Minimizing Lost Time at Automobile Intersections

Alexander Stanoyevitch
Associate Professor, University of Guam
Division of Mathematical Sciences
UOG Station
Mangilao, GU 96923, USA
alex@math.hawaii.edu

Keywords: Transportation logistics, automobile intersections, MATLAB

Abstract: We will set up a model of traffic flow into an intersection and compare the two-way stop traffic control and the four-way stop traffic control with regard to the average time delay per car. We will use simulations to demonstrate that a two-way stop system is always more efficient than a four-way stop system, as far as average time delay per car is concerned. We will explain the validity of our model, and show how to extend it to study other parameters of interest. The simulations were run on a personal computer using the MATLAB computing environment. Details of setting up efficient codes, in general, will be provided. Moreover, elements of MATLAB coding, a language which is very intuitive and similar to the C-family, will also be given.

INTRODUCTION

The basic object under consideration, an automobile intersection, is shown in Figure 1.

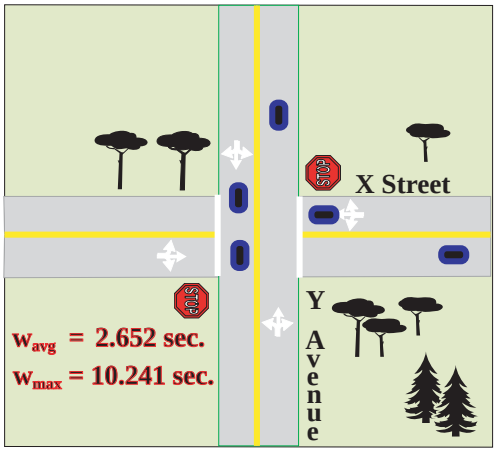

Figure 1. A two-way stop controlled automobile intersection, with simulated data displayed.

Even for this simple model, there are many variables and an infinite number of interesting problems that one might consider. The two quantities shown in Figure 1 have the following meanings: w_{avg} = the average time lost at the intersection by ALL cars, w_{max} = the maximum time lost at the intersection by a car approaching on X Street.

To set up the model, the following intrinsic parameters and basic variables will be required. We distinguish between X-cars (cars approaching the intersection on X Street) and Y-cars (cars approaching on Y Avenue). f_x = the flow rate on X Street (= the number of cars/unit time entering the system through either end of X Street), and f_y = the flow rate on Y Avenue. In our model, we will assume that the cars arriving from either end of X Street have the same homogeneous Poisson process distribution, and likewise for the cars arriving on either end of Y Avenue. We will assume that all cars arriving into the intersection will continue in the same direction. We make this assumption only to simplify the computations of the delay times, which for any car will simply be the difference in actual time that it takes the car to run the length of the road less the time it would take if it were to maintain the (constant) speed limit. To make up for this assumption of convenience, we will work with the standard (legal) protocol for priorities at an intersection: cars that arrive first have first priority, and the next car in priority must wait until its predecessor has cleared the intersection. In particular, we ignore the common, if not strictly legal practice of cars negotiating the sharing of an intersection. We will arbitrarily set up a priority system to handle cases where two or more cars arrive at their stops signs simultaneously. Although there are specific roadway laws governing such priorities, different priority systems will result in the same average time delays, so any will be sufficient for our purposes. We assume that all cars will maintain the speed limit unless, slowing down for, moving away from, or resting at a stop sign. We assume that speed limits on both streets will be equal, but removing this restriction is not difficult.

Our methodology will be to create simulation programs using the MATLAB software. The explanations will be presented in a general fashion so as to allow the creation of simulation programs in any feasible computing platform. These programs can be run on personal computers with acceptable speed and graphical output. The lion's share of the computer time, however, will be taken up by the graphical output. The basic strategy in designing our simulation programs will be to first design one with a full graphical account of all of the events taking place, and then (after checking to see that it works the way we intended it to work), we disable all of the graphics from it to get a streamlined core program. The latter program will be more robust and from it we will be able to collect a sufficient amount of data to draw conclusions from. The graphically enhanced original program is vital. The simulations are quite complicated for such traffic

 ISBN: 1-5655-283-0

problems, and a graphical output will allow us a most effective way to detect and eliminate any bugs that the program might have.

