A General Simulation Model for Traffic Intersection Performance

Brendan P. Hogan
CSCI 710 Report

April 29, 2002

Acknowledgements

The author would like to recognize the contribution of a couple of individuals without whose help this project would not have been possible. To Professor Brian Park of the Virginia Transportation Research Council and the University of Virginia Civil Engineering Department, much thanks for your countless emails providing traffic modeling advice and comments on my work. To Professor Larry Leemis of the College of William and Mary Mathematics Department, your continued guidance and encouragement over the past months has been outstanding. Thank you.

Abstract

The objective of this project is to develop a simulation model for the analysis of traffic intersection performance. The model was built in a general way so that changes to an intersection, and the analysis of different types of intersections, can be easily supported. Factors considered in the model include the physical characteristics of an intersection, such as the number of lanes in each direction, and the turning classification of each lane (e.g. straight or for turn only). Also modeled are the arrival process of vehicles to the intersection and the traffic signal algorithm used in controlling traffic flow. The primary output from the simulation model is the wait time for vehicles to get through an intersection. Statistical measures calculated include the mean, median, and 95th-percentile of wait times, as well as the cumulative distribution function of wait times.

As a test of the simulation model, a steady-state analysis was performed of rush hour conditions at the intersection of Jamestown Road and Route 199 in Williamsburg, Virginia. Model development, experiments completed, and the results of this study are presented.

Contents

1 Introduction **1**

2 Literature Review **2**

3 Model **4**

3.1 Static Variables - Physical Layout 4
3.1.1 Number of Lanes 6
3.1.2 Size of Lanes 6
3.1.3 Turning Information About Lanes 6
3.1.4 Placement of Traffic Sensors 7
3.2 Static Variables - Stochastic Input Model 8
3.2.1 Arrival Process 8
3.2.2 Vehicle Size Distribution 8
3.2.3 Turn Probabilities 9
3.2.4 Light Sequence Algorithm 9
3.3 Dynamic Variables - System State 11
3.3.1 Traffic Signal 11
3.3.2 Cars 13
3.3.3 Cars in the Intersection 13
3.4 Simulation Implementation 16
3.4.1 Pseudo Code 16
3.4.2 Model Output 19

4 Discussion of “good” intersection performance **21**

5 Test Case - Jamestown Road / Route 199 **24**

5.1 Overview 24
5.2 Model 24

5.2.1 Intersection Geometry 24
5.2.2 Stochastic Input Model 26
5.3 Experiments/Results 28
5.3.1 Current Intersection Arrangement 29
5.3.2 Excursion A: Southbound Shared Straight/Right-Turn Lane 34
5.3.3 Excursion B: Southbound Four Lanes 35
5.3.4 Excursion C: Westbound Four Lanes 36

6 Conclusions/Future Work 38

A Program Code 40
A.1 Car Class: car.cpp 40
A.2 CarQueue Class: carQueue.cpp 43
A.3 Simulation Function Definitions: functions.cpp 51
A.4 Simulation Main Program: traffic.cpp 66

Chapter 1

Introduction

The purpose of this project is to provide a tool for the analysis of traffic intersection performance. Factors considered in the study include intersection size and shape, as well as the traffic signal timing algorithm. With a discrete-event simulation, the effects of varying any of these factors can be easily monitored with perfomance measures such as the average throughput time or the average delay time at the intersection.

The scope of this project is limited to four-way signal-controlled intersections. The effect of a network of multiple intersections linked together is not considered in this study. Intersections studied are assumed to be isolated from any neighboring intersections. Consistent with this assumption, a car that reaches the front of its lane of traffic and has a green signal is assumed to be able to leave the intersection with no problem. Exit traffic from the intersection is not modeled. At present the model assumes that there is at most one lane from which cars can turn left and one lane from which cars can turn right for each direction of traffic. Lanes from which cars can go straight are assumed to be infinite in length. Lanes from which cars can turn right are also assumed to be infinite in length. That is, the number of cars that can fit in straight or right turn lanes are not limited by their size. The model can be expanded to include more specialized intersections, but this is the current state of the simulation program. Also, at present the simulation code only incorporates fixed signal timing plans. However, the code has been written in such a way that an actuated signal timing plan, based on cameras or metal detectors, for a specific intersection can be easily implemented.

Chapter 2

Literature Review

A large amount of work has been done in studying and trying to improve traffic intersections. For a good overview of intersection history, terminology, and current techonology, see Black and Wanat [2]. A discussion of many recent traffic analysis tools can be found in Weiss [20].

Originally, much traffic intersection research focused on analytically describing vehicles' delay in terms of the intersection characteristics. One of the most prominent early works is Webster [18]. For more recent analytical delay models and queueing theory approaches see Hurdle [6] and Hagen and Courage [5]. A discussion of more complex intersection models can be found in Meneguzzer [10].

For a detailed study of intersection characteristics and performance, the flexibility of computer simulation is preferable to analytical approaches. Many privately and publicly funded simulation models exist for the purpose of studying traffic networks. A few examples are UTCS, TRANSYT, SCOOT, DITCS, RT-TRACS, and RHODES [2]. Most of these programs are intended as a tool to aid in setting traffic signal timing algorithms for a network of intersections. One program developed exclusively for the purpose of evaluating traffic network management systems is MITSIM, described in Yang and Koutsopoulos [21].

Two simulation studies that focused on optimizing intersection performance with respect to traffic signal phases are Sen and Head [15], and Asante, Ardekani, and Williams [1]. The focus of both of those works is the definition of general rules for setting traffic signal timing algorithms. The primary measure of performace in Sen and Head [15] is the average delay of all vehicles that use an intersection. The primary performance measure in

Asante, Ardekani, and Williams [1] is the average delay among vehicles that turn left at an intersection.

The focus of this work differs from those mentioned above in that this simulation model offers the user the chance for more detailed intersection analysis. Performance measures of an intersection's performance can be split between individual lanes of traffic, distinct turn patterns, or both. Additionally, the advantages and disadvantages of a number of intersection characteristics can be easily measured, including, size, number, and type of lanes, as well as the signal timing algorithm. The author feels the general nature of this model is its biggest asset for intersection analysis.

Chapter 3

Model

This chapter describes the data structures and algorithms necessary to implement the traffic model in the C++ programming language. In my discussion of the program implementation, the distinction is made between static and dynamic variables. Static variables are those that define the size, shape, and other qualities of the intersection that do not change over time. Other static variables describe the input model for a particular intersection. Characteristics such as the stochastic model for arrivals and turn probabilities are input once at the beginning of the study and also do not change with time. Dynamic variables include the data structures necessary to completely describe the state of the intersection at any point in time. These include information about the current state of the traffic light and the cars waiting at the intersection. These characteristics change dynamically with time, and the variables describing them are updated accordingly.

3.1 Static Variables - Physical Layout

This section will explain the meaning of the static variables used in the model to describe the physical layout of an intersection. The example intersection in Figure 3.1 is used to illustrate the purpose of each of the variables as they are discussed.

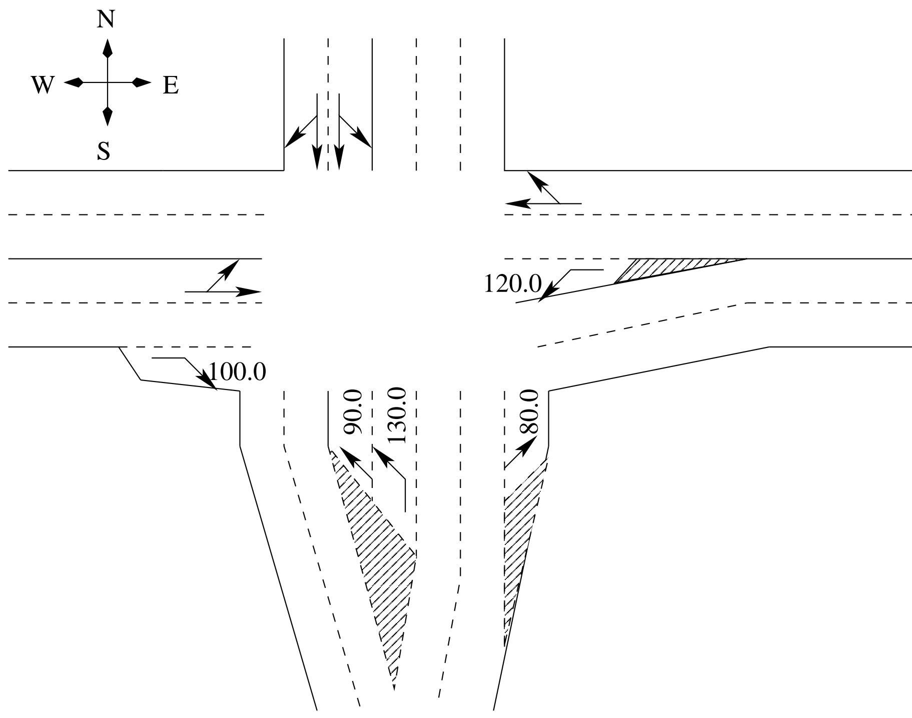


Figure 3.1: Sample intersection - static information

3.1.1 Number of Lanes

The integer-valued vector `Lanes` gives the number of approaching lanes at the intersection for each of the four directions. By convention, this vector and other data structures refering to the directions of traffic will ordered as follows: northbound, southbound, eastbound, westbound. The example in Figure 3.1 would be represented as `Lanes` $= \{5, 2, 3, 3\}$. The integer `maxLanes` gives the maximum number of lanes from any direction of traffic. It is computed as `maxLanes` $= \max($`Lanes`$)$. For this example, `maxLanes` $= 5$.

3.1.2 Size of Lanes

The floating point-valued matrix `Geometry` gives the size of each lane in feet measured from the corner of the intersection backwards. This matrix has dimensions $4 \times$ `maxLanes` and includes information about each lane in each direction of traffic. Lanes that are merely a turning bay will inherently have a finite length, lanes that are straight will be considered to have infinite length. The elements of the `Geometry` matrix corresponding to lanes that do not exist will have a value of 0.0. Then for the example in Figure 3.1 this matrix is

$$\texttt{Geometry} = \begin{bmatrix} 90.0 & 130.0 & \infty & \infty & 80.0 \\ \infty & \infty & 0.0 & 0.0 & 0.0 \\ \infty & \infty & 100.0 & 0.0 & 0.0 \\ 120.0 & \infty & \infty & 0.0 & 0.0 \end{bmatrix}.$$

Note that for this matrix the order for the directions of approaching traffic (rows) is Northbound, Southbound, Eastbound, Westbound. Within each row of the matrix the lanes are represented in the order of left to right from the perspective of a car traveling in that direction. A sequence of 0.0 entries fills out each row. Similar data structures containing information about a direction of travel will be ordered the same way.

3.1.3 Turning Information About Lanes

To make the distinction between lanes that are for turn only, straight only, or straight/turn, the integer-valued matrix `Direction` contains the appropriate

information for each lane. `Direction` also has dimensions $4 \times$ `maxLanes`. By convention, a value of 0 indicates straight only, 1 indicates straight/turn, and 2 indicates turn only. Positive values indicate right turn while negative values indicate left turn. The elements of the `Direction` matrix corresponding to lanes that do not exist will have a value of 99. Then for the example in Figure 3.1 this matrix will be

$$\texttt{Direction} = \begin{bmatrix} -2 & -2 & 0 & 0 & 2 \\ -1 & 1 & 99 & 99 & 99 \\ -1 & 0 & 2 & 99 & 99 \\ -2 & 0 & 1 & 99 & 99 \end{bmatrix}.$$

Therefore, a car approaching from direction `i` and desiring to turn left will need to be in a lane `j` such that $-2 \leq \texttt{Direction}[\texttt{i}][\texttt{j}] \leq -1$. Likewise, cars going straight need to be in a lane `j` such that $-1 \leq \texttt{Direction}[\texttt{i}][\texttt{j}] \leq 1$, and cars turning right need a lane `j` such that $1 \leq \texttt{Direction}[\texttt{i}][\texttt{j}] \leq 2$.

