to maintain when there is no car in front.

current_desired_speed

This variable is the speed that the driver wants to be at after he has taken into account his surrounding such as the distance to car in front.

lane, targetlane

Lane is a pointer to lane object that the car is running in and targetlane is a pointer to the lane object that the car might want to go to.

min_d_to_frontcar

This variable represents minimum distance to frontcar that the car will tolerate

d_to_frontcar:

The current distance to front car in current lane

d_to_frontcar_targetlane

Potential frontcar, other car with targetlane in current lane

d_to_backcar

Current distance to back car in current lane

d_to_backcar_targetlane

Current distance to back car in target lane

d_to_backcar_potential;

Potential backcar, other car with targetlane in current lane

lanechange_turn_radius

Turn radius this car will use when changing lane. (Explicitly used for this)

lane_end_adjustment

Boolean to show if angular speed calculation at the end of lane has been done or not

vector<Lane*> waypoints

The vector storing pointers to lanes that the car object should enters after reaching the end of current lane.

vector<int> endlane_transition

This is coupled to waypoints vector. It determines the type of transition the car will do when going to the next lane specified in waypoints vector. Types of transition include straight (0), left turn (1), right turn (2), and exit system (8). Because exit transition is also specified, this vector should be one element longer than waypoints vector.

changing_lane_right/left

When this Boolean is true, the car will try to change lane right/left right away and will slow down to change the lane if necessary. This Boolean will remains true throughout the lane change process.

Methods:

placeCar():

After the car has been created by the constructor, we have to specify its parameters and place it into the system. This placement job is exactly what this method is built for. There is also

another placeCar() method which extends the original method but take in extra platoon_mode Boolean, which is used in platoon case.

Move():

This function basically moves the car according to the acceleration and angular movement stored inside car object.

getFrontCar():

This is a function that will return the car in front of the current car when given car array. The function parameter, "lane_or_targetlane", let the function user choose if he/she wants to look in the current car's lane or its targetlane. "lane_or_targetlane_others" let user choose if he/she want to have the current car's driver look at other cars' actual lane, or the lane other cars are aiming for or targetlane. "lane_or_targetlane_others" becomes useful when the car has to check for car cutting in front of it to prevent crash. And similar to any Boolean variable names in this code, the front part of name before or represent the case when the Boolean variable is true. getBackCar() is basically the same type of function except that it gets the back car instead. getBackMostCar() is also similar, except that it gets the back most car in the lane instead, which is useful for case with periodic boundary condition.

React():

find distance to front car

This part basically finds distance to the nearest front car in the current lane as well as cars in other lane that is changing lane into the current lane.

Cut in front code

This part can be turned on to make the car start the cutting in front behavior. Judgment criteria driver use for front car is:

- coefficient*((speed of front car in our lane)*time +distance to front car in our lane))
- (speed of front car target lane)*time +distance to front car in target lane

Where time, or del_t in code, is any set time, depending on driver's experience. "Coefficient", or shorted "coeff" in the code, allows us to set how much we want the driver to cut. After looking at front car, the driver then looks at back car. The driver basically estimates, with the speed of back car and the distance to back car, if the car will hit him if he cuts in. This behavior translates to the following criteria:

- (coefficient*distance) >((back car speed)*(cut time))

If the above is true, there should be enough distance for the car to cut (at least according to driver's judgement).

Roadblock added code:

This part is specifically for road block. It contains two Booleans, warningsign and roadblock. Turn on roadblock when simulating roadblock. And put warningsign to true if the case is with warning sign and false if it is a sudden bottleneck.

Calculate the current desired speed

This part obtains the current desired speed of car and the acceleration that the car will want at current time. First it calculate the current desired speed, and then make sure that we do not get negative current desired speed by setting current desired speed to zero if it happens. (Note that negative current desired speed can occur if the vehicle in front suddenly stop and the current vehicle did not manage to stop before becoming closer to front vehicle than specified min_d_to_front_car.)

Acceleration is then calculated if it was not done before. (acc_adjusted tells the code if acceleration has been adjusted in prior part) Current desired acceleration is calculated using formula explained in previous formula part. Other than the formula, the car's acceleration cannot exceed the maximum acceleration of car determined by its engine limitation. So if the current desired acceleration is higher than maximum possible acceleration, the car's acceleration is set to this acceleration limit.

However, for some reason, if the speed is more than current desired speed, the car will be forced to slow down faster than usual with factor of five added to prevent crashing with car in front. (Note factor of five is arbitrary) Max brake could have been included as an extra cap on deceleration, but since this type of trigger—where speed is above maximum—does not happens a lot, it was left out.

