A simple simulation for Manhattan traffic

Eleven (Shiyi) Chen sc7825@nyu.edu

May 15, 2021

Contents

1 Introduction

In this project, a simple traffic grid is built to mimic the traffic system at Manhattan in New York City. Manhattan has 214 numbered streets and 11 numbered avenues[1]. The streets go in the East-West direction while the avenues go from South-North. The directions here are all aligned with the Hudson river. In real Manhattan, some streets and avenues can go in both directions. To simplify the problem, only alternating-direction one way streets and avenues are considered.

The motivation of this project is to have a descriptive model to the Manhattan traffic flow and some qualitative insights about the capacity of the grid system.

2 Notations and assumptions

The notations used in this report can be found in Table 1. The assumptions of the model includes:

- Our idealized Manhattan consists only one way avenues and streets. The length and the width for each block is the same. So that the total length for each streets and avenues only depend on the number of blocks in the vertical and horzental directions. An example grid can be found in Figure 1
- The cars on the streets and avenues are randomly assigned for the departures and destinations, cars find the closest path toward its destination. The detail guidance system is discussed in Section 4.1. Once the car reaches its destination, it automatically disappears
- There is no lights on the cross. To make the turns, there should be no cars within the range of entering zoom on the other road or the speed of cars on the other road is 0. The detailed cross rules can be find in Section 4.1.
- The speed of the car is calculated using the formula below:

$$v = \begin{cases} 0 & \text{if } d < d_{\min} \\ \frac{v_{\max} \log(d/d_{\min})}{\log(d_{\max}/d_{\min})} & \text{if } d_{\min} < d < d_{\max} \\ v_{\max} & d > d_{\max} \end{cases}$$

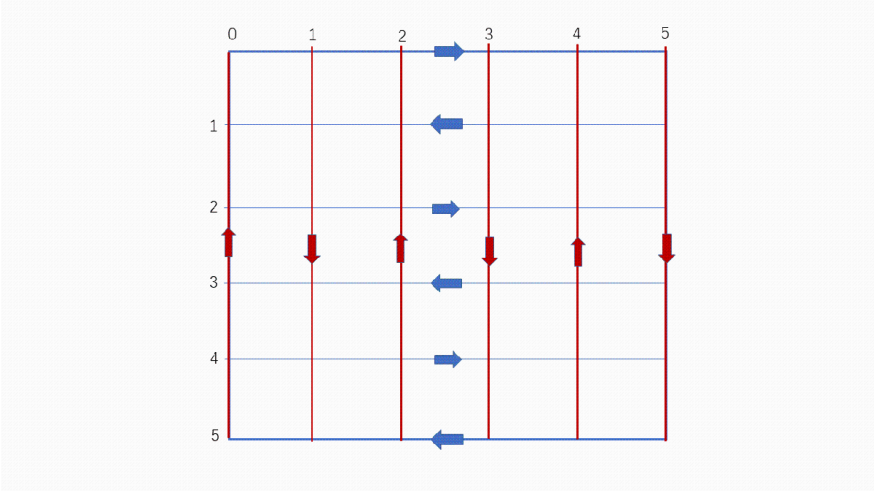

Figure 1: Schematic of the model system

Notation	Discription	Default value
Na	Number of avenues	4
Ns	Number of streets	4
dt	Time step	1 second
initNc	Initial car in the system	10
d_{min}	Distance within which car stops	10 meters
d_{max}	Distance above which cars goes at max speed	100 meters
v_{max}	Maximum speed	5 m/s
c_{max}	Maximum number of cars in the system	Na*Ns*L*L/dmin
t_{max}	Simulation time	5e4 seconds
L	The length of each block	50 meters
$rate$	The rate of imcoming cars per second	0.4 s^{-1}

Table 1: Notations used in this report

3 Object oriented implementation in the program

Object-oriented programming is a great implementation of a programming language to organize its data structure. Since the system that we are trying to simulate contains a non-trivial grid structure, the object-oriented programming style is employed to organize the variables thus making the life of coding easier. Although object-oriented programming has the chance to blow up the run time of the program, considering the sum of programming time, debugging time, and run time, I decide to adopt this method.