3.1.4 Placement of Traffic Sensors

To capture the quality of data sensors used for actuated control of an intersection there is a floating point-valued matrix `Sensor` that stores the information for each lane in the intersection. The assumption here is that the type of sensor used is a metal induction loop which extends from the corner of the intersection a certain distance backwards in each lane. This technology is not able to distinguish between one car and ten cars. It makes a simple boolean decision either yes, there is traffic waiting, or no, there is not. The dimensions of these induction loops are typically $6' \times 30'$ or $6' \times 40'$ [14]. Then the element in `Sensor` corresponding to approach direction `i` and lane `j` will contain the length (if any) of the induction loop for that lane.

Note that since the current study was a steady state analysis of rush hour conditions (see Section 5.2 for details), data sensors and dynamic control of the intersection are not included in the model. However, if dynamic signal control of the intersection were to be implemented in the future, advanced sensor systems incorporating more than the simple boolean detection described above could be easily modeled. This is due to the fact that data including the size and current position of every car in the intersection is stored by the implementation. For more details see the discussion of dynamic variables in Section 3.3.

3.2 Static Variables - Stochastic Input Model

The physical attributes comprise only one portion of the static description of the intersection. The remaining portion describes the four probabilistic aspects of the model: (1) the arrival process for each of the four arrival directions, (2) the probabilities that a car is classified as passenger or heavy vehicle, (3) the probabilities that a car intends to turn left, go straight, or turn right for the four arrival directions, and (4) the light sequence algorithm.

3.2.1 Arrival Process

For each direction of approaching traffic to an intersection, there is a stochastic point process model that models the arrival stream. These models are not saved in memory, rather they are included in a `GetArrival` function that generates the next time of arrival based on the approach direction and a specified probability distribution. Only the most imminent time of arrival for each direction is saved in the discrete-event simulation's event calendar.

3.2.2 Vehicle Size Distribution

For each direction of approaching traffic, there is a specific distribution of the types of vehicles that arrive. A motorcycle or a passenger car take up much less room in a lane of traffic than a large delivery truck or a tractor trailer. The effects these vehicles have on a traffic queue are unique and it is important to include this distinction in an intersection model. For the purposes of this model there are two classes of vehicles, passenger cars and heavy vehicles [1]. It is also assumed that passenger cars (and the gap between them) take up 25 feet in the lane and heavy vehicles take up 35 feet [11]. More complicated vehicle distribution schemes can easily be implemented if the simulation in question requires it. For the types of studies handled so far, the implementation of two vehicle classes has been sufficient. The floating point-valued array `SizeProb` captures the input model for the distribution of passenger cars and heavy vehicles. For each direction of approaching traffic `i`, `SizeProb[i]` contains the proportion of arriving vehicles that are considered passenger cars. Then $(1 - \texttt{SizeProb}[\texttt{i}])$ contains the corresponding proportion of heavy vehicles.

3.2.3 Turn Probabilities

For any direction of approaching traffic, a certain proportion of cars will turn left, go straight, or turn right. This information is stored in the 4×2 floating point-valued matrix `TurnProb`. Each row in the matrix corresponds to a specific direction of approaching traffic (i.e. Northbound, Southbound, Eastbound, or Westbound). The first column corresponds to the proportion of cars that turn left. The second column corresponds to the cumulative proportion of cars that turn left or go straight. Since this matrix contains cumulative turn information, a third column to represent right turns is unnecessary since the cumulative probability of turning left, going straight, or turning right will be 1.0 in all situations.

$$\texttt{TurnProb} = \begin{bmatrix} 0.20 & 0.70 \\ 0.30 & 0.60 \\ 0.15 & 0.85 \\ 0.21 & 0.76 \end{bmatrix}$$

In the example above, northbound cars have a 0.20 probability of turning left, a $0.70 - 0.20 = 0.50$ probability of going straight, and a $1.0 - 0.70 = 0.30$ probability of turning right. The turn probabilities for the other directions are calculated in the same way.

3.2.4 Light Sequence Algorithm

For any intersection that is modeled, it is necessary to describe the algorithm by which the traffic signal is updated. Before discussing the computational methods for implementing this, consider the behavior of traffic during the respective signal settings of green, yellow, and red. When a signal first turns green, there is a delay of a couple seconds before the first car in line accelerates and leaves the intersection. In the civil engineering community, this is referred to as the start-up lost time. When a signal changes from green to yellow, there are a couple seconds during which cars continue to pass through the intersection. This is referred to as the end-gain time. During the time between the start-up lost time and the end-gain time, the signal is effectively green and cars leave the intersection at a steady rate. This rate of discharge from the intersection is the saturation flow rate, typically measured in units of vehicles per hour per lane. The time that the signal is not effectively green consists of a portion of yellow light and a portion of red light. As far

as this simulation model (and for that matter a driver at an intersection) is concerned, that time is effectively red and no cars leave the intersection. The relationship between actual signal settings and the effective result on an intersection is graphically described in Figure 3.2. For the remainder of this report, the word "green" will be used to refer to the time when the signal is effectively green. The word "red" will be used to refer to all other times. As an intermediate (yellow) phase is not included in the model, it will not be refered to for the remainder of this report.

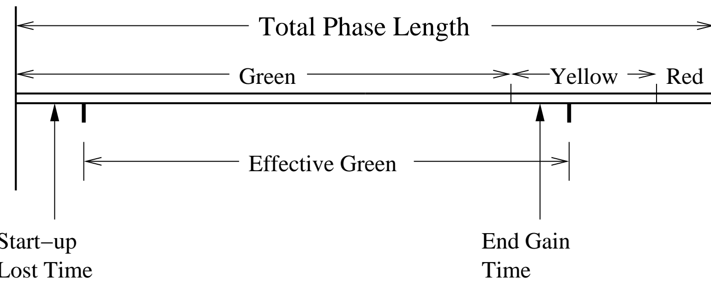


Figure 3.2: Signal settings vs. effective phases

The light sequence algorithm for an intersection consists of a definition of the signal phases that are possible, an ordering of those phases, and the length of each phase. Since most intersections have traffic sensors that fine tune the signal settings based on the presence of cars, the light sequence algorithm defines these more complicated interactions as well. For example, each signal phase may have a minimum and maximum length, and the actual time within that range that is alloted to the phase depends on whether cars are detected by the sensors. Another example would be the case when a specific left turn phase is only included in the cycle if the presence of vehicles in the left turn lane is detected by the sensors. A simpler light sequence algorithm would have fixed length signal phases that are not subject to feedback from the traffic sensors. For the example intersection in Figure 3.1, a feasible light sequence algorithm is described in Figure 3.3.

A discussion of the implementation details of the light sequence algorithm is heavily dependent on the simulation event structure defined in Section 3.4 and the dynamic variables defined in Section 3.3. Therefore the treatment of the computational techniques involved is deferred to later in this paper.

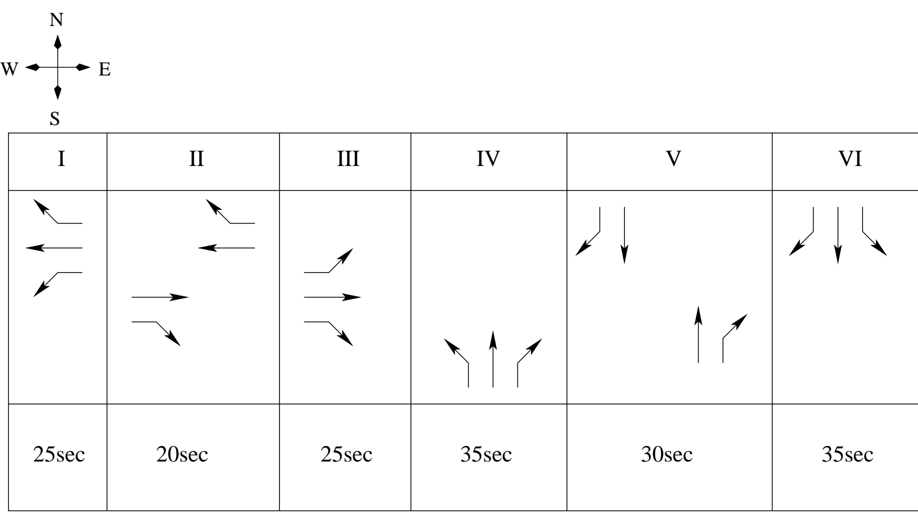


Figure 3.3: Example light sequence algorithm

3.3 Dynamic Variables - System State

The dynamic variables in the simulation are those that are necessary to completely decribe the state of the intersection at any point in time. Section 3.3.1 describes how the traffic signal settings are captured in the simulation, Section 3.3.2 describes the data that is kept for each individual car in the intersection, and Section 3.3.3 describes how the information for each direction and lane of the intersection is stored in memory. Due to the discrete-event nature of this simulation, the appropriate dynamic variables will be updated as each event occurs. The methods by which these variables are updated are discussed later in Section 3.4.

3.3.1 Traffic Signal

The boolean matrix `Light` contains information on which lanes of traffic have red or green lights. A value of 0 indicates a red signal and a value of 1 indicates a green signal. This $4\times$`maxLanes` integer-valued matrix is organized according to the approaching direction of traffic and the lanes within that direction in the same manner as the static variables. The elements of the `Light` matrix corresponding to lanes that do not exist will always have a value of 0. Then for the snapshot of the intersection shown in Figure 3.4,

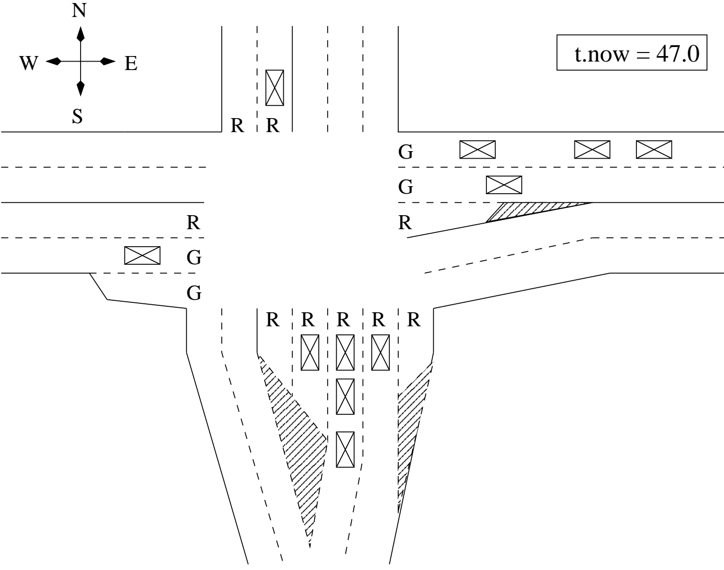


Figure 3.4: Sample intersection - snapshot of dynamic information

this matrix is

$$\texttt{Light} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix}.$$

Updating the Traffic Signal

As described in Section 3.2.4, the specific signal phases an intersection contains, and the ordering of those phases are included in the input model. The way in which the simulation handles this is through the C++ functions `Change2Red` and `Change2Green`.

If a lane of traffic had a green signal in a previous phase and a red signal in an upcoming phase, the function `Change2Red` will turn it red. That is, it will set $\texttt{Light}[\texttt{i}][\texttt{j}] = 0$ for the appropriate direction `i` and lane `j`.

Likewise, if a lane of traffic had a red signal in a previous phase and a green signal in an upcoming phase, the function `Change2Green` will turn it

green. That is, it will set `Light[i][j]` = 1 for the corresponding direction `i` and lane `j`.

Lanes of traffic that have either a continued green signal or a continued red signal from one phase to the next will be left alone by these two functions. For more detail on the algorithm, see the pseudo-code in Section 3.4 or the program listing in Appendix A.

3.3.2 Cars

The state of any car in the simulation which is at the intersection is defined by the following characteristics.

- Arrival time to intersection
- Turning Intention (left turn, straight, right turn)
- Current distance (in feet) from the intersection
- Vehicle length

The data structure for a car is implemented using a C++ class `Car` with four private data members corresponding to the four characteristics above. Note that information about the car's approaching direction of travel and current lane within that direction are not stored here. Instead they are implied by the location where each specific `Car` data structure is stored.

3.3.3 Cars in the Intersection

At any point in the simulation the implementation must keep track of a possible queue of cars in each lane of the intersection. The conceptual organization of this information is shown in Figure 3.5.