Code for adjusting lane:

This part of the code will triggers if it detects that changing_lane_left or changing_lane_right Boolean variable is true. It will then check the stage of the turn the vehicle is in and adjust the angular_movement accordingly. If the vehicle is tested to be closer to the middle of the old lane, then the car will turns the wheel toward the lane it wants to go to. This action correlates to a positive angular_movement when turning to left lane and negative angular_movement when turning to the right lane. If the car is now closer to the new lane, then the driver would want to turn its wheel back and adjust the car into the lane. This action is then opposite of the angular_movement change stated before. To calculate magnitude of angular_movement, the variable lanechange_turn_radius was used as a reference turn radius.

lane end:

This part of the react() function will be triggered when the car recognized that it is at the end of its lane. The car will then take the next endlane_transition element in the vector and determine the type of lane transition whether it is straight, turn left, or turn right from the variable. (0=straight,1=turn right,2=turn left, 8=exit system) After that, the car will take the

next lane it should go to from the waypoints vector and calculate the according turn or simply go straight.

Platoon mode:

Basically when platoon mode is turned on, this last part of react() function will modify the current vehicle's velocity to be the exact same as front vehicle's velocity. This is equivalent to having vehicle accelerates and brakes at the same time as front vehicle which is exactly what happens in car platoons.

Lane Class:

Variables:

x_or_y, pos_or_neg_dir

Boolean that tells which direction the lane is. For instance, if they are both true, the lane is in positive x direction. If only x_or_y is true, the lane is negative x direction etc.

Position:

Position of the lane can mean x or y coordinate depending on the alignment.

Width

Width of lane is used when doing lane change. With width, the car can recognize when the car center has already reached targetlane and should start steering the wheel back.

start,end

Start and end of lane, again it can means start and end in x or y direction depending on alignment specified by the first two Booleans.

leftlane,rightlane

These two variables are pointer to the lanes adjacent to the current lane which lets the car in the lane detect other cars in adjacent lanes.

redlight

If this is true—which means the red light is on—the car in lane will pretend that there is another car blocking at the lane end and as a result, stop at the lane end.

Methods:

getDirection()

This method converts the x_or_y and pos_or_neg_dir to proper direction angle for convenience.

Creator Class:

This class will basically create a car given that there are no cars within the critical radius equals to the distance_between_cars variable. If there is no car within radius, chance_to_produce will also determine if the car will be produced. 0.05 chances to produce correspond to producing around one car every 2 seconds assuming there are no cars in the critical radius.

Variables:

Variables here are mostly similar to Car class because those variables will then be placed in car class when creating car. Most other variables are unambiguous and commented well in the code.

Methods:

closeByCar() method verifies if there is any car in the critical radius and returns true if so.

Three creation methods are available for this class for different uses. "createCar()" function creates car strictly according to the parameter specified. "createPlatoon()" function creates car with platoon_mode sets to true, and "createCarRand()" create car with slight variations in its parameters. "createCarRand()" is mostly used for a system that needs more variation such as the system that we used to demonstrate cutting in front behavior.

Main function:

Main function basically makes use of the classes shown above and builds the system we want to simulate. First the lanes are specified and adjacent lanes are inserted. Then general parameters that will be inserted into cars are initialized. The vector of car creator is built. For each case, this process starts with clearing the current waypoints and endlane_transition vector inputs, and then building/inserting specific way points for the creator. Creator is then pushed the creator into the creators vector. This process is repeated until all creators at incoming lanes are specified.

The function then proceeds to loop through time step of 0.1 second each. 0.1 second per time step is chosen because human reaction speed is around 0.2 second. During the loop, each creator will be checked if it should produce car. Out of bound cars will be terminated and tallied. And most importantly each car will be updated with new parameters.

After the time has reached specified maximum and loop is exited, tally is calculated and displayed. Lastly the memory is cleared to prevent memory leaks.

Data, Observation and Analysis for each types of phenomena:

Shockwave:

Road view shockwave video:

The car shockwave video shows the traffic jam propagating backward. As more cars arrive at the back of the jam, they must slow down and stop. On the other hand, cars in front of the jam start to clear. As jam in front clears and jam in the back builds up, the jam propagate backward as clearly show in the video.

Video of number of cars per bin vs position:

Another video is the video of number of cars per bin where the 300 meters system is separated into 200 bins. Each car is represented by Gaussian distribution and when tallied, instead of tallying +1 for a car into the bin where the car is positioned. The tally part put in only little fraction (calculated using Gaussian distribution function) into each bin. If we graph one car over all bins, we will see a Gaussian distribution. This approach turned out the smoothed the curve by quite a lot and we no longer see points that jump up and down because it can only exist as integer.