Going back to our model, only two variables are used - `strs, aves` to describes all the information for cars in the system. That information includes their current locations, their current speeds, their departures, and their destinations. In our model, the smallest unit is a car. For each car, it is described using a variable of class `Car`. In Matlab, the self-defined class is supported. Figure 3 shows the structure of the class `Car`. It has an encapsulated data field containing four variables which are regular arrays. `depart, destin, loc, speed` represent the departure location, destination location, and its current location, and current speed respectively. Besides its data field, it has a constructor method that takes the `Ns` and `Ns` as input and constructs a car object with a random departure in the grid and a random destination.

Matlab also offers a handy functionality called object array which is used to represents each street or avenue in this project. Object array is similar to the ordinary arrays in many ways, it is indexed using curly bracket, it only allows the elements within one object array has the same type. So for each street or avenue, it is represented by a 1 by n `Car` array, while the n is the number of cars on the street/avenue.

All the streets then constitutes the 1 by `Ns` cell array called `strs`, a cell array in Matlab allows any type of elements within one cell and it is indexed using { }. Similarly, all the avenues constitutes `aves`.

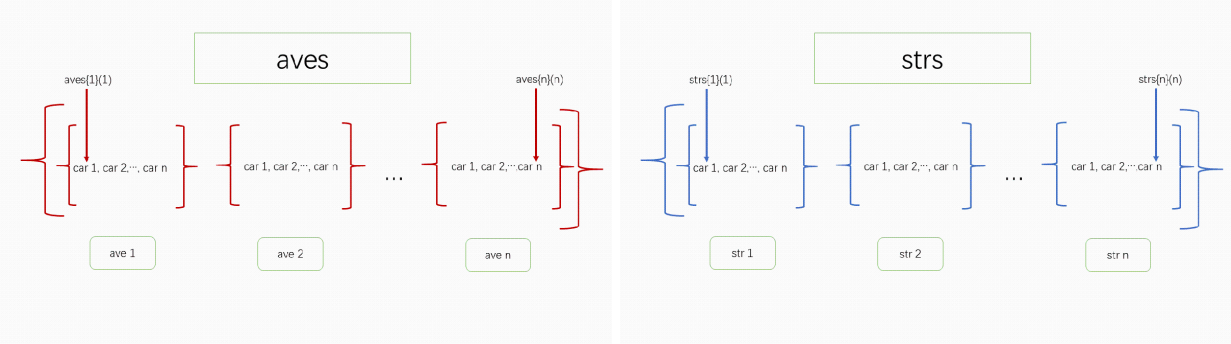

Figure 2: `aves` and `strs`

In summary, in order to retrieve i's car on the j's avenue, we can just call:

```
aves{j}(i)
```

4 Self-guidance system and cross rules

4.1 Self-guidance system

Cars use a self-guidance system to guide themselves to the destination. This mechanism is being implemented whenever there is a car arriving at the cross. The self-guidance system relies on a vector pointing from the current location to the destination. At the cross, there are only two possible directions the car could go, either to the avenue direction or to the street. In Figure 4, for example, the car is currently at the cross star, and the direction vector is $(-1, 3)$. The base vector on the street is $(0, -1)$ and the base vector on the avenue is $(-1, 0)$. Since the dot product on avenue 1 is larger than the dot product on street -3. Thus, the car will choose on the avenue no matter its previous location on the street or avenue.