The implementation of this queue is handled by the combination of different data structures. The queue of traffic for each lane of the intersection is implemented using a C++ class `carQueue`. There are two private data members in this class: an integer that gives the size of the queue at any time, and a pointer to the last car in the queue. This class is a circular queue structure in which the data members are items from the `Car` class. The pointer-valued matrix `Queue` with dimensions $4 \times$ `maxLanes` contains references to the memory location of the `carQueue` element for each lane. This matrix is organized

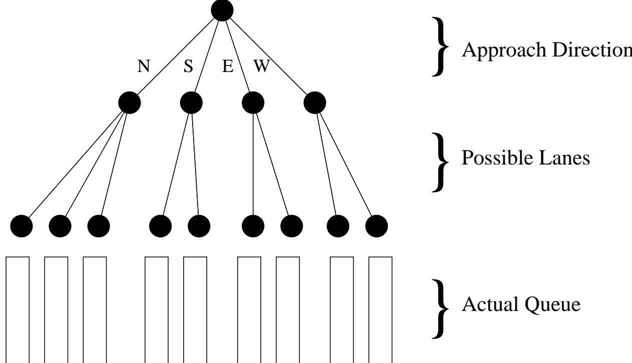


Figure 3.5: Conceptual Queue Organization

in the same manner as the other data structures and can be referenced by direction and lane.

For the example shown in Figure 3.4, the information stored about the current state of the intersection is shown below. In this `Queue` matrix, an "&" symbol indicates a pointer to the location in memory where data on the cars in the specific lane are stored. A "–" symbol indicates a NULL pointer, that there are no cars in that element of the queue, or that that lane does not exist.

$$\texttt{Queue} = \begin{bmatrix} 0/- & 1/\& & 3/\& & 1/\& & 0/- \\ 1/\& & 0/- & - & - & - \\ 0/- & 0/- & 0/- & - & - \\ 0/- & 0/- & 0/- & - & - \end{bmatrix}$$

An example of a `carQueue` class from the intersection snapshot of Figure 3.4 is shown in Figure 3.6. The `carQueue` element shown corresponds to the Northbound middle lane, which is `Queue[0][2]`. Note that each node in the linked list corresponds to an element of the `Car` class defined in the previous section.

Cars arriving to a lane with no traffic queue built up and a green signal are not stored in a queue. They simply move through the intersection and a counter of the total number of cars served is incremented.

Regarding the cars currently at the intersection, the floating point-valued matrix `NextDeparture` contains the time of next departure from each individual lane. This is a also a $4 \times$ `maxLanes` matrix and is referenced by direction

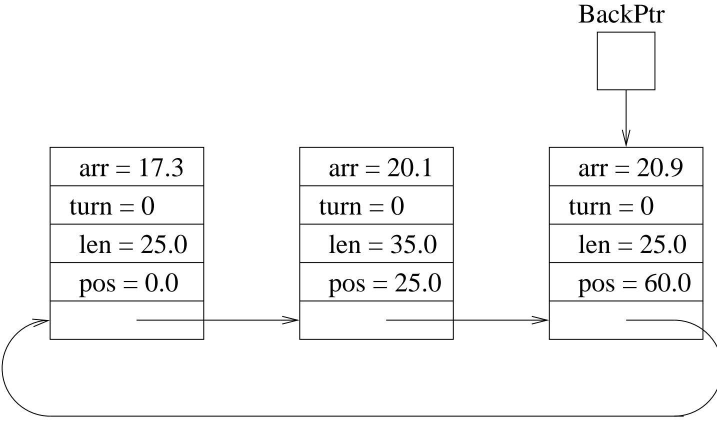


Figure 3.6: Northbound middle lane `carQueue` element from Figure 3.4

and lane. Keeping track of this information will simplify the process of moving cars through the intersection since for any row of the matrix (direction of traffic) the column with the minimum value will indicate the lane from which the next car will leave. The `NextDeparture` matrix is only updated and checked for lanes of traffic that have a green signal. For lanes that have a red signal, no departure will occur until after the signal is changed. So there is no need to keep the departure matrix up to date for those lanes. Lanes of traffic that either do not exist in the intersection, have a red signal, or have no cars currently in them are assumed to take on an infinite `NextDeparture` time.

Suppose that the snapshot of an intersection given in Figure 3.4 occured when the simulation clock was at `t.now` = 47.0. Then the `NextDeparture` matrix corresponding to the state of that intersection would be given by

$$\texttt{NextDeparture} = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 48.2 & \infty & \infty & \infty \\ \infty & 51.1 & 49.6 & \infty & \infty \end{bmatrix}.$$

Table 3.1: Event List

0	`t`	`x`	Northbound Arrival
1	`t`	`x`	Southbound Arrival
2	`t`	`x`	Eastbound Arrival
3	`t`	`x`	Westbound Arrival
4	`t`	`x`	Pass Through
5	`t`	`x`	Signal Change-2-Red
6	`t`	`x`	Signal Change-2-Green

3.4 Simulation Implementation

The intersection is modeled using a next-event simulation [12]. That is, the simulation only considers a set of finite events that can change the state of the system. The state of the system is constant between occurences of these events. For that reason the simulation clock advances in discrete steps to the time of next occurence for one of these events. The program contains modules that update the state of the system according to which event occured, and the next occurence of the same type of event is scheduled for some time in the future. The event list, or calendar, for this simulation is shown in Table 3.1.

The `t` field in each event structure is the scheduled time of next occurence for that event. The `x` field is the current activity status of the event. This variable is a boolean indicator of whether the event has the possibility of occuring at a given point in time. For example, if the only cars at an intersection are sitting at a red signal, there is no sense continually checking if a `Pass Through` will be the next event. The event will be set to impossible (`event[4].x = 0`) until a `Signal Change-2-Green` occurs, which would then make a `Pass Through` event possible.

3.4.1 Pseudo Code

This section outlines in pseudo code the simulation algorithm. The emphasis in this pseudo code is on explaining the flow of vehicles through the intersection, and detailing how each type of event (`e = 0...6`) is processed. The details regarding the calculation of output statistics are not explained here for the sake of keeping this code as neat as possible. Instead, this explanation is deferred to Section 3.4.2.

```
While (t.now < STOP) {
    e ← {i*|event[i*].t ≤ event[i].t; i = 0, ..., 6}          [Get next event]
    t.now ← event[e].t                                        [Update simulation clock]
    If (e < 4) {                                              [Process an Arrival from direction e]
        event[e].t ← t.now + GetArrival(e)                    [Schedule next arrival]
        NewCar.arr ← t.now                                    [Save car's arrival time]
        NewCar.turn ← PickDirection(e)                        [Assign turn direction]
        NewCar.len ← PickSize(e)                              [Assign length of vehicle]
        newLane ← PickLane(Queue, NewCar)                     [Place in appropriate lane]
        if (Light[e][newLane] = 1)                            [If green signal]
            Stats[e][newLane].count[0] + +                    [Increment frequency count]
        else                                                  [If red signal]
            Queue[e][newLane].insert(NewCar)                  [Save car to memory]
    }
    else if (e = 4) {                                         [Process a Departure]
        PassThru()                                            [See documentation]
        For all (i, j) ∋ Light[i][j] = 1                      [Check green lanes for next departure]
           (approach, lane) ← {(i*, j*)|NextDeparture[i*][j*] ≤ NextDeparture[i][j]}
        nextTime ← NextDeparture[approach][lane]
        If (!Queue[approach][lane].isEmpty())                 [If more waiting]
            event[4].t ← nextTime                             [Schedule next departure]
        else
            event[4].x ← 0                                    [Turn departure off]
    }
    else if (e = 5) {                                         [Change signal to red]
        For all i, j
            If (Light[i][j].OldPhase = 1)&(Light[i][j].NewPhase = 0)
                Light[i][j] ← 0                               [New red]
        event[5].x ← 0                                        [Turn off Change to red]
        event[6].t ← t.now + blockTime                        [Schedule Change to green]
        event[6].x ← 1                                        [Turn on Change to green]
    }
    else {                                                    [Change signal to green]
        For all i, j
            If (Light[i][j].OldPhase = 0)&(Light[i][j].NewPhase = 1)
                Light[i][j] ← 1                               [New green]
        event[6].x ← 0                                        [Turn off Change to green]
        event[5].t ← t.now + phaseLength                      [Schedule Change to red]
```

$event[5].x \leftarrow 1$ [Turn on Change to red]
}
} [End While loop]

Further discussion of two significant subroutines are required at this point.

PickLane The `PickLane` function is used to determine which lane of an approach direction a car should be placed in. This decision is dependent on a few factors: turning intention of the vehicle, size of the vehicle, size and turning classification of the possible lanes, and the current state of the traffic queue.

For cars that are traveling straight, this decision is generally easy since the straight lanes are assumed to be infinite in length. The car is simply placed in the lane with the shortest line of traffic. Ties are broken by selecting the left-most from among the possible lanes.

For cars that are turning right, this decision is also trivial since at present the model assumes that right turn lanes are also infinite in length.

For cars that are turning left, the presence of finite sized left turn only lanes complicates this question. Obviously, if there is room in the turning lane and access to that lane, the car is placed there. If there is not access to that lane (i.e. traffic is backed up preventing the car from reaching the turning lane) or if there is no room in the turning lane, the car is placed in the adjacent lane as close as possible to the entrance to of the turning lane.

The programming logic for implementing this is rather complicated, see Appendix A.3 for the complete details.

PassThru The `PassThru` function serves to remove a car from the intersection, and update the position of any cars that may be behind it in line, and schedule the next departure time for that lane of traffic if necessary. As mentioned above, it is possible for a lane of traffic to be blocked due to the finite length of turning lanes. Also, a departing car may cause a previously blocked lane of traffic to become unblocked. When updating the position of remaining cars in a lane, the `PassThru` function must check for and handle both of those possible situations. The complete programming details are given in Appendix A.3, if interested.

3.4.2 Model Output

The purpose of a simulation is to gain insight about the system being modeled. To that end, there are a number of performance measures reported by the program at the end of a simulation experiment. For each performance measure discussed here there is the possibility to view and compare output from the whole simulation, individual lanes of traffic, individual turning patterns, or a combination of the above.

Each performance measure concerns the wait time of vehicles at the intersection. The statistics that are calculated are the mean, median, and 95th-percentile of the wait times. See Section 4 for a discussion of the pros and cons of each performance measure and why it is advantageous to consider more than one of these. In an effort to capture the whole picture regarding intersection performance, the cumulative distribution function of wait times is calculated as well and plotted from the output of a simulation experiment. Note that the actual calculation of these performance measures is done by interfacing the data from the C++ simulation with the data analysis and programming language S-Plus.

Computing these statistics is accomplished by keeping wait time data for every vehicle in the simulation. The histogram-type integer-valued vector `count[maxTime]` is used to store this information[1]. The integer `maxTime` is the largest wait time recorded by the simulation and is set by the user depending on the experiment at hand. By default, any wait times that may be greater than `maxTime` are considered equivalent to `maxTime` for the purpose of collecting statistics. For any $i = 0, 1, \ldots, maxTime - 1$, $count[i]$ is the number of vehicles that had wait time, wt, such that $i \leq wt < i+1$. In other words, each time a vehicle departs the intersection, the wait time is recorded by the statement $count[min(\lfloor wt \rfloor, maxTime-1)]++$. This discussion refers to the case of collecting statistics on every car in the simulation. The same approach is just as easily applied to compute statistics for more specific studies such as comparing the performance of the left, middle, and right lane of an approach direction, or comparing the performance of left turns versus all other movements. The specific breakdown of the analysis is dictated by the intersection being modeled, and the questions that hoped to be answered by the study.

Since it is histogram-type count data that is being recorded, the sam-

[1]Much thanks to Prof. Weizhen Mao, Computer Science Department, College of William and Mary, for the suggestion of this clever data collection mechanism

ple mean can be calculated according to the method of histogram means described in Park and Leemis [12]. That is,

$$\bar{t} = \frac{1}{N} \sum_{i=0}^{maxTime-1} i \cdot count[i],$$

where N is the total number of observations recording in *count*.

The empirical cumulative distribution function of wait times can be calculated from the histogram by starting at zero and adding the contribution of $\frac{1}{N} count[i]$ for each i such that $count[i] > 0$. Let $\hat{F}(i)$ represent the empirical CDF of the wait times at any (integer) time value i. Then the median, $t_{0.5}$, is the minimum i such that $\hat{F}(i) \geq 0.5$. Likewise, the 95-th percentile is the minimum i such that $\hat{F}(i) \geq 0.95$.