The result is that we see the shockwave propagating backward as we have expected. After propagating for a while, the wave started to become slanted in the front and then broke up into two waves which then decayed away. It is still uncertain whether this can actually happens in real road.

Steady State Test:

Motivation:

In the earlier part we did not use car creator class to manage incoming cars at the start of the lane, but use placeCar() function to manually place cars using the for-loop. However, this process would be extremely daunting and time consuming if we were to do it, for all the cases and for all the phenomena. If we want to get the data obtained in sections after this, and must place cars using for-loop manually we would have to determine the proper position and speed of every cars in the system at the start and write many loops in main time step for loop to get new incoming cars in. So knowing this, creator class was created in order to solve this time consumption problem and significantly simplify what we have to set in main() function to make a simulation case work.

Is using car creator idea actually valid?

Simplicity is always welcomed, but we also need to verify if using a car creator that spits out car with some distance interval will actually give us good results (car creator description is explained in code section). So this steady state tests are exactly done for this purpose. So

from the equations derived in class, we can solve for the distances between cars where maximum flux can be calculated as followed.

$$flux = \rho \cdot v = (1 - \rho) \cdot \rho$$

$$\frac{d}{d\rho} flux = \frac{d}{d\rho}(1 - \rho) \cdot \rho = 0$$

$$\rho = 0.5 = \frac{d_{min}}{d}$$

$$d = 2 \cdot d_{min}$$

So we can see that the distance between cars that will give maximum flux is just double the minimum distance the car will leave to another car which is set to 7 meters for this simulation. (Note that this distance counts the center to center of cars, so actual distance between front car's back and back car's front would be around 2-3 meters) Next, the car creator was placed on a lane to unleash the cars with some fixed distance between them. This was tried with different distances-between-produced-cars to see how the car flux varies. Car flux here is defined as the amount of cars exiting the system per second.

For this case, the random variation that has been built into system with the purpose of making it looks more real has been discarded so we can get more exact measurement.

Table 1: This table indicates the relationship between distances between cars that was created by creator object, and the car flux (number of cars per second).

Distance between cars (m)	Average flux (cars/s)
24	1.06
19	1.18
14	1.27 (calc. = 1.285)
9	1
4	0.66

Analysis of results:

As we have expected, the average flux is highest at the distance of 14 meters. At this point, the flux is 1.27 cars/s which is only 0.015 off from 1.285cars/s which can be obtained from plugging 14 into the original flux equation. The slight difference probably arises from the fact that the front most car will speed up slightly before exiting the system. This happens because

it sees no front car after its front car has been eliminated from the system (out of bound elimination).

Platoon:

Next we move on to examining how platoon can work on the system. We take the optimal car flux case from before and added platoons to the left lane. For this case, it was assumed that there is 50% chance that right lane's car creator will produce a single car and 50% chance that it will produce a platoon of ten cars. The resulting flux and average final time is shown below. Average final time represent the average number of seconds that each car spent from entering the system to exiting the system.

Table 2: This table shows the average final time and flux for the case where there flux is maximized for lane without platoon and above assumption was used for lane with platoon.

	Average final time (s)	**Average flux (cars/s)**
Platoon	24.1601	1.73
No Platoon	27.174	1.27

Analysis of results:

As shown, the platoon allows for higher than usual maximum flux as we would expect. More flux could have been squeezed in if we did a longer platoon and/or both lanes platoon.

Road Block:

Next, we examined road block and how placing warning sign before road block can make a difference in traffic flow through the bottleneck.

Table 3: This table shows the effectiveness of placing warning sign 100 meters before road block.

	No Warning Sign		Warning Sign	
Production distance	average final time (s)	flux (n/s)	average final time (s)	flux (n/s)
24	70.7099	0.37	51.7759	0.36
34	41.8327	0.366667	28.6344	0.426667
44	36.5018	0.37	23.2858	0.423333

Analysis of results:

The table shows that in general, placing the warning sign around 100 meters before a bottleneck can help with traffic flow through it. Although this is not too apparent in flux column, it is very apparent in average final time column in the table. This correspond to the video attached where we can see that traffic arranges nicely without as much cutting in front compare to the case without warning sign.

Cut in front:

Next we move to examining the behavior of cutting in front. In this case, the delta_t of 1 is used for when judging lane change. (This variable is explained in code section)

Table 4: This table shows how average final time and flux varies with cut-in-front coefficient.