After running the graphics-based version to show its plausibility, we will use a non-graphics version of the program to create some large sets of simulation data that we will analyze and draw conclusions from. Our simulations will demonstrate the rather surprising conclusion that the two-way stop sign configuration (see Figure 1) (in case $f_x \geq f_y$) is always advantageous over a four-way stop sign system, as far as minimizing average lost time is concerned. For an assortment of various flow rates, we will run simulations for very long time intervals so that the simulated values of average time lost either stabilize or grow without bound. The latter cases correspond to the traffic control system being unfeasible to control the given traffic. In particular, our simulations will indicate that for a given traffic flow, unfeasibility of a two-way stop system will be sufficient for that of a four-way stop system, but not conversely.

If traffic flows are too high for stop sign systems, one can then consider traffic lights, or, in extreme cases, even an overpass. One obvious fact is that if a four-way traffic light is used, the red light proportions in a given light cycle should be approximately inversely proportional to the ratio of the traffic flows. Our methods should enable the reader to set up similar simulations to examine many other related questions such as: When are traffic lights more efficient?, Once a traffic light is deemed necessary, how long should the cycle last? Any 4-way traffic light system can be reduced to any sort of 2-way or 4-way stop sign system using flashing lights (and many municipal systems revert to this during the graveyard shift hours), and furthermore, traffic lights can be programmed to adapt to changes in traffic flows.

Computer simulation of traffic dates back to the 1955 dissertation of D. L. Gerlough [1], and since this time it has developed into a vast area of research with many facets and is helping to make our increasingly congested roadways more safe and efficient. Traffic simulations are often dynamic in nature and the most efficient traffic control systems react to the dynamics of the system rather than the other way around. Even the problem of optimal traffic control for a single traffic intersection is still not fully resolved and remains an area of active research, see, e.g., [2]. The complexity of even the most simple sort of intersection (see Figure 1) is analytically unsolvable so that simulation is the only feasible approach. Our models will ignore the possibility of automobile accidents due to driver error, but our systems will be designed so as to not allow drivers to enter into an accident provided that the rules and signals are followed. Safety is a serious issue that cannot be ignored, and inevitably accidents do happen (even on the best designed traffic systems). In practice, most traffic simulation programs that have been developed are constructed to avoid accidents. Although some work has been done to include accidents in traffic simulation programs (see, e.g., [3]), effective general methods remain to be developed. For a good general survey of recent trends in simulations in traffic control, we cite [4].

PREPARATION AND THE ARRIVAL PROCESS

For an introduction to the MATLAB computing environment, see [5]. We need to work in a specific reference frame, for simplicity we will use the one shown in Figure 2.

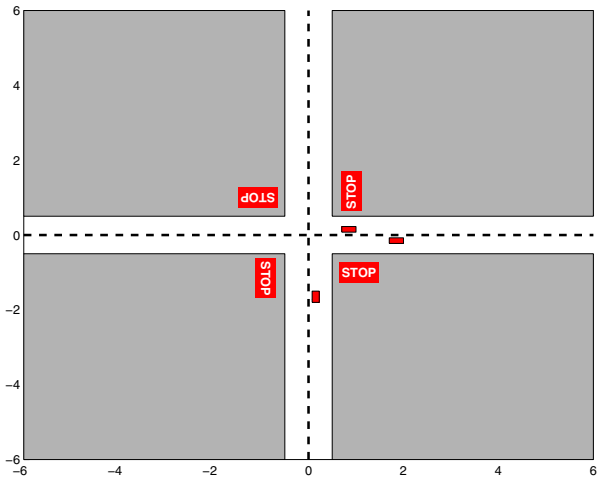

Figure 2 A MATLAB graphic of a generic (four-way) stop signed intersection, with three cars.