Chapter 4

Discussion of "good" intersection performance

As detailed in the previous section, the performance measures from the model include the mean, median, 95th-percentile, and cumulative distribution function of the wait times from cars in the simulation. It is appropriate at this point to discuss what can be implied from each of those statistics, the pros and cons of each as a measure of intersection performance, and the benefit of considering multiple of these measures. The following sections explain these points.

The sample mean is probably the most classic measure of central tendancy. However it offers little information about the distribution of the underlying population.

The sample median is another common measure of central tendancy. It has the nice feature of being inherently linked to the distribution of a population as 50% of the population is below the median and 50% is above. Similar to the sample mean, the median provides little information about the tails of a distribution. When considering a real world system, as in the case of a discrete-event simulation, the tails of a distribution are often the most important part of the model [12]. To put this in context of a traffic intersection, a driver that waits for the mean or median amount of time is pretty indifferent about their experience at that intersection. On the other hand, a driver that experiences a wait time from one of the tails of a distribution is going to be much more excited (for better or for worse) than the previous driver that fell in the center of the distribution. This is why it is desireable to acquire knowledge about the tail behavior of a system, in addition to measures of

central tendency.

The 95th-percentile wait time is the measure used in this study as it seems like a reasonable guage of intersection performance. There are always going to be some people who have a bad experience, show up at the intersection at the wrong time and have an unusually long wait. Minimizing the 95th-percentile of the wait times is equivalent to trying to please 95% of the users of an intersection, not a bad objective.

The cumulative distribution function of wait times gives the whole picture of the proportion of cars that short wait times and long wait times. It is still beneficial to have the numerical measures of the mean, median, and 95th-percentile for ease of comparison between systems. The CDF of wait times will be used in some cases for visualizing the results of a simulation.

Consider the CDFs of wait times for the two hypothetical Systems A and B shown in Figure 4.1. The median and 95th-percentile wait times for each system are sketched on the graph as well. The two systems have identical 95th-percentile wait times, but vastly different distributions aside from that point. If you were a driver with a choice between the two Systems A and B, which one would you choose? System A has a much better median wait time than system B. However, System A also has the chance of producing a much worse wait time than System B ever would. Considering that drivers who experience about the median wait time are generally apathetic about the situation, where drivers who wait the maximum amount of time are often extremely angered, it may be advantageous to play it safe and select System B.

The definition of "good" intersection performance with respect to the statistics considered in this study is not entirely clear. Ideally, each of the mean, median, and 95th-percentile of wait times should be as small as possible to get the most cars through the intersection with as little delay as possible. Clearly, an intersection arrangement that has each of those three statistics smaller than an alternative intersection arrangement should be the preferred choice. What is unclear is which intersection is preferable if the first has superior mean and median wait times, and the second intersection has the superior 95th-percentile wait time. Or which is preferable if the first intersection has a superior median and 95th-percentile of wait times, while the second intersection has a superior mean wait time. The permutation of these three statistics among the available intersection arrangements make this a complicated question. The author suggests that a meaningful description of the trade-offs between the mean, median, and 95th-percentile wait

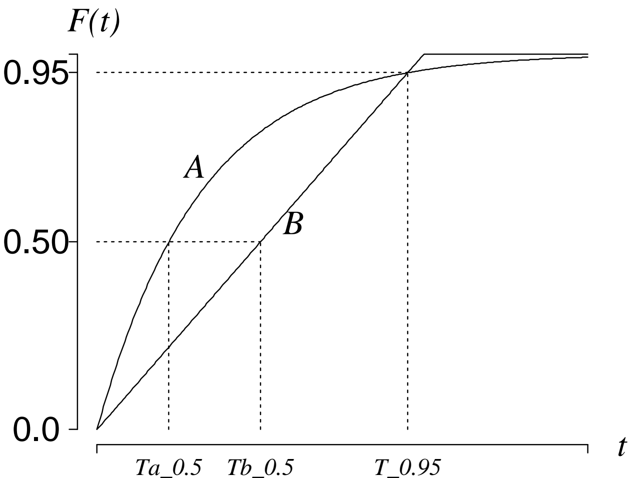


Figure 4.1: Hypothetical wait time CDFs

times may be an open research question. A detailed study of these relationships would depend on the objectives of the researcher. For example, maximizing intersection safety and maximizing driver satisfaction could lead to two different answers to the question of what is the best intersection arrangement. A philosophical discussion of these points is outside the scope of this report. The author merely intends to point out that some ambiguity does exist regarding the definition of a "good" intersection.

Chapter 5

Test Case - Jamestown Road / Route 199, Williamsburg, VA

5.1 Overview

The intersection of Jamestown Road and Route 199 in Williamsburg, Virginia is a major crossroads of the town and a major headache for many travelers. Proposed improvements to the intersection consistently draw the attention of local newspapers. See Virginia Gazette articles "Your turn next at Jamestown Road," Dec. 26, 2001, "Privatizing work a quicker fix for 199," Jan. 19, 2002, and "Green light for intersection plan," Jan. 26, 2002, among others. For those reasons this intersection provides an interesting test case for the simulation model described above.

The following two sections describe how the physical and stochastic elements of the Jamestown/Route 199 intersection are captured using the data structures outlined in Sections 3.1 and 3.2. The next sections describe the experiments performed with the model, their results, and a discussion of what was learned.

5.2 Model

5.2.1 Intersection Geometry

This section describes how the static, physical characteristics of the actual intersection are modeled using the variables defined in Section 3.1. For ref-

erence, a diagram of the intersection is given in Figure 5.1.

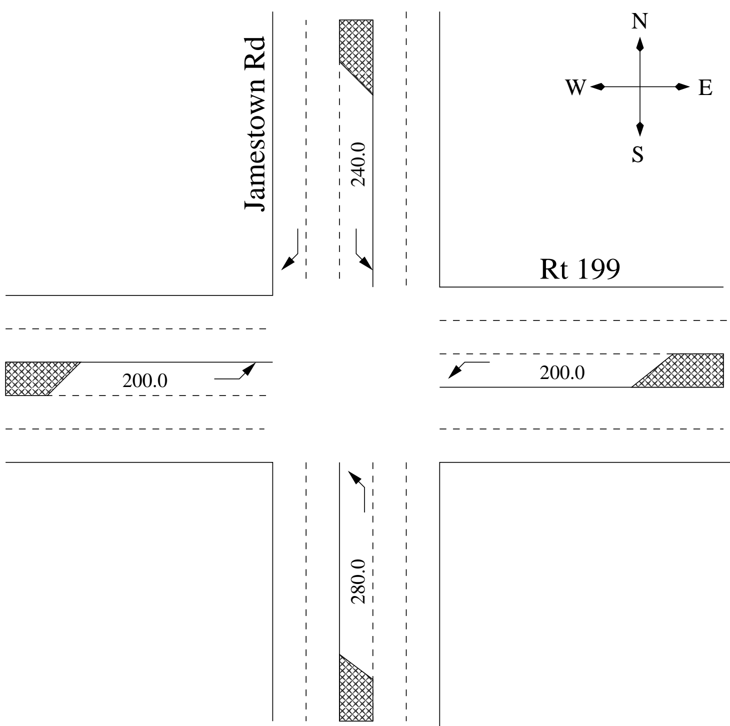


Figure 5.1: Intersection geometry at Jamestown/Route 199

The intersection of Jamestown Road and Route 199 has three lanes of approaching traffic from each of the four directions. Therefore `Lanes` $= \{3, 3, 3, 3\}$, and `maxLanes` $= 3$. The turning characteristics of each lane are defined as

$$\texttt{Direction} = \begin{bmatrix} -2 & 0 & 1 \\ -2 & 0 & 2 \\ -2 & 0 & 1 \\ -2 & 0 & 1 \end{bmatrix}.$$

The size of each lane, measured in feet, is given by

$$\texttt{Geometry} = \begin{bmatrix} 240.0 & \infty & \infty \\ 280.0 & \infty & \infty \\ 200.0 & \infty & \infty \\ 200.0 & \infty & \infty \end{bmatrix}.$$

Note that from the `Direction` matrix and Figure 5.1 it is clear that each approaching direction of traffic has a left-turn only lane. This is consistent with the fact that the left-most lane from each approaching direction (i.e. column 0 in `Geometry`) all have a finite length.

Information about the placement of traffic sensors is not included here as the focus of this study is an analysis of steady state rush hour conditions. The assumption applied is that during rush hour the intersection is constantly busy, traffic will constantly be detected by the sensors, and the signal phases will be appropriately lengthened during each signal cycle. The author feels comfortable making this assumption since rush hour data on the signal changing patterns of the intersection revealed remarkably constant lengths of phases from one signal cycle to the next.

5.2.2 Stochastic Input Model

The stochastic input model was developed from data collected at the intersection in December 2001. All observations were made during the period of the afternoon rush hour from 5:30pm to 6:00pm. The individual data collecting methods and modeling techniques for each component of the input model are discussed in the following sections.

Arrival Process

To model the arrival process, 100 interarrival times were collected from each direction of approaching traffic. The distinction between arrivals to a specific lane, or arrivals with a specific turning intention, is not made at this level. The data of interarrival times were fitted to statistical distributions using the data fitting package in Arena [13]. The output from the Arena package consists of the statistical distribution which provided the best fit to the data, as well as the maximum likelihood estimators for the parameters of that distribution. The results of this process, as well as the mean times between

arrivals, are shown in Table 5.2.2. Note that the parameterization of the log normal distribution that is used here is $lognormal(\mu_l, \sigma_l)$ where

$$f(t) = \frac{1}{\sigma t\sqrt{2\pi}} e^{-\frac{(\log t-\mu)^2}{2\sigma^2}}$$

for $t > 0$. The mean is $\mu_l = e^{\mu+\sigma^2/2}$ and the variance is $\sigma_l^2 = e^{2\mu+\sigma^2}(e^{\sigma^2} - 1)$.

Table 5.1: Arrival process model for each approaching direction

Approach Direction	Distribution	Mean T.B.A. (sec)
Northbound	log-normal(1.3515, 0.8938)	5.76
Southbound	log-normal(1.4495, 1.0647)	7.51
Eastbound	log-normal(0.9069, 0.8745)	3.63
Westbound	log-normal(0.4963, 0.9584)	2.60

It is interesting that the log-normal distribution was selected to model the arrival process from each of the four directions, despite the large difference in traffic volume between them. (Westbound has nearly three times the volume of traffic as Southbound.) It was reassuring to the author to find that the log-normal distribution has been used often times before to model the arrival process in traffic studies. With reference to their simulation study, Asante, et. al. state "a lognormal headway distribution model was used as it represents realistic arrival patterns under a broad range of volumes" [1, p. 133]. Further information about the use of the log-normal distribution in traffic modeling can be found in May [9] and Tolle [17].

Vehicle Size Distribuion

The vehicle size distribution, i.e. the fraction of passenger cars and heavy vehicles, was determined by a count of the number of each classification of vehicles that approached the direction during a period of rush hour conditions. The resulting fraction of passenger cars for each direction of approaching traffic is `SizeProb` $= \{0.93, 0.94, 0.88, 0.95\}$.

Turn Probabilities

The turn probabilities stated in Table 5.2 correspond to the following values of the 4×2 cumulative probability matrix in the simulation.

Table 5.2: Turn Distribution Input Model

Approach Direction	Turn Proportion		
	Left	Straight	Right
Northbound	0.32	0.41	0.27
Southbound	0.19	0.52	0.29
Eastbound	0.14	0.64	0.22
Westbound	0.28	0.62	0.10

$$\texttt{TurnProb} = \begin{bmatrix} 0.32 & 0.73 \\ 0.19 & 0.71 \\ 0.14 & 0.78 \\ 0.28 & 0.90 \end{bmatrix}$$

Light Sequence Algorithm

The signal timing algorithm under rush hour conditions at the intersection is described in Figure 5.2.