	Distance = 14 m			**Distance = 54 m**		
Coefficient	average final time (s)	flux (n/s)	Number of cuts per car	average final time (s)	flux (n/s)	Number of cuts per car
1.1	48.9568	1.45	1.69195	38.8109	0.95	0.22807
1.5	48.3438	1.49	1.05593	38.7677	0.95	0.157895
2.0	46.417	1.48667	0.44843	38.7014	0.95	0.101754
3.0	45.8578	1.49333	0.176339	38.7028	0.95	0.0982456
5.0	45.7897	1.49	0.152125	38.7021	0.95	0.0982456

Table analysis:

As the table above indicates, cutting in front increases the average final time spent in system for vehicles. This is most likely due to the fact that cutting in front forced the back car in the lane to slow down. As shown above, this effect is more prominent in the case where the cars are closer to each other compare to the case where cars are farther away from each other. This is not unexpected since if there is less distance between cars, there is higher chance that some car will have to slow down to give way to the car cutting in front.

System of four-way and thre-way:

So the first run on a system of four-way intersection attached to three-way intersection was done with the following data as the results.

Table 5: This table shows the performance of red-light and stop sign cases at different production distances. This is done on a system with coupled four-way and three-way intersections.

	Red light/green light		**Stop signs**	
Production Distance	average final time (s)	flux (n/s)	average final time (s)	flux (n/s)
7	102.126	1.87143	110.135	0.485714
15	97.4733	1.87143	104.56	0.614286
30	90.122	1.81429	92.4511	0.642857
50	83.9205	1.67143	74.2911	0.642857

Why is there a decrease in average final time?

At first, this fact seems very weird because if the road clogged up and time is counted when car enters the system. Intuitively if the road is completely clogged, and the production of cars from car creator decreased down to the same flux, the time it takes for vehicle to go from one end to the other end should be around equal. However why are we seeing this noticeable decrease in average final time, even after system was retested for end time of 2000s and tally start time of 1500s. After observing the actual simulation, we found this is caused by the way the creator works. Take the case with production distance of 50 meters, if the actual simulation is observed, we can see that the car creator would leave 50 meters into the lane where there is no car (because creator test for if there is car close by before producing new cars). The car that came out on the 50 meters case will have some free run time while the car that came out from the 7 meters creator will have to stop right away. Because of this extra free run time, the average final time the car stays in system decrease as we increase the production distance. Of course this is not true with real world system.

This reemphasizes the fact that this type of set up of simulation has its limitation when the road starts clogging. To fix this to make it usable in real world, we should choose the system to make sure that the boundary has no clog. Of course, this does not mean that we cannot simulate clogged traffic. This can still be done (with inaccurate time but accurate flux), given

that we can simulate the system to extend upstream enough to where traffic isn't as severe. In other word, we can go upstream on the road until it reaches the point traffic clears and start to simulate there. Of course this will requires us to build quite a large system and consume a lot of computing power.

If the circumstance does not allow us to choose system that has no clog, extending the simulation back upstream should still help. As the length of the jam becomes a larger than the production distance, the relative error decreases.

Why is red light case's flux decreasing with increasing production distance?

The answer to this again lies in the way the car creator is set up. Again creator left space equals to production distance between the end of traffic jam and itself. This means, in the case where creator left a lot of distance, the traffic jam will be shorter compare to the case where creator left less distance. In other word, there would be less cars waiting for red light for long production distance case, and consequently, when the green light is up, there would also be fewer cars that got freed, and therefore less flux overall.

This problem again can be mitigated by extending the road farther. In fact, it should be possible to get the accurate flux in clogged road if we extend the road back long enough that one green light will not free the whole road to the end close to where car creator is.

System of four lane (Second Trial):

So to since the last attempt failed to give us satisfying data, we went on to truncate the right three-way intersection part of the system in order to simplify the system and decrease computer run time. Then the road was extended 200 meters in each direction so the distance between cars can be set to higher values. The total time and the tally start time were also increased considerably to make sure the calculation would be as accurate as possible. The time max is now set to 1000 seconds while the start tally time is set to 500.

Table 6: This table shows the performance of red-light and stop sign cases at different production distances. This is for system with only one four-way intersection with assumptions stated earlier.

	Red light/green light		**Stop signs**	
Production Distance	average final time (s)	flux (n/s)	average final time (s)	flux (n/s)
50	116.272	5.385	171.83	1.69
100	60.6624	4.65	108.225	2.14
125	58.2593	3.945	84.7351	2.06
150	57.1709	3.425	56.1816	1.965
230	55.0494	2.431	49.397	2.175

Analysis of results:

For stop sign case, the 150 meters trials has around 55-61 cars in system. And for red light/green light case, the 150 meters trials has around 59-74 cars at saturated point for red light/green light case.