The following MATLAB commands will produce the intersection in Figure 2:

```
axis(2*[-3 3 -3 3])
hold on
xl=[-6 -.5 -.5 -6];, yb=[-6 -6 -.5 -.5];
xr=[.5 6 6 .5];, yt=[.5 .5 6 6];
fill(xl,yb,[.7 .7 .7]), fill(xl,yt,[.7 .7 .7]),
fill(xr,yb,[.7 .7 .7]), fill(xr,yt,[.7 .7 .7])
t=-6:.25:6;, u=zeros(size(t));
plot(u,t,'--y','LineWidth',2)
plot(t,u,'--y','LineWidth',2)
```

(The semicolon at the end of a MATLAB command suppresses its output.)

Each stop sign can be inserted using a command like the following (which installs the southbound lane's stop sign):

```
Sstop = text(-.7,1, 'STOP');
set(Sstop, 'Rotation', 180, 'BackgroundColor',...
[1 0 0],'FontWeight', 'bold', 'Color', [1 1 1]);
```

(The ellipsis '…" in MATLAB is used to continue a long command line on the next line of code.)

We leave it to the reader to create the other three stop signs. We distinguish the four lanes: East, West, North, and South. Note that each lane has length 12 and width 0.5. The cars have length 0.3 and width 0.15. The basic parameters for the East and West cars are as follows:

```
EcarX=.15*[-2 0 0 -2]; EcarY=0.075*[-3 -3 -1 -1];
WcarX=.15*[0 2 2 0]; WcarY=0.075*[1 1 3 3];
```

The cars are the objects that will be changing in the graphic display of this simulation. To place the East(bound) car that appeared in Figure 2, we could have entered the command:

```
Car1 = fill(EcarX+2,EcarY,'r')
```

Try this, and see the eastbound car appear in your MATLAB window. We could have just entered the latter part of the above command '`fill...`' to have obtained the same result, but it is

convenient to give graphical objects names, since then it will be a simple matter to change their position (or many other of its attributes). For example, if we wanted to change the above eastbound car's position so that it advanced .5 units forward (eastbound), we could simply enter:

```
set(Car1(E(i,1)), 'XData', EcarX + 2.5)
```

This will cause the current East car to disappear and be replaced by a new East car with the desired position. We leave it to the reader to construct corresponding parameters for North and South cars, and to append the plot of the northbound car shown in Figure 2. We also would like to include a sign for the real-time updated average delay (as shown in Figure 1). The following commands will set this up:

```
DelaySign=text(-5,4,'Average Delay');
AvgDelayTracker=text(4.5,3.4,num2str(0,5),...
'FontWeight','bold', 'BackgroundColor','c');
```

We are assuming that cars arrive (from both sides) of each road according to a homogeneous Poisson process. This necessitates establishing some unit of time as a reference frame. The governing variable in our model is the time t, and the simulation starts at t = 0. We recall the standard algorithm for generating events of a homogeneous Poisson process with rate λ from t = 0 to t = T: (see, Section 5.4 of [6]):

Set t = 0, counter = 0. Continue to generate (uniformly distributed) random numbers $U_1, U_2, \cdots$ in (0, 1) and advancing t by $t = t - (\log U_i)/\lambda$ along with incrementing the counter until the time advances past T. These time values constitute the event times of the Poisson process.