5.3 Experiments/Results

An analysis was performed of the steady state rush hour conditions at the intersection of Jamestown Road and Route 199. First, the current arrangement of the intersection geometry and the current signal timing plan were evaluated. The goal in doing this is twofold. First, some confidence in the model is established by comparing simulation results with observations from the actual system being modeled. Once convinced that the model is capturing system behavior adequately, troublesome features from the system can be identified and proposed solutions evaluated. With respect to the intersection being modeled in this study, the goal of a carefull analysis of the current intersection properties is to identify specific lanes of traffic or turning movements that are problems for drivers. Once bottlenecks are identified, modifications to the intersection aimed at easing those bottlenecks can be proposed. The following sections detail the analysis of the current intersection arrangement, the decisions for proposed improvements to the intersection, and the results

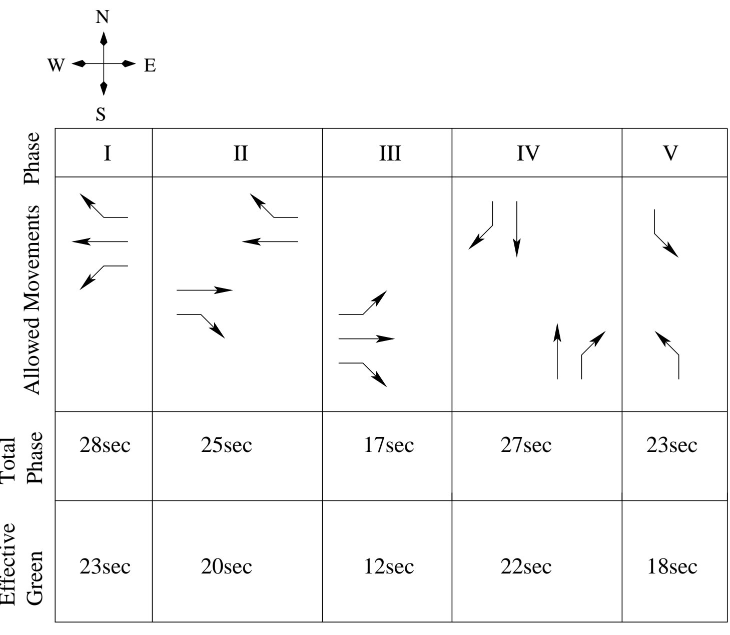
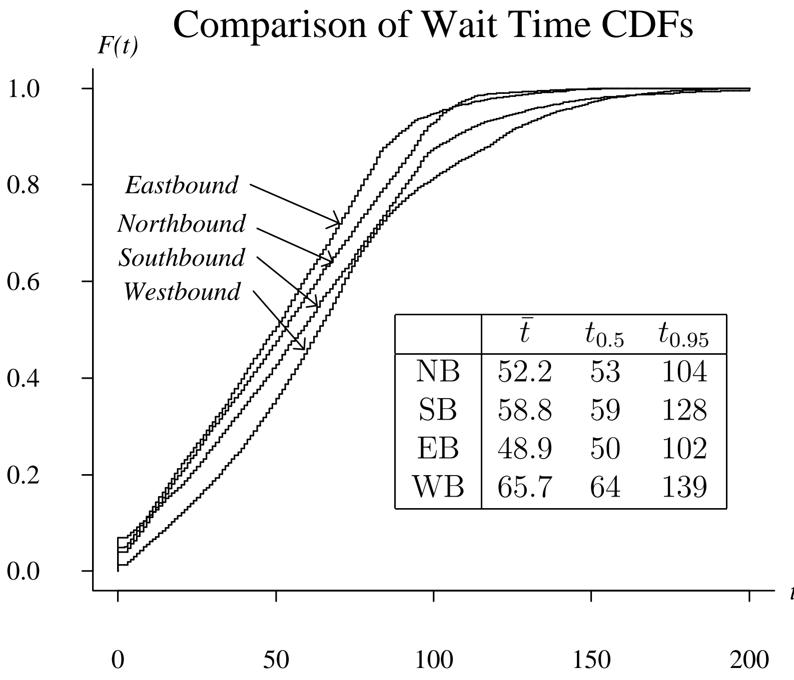


Figure 5.2: Signal phases at Jamestown/Route 199

of evaluating those improvements. Note that all of the data that follow are the result of 30 hours of simulated time, of which statistics were accumulated for the last 29 hours.

5.3.1 Current Intersection Arrangement

As described in Section 5.2, the current arrangement of the intersection of Jamestown Road and Route 199 was tested under rush hour conditions. The results split between approaching directions of traffic are shown in Figure 5.3. The equivalent information distinguished by turning movement is shown in Figure 5.4.



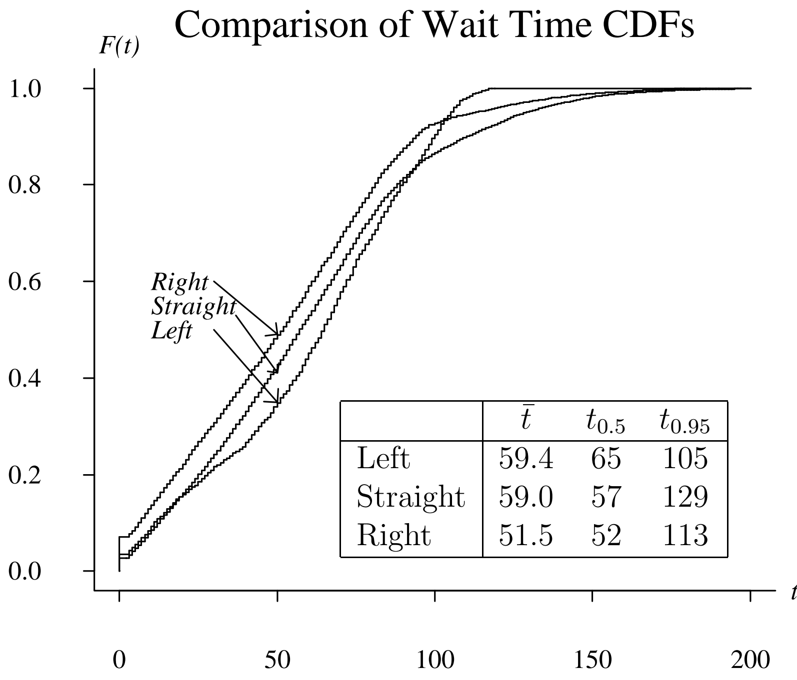
	$\bar{t}$	$t_{0.5}$	$t_{0.95}$
NB	52.2	53	104
SB	58.8	59	128
EB	48.9	50	102
WB	65.7	64	139

Figure 5.3: Comparison of performance between approach directions for current intersection arrangement

Note in Figure 5.3 that $F(0) > 0$ for each approaching direction. This is to be expected since some cars will be lucky enough to arrive to an intersection during a time when their lane has a green signal and there is no previous queue built up. Then $F(0)$ from the empirical CDF of wait times is equivalent to the fraction of cars that get through an intersection without waiting in a queue. From this first glance at the CDFs and the performance measures it seems the Eastbound and Northbound cars get through the intersection with the least trouble while Southbound and Westbound are delayed the most.

A different perspective of the same simulation experiment is shown in Figure 5.4. In this analysis, the data is split between the possible turning movements of left, straight, and right. Not surprisingly, right turn cars get through the intersection more quickly than others, at least based upon the mean and median wait times. When looking at the 95th-percentile of wait times, the intersection actually performs best for left turn cars. This seems counterintuitive at first but consider the following. As detailed in Figure 5.2, left turn cars are the only ones in this model that have a distinct phase just

for that type of turn movement. Granted, those phases are shorter than the others, but there are also fewer left turn cars in most cases. There is a chance once per signal cycle, almost regardless of the presence of other cars, to flush the left turn vehicles from the intersection. This prevents the major delays to left turn cars that would otherwise drive the 95th-percentile to be larger than it is.



	$\bar{t}$	$t_{0.5}$	$t_{0.95}$
Left	59.4	65	105
Straight	59.0	57	129
Right	51.5	52	113

Figure 5.4: Comparison of performance between turning movements for current intersection arrangement

As the CDF graphs shown in Figures 5.3 and 5.4 tend to get crowded and difficult to distinguish between curves, a detailed analysis of specific lanes of the intersection will rely on purely the numerical measures of the mean, median, and 95th-percentile.

The results of this study of the current intersection arrangement are given in Table 5.3. Note that the first three rows within each direction of approaching traffic correspond to the lanes in the intersection. The last two rows in each direction correspond to all cars that traveled straight and all cars that turned right. Since each left turn lane in this model is a left turn-only lane, and there is only one such lane per direction, the first row within each group also contains the data for each car that turned left. By organizing the data in this way, it is easy to spot anomalies away from the trends that you would expect. For example, as a rule of thumb you might expect left turn cars to have longer wait times than straight cars, and straight cars in turn to have longer wait times than right turn cars. Significant deviation from this trend

may be a clue that something is wrong with either the model or the system itself.

Table 5.3: Output data for current intersection arrangement

		$\bar{t}$	$t_{0.50}$	$t_{0.95}$
Northbound	Lane 0 (L)	56.10680	58	106
	Lane 1 (S)	49.19620	48	105
	Lane 2 (S/R)	53.98052	55	102
	All Straight	51.61345	52	103
	All Right	50.17777	50	98
Southbound	Lane 0 (L)	56.32021	58	109
	Lane 1 (S)	55.11691	54	117
	Lane 2 (R)	43.22550	44	93
	All Straight	53.10304	52	108
	All Right	43.22550	44	93
Eastbound	Lane 0 (L)	65.97712	79	107
	Lane 1 (S)	44.33580	44	91
	Lane 2 (S/R)	48.62017	49	96
	All Straight	46.83216	48	92
	All Right	47.00158	48	96
Westbound	Lane 0 (L)	64.74291	68	90
	Lane 1 (S)	59.39484	58	128
	Lane 2 (S/R)	70.73550	63	174
	All Straight	66.18395	61	154
	All Right	68.05123	60	171

Among the more interesting results in Table 5.3 is Southbound Lane 1. This straight-only lane has the longest 95th-percentile wait time of any lane in that direction. This is somewhat unexpected and is the motivation for the adapted intersection studies in Section 5.3.2 and Section 5.3.3. Another interesting result is that Westbound right-turning cars have by far the worst 95th-percentile statistic among any other turning movements for that direction. A proposed intersection improvement addressing this problem is given in Section 5.3.4.

5.3.2 Excursion A: Southbound Shared Straight/Right-Turn Lane

One possible explanation for the large 95th-percentile wait time for Southbound straight cars is that the majority of cars from that direction (52%) continue straight. Also, since there is only one lane from which cars can leave that direction heading straight, perhaps the demand for this traffic movement is too close to the hourly throughput of one lane to be effective. As an attempt to overcome those problems, it is proposed that the rightmost Southbound lane be changed from right turn-only to a straight/right combination. The results of this excursion are given in Table 5.4.

Table 5.4: Output data for Southbound shared straight/right-turn lane

		$\bar{t}$	$t_{0.50}$	$t_{0.95}$
Northbound	Lane 0 (L)	56.10680	58	106
	Lane 1 (S)	49.19620	48	105
	Lane 2 (S/R)	53.98052	55	102
	All Straight	51.61345	52	103
	All Right	50.17777	50	98
Southbound	Lane 0 (L)	55.16605	56	108
	Lane 1 (S)	45.10379	44	97
	Lane 2 (S/R)	49.41367	51	95
	All Straight	47.81444	48	97
	All Right	46.46089	48	94
Eastbound	Lane 0 (L)	65.97712	79	107
	Lane 1 (S)	44.33580	44	91
	Lane 2 (S/R)	48.62017	49	96
	All Straight	46.83216	48	92
	All Right	47.00158	48	96
Westbound	Lane 0 (L)	64.74291	68	90
	Lane 1(S)	59.39484	58	128
	Lane 2(S/R)	70.73550	63	174
	All Straight	66.18395	61	154
	All Right	68.05123	60	171

According to what you would expect, right turn cars have slightly larger wait times in this model than they did in the current intersection arrangement. This is due to the fact that now they are sharing a lane of traffic that

was previously for right turns only. As intended, the performance measures for Southbound straight cars improve significantly over the current arrangement. These results suggest that some improvement can be obtained by altering the intersection geometry in this way.

5.3.3 Excursion B: Southbound Four Lanes

Another proposed improvement for Southbound travelers would be the expansion to four lanes from the current arrangement of two. In this setup there is one left turn lane, two straight lanes, and one right turn lane. The results of this experiment are given in Table 5.5.