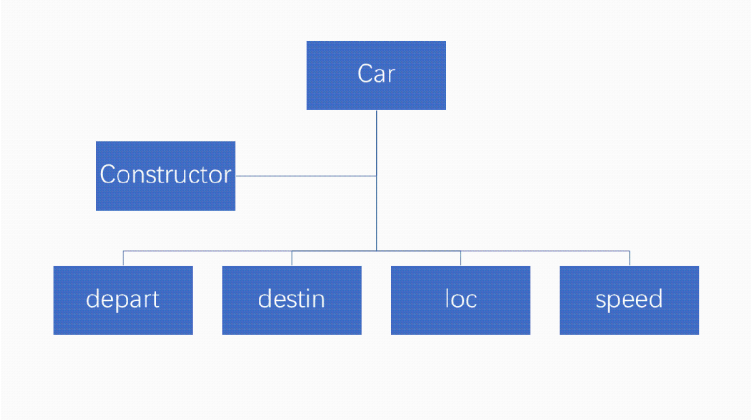

Figure 3: A diagram illustrating the class Car

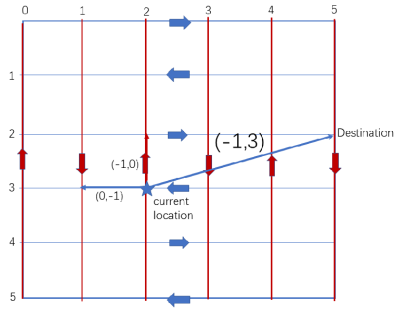

Figure 4: The self-guidance system

4.2 Crossing rules

A magnified version of the cross is shown on the right of Figure 5. The orange areas with length $2dt \times v_{max}$ on both avenues and streets represent the cross area. Once a car enters this area, the mechanism will be triggered as shown in the left of Figure 5. The second step of the flow chart has already been discussed in Section 4.1. The third step judges if there are cars in the cross zoom on the other street/avenue. If there are no cars, this car will go straight or make the turn as normal. Otherwise, we need to enter the 4th step. The 4th step judges whether the car on the other cross all have zero speeds. If cars on the other road have zero speed, meaning cars on the side are waiting for this car to cross, then this car will go first. Otherwise, this car will stop to let the cars on the other road go first.

5 Simulation result

This section includes the experiments which have been run using the current model. The real Manhattan size-214 by 11 is too big for the personal computer to run, thus this simulation only considers a much small system which is 6 by 6. By varying the value of the incoming rate, the behavior of the total cars in the system evolving with time is shown in Figure 6. When *rate* is lower, 0.2 or 0.3 for example, the car in the system keeps fluctuating. However, when *rate* surpasses a certain critical value, the total number in the system goes up forever which suggests a maximum capacity of the traffic system.

To better see the distribution of the number of cars in the system, 100 times of test with each one run 5e4 seconds. To validate the model, $dt = 0.1, 0.5, 1$ are adopted and plotted along with $rate = 0.3, 0.35$. In

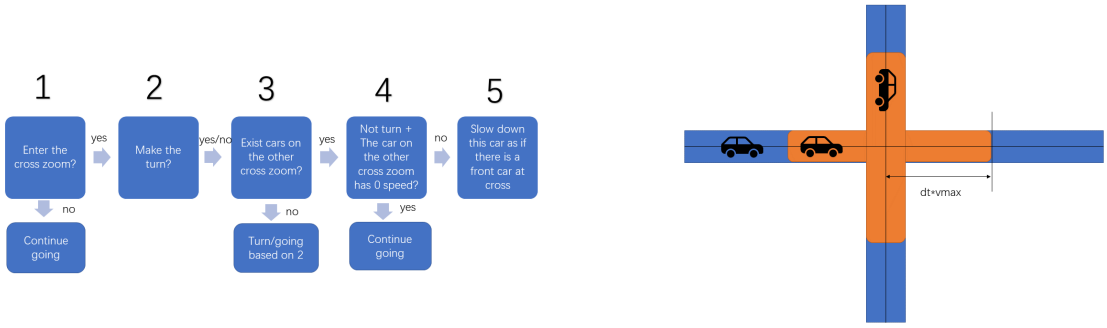

Figure 5: The Mechanism when a car arrives at the cross, the orange area on the right shows the cross zoom

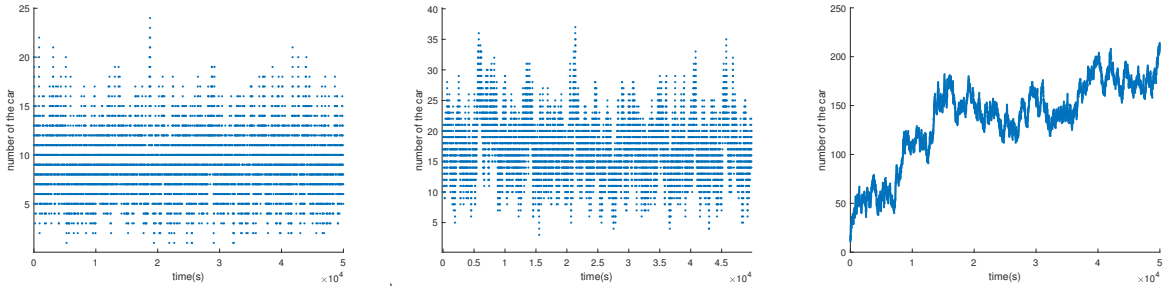

Figure 6: The number of car in Manhattan v.s. time with rate = 0.2, 0.3, 0.4 respectively