In our simulation, we will advance t by whole numbers: t = 0, t =1, t = 2, ….(called **discrete time units**, or **du**'s). We will assume that when traveling at normal speed, a car will cover one quarter of its length (dx = 0.075) as t advances by 1du. Also, we assume that at the speed limit, the minimum gap between consecutive cars (on the same lane) is 1.5 carlengths (= 6dx). The following MATLAB program uses a slight modification of the Poisson process to generate arrival times that satisfy a minimum gap requirement.

```
function Times = hompoissongap(lambda, T, mingap)
Times = [ ]; t=0;
while 1
    U = rand;
    t = t-1/lambda*log(U);
    if t > T
        break
    else
        Times = [Times t];
    end
end
len=length(Times);
for i=1:len-1
now=Times(i); next=Times(i+1);
while next-now<mingap
Times(i+1)=now+mingap; i=i+1;
if i+1>len, return, end
now=next; next=Times(i+1);
end
end
```

CREATION OF THE SIMULATION PROGRAMS

We will provide a detailed outline of the construction of a graphically enhanced simulation program for the two-way stop controlled intersection with traffic flow as described in the last section, along with many details of the MATLAB code. In order for the graphical animations that will be produced by such a program to display smoothly in MATLAB , it is helpful to enter the following in the command window (after a graphics window for the intersection has been set up as in the last section):

```
set(gcf, 'DoubleBuffer', 'on')
```

Without getting into technical details, this command changes the way MATLAB's graphics window draws its displays in a way that is more conducive to animated rather than static displays.

We will measure our traffic flow rates f_x and f_y in terms of cars per 800 du. To put things into a practical perspective, let us take a typical car to be 14 feet in length. The speed limit of the cars listed in the last section would translate into $2.\overline{12}$ miles per 800 du. Thus, if we were to take 800 du to be 4 minutes of real time, this would translate to a speed limit of approximately 30 m.p.h ($31.\overline{81}$ to be exact). Once the flow rates have been entered as variables '`fx`' and '`fy`' into the program, as well as a variable '`T`', indicating the number of du's over which the program should run, we can use the program of the last section to create simulated arrival times for cars in each of the four directions. In MATLAB, the arrivals of the eastbound cars could thus be constructed with the following command:

```
Etimes=hompoissongap(fx/800,T,10);
```

In the same fashion, the simulated arrival times '`Wtimes`', '`Ntimes`', and '`Stimes`', for the other lanes can be created. It would consume less storage to simply create these simulated times as needed (and delete them when we are finished with them); we will leave this embellishment to the reader as its implementation will become relatively clear after the program is understood.

We also pre-allocate corresponding vectors for the northbound and southbound departure times (whose entries will be constructed by the program), as well as variables that will sum the total delay times for northbound and southbound cars (the only cars that will be delayed by the stop signs).

```
NtimesDep=zeros(size(Ntimes));
StimesDep=zeros(size(Stimes));
delayN=0; delayS=0;
```

It will be convenient to add an extra imaginary car that will arrive at t = ∞ to each of the four vectors of arrival times:

```
Etimes=[Etimes Inf]; Wtimes=[Wtimes Inf];
Ntimes=[Ntimes Inf]; Stimes=[Stimes Inf];
```

In order to make the appropriate changes in our system at each du time increment, it will be necessary to keep track of the cars that are in the system, the system being the intersection of Figure 2 (with only a two-way stop for the North/South cars). We next describe and initialize some variables that will help to make this task more manageable:

SYSTEM VARIABLES:

Indices for each type of car:

```
iE=1; iW=1; iN=1; iS=1;
```

Variables for the "next" arrival times:

```
nextE=Etimes(iE); nextW=Wtimes(iW);
nextN=Ntimes(iN); nextS=Stimes(iS);
```

Information matrices for cars currently in the system, and corresponding variables tracking the number of rows:

```
E=[]; W=[]; N=[]; S=[];
rowsE=0; rowsW=0; rowsN=0; rowsS=0;
```

Note: Initially, of course, these matrices are empty since there are no cars in the system. In general, these matrices will have two columns and a row for each car in the system in its corresponding lane. For example, the first column entry of an eastbound car in the matrix '`E`' indicates the index of the car, that is, the index of the car's arrival time in the '`Etimes`' vector. The second column entry for the car in the system indicates the x-coordinate of the front of the car at the current time t. We will arrange it so that the first column entries of E decrease. This corresponds to new cars entering the system having their information put in the first row of E.