Table 5.5: Output data for Southbound four lanes

		$\bar{t}$	$t_{0.50}$	$t_{0.95}$
Northbound	Lane 0 (L)	56.10680	58	106
	Lane 1 (S)	49.19620	48	105
	Lane 2 (S/R)	53.98052	55	102
	All Straight	51.61345	52	103
	All Right	50.17777	50	98
Southbound	Lane 0 (L)	54.95051	56	108
	Lane 1 (S)	41.76321	40	96
	Lane 2 (S)	47.20773	47	92
	Lane 3 (R)	43.22550	44	93
	All Straight	43.98450	43	95
	All Right	43.22550	44	93
Eastbound	Lane 0 (L)	65.97712	79	107
	Lane 1 (S)	44.33580	44	91
	Lane 2 (S/R)	48.62017	49	96
	All Straight	46.83216	48	92
	All Right	47.00158	48	96
Westbound	Lane 0 (L)	64.74291	68	90
	Lane 1 (S)	59.39484	58	128
	Lane 2 (S/R)	70.73550	63	174
	All Straight	66.18395	61	154
	All Right	68.05123	60	171

In looking at the results from the study with four Southbound lanes, there are slight improvements over the results from Excursion A, the case with three

Southbound lanes and a shared straight/right lane. The differences between the two studies is not as great as expected, and the author would make the recommendation that it is not worth the huge expense to add a lane to the intersection. The relatively inexpensive improvement of reclassifying the lanes presented in Section 5.3.2 still provides improvement over the current traffic arrangement. Therefore, this would be the change recommended for the Southbound approach direction.

5.3.4 Excursion C: Westbound Four Lanes

The fact that the current intersection arrangement has such a large chance for Westbound traffic to back up in the right-most lane leads the author to believe that improvements to this approach direction could be very beneficial. Due to the large quantity of arrivals, of which 62% continue straight, any reduction in the flow potential for straight cars would be a mistake for this intersection. Therefore, the case of switching to a left turn-only/straight-only/right-only arrangement is not considered. The proposed improvement for this direction of traffic is to increase the number of Westbound lanes to four, and use the configuration left-only, straight-only, straight-only, right-only. The results of this study are given in Table 5.6.

The effect of adding a lane to this direction, which already has the most green time per signal cycle, is amazing. Roughly 20 seconds have been trimmed from the mean and median wait times for straight and right turning vehicles. The troublesome 95th-percentile wait time for right turn vehicles has been improved by about 100 seconds. With the amount of effective green time that the Westbound straight and right turn lanes have, 48 seconds out of each 120 second cycle, three lanes of outgoing traffic is clearly enought to prevent any chance for severe delays. In fact, this improvement is probably unnecessary as it lowers mean and median wait times to levels that few people would expect to find at an intersection. In this case, the large expense of adding an extra lane could probably be best utilized by improvements in other areas of the intersection.

Table 5.6: Output data for Westbound four lanes

		$\bar{t}$	$t_{0.50}$	$t_{0.95}$
Northbound	Lane 0 (L)	56.10680	58	106
	Lane 1 (S)	49.19620	48	105
	Lane 2 (S/R)	53.98052	55	102
	All Straight	51.61345	52	103
	All Right	50.17777	50	98
Southbound	Lane 0 (L)	56.32021	58	109
	Lane 1 (S)	55.11691	54	117
	Lane 2 (R)	43.22550	44	93
	All Straight	53.10304	52	108
	All Right	43.22550	44	93
Eastbound	Lane 0 (L)	65.97712	79	107
	Lane 1 (S)	44.33580	44	91
	Lane 2 (S/R)	48.62017	49	96
	All Straight	46.83216	48	92
	All Right	47.00158	48	96
Westbound	Lane 0 (L)	65.91802	70	95
	Lane 1 (S)	38.32508	39	83
	Lane 2 (S)	41.72637	39	89
	Lane 3 (R)	23.13552	18	67
	All Straight	42.01907	41	88
	All Right	23.13552	18	67

Chapter 6

Conclusions/Future Work

A discrete-event simulation has been built for the purpose of general traffic intersection analysis. The logic in the model is sufficient to easily compare the performance of different intersection arrangements. Examples of intersection parameters that can be changed and evaluated by the model are the number of lanes, turning classification of lanes (e.g. straight only, turn only, straight/turn), and the signal timing algorithm. The effect of altering the above parameters is measured in terms of the wait times of vehicles using the intersection. Specific performance measures computed include the mean, median, and 95th-percentile of the wait times, as well as the cumulative distribution function of wait times.

Future work in this area could include further verification and validation that the model is performing as it should be. A comparison between model output and actual wait time data from the intersection would be an appropriate next step for establishing credibility for the model. Besides pure data of wait times, the frequencies with which certain key situations occur at the intersection should be compared between the model and the real system. Examples of this type of validation would be to track the number of times traffic from left turn lanes overflow and block traffic in the adjacent straight lane. The number of times this occurs in the real intersection should be consistent with the number of occurances for an equivalent amount of simulated time in the model. Another appropriate consistency check would be to track the proportion of cars that do not make it through the intersection in one signal cycle. Having to wait for a second green light is a common cause of frustration for many drivers. Therefore it would be interesting to check that the actual frequency with which this event occurs is captured in the model

behavior.

Regarding the intersection of Jamestown Road and Route 199, interesting research questions remain as well. One point of interest would be to compare the intersection performance during the morning rush hour with that of the afternoon rush hour which was the focus of Section 5.3. Given that a large amount of the afternoon traffic either turns off of Route 199 to proceed Southbound on Jamestown Road, or continues straight if originally heading Southbound on Jamestown Road, it would be interesting to see if that trend is reversed in the morning. Due to the location of many residences on that side of town, it is possible that this trend is reversed and that morning Northbound arrivals to the intersection incur worse wait times than other directions. Perhaps a more meaningful study of that intersection would be to access the effectiveness of the proposed improvements currently being discussed by local authorities. For example, one idea under consideration is a massive widening of the Northbound approach to the intersection that would include removing a couple shops that are close to the road and gas pumps from the 7-11 station on the corner. This simulation model could provide numerical evidence of the effects of such a project.

As part of a collaboration with B. Brian Park of the Virginia Transportation Research Council and the University of Virginia, this simulation model will be expanded in the coming weeks and used in further traffic intersection research. Likely additions to the model include the use of actuated (sensor-dependent) as opposed to pre-timed traffic signals, as well as model calibration and validation using field data. Possible areas of research include the effectiveness of turn-only lanes, and a comparison of signalized intersection performance with unsignalized (stop-controlled) intersections.

Appendix A

Program Code

A.1 Car Class: car.cpp

This file provides the definition of the C++ class `Car`, used to store data about a car waiting at an intersection.

```
//********************************************************
// Implementation file car.cpp for the Car class
// 11/22/01
// Brendan Hogan
//*********************************************************

#include "car.h"       // defines the class Car and
                       //includes necessary headerfiles
#include <iostream.h>
#include <iomanip.h>


  // constructor
  // ===========
Car::Car(double arr = 0.0, int turn = 0, double pos = 0.0,
         double len = 15.0): _arr(arr), _turn(turn), _pos(pos), _len(len)
  // initializes the car object either with user-specified
  // characteristics or default values
{
```

```
 }  // end constructor

  // functions that query
  // ====================
  double Car::arr() const {   // return the arrival time
      return _arr;
  }

  int Car::turn() const {      // return the turning intention
      return _turn;
  }

  double Car::pos() const {   // return the position in line
      return _pos;
  }

  double Car::len() const {   // return the length
      return _len;
  }

void Car::displayCar() const {
      cout << "Arr = " << _arr << "; Turn = " << _turn << "\n";
      cout << "Pos = " << _pos << "; Len = " << _len << "\n";
}



  // functions that modify
  // =====================
void Car::replace_pos(double new_pos) { // replace value of pos with new_pos
    _pos = new_pos;
}

void Car::advance_pos(double posChange) { // decrease value of pos by posChange
    _pos -= posChange;
}

void Car::transfer(const Car& C) {    // copy in data members of C
```

```
    _arr = C.arr();
    _pos = C.pos();
    _len = C.len();
    _turn = C.turn();
} // end transfer
```

A.2 CarQueue Class: carQueue.cpp

```
//*******************************************************
// Implementation file carQueue.cpp for the carQueue class.
// Circular linked list implementation.
//*******************************************************
#include "carQueue.h"  // header file
#include <stddef.h>  // for NULL
#include <new.h>     // for new exceptions
#include "matrix.h"
#define new new(nothrow)

// The queue is implemented as a circular linked list
// with one external pointer to the back of the queue.
struct node {
  Car       TheCar;
  ptrType   Next;
}; // end struct

carQueue::carQueue() : _size(0), BackPtr(NULL) {
} // end default constructor

carQueue::~carQueue() {
  bool Success;

  while (!QueueIsEmpty()) {
    QueueDelete(Success);
  } // end while
  // Assertion: BackPtr == NULL
} // end destructor

bool carQueue::QueueIsEmpty() const {
  return BackPtr == NULL;
} // end QueueIsEmpty

void carQueue::displayCarQueue() const {
    if (!QueueIsEmpty()) {
        ptrType Cur = BackPtr->Next;
```

```
        while (Cur != BackPtr) {
            Cur->TheCar.displayCar();
            cout << "*****************************\n";
            Cur = Cur->Next;
        }
        // copy last one in line
        Cur->TheCar.displayCar();
        cout << "*****************************\n\n";
    } // end if
    else
        cout << "No cars in queue.\n";
} // end displayCarQueue

int carQueue::cardinality() const {
    return _size;
}

void carQueue::GetQueueFront(Car& QueueFront,
                             bool& Success) const {
  Success = !QueueIsEmpty();

  if (Success) { // queue is not empty; retrieve front
    ptrType FrontPtr = BackPtr->Next;
    QueueFront = FrontPtr->TheCar;
  } // end if
} // end GetQueueFront

void carQueue::GetQueueBack(Car& QueueBack,
                            bool& Success) const {
    Success = !QueueIsEmpty();

    if (Success) { // queue is not empty; retrieve back
        QueueBack = BackPtr->TheCar;
    } // end if
} // end GetQueueBack

void carQueue::QueueInsert(Car NewCar,
```

```
                         bool& Success) {
  // create a new node
  ptrType NewPtr = new node;
  Success = NewPtr != NULL;  // check allocation

  if (Success) { // allocation successful;
    NewPtr->TheCar = NewCar; // set data portion of new node
    // insert the new node
    if (QueueIsEmpty()) {  // insertion into empty queue
      NewPtr->Next = NewPtr;
    } // end if
    else{                  // insertion into nonempty queue
      NewPtr->Next = BackPtr->Next;
      BackPtr->Next = NewPtr;
    } // end else
    BackPtr = NewPtr;      // new node is at back
    _size++;
  } // end if
} // end QueueInsert

void carQueue::QueueDelete(bool& Success) {
  Success = !QueueIsEmpty();

  if (Success) { // queue is not empty; remove front
    ptrType FrontPtr = BackPtr->Next;
    if (FrontPtr == BackPtr) {  // special case?
      BackPtr = NULL;           // yes, one node in queue
    } // end if
    else {
      BackPtr->Next = FrontPtr->Next;
    } // end else

    FrontPtr->Next = NULL;      // defensive strategy
    delete FrontPtr;
    _size--;
  } // end if
} // end QueueDelete
```

```
void carQueue::QueueDelete(Car& OldCar,
                           bool& Success) {
  Success = !QueueIsEmpty();
  if (Success) { // queue is not empty; retrieve front
    ptrType FrontPtr = BackPtr->Next;
    OldCar = FrontPtr->TheCar;

    QueueDelete(Success);             // delete front
  } // end if
} // end QueueDelete

void carQueue::UpdatePosition(matrix<double>& Geometry, matrix<int>& Direction,
                              int approach, int lane, double posChange,
                              int TheTurn, bool& Waiting, bool& CkBlocked) {
/* This function scans a lane from which there was just a departure.
   If cars remain, their position is advanced forward in the lane an
   appropriate amount.  If there is a car in the lane that is waiting to
   get into a left turn lane and therefore blocking traffic in the
   current lane, the position of that car (and any behind it) is not
   updated.
   Input:  Geometry: matrix defining the size of each lane
           Direction: matrix defining the turn movements of each lane
           approach/lane: indices of the approaching direction and lane
                          to be searched
           posChange: the distance (size of departing car) that
                      remaining cars should be moved forward
           TheTurn: indicator of turning movement of departing car
   Output: Waiting: indicator of whether cars remain in the lane
           CkBlocked: indicator about a possible unblocked car */

    Waiting = !QueueIsEmpty();
    CkBlocked = false;
    bool blocked = false;
    double movedist = posChange;
    double position;

    if (Waiting) {
        ptrType Cur = BackPtr->Next;
```

```
if (Cur == BackPtr) {   // only 1 car in lane

    if ( (lane == 1) && (Cur->TheCar.turn() == -1) ) {
        position = Cur->TheCar.pos();
        if (position == Geometry[approach][0])
            blocked = true;
        else
            if ( (position > Geometry[approach][0]) &&
                  (position < Geometry[approach][0]+movedist)) {
                movedist = position - Geometry[approach][0];
            }
    } // end if
    if (!blocked) {
        Cur->TheCar.advance_pos(movedist);
    }
}  // end if only 1 in lane
else {                  // more than 1 in lane
    while ( (Cur != BackPtr) && (!blocked)) {
        if ( (lane == 1) && (Cur->TheCar.turn() == -1) ) {
            position = Cur->TheCar.pos();
            if (position == Geometry[approach][0])
                blocked = true;
            else
                if ( (position > Geometry[approach][0]) &&
                      (position < Geometry[approach][0]+movedist)) {
                    movedist = position - Geometry[approach][0];
                    CkBlocked = true;
                }
        } // end if

        if (!blocked) {
            Cur->TheCar.advance_pos(movedist);
            Cur = Cur->Next;
        }
    } // end while

    // now pointing at BackPtr, check the last one
    if ( (lane == 1) && (Cur->TheCar.turn() == -1) ) {
```

```
                position = Cur->TheCar.pos();
                if (position == Geometry[approach][0])
                    blocked = true;
                else
                    if ( (position > Geometry[approach][0]) &&
                         (position < Geometry[approach][0]+movedist)) {
                        movedist = position - Geometry[approach][0];
                        CkBlocked = true;
                    }
            } // end if

            if (!blocked)
                Cur->TheCar.advance_pos(movedist);

        } // end else more than 1 in lane
    }  // end if waiting
} // end UpdatePosition


void carQueue::Unblock(matrix<double>& Geometry, int approach,
                       double gap, Car SwitchedCar, bool& Success) {
/* This function scans the lane adjacent to a left turn-only lane to see
   if an opening for a previously blocked car exists.  Two conditions
   must be satisfied in order for a car to be moved from this lane
   to the left turn-only lane:
   1. The car must be waiting at the entrance to the left turn-only lane.
   2. There must be room in the left turn-only lane for the car to fit.
   Input:   Geometry: matrix defining size of lanes
            approach: index of approaching direction for the lanes involved
            gap: the space available in left turn-only lane
   Output:  Success: indicator of whether a previously blocked car was
                      unblocked
            SwitchedCar: the car being moved between lanes */

    ptrType Prev, Cur;
    Success = false;

    Prev = NULL;
```

```
Cur = BackPtr->Next;

if (Cur == BackPtr) {   // special case, only 1 car in queue
    if ((Cur->TheCar.pos() == Geometry[approach][0]) &&
        (Cur->TheCar.len() <= gap)) {
        // if car at entrance and room in left turn-only lane
        SwitchedCar.transfer(Cur->TheCar);  // copy car to be moved
        BackPtr = NULL;             // remove it from this queue
        Cur->Next = NULL;
        delete Cur;
        Cur = NULL;
        Success = true;             // set indicator
        _size--;                    // decrease count of # in queue
    } // end if car to be unblocked
} // end if only 1 car in queue
else {   // more than 1 car in queue
    while ((Cur->TheCar.pos() < Geometry[approach][0]) &&
           (Cur != BackPtr)) {
        Prev = Cur;
        Cur = Cur->Next;
    }

    // at this point, if there is a car at the entrance to the left
    // turn-only lane, Cur is pointing to it
    if ((Cur->TheCar.pos() == Geometry[approach][0]) &&
        (Cur->TheCar.len() <= gap)) {
        // if car at entrance and room in left turn-only lane
        SwitchedCar.transfer(Cur->TheCar);  // copy car to be moved
        if (Cur == BackPtr)     // special case, remove queue back
            BackPtr = Prev;
        Prev->Next = Cur->Next;     // remove it from this queue
        Cur->Next = NULL;
        delete Cur;
        Cur = NULL;
        Success = true;             // set indicator
        _size--;
    } // end if car is switched
} // end else more than 1 can in queue
```

```
} // end Unblock


carQueue& carQueue::operator=(const carQueue& Q) {
// Overloaded assignment operator

// copy in any and all elements of Q
   ptrType Cur = Q.BackPtr;
   bool Success;
   while (Cur != NULL) {
     QueueInsert(Cur->TheCar, Success);
     Cur = Cur->Next;
   }

   return *this;
} // end operator=(carQueue Q)
```