Figure 7, the blueish bins are the result from the $rate = 0.3$ while the reddish bins are the result from the $rate = 0.35$. When we take a closer look at the edges at those peaks, the differences in the distribution of different dt is not very significant which serves as a validation of our model.

The above experiments with $rate = 0.3, 0.35$ suggest a critical value for $rate$ which will cause the crackdown of the system. To better show this effect and get the critical value, the $rate$ is randomly generated from 0 to 0.4 repeated several hundred times. The average speed of all the cars and the total number of cars at the steady-state is recorded. Figure 8 shows the final number in the system and their average speed. As we can see from this figure, the average speed of the car suddenly dropped to almost 0 around $rate = 0.38$ and the number of the car also blows up very fast around the same point.

6 Discussion and limitation

The experiments above show a limited capacity for this traffic grid. There are still a lot of questions to be explored using the current model. For example, an even close to the real-life scenario - the morning peak and the evening peak. Since our program allows the assignment of the initial departure location, a distribution of the car could be allocated to mimic the departure from the living area to the working area for the morning peak and vise versa for the evening peak. Also, $rate$ can be a function of time to mimic the real cars generated in Manhattan.

One of the main limitations of this project is - we only build up the single lane in our model which is not the case in real Manhattan. One possible direction is to build up some parts of road as double-way, also put the traffic lights on which could better approximate the real Manhattan. The current model assumes that the length and the width of each block to be the same. Another possible improvement is to make the ratio of the width and the length and width to be adjustable. This adaptation can better approximate Manhattan's