Variables to keep track of the priority in the intersection:

```
tracker = [];, busyflag=0; justcN=0; justcS=0;
```

Note: The variable '`tracker`' (initially empty), is a vector that will keep track of who has priority in the intersection. This vector will have at most three components among the numbers 1 (for North) and 2 (for South). The first entry in the vector has first priority or is currently clearing the intersection. The variable '`busyflag`' is a number being either 0, if no north/southbound cars are clearing the intersection at the current time t, 1 if a northbound car is currently clearing the intersection, or 2 if a southbound car is currently clearing the intersection. The last two variables will either take on the values 0 or 1; most of the time they will be zero. The variable '`justcN`' will be 1 for the first few du's after a northbound car has just cleared the intersection. We will make the notion of "clearing the intersection" more precise in a short while.

With the variables and graphics initialized we are now ready to describe the main simulation program. It will run from t = 0 to t = T du's (in MATLAB: `for t=0:T`). At each time increment we will need to do two things: (i) Update the current cars that are in the system, and (ii) check for any new cars that will enter the system. At each iteration, (i) should be done before (ii) because we need to assure that the new car will have an appropriate gap from its immediate predecessor. While this will be automatic for eastbound and westbound cars by the way the arrival times were constructed, it may be a problem for the north and southbound cars, in case the traffic is backed up.

(i) Updating Current Cars in the System:

We begin with how to deal with eastbound cars. Westbound cars can be dealt with similarly. Such cars proceed through the system at constant speed, so are quite simple to track. The following loop will perform the required updates in the system variables:

```
for i=1:rowsE
     if E(i,2)>6+4*dx %E car has just left system
         delete(Ecar(E(i,1)))
         rowsE=rowsE-1;
         E=E(1:rowsE,:);
     else
         E(i,2)=E(i,2)+dx;
         set(Ecar(E(i,1)), 'XData', E(i,2)+EcarX)
        end, end
```

(The percent symbol '`%`' in MATLAB indicates a comment that will be ignored by the compiler.)

There are two easy cases. In one case, the car has left the system (we take this to be when the front of the car is greater than a car length past 6), and we delete the graphic element of the car and the last row of E. The graphic element for the car '`Ecar(E(i,1))`', will have been previously constructed in Part (ii) of the code. The other case corresponds to the car still remaining in the system. Here we simply need to update its information in the matrix E and reset the position in its graphical object.

North and southbound cars are more complicated since each need to stop at the intersection. Furthermore, the distance that a given car will advance will depend on various factors such as: is it clearing the intersection (so speeding up)?, does it have traffic backed up before it? We explain in detail how to deal with updating the matrix N corresponding to the northbound cars. When such a car is still in the system, in order to correctly update its position, we need to separate into several cases depending on where the "front" of the car is located. Firstly, to account for backed up traffic, it is important that the positions of cars be updated starting with the first ones to enter the system. In MATLAB, the updating loop for the N matrix could be indexed as follows:

```
for i=rowsN:-1:1
```

We separate into different cases depending on where the front of the car (y-coordinate N(i,2)) is located. The case where the car has just left the system is done in the same way as was done for eastbound cars above. The only difference is that we will need to record the departure time, compute the delay that the car experienced and add this onto the '`delayN`' variable.

```
if N(i,2)>6+4*dx %N car has just left system
      NtimesDep(N(i,1))=t;
      delayN=t-Ntimes(N(i,1))-163+delayN;
      delete(Ncar(N(i,1)))
      rowsN=rowsN-1;
      N=N(1:rowsN,:);
else %N car is still in system, we will have to
     %reset its y-data. There are several cases.
```

Note: The 'delay' that is added to the '`delayN`' variable above equals the difference in the actual time the car spent in the system less the time it would take to run the same course at normal speed (as if there were no stop sign). For example, if the northbound car had index iN=8, the time the car spent in the system is NtimesDep(8) – Ntimes(8). This time corresponds to when the vehicle's front end was at position y = -6 + dx until it reached y = 6 + 4 · dx. Since dx = .3/4, we get 160dx = 12, so it would take 163 time units for a car to run this course at normal speed. We ignore round-off errors arising from rounding arrival times to their floors of the discrete time units.