A.3 Simulation Function Definitions: functions.cpp

```
#define infty 10000.0
#define NumPhases 5
#define NumEvents 7
#define WarmUp  3600       // system warm-up time before stats are collected
#define hour    3600       // # of seconds in one hour
#define maxTime 300        // largest pos. wait time recorded in histogram
#define blockTime 5.0      // dead time where no cars get through
// equivalent to = (yellow - end gain) + (all red) + (start-up loss)



typedef struct {                        /* the next-event list    */
  double t;                             /*   next event time      */
  int    x;                             /*   event status, 0 or 1 */
} event_list[NumEvents];

typedef struct {                           /* accumulated sums of        */
  double wait;                    /*   wait times                        */
  int   served;                  /*   number served                     */
} sum;

typedef struct {                      /* definition of a vector of ints */
  int count[maxTime];                 /* so that I can make a matrix of*/
} wait_times;                         /* these */




double GetArrival(int approach)
/* ---------------------------------------------------------------------
 * generate the next arrival time from approach = 0..3 (NB,SB,EB,WB)
 * ---------------------------------------------------------------------
 */
```

```
{
    SelectStream(approach);
    double u = Random();
    double temp;

    switch (approach) {
    case 0:
        temp = idfLognormal(1.3515, 0.8938, u);
        break;

    case 1:
        temp = idfLognormal(1.4495, 1.0647, u);
        break;

    case 2:
        temp = idfLognormal(0.9069, 0.8745, u);
        break;

    case 3:
        temp = idfLognormal(0.4963, 0.9584, u);
        break;
    };

    return(temp);

}  // end GetArrival

    int PickDirection(matrix<double>& TurnProb, int approach) {
/* Stochastic function to assign intended direction of travel
   Input: (4x2) matrix TurnProb which contains CUMULATIVE probabilities
   * of turn left in the first column and driving straight in the second
   * column.  It is assumed that the difference between 1.0 and the
   * cumulative probability in the second column is the probability of
   * turning right.  The integer approach contains the index of the
   * approach direction to the intersection, this is used to reference
   * the appropriate row within the TurnProb matrix.
   Output: An integer -1, 0, or 1, indicating left, straight, or right. */
```

```
    SelectStream(approach+4);
    double u = Random();
    if (u < TurnProb[approach][0])
        return (-1);  // left
    if (u < TurnProb[approach][1])
        return (0);   // straight
    else
        return (1);   // right
}  // end PickDirection


double PickSize(double proportions[], int approach) {
/* Stochastic function to assign a vehicle size, either passenger car,
 * or heavy vehicle, based on a specified percentage of each.
 * Note that the length returned is the length of the vehicle plus the
 * buffer distance between vehicles.
 *     Vehicle Type   Vehicle Assumed Length     Total Space in Lane
 *******************************************************************
 *     passenger car         19 ft                        25 ft
 *     heavy vehicle         30 ft                        35 ft
 *
 * Input:  Vector of passenger car proportion for each approach dir.
 *         Integer indicating approach direction
 * Output: Double length taken up in lane by this vehicle  */

    SelectStream(approach+8);
    double u = Random();
    if (u < proportions[approach])
        return 25.0;
    return 35.0;
}




void PlaceEmptyLane(matrix<int>& Light, matrix<carQueue>& Queue,
                    Car TheCar, int approach, int lane,
                    int MaxLanes, double curtime, matrix<wait_times>&  A)
```

```
{
    /* Function to place a car in empty lane.
     * If the light is green, car will pass through and update stats,
     * if light is red, car will stop at front. */
    bool Success;

    if (Light[approach][lane] == 1)  {// light is green
        if (curtime > WarmUp) {               // collect frequency stats
            A[approach][lane].count[0]++;
            if (TheCar.turn() > -1)
                A[approach][MaxLanes+TheCar.turn()].count[0]++;
        }
    }
    else {   // light is red, stop
        TheCar.replace_pos(0.0);  // just to be safe, this can be taken out
        Queue[approach][lane].QueueInsert(TheCar, Success);
    }
} // end EmptyLane


void PickLane(matrix<int>& Direction, matrix<double>& Geometry,
              matrix<carQueue>& Queue, Car NewCar, int approach,
              int NumLanes, bool& CarPlaced, int& NewLane) {
/* function to place a car in a lane,
   if lane lengths are infinite this decision is based solely on
   * intended direction of travel,
   if lane lengths are finite this decision is based on intended
   * direction of travel as well as available space in the lanes*/


    int startlane = -26;  // two local variables used to track avail. lanes
    int countlane = 0;
    bool Success, SuccessB;
    Car LastCar, LastCarB;
    int turn = NewCar.turn();
    CarPlaced = false;
    double backA, backB, cutoffA;
```

```
NewLane = -99;

switch (turn) {
case -1:              // left turn
    // check avail. lanes  (ie. Direction[approach][i]<0)
    for (int i = 0; i < NumLanes; i++) {
        if ((Direction[approach][i] < 0) &&
            (Direction[approach][i] >= -2)) {
            countlane++;
            if (startlane == -26)
                startlane = i;
        }
    }

    // Note: for now I will assume that there will be at most 1 left
    // turn lane, the logic for more left turn lanes can be
    // added later.

    // Get copy of last car in that lane and adjacent lane
    Queue[approach][startlane].GetQueueBack(LastCar, Success);
    Queue[approach][startlane+1].GetQueueBack(LastCarB, SuccessB);

    if ((!Success) && (!SuccessB))  // both lanes empty
        NewLane = startlane;        // put it in first lane by default
    else {              // at least 1 lane has traffic, need to look closer
        // back of traffic in lane A and B
        backA = LastCar.pos() + LastCar.len();
        backB = LastCarB.pos() + LastCarB.len();

        // critical position for fitting in the lane
        cutoffA = Geometry[approach][startlane] - NewCar.len();

        if (max(backA, backB) > cutoffA) {    // A blocked, put in B
            NewCar.replace_pos(max(backB, cutoffA));
            Queue[approach][startlane+1].QueueInsert(NewCar,
                                                     Success);
            CarPlaced = true;
        }
```

```
        else {   // put in A
            if (backA == 0.0)
                NewLane = startlane;
            else {
                NewCar.replace_pos(backA);
                Queue[approach][startlane].QueueInsert(NewCar,
                                                       Success);
                CarPlaced = true;
            }
        } // end else put in A
    } // end else at least 1 lane has traffic
    break;  // end left turn car

case 0:    // straight car

    // check avail. lanes  (ie. -1 <= Direction[approach][i]<= +1)
    for (int i = 0; i < NumLanes; i++) {
        if ((Direction[approach][i] <= +1) &&
            (Direction[approach][i] >= -1)) {
            countlane++;
            if (startlane == -26)
                startlane = i;
        }
    }

    if (countlane == 1) {   // 1 straight lane, put it there
        // Get copy of last car in that lane
        Queue[approach][startlane].GetQueueBack(LastCar, Success);

        if (Success) {  // there is a queue built up
            NewCar.replace_pos(LastCar.pos() + LastCar.len());
            Queue[approach][startlane].QueueInsert(NewCar, Success);
            CarPlaced = true;
        }
        else
            NewLane = startlane;
    } // end 1 straight lane
```

```
    if (countlane == 2) { // 2 straight lanes, pick shortest queue
        // Get copy of last car in that lane and adjacent lane
        Queue[approach][startlane].GetQueueBack(LastCar, Success);
        Queue[approach][startlane+1].GetQueueBack(LastCarB, SuccessB);

        if ((!Success) && (!SuccessB))   // both lanes empty, put in A
            NewLane = startlane;
        else {        // at least 1 lane has traffic

            // pick lane with shortest queue
            backA = LastCar.pos() + LastCar.len();
            backB = LastCarB.pos() + LastCarB.len();

            int TheLane = startlane;
            double back = backA;
            if (backB < backA) {
                TheLane++;
                back = backB;
            }

            if (back == 0.0)   // shortest lane is empty
                NewLane = TheLane;
            else {             // shortest lane has queue
                NewCar.replace_pos(back);
                Queue[approach][TheLane].QueueInsert(NewCar,
                                                     Success);
                CarPlaced = true;
            }  // end shortest lane has queue
        } // end at least 1 lane has traffic
    } // end 2 straight lanes
    break;  // end straight car

case 1:  // right turn car

    // check avail. lanes  (ie. Direction[approach][i] > 0)
    for (int i = 0; i < NumLanes; i++) {
        if ((Direction[approach][i] > 0) &&
            (Direction[approach][i] <= 2)) {
```

```
                countlane++;
                if (startlane == -26)
                    startlane = i;
            }
        }

        // Note: for now I will assume that there will be at most 1 lane
        // from which right turns are possible, and that that lane has
        // infinite length.
        // Logic for more complicated situations can be added later.

        // Get copy of last car in that lane
        Queue[approach][startlane].GetQueueBack(LastCar, Success);

        if (Success) {  // there is a queue built up
            NewCar.replace_pos(LastCar.pos() + LastCar.len());
            Queue[approach][startlane].QueueInsert(NewCar, Success);
            CarPlaced = true;
        }
        else
            NewLane = startlane;

        break;  // end right turn car
    }; // end switch(turn)
}  // end PickLane



void PassThru(matrix<carQueue>& Queue, matrix<double>& Departure,
              matrix<double>& Geometry, matrix<int>& Direction,
              int approach, int lane, double curtime,
              int MaxLanes, matrix<wait_times>& A) {
/*  Sends a previously waiting car through the intersection.  That is,
 * it is removed from memory, the physical position of cars behind it in
 * line (if any) is moved forward, and the next time of departure for
 * the lane in question is set.
 * Input: Queue structure, Stats data structure, Departure - the array
 * of next departure times for each lane, the direction and lane that
```

```
* the depature occurs from, and curtime - the current value of the
* simulation clock.
* Output: Updated Queue, Stats, Departure structures.  */

   bool Success, SuccessB;
   bool CarsWaiting, CkBlocked;

   /* remove from memory */
   Car OldCar;
   Queue[approach][lane].QueueDelete(OldCar, Success);
   double posChange = OldCar.len();   // length of departing car
   int OldTurn = OldCar.turn();         // turning intention of old car

   /* check rest of lane and if cars remain, update their position */
   Queue[approach][lane].UpdatePosition(Geometry, Direction, approach,
                                         lane, posChange, OldTurn,
                                         CarsWaiting, CkBlocked);
   if (CarsWaiting) {   /* if more cars waiting */
       Car NewFrontCar;
       Queue[approach][lane].GetQueueFront(NewFrontCar, Success);
       if (NewFrontCar.pos() < 10) {   // if close to front
           /* schedule next departure (approach, lane) */
           Departure[approach][lane] += 2.0;  /* buffer time between cars */
       }
       // need to check for possible unblocking of left lane, or
       // creation of access to the left lane, and if so handle it
       if ((lane == 0) || ((lane == 1) && CkBlocked))  {
       // if left turn-only lane, or appropriate straight
           Car BackCar;
           Queue[approach][0].GetQueueBack(BackCar, Success);
           double back = BackCar.pos() + BackCar.len(); // back of left lane
           double gap = Geometry[approach][0] - back;  // space in left lane

           Car SwitchedCar;
           Queue[approach][1].Unblock(Geometry, approach, gap,
                                      SwitchedCar, Success);
           if (Success) {
               // set new pos and add to left lane
```

```
                SwitchedCar.replace_pos(back);
                Queue[approach][0].QueueInsert(SwitchedCar, SuccessB);
            }
        } // end if possible unblocking
    }  // end if CarsWaiting
    else     /* no cars waiting, so no departure scheduled */
        Departure[approach][lane] += infty;


    /* collect stats */
    if (curtime > WarmUp) {

        // counts for each lane of traffic
        int waittime = (int) (curtime - OldCar.arr());
        int vindex = min_ints(waittime,maxTime-1);
        A[approach][lane].count[vindex]++;

        // counts for each turn pattern
        // (left lane data will suffice for left turn cars)
        if (OldTurn > -1) {
            int mindex = MaxLanes + OldTurn;
            A[approach][mindex].count[vindex]++;
        }
    }
} // end PassThru


void Change2Red(matrix<int>& Light) {
    /* This function updates the boolean matrix of light data indicating
     * which lanes have a green signal (1) or a red signal (0).
     * Turns off green lanes which are ending, green lanes that continue green
     * are left alone.  The next signal event (Change2Green) will be
     * scheduled for a buffer length of time in the future.
     * Input/Output: The boolean matrix Light.   */

    static int phase = 0;

    phase = (phase + 1)%NumPhases;
```

```
    switch (phase) {
    case 0:
        Light[0][0] = 0;
        Light[1][0] = 0;
        break;
    case 1:
        Light[3][0] = 0;
        break;
    case 2:
        Light[3][1] = 0;
        Light[3][2] = 0;
        break;
    case 3:
        Light[2][0] = 0;
        Light[2][1] = 0;
        Light[2][2] = 0;
        break;
    case 4:
        Light[0][1] = 0;
        Light[0][2] = 0;
        Light[1][1] = 0;
        Light[1][2] = 0;
        break;
    } // end switch
} // end Change2Red


void Change2Green(matrix<int>& Light, double& phaselength) {
    /* This function updates the boolean matrix of light data indicating
     * which lanes have a green signal (1) or a red signal (0).
     * phaselength is set to the length of time signal settings will
     * remain in this state,
     * i.e. the effective green time = total phase length - blockTime
     * Input: The boolean matrix Light.
     * Output: Updated matrix Light, length of current phase in order to
     * schedule the next signal change event.  */
```

```
    static int phase = -1;

    phase = (phase + 1)%NumPhases;

    switch (phase) {
    case 0:
        Light[3][0] = 1;
        Light[3][1] = 1;
        Light[3][2] = 1;
        phaselength = 28.0-blockTime;
        break;
    case 1:
        Light[2][1] = 1;
        Light[2][2] = 1;
        phaselength = 25.0-blockTime;
        break;
    case 2:
        Light[2][0] = 1;
        phaselength = 17.0-blockTime;
        break;
    case 3:
        Light[0][1] = 1;
        Light[0][2] = 1;
        Light[1][1] = 1;
        Light[1][2] = 1;
        phaselength = 27.0-blockTime;
        break;
    case 4:
        Light[0][0] = 1;
        Light[1][0] = 1;
        phaselength = 23.0-blockTime;
        break;
    } // end switch
} // end Change2Green

bool UpdateDepartures(matrix<int>& Light, matrix<carQueue>& Queue,
                      matrix<double>& Departure, int NumLanes,
                      double curtime, int& newApproach, int& newLane) {
```

```
    /* This function is called during a signal change after the light
     * settings are updated.  It's purpose is to update the necessary
     * elements of the NextDeparture matrix, and if a car is waiting, to
     * output the approach direction and lane of the departure. */

    bool waiting = false;

    for (int i = 0; i < 4; i++)                  // ck each approach direction
        for (int j = 0; j < NumLanes; j++)       // ck each lane
            if (Light[i][j] == 1) {              // if green
                if (Queue[i][j].QueueIsEmpty())      // no cars waiting
                    Departure[i][j] = curtime + infty;
                else  {                              // cars waiting
                    Car FrontCar;
                    bool Success;
                    Queue[i][j].GetQueueFront(FrontCar, Success);

                    // if car waiting at new green,
                    // and the car is not blocked
                    if ( ( (curtime > Departure[i][j]) ||
                           (Departure[i][j] > curtime+2)) &&
                          ( FrontCar.pos() < 10) ) {
                        Departure[i][j] = curtime;   // depart.time=NOW
                        if (!waiting) {
                            newApproach = i;
                            newLane = j;
                            waiting = true;
                        }
                    } // end if new departure should be scheduled
                } // end else cars waiting
            } // end if green
    return (waiting);
} // end UpdateDepartures


void ShowState(matrix<carQueue>& Queue, int NumLanes) {
    /* This function outputs the number of cars waiting in each lane of
     * the intersection at a given time.
```

```
     * Input: Queue matrix of links to the actual stored data of cars
     * waiting there, NumLanes which is the maximum number of lanes in
     * any intersection.  Note the dimensions of the Queue matrix should
     * be 4xMaxLanes.
     * Output: text written to screen. */
    int m, n;

    cout << "\n Number of Cars in Each Lane  \n";
    cout << " Dir \\ Lane    ";
    for (n = 0; n < NumLanes; n++)
        cout << "   " << n << "  ";
    cout << "\n";
    for (m = 0; m < 4; m++) {
        cout << " " << m << "              ";
        for (n = 0; n < NumLanes; n++)
            printf("%5d", Queue[m][n].cardinality());

        cout << "\n";
    }
} // end ShowState

template <class Data>
void ShowMatrix( matrix<Data> & TheMatrix, int rows, int cols) {
    /* Code to output the contents of a matrix to the screen, regardless
     * of the type of info it contains. */
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            cout << TheMatrix[i][j] << "  ";
        }
        cout << "\n";
    }
} // end ShowMatrix


   int NextEvent(event_list event)
/* --------------------------------------
 * return the index of the next event type
 * --------------------------------------
```

```
 */
{
  int e;
  int i = 0;

  while (event[i].x == 0)       /* find the index of the first 'active' */
    i++;                        /* element in the event list            */
  e = i;
  while (i < NumEvents-1) {     /* now, check the others to find which  */
    i++;                        /* event type is most imminent          */
    if ((event[i].x == 1) && (event[i].t < event[e].t))
      e = i;
  }
  return (e);
}
```