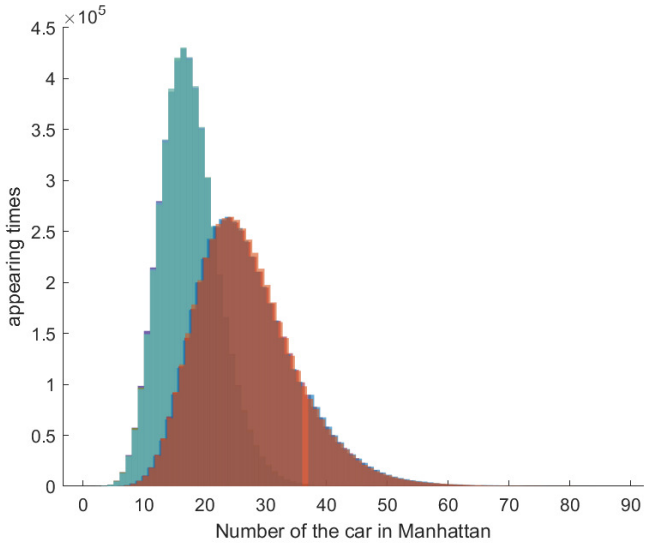

Figure 7: Distribution of the number of cars with rate = 0.3, 0.35

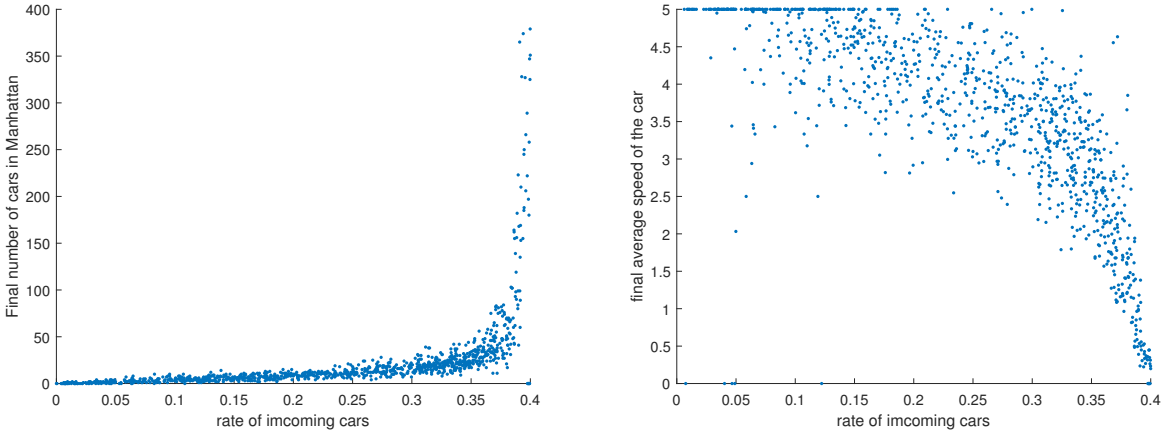

Figure 8: Final numbers of the cars and their speeds

elongated feature.

The traffic grid system is a common feature in American cities, it is not the case in other parts of the world, however. For example, in Beijing or Chengdu, the traffic web is circular with a radiation pattern. There are still some other cities, Chongqing, for example, doesn't have any pattern due to its mountainous and complex terrain. A future project may focus on how to simulate the traffic with any arbitrary structure.

7 Conclusion

To conclude from this project, a simple traffic grid was built to mimic Manhattan's traffic system. The cars in the system were randomly generated and disappear until the destination is reached. Despite the oversimplified feature of the traffic system, the model successfully picked up the critical value which the traffic breaks down for a 6 by 6 grid system. The object-oriented practice facilitates the organization of the data and reduces the time for programming and debugging. Future work for this model may include some more detailed implementation as discussed in Section 6.

References

[1] Wikipedia. *List of numbered streets in Manhattan*. 2021. URL: https://en.wikipedia.org/wiki/List_of_numbered_streets_in_Manhattan (visited on 05/15/2021).

Appendices

A Car.m

```
% Car class define the properties for each car

classdef Car
    properties
        depart(2,1) double % depart location of the car
        destin(2,1) double % destination of the car
        speed(1,1) double  % speed of the car
        loc(2,1) double    % loction of the car
    end

    methods
        function car = Car(Na,Ns,L)
            if nargin == 0
                car.depart = zeros(2,1);
                car.destin = zeros(2,1);
                car.speed = 0;
                car.loc = zeros(2,1);
                return
            end

            % random assign the departure and destination
            car.depart = [randi(Ns)-1;randi(Na)-1];
            car.destin = [randi(Ns)-1;randi(Na)-1];

            % check whether the two are the same, if it is, generate new
            % destin
            while all(car.depart == car.destin)
                car.destin = [randi(Ns)-1;randi(Na)-1];
            end

            car.loc = car.depart*L;
            car.speed = 0;
        end
    end
end
```