We will declare the "speed up" (clearing) zone of the intersection to be when the front of the car is in the range y = -0.5 to y = 1. The following commands will introduce the gap between the current car and it's immediate predecessor (if it has one):

```
front=N(i,2); back=front-4*dx;
  if i<rowsN %another N car in system is in front
    gap=N(i+1,2)-4*dx-front;
  end
```

When a car is out of the speed up zone, the treatment is quite simple (no possible traffic), we just need to reset the '`justcN`' variable back to 0, in case the car "just enters" into the zone:

```
if front>1 %out of speed up zone
   N(i,2)=N(i,2)+dx;
   if front<1+2*dx & justcN==1,justcN=0; end
```

 ISBN: 1-5655-283-0

The speed up zone is handled by the code below, we (realistically) give the car a linear increase in speed (with even a slight overshoot in speed at the end). We also add in provisions to change the intersection priority variables when the car just leaves the clearing zone.

```
elseif front>-.5 & front <=1 %speed up zone
   N(i,2)=N(i,2)+dx*(front+.85)/1.5;
   if busyflag==1&justcN==0&front>.5+4*dx
     %N car just clears
     justcN=1; busyflag=0; tracker=tracker(2:end);
   end
```

Before moving on to updating the north/southbound cars, we will need to form two sets SN and SS that will help us to later see if any north/southbound cars will be able to clear the intersection. We will give only the construction of SN, that for SS is similar. The set SN will be nonempty precisely when it is unsafe for a northbound car to cross the intersection. It will consist of indices of any eastbound car whose front is in the range x = -2.75 to x = .5 + 4dx (see Figure 2) as well as the indices of any westbound cars in the corresponding "danger zone".

```
SN=[]
if ~isempty(E)
 SN=find(E(:,2)>-2.75 & E(:,2)<.5+4*dx);, end
if ~isempty(W),
SN=union(SN,find(W(:,2)<3 & W(:,2)>-.5));, end
```

We set the "slow down zone" for the stop sign to be within the ranges y = -1.5 to y = -.58. The program segment below takes care of the pre-slow down zone. The case where traffic is backed up will result in a slower speed increment depending on the gap to the next car. In slowing down, the minimum gaps between cars is allowed to (realistically) decrease down to 0.05 (=2/3 dx).

```
elseif front<-1.5 %before slow down zone
   if i==rowsN %no gap
      N(i,2)=N(i,2)+dx;
   elseif gap>=6*dx
      N(i,2)=N(i,2)+dx;
   else, N(i,2)=N(i,2)+dx*(gap-.05)/(6*dx);
end
```

When the car is in the slow down zone, elements of the previous two cases are applied to produce the code segment below.

```
elseif front>=-1.5 & front<-.58
  %slow down for stop zone
  if i==rowsN %no gap
    N(i,2)=N(i,2)+dx*(-.5-front);
  elseif gap>=6*dx
    N(i,2)=N(i,2)+dx*(-.5-front);
  else
    N(i,2)=N(i,2)+dx*(-.5-front)*(gap-.05)/(6*dx);
  end
```

This takes care of all cases except when the car is actually at the stop sign. The code below takes care of this final case.

```
else %N car is on stop line, driver needs to wait
     %for clearing/turn
if sum(ismember(tracker,1))==0,
    tracker=[tracker 1];, end
if busyflag==1&sum(ismember(tracker(2:end),1))==0
    tracker=[tracker 1]; end
if busyflag==0&tracker(1)==1&isempty(SN) %go
     N(i,2)=N(i,2)+.09; busyflag=1; end
%else stay put
end
```

As all cases having been taken care of, we may now update the graphic of the current car '`set(Ncar(N(i,1)), 'YData', N(i,2)+NcarY)`', and end the loops.