For stop sign case, the 230 meters trials has around 38-41 cars in system. And for red light/green light case, the 230 meters trials has around 38-45 cars at saturated point for red light/green light case.

For red light green light case the cars never really build up even in 100 meters case where it saturates around 88-102 cars. Similarly to the four-way, three-way road this shows just how much more efficient the red light and green light system is compare to all-way stop signs.

Nevertheless, as before, we see cars starts to build up on 125 meters and 100 meters tries for stop sign case.

At that point, the measurement of average final time will again no longer be accurate. Because the road is very long, we also observed that the takes sometimes to clog the whole road, and therefore it uses quite a large amount of time to reach equilibrium. (Equilibrium where road is saturated with cars and car creator no longer produce car because of distance between cars condition)

The most important characteristic that can be seen in this data is that as the distances between cars becomes very large; the performance of stop sign cases rivals that of red-light/green-light cases.

Does lane amount and intersection size affects the results?

The results above might not be accurate for all types of road. We actually used three-lane road which has larger intersection area. As a result, it takes longer time for the cars to pass the middle of the intersection area and therefore each turn for letting cars go becomes longer. As a result, cases of stop signs with more lanes should be much easier to clog compare to cases of stop signs with fewer lanes.

To verify and demonstrate this point, small intersection of only one lane with minimal intersection area size has been built. Then again the average final time and flux has been compared as shown in table below.

Table 7: This table shows the performance of stop sign cases, with different number of lanes, at different production distances. This is for system with only one four-way intersection with assumptions stated earlier.

	Stop signs one lane		**Stop signs three lanes**	
Production Distance	average final time (s)	flux (n/s)	average final time (s)	flux (n/s)
100	52.1923	1.55	108.225	2.14
230	49.8251	0.705	49.397	2.175

Analysis of results:

This table help corroborates that our hypothesis about the intersection size affecting the stop sign's efficiency is right. Note that although there is only one lane compared to three lanes, the flux at 100 meters production distance is 1.55 cars/s compare to 2.14 cars/s. To make the number comparable, the 2.14 should be divided by 3 to get 0.713 cars/s which is only half as much as 1.55 cars/s. This shows the increased effectiveness of all-way stop sign, on roads with fewer lanes. At very high production distance, the flux pretty much turns out the similar at 0.725cars/s (2.175/3) in three lanes case, and 0.705cars/s in one lane case.

This is probably partly related to how we will almost never see an all-way stop sign on a road larger than two lanes. (Other than the reason that, stop sign will usually be placed in light traffic area and light traffic area usually does not have more than two lanes.)

Intersections Overall Analysis:

All-way stop sign should be placed in areas with less traffic intensity and less lanes. Fortunately, these two characteristics are usually coupled since fewer lanes are usually built for places that are predicted to have less intensity. Another important point is that, from these

data and analysis, we can conclude that it would be wise to use stop sign in the case where cars are very sparse.

Conclusions:

Steady State:

-Maximum flux obtained from using equation and using simulation is almost exact. (Which should be, given we are pretty much using the same equation)

-Using creator class to create car does work quite well and could simplify the tedious work by quite a lot.

Platoon:

-Platoon can be used to increase the traffic flux and exceed the usual peak flux in the case when cars are not arranged into platoons

Road Block:

-Putting warning sign before roadblock or bottleneck can help alleviate traffic through it. With warning sign, driver is given time to slide into another lane without having to urgently cut into another lane and slow down other car from doing so.

Cut in Front:

-The behavior of cutting in front slows down the overall traffic especially in the high car density case.

Intersections:

-Red-light/green-light is almost always better than stop sign in terms of relieving traffic. However stop sign has its advantages in cost and safety.

-Stop sign's performance converged very close to that of red-light/green-light at larger production distance (or car density).

-Stop sign with fewer lanes is more effective than with more lanes.

Possible extensions:

-Add in graphics directly from C++ code to makes movie making code runs faster.

-Add car turning left in four-way intersection.

-We can define lane as several points instead of what is done in currently. This way we can create lane that can curve around the way we want and adjust to any type of lane in real world.

-The lane end to another lane start, transition algorithm can also be modified to make it possible to build a non 90-degree intersection.

- Turn radius calculation: (for future improvement)
    - 1. Find intersection using y = mx+b etc.
    - 2. Find distance from lane end to intersection
    - 3. Find angle 3
    - 4. Find turn radius