B MoveManhattan.m

```
% move the cars in the city in dt time

function [aves,strs] = MoveManhattan1(aves,strs)
global dt vmax L dmax dmin Ns Na;
tol = vmax*dt; % the tolerance considering the two points to be the same

%% Dealing with avenues
for i = 1:length(aves)
    ave = aves{i};% i is the index for the avenue

    if isempty(ave)
        continue
    end

    locs = [aves{i}.loc];
```

```
    locs = locs(1,:); % all the location of the car on this road
    % sorting the locs and aves{i}
    [locs, indx] = sort(locs);
    ave = ave(indx);
    aves{i} = ave;
    delList = []; % the list of element to be deleted
    dir = 1-2*mod(i,2);

    for j = 1: length(locs)
        car = ave(j); % the car on ave i and is j's car
        % check whether the car arrives the destiny
        direc = car.destin*L-car.loc; % vector pointing to the destiny

        if sum(abs(direc))<tol % if the car reaches the destination
            delList(end+1) = j;
            continue
        end

        % calculate the distance to the front car
        if (j+dir<=length(locs))&&(j+dir>0)
            % if the front car exists
            d = abs(locs(j)-locs(j+dir));
        else
            d = dmax;
        end

        if (mod(locs(j),L)>L-tol)||(mod(locs(j),L)<tol) % if the cars arrives at the cross
            r = round(locs(j)/L)+1; % r: the index of the street at the cross
            dir1 = 2*mod(r,2)-1;
%             if (r==Ns&&i==1)||(r==1&&i==Na)||(r==Ns&&i==Na)||(r==1&&i==1)
%                 % arriving at the four conners
%                 delList(end+1) = j; % delete this car on the str
%                 continue
%             end

            if (r==Ns && dir==1) || (r==1 && dir==-1)
                % if car goes beyond the bound of the ave
                car.loc = [(r-1)*L;(i-1)*L+(dir1)*(tol+v(d)*dt)];
                delList(end+1) = j; % delete this car on the ave
                strs{r} = [strs{r},car]; % reappearing on the str
                continue
            end

            if (dir*direc(1)<(dir1)*direc(2))
                % if the car has a higher tendency to go on str

                % decide whether there is car on the other way:
                if ~isempty(strs{r})
                    sloc = [strs{r}.loc];sloc = sloc(2,:);
                    sspeed = [strs{r}.speed];
                    sspeed = sspeed(sloc>(sloc>car.loc(2)-dmin)&(sloc<car.loc(2)+dmin));
                    % speed of cars in the cross zoom on the other road
                    if any((sloc>car.loc(2)-dmin)&(sloc<car.loc(2)+dmin))&&...
                            (mod(locs(j),L)~=0)&&~any(sspeed==0)
                        % if any car is in the entering zoom on the other road
                        aves{i}(j).speed = v(abs(car.loc(1)-(r-1)*L));
                        aves{i}(j).loc = [locs(j)+dt*aves{i}(j).speed*dir;(i-1)*L];
                        continue
                    end
                end

                % otherwise, enter the other road:
                car.loc = [(r-1)*L;(i-1)*L];
                delList(end+1) = j; % delete this car on the ave
                strs{r} = [strs{r},car]; % reappearing on the street
            else % else: move the car out of the cross zone
                aves{i}(j).loc = [locs(j)+(dt*v(d)+tol)*dir;(i-1)*L];
            end
```

```
        else % if not at the cross, continue going
            aves{i}(j).speed = v(d);
            aves{i}(j).loc = [locs(j)+dt*v(d)*dir;(i-1)*L];
        end

    end
    if ~isempty(delList)
        aves{i}(delList) = [];
    end
end

%% Dealing with streets
for i = 1:length(strs)
    str = strs{i};% i is the index for the street

    if isempty(str)
        continue
    end
    locs = [strs{i}.loc];
    locs = locs(2,:); % all the location of the car on this road
    % sorting the locs and strs{i}
    [locs, indx] = sort(locs);
    str = str(indx);
    strs{i} = str;
    delList = []; % the list of element to be deleted
    dir = 2*mod(i,2)-1;

    for j = 1: length(locs)
        car = str(j); % the car on str i and is j's car
        % check whether the car arrives the destiny
        direc = car.destin*L-car.loc; % vector pointing to the destiny
        if sum(abs(direc))<tol % if the car reaches the destination
            delList(end+1) = j;
            continue
        end

        % calculate the distance to the front car
        if (j+dir<=length(locs))&&(j+dir>0)
            % if the front car exists
            d = abs(locs(j)-locs(j+dir));
        else
            d = dmax;
        end

        if (mod(locs(j),L)>L-tol)||(mod(locs(j),L)<tol) % if the cars arrives at the cross
            r = round(locs(j)/L)+1; % r: the index of the ave at the cross
            dir1 = 1-2*mod(r,2);

%             if (r==Na&&i==1)||(r==1&&i==Ns)||(r==Na&&i==Ns)||(r==1&&i==1)
%                 % arriving at the four conners
%                 delList(end+1) = j; % delete this car on the str
%                 continue
%             end

            if  (r==Na && (dir)==1) || (r==1 && (dir)==-1)
                % if car goes beyond the bound of the street
                car.loc = [(i-1)*L+(dir1)*(tol+v(d)*dt);(r-1)*L];
                delList(end+1) = j; % delete this car on the str
                aves{r} = [aves{r},car]; % reappearing on the ave
                continue
            end

            if (dir1)*direc(1)>(dir)*direc(2)
                % if the car has a higher tendency to go on ave

                % decide whether there is car on the other way:
                if ~isempty(aves{r})
                    aloc = [aves{r}.loc];aloc = aloc(1,:);
```

```
                    aspeed = [aves{r}.speed];
                    aspeed = aspeed(aloc>(aloc>car.loc(1)-dmin)&(aloc<car.loc(1)+dmin));
                    if any((aloc>car.loc(1)-dmin)&(aloc<car.loc(1)+dmin))&&...
                            (mod(locs(j),L)~=0)&&~any(aspeed==0)
                        % if any car is in the entering zoom on the other road
                        strs{i}(j).speed = v(abs(car.loc(2)-(r-1)*L));
                        strs{i}(j).loc = [(i-1)*L;locs(j)+dt*strs{i}(j).speed*(dir)];
                        continue
                    end
                end

                car.loc = [(i-1)*L;(r-1)*L];
                delList(end+1) = j; % delete this car on the str
                aves{r} = [aves{r},car]; % reappearing on the ave

            else % else: move the car out of the cross zone
                strs{i}(j).loc = [(i-1)*L;locs(j)+(dt*v(d)+tol)*(dir)];
            end
        else % if not at the cross, continue going
            strs{i}(j).speed = v(d);
            strs{i}(j).loc = [(i-1)*L;locs(j)+dt*v(d)*(dir)];
        end

    end
    if ~isempty(delList)
        strs{i}(delList) = [];
    end
end
end
```