(ii) Check for any New Cars that Will Enter the System:

For east/westbound cars, the procedure is quite straightforward, the MATLAB code is demonstrated below for eastbound cars.

```
if nextE<t+1 %need to add a new first row of E and
             %update other variables
    E=[iE -6+dx; E]; iE=iE+1; rowsE=rowsE+1;
    if iE<=length(Etimes), nextE=Etimes(iE);, end
  %next, install graphic w/ handle
Ecar(E(1,1))=fill(E(1,2)+EcarX,EcarY,'r',...
'EdgeColor', 'None');
```

The treatment for north/southbound cars needs to take into consideration the gap between the front of the new car and the end of its immediate predecessor (if there is one). If the gap exceeds 10dx, the treatment is as above, but for smaller gaps, the speed should be decreased (as was done in some of the previous code elements). If the gap is too small, we treat this as a traffic overflow (it would correspond to a backup of approximately 16 cars). In such a case, we would end the program with either an error message ("traffic jam"), or give an output of an infinite delay. We leave this choice and the remaining writing of this code segment to the reader. Of course, larger backups could be accommodated by using an intersection with a longer Y Avenue.

The final MATLAB segment below will compute the current average delay and display it on the graphic window.

```
delay=delayN+delayS;
 if delay>0
   NumOutCars=iE+iW+iN+iS-rowsE-rowsW-rowsN-rowsS;
   AvgWait=delay/NumOutCars;
set(AvgDelayTracker,'String',num2str(AvgWait,5));
    end
 drawnow
```

CONCLUSIONS

Once your programs have been debugged to work well, the graphics can be disabled and simulations can be run for time periods of over 100,000 du's (which equates to roughly an 8 hour real time simulation) in a few minutes on a personal computer. We ran 20 such simulations for each flow rate ranging from fx = fy = 1 to 6.5 in increments of 0.5, and used the means of the resulting outputted sets to obtain the plot shown in Figure 3. The standard deviations for each such data set were tolerable, ranging from about 2.5% to about 8%. The variances clearly reduced in a nice accordance with the central limit theorem as the length of the simulation increased. We took the average wait time to be infinite if more than half of the 20 trials resulted in a traffic overflow in the program (meaning about 16 cars were queued up at one point). A suitably modified and more lengthy simulation could be used to discern the threshold traffic flows for actual unstable traffic control with a given system versus a very long average waiting time.

It seems quite intuitive that if a four-way stop configuration were ever to be more efficient than a two-way system with the equivalent traffic flows, then the same should be true in an equal traffic flow situation for the two roads. Even for such a plausible statement, an analytic proof does not seem to be available. In Figure 4, we give some simulated evidence for validity of this statement. (Of course, in cases where the flows are different, it is clear that the stop signs on a two-way system should be placed on the road with less traffic).