C BuildeManhattan.m

```
% build the traffic system in Manhattan

global Na Ns initNc cararr L;
aves = cell(1,Na);
strs = cell(1,Ns);
for i = 1:Na; aves{i} = {}; end % initialize for avenues
for i = 1:Ns; strs{i} = {}; end % initialze for streets

% initialize the cars in Manhattan
if initNc >=1

    while size(cararr,2)<=initNc
        car = Car(Na,Ns,L);
        loc = car.depart;
        while any(all(cararr==loc)) % if the new generated loc already exist
            car = Car(Na,Ns,L);
            loc = car.depart;
        end
        cararr = [cararr,loc];

        if randi(2)==1 % half chance to put this car on the Avenue
            aves{loc(2,1)+1} = [aves{loc(2,1)+1},car];
        else           % another half chance to put it on the street
            strs{loc(1,1)+1} = [strs{loc(1,1)+1},car];
        end
    end
end
```

D RunManhattan.m

```
% Run manhattan


global Na Ns L dt initNc cararr dmin dmax vmax;
Na = 4;     % number of avenues
Ns = 4;     % number of streets
cararr = [Na+1;Ns+1];
rate = 0.35; % incoming rate
dt = 1;      % (second)
initNc = 10; % initial number of car in the city
L = 50; % the length of each block
dmin = 10; % distance within which car stops (meters)
dmax = 100; % distance above which cars goes at max speed (meters)
vmax = 5; % maximum speed (meters/second)
cmax = Na*Ns*L*L/dmin/10;
tmax = 5e4; %time of simulation (seconds)
clockmax=ceil(tmax/dt);

% initialize the city and see it
BuildManhattan
ViewManhattan(aves,strs)
Nsave = [];
speedsave = [];

for i = 1:clockmax
    if rate*dt<1
        if rand<rate*dt % generate a car if the rate satisfies
            [aves,strs] = addCar(aves,strs);
        end
    elseif rate*dt>=1
        % The integer part of the car number
        for j = 1:floor(rate*dt)
            [aves,strs] = addCar(aves,strs);
        end

        if rand<rate*dt-floor(rate*dt)
        % generate a car if the rate satisfies
            [aves,strs] = addCar(aves,strs);
        end
    end

    [aves,strs] = MoveManhattan1(aves,strs);
%     if mod(i,100)==0
%          ViewManhattan(aves,strs)
%         drawnow
%     end

    %get the total number of cars and their average speed
    [N,speed] = getprops(strs,aves);

    if N>cmax
        error("Too many cars on the road!")
    end

    Nsave = [Nsave,N];
    speedsave = [speedsave, speed];

end
```