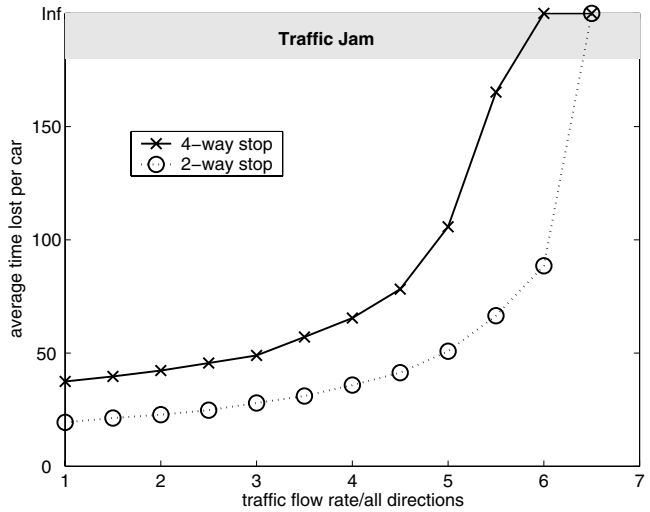

Figure 3 A plot of simulated delay times for the four-way stop (solid) and the two-way stop (dotted) controlled intersection as a function of the uniform flow rates.

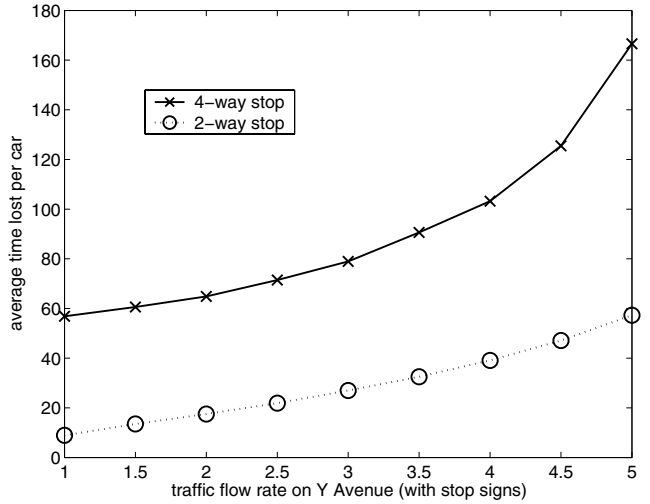

Figure 4 A plot of simulated delay times for the four-way stop (solid) and the two-way stop (dotted) controlled intersection as a function of the traffic flow rates on Y Avenue. The traffic flow on X street is maintained at a constant rate of 6. Notice how the differences for the two systems are even more extreme than in Figure 3.

REFERENCES

[1] Gerlough, D. L., *Simulation of freeway traffic on a general-purpose discrete variable computer*, PhD Dissertation, UCLA (1955)

[2] McDonald, M., M. Brackstone, and B. Sultan, *Instrumented vehicle studies of traffic flow models*, Proceedings of the Third International Symposium on Highway Capacity, Volume 2, (editor: R. Ryysgaard), pp. 755-774. Copenhagen Transportation Research Board and Danish Road Directorate, Copenhagen (1998)

[3] Sayed, D., *Estimating the safety of unsignalizedintersections using traffic conflicts*, Proceedings of the third international conference on intersections without traffic signals, (editor: M. Kyte), pp. 230-235 Portland, OR (1997)

[4] Pursula, M., *Simulation of Traffic Systems - An Overview*, Journal of Geographic Information and Decision Analysis, vol.3, no.1, pp. 1-8 (1999)

[5] Stanoyevitch., A., *An Introduction to MATLAB with Numerical Preliminaries*, John Wiley & Sons, New York, to appear in late (2004)

[6] Ross, S., *Simulation-Third Edition*, Academic Press, San Diego (2002)

Biography

Alexander Stanoyevitch earned his PhD in 1990 from the University of Michigan-Ann Arbor in mathematical analysis. Since then, except for a recent visiting appointment at the University of Missouri-Columbia, he has worked in academia in the tropics (at the Universities of Hawaii and Guam). He has published numerous papers in geometric analysis and partial differential equations. He uses MATLAB (extensively) in his teaching as well as his research, and has recently completed two MATLAB-based books (one on numerical differential equations and the other a general introduction to the software) that will soon be published by John Wiley & Sons. Dr. Stanoyevitch enjoys traveling and has spent extended periods visiting mathematical institutes in Finland, France, Norway, Germany, The Czech Republic, and Ireland.