A.4 Simulation Main Program: traffic.cpp

```
/*******************************************************************
    Title: traffic.cpp
    Author: Brendan Hogan
    Date: Oct 29, 2001
    Modified: April 6, 2002
    Purpose: discrete event simulation of a generic traffic intersection
*******************************************************************/
#include <stdio.h>
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <math.h>
#include "matrix.h"     // Matrix class, from Weiss, 2nd Ed.
#include "rngs.c"       // Multi-stream Random Number Generator, from S. Park
#include "rvms.c"       // Random Variate Generator, from S. Park
#include "routines.cpp" // Definition of min, max functions
#include "car.h"
#include "car.cpp"      // Definition of C++ Car class
#include "carQueue.h"
#include "carQueue.cpp" // Definition of C++ carQueue class
#include "functions.cpp"  // all functions defined in external file





int main(void) {

    /******* Static Variables ***********/
    int const Lanes[4] = {3,3,3,3};      // number lanes in each direction
    int maxLanes = Lanes[0];
    for (int i = 1; i < 4; i++)
        if (Lanes[i] > maxLanes)
            maxLanes = Lanes[i];
```

```
// geometry of each direction of travel  (length in feet of lane)
matrix<double> Geometry(4, maxLanes);
// initialize it
for (int i = 0; i < 4; i++)    // for each direction
    for (int j = 0; j < Lanes[i]; j++)   // for each possible lane
        Geometry[i][j] = infty;

// data for lanes that have finite length
Geometry[0][0] = 240.0;
Geometry[1][0] = 280.0;
Geometry[2][0] = 200.0;
Geometry[3][0] = 200.0;

//turning configuration of lanes   (99 if lane does not exist)
matrix<int> Direction(4, maxLanes, 99);
Direction[0][0] = -2;
Direction[0][1] = 0;
Direction[0][2] = 1;
Direction[1][0] = -2;
Direction[1][1] = 0;
Direction[1][2] = 2;
Direction[2][0] = -2;
Direction[2][1] = 0;
Direction[2][2] = 1;
Direction[3][0] = -2;
Direction[3][1] = 0;
Direction[3][2] = 1;

matrix<double> TurnProb(4, 2);
TurnProb[0][0] = 0.32;
TurnProb[0][1] = 0.73;
TurnProb[1][0] = 0.19;
TurnProb[1][1] = 0.71;
TurnProb[2][0] = 0.14;
TurnProb[2][1] = 0.78;
TurnProb[3][0] = 0.28;
TurnProb[3][1] = 0.90;
```

```
 // Distribution of small vs. large vehicles from each approach direction.
 double SizeProb[4] = {0.93, 0.94, 0.88, 0.95};


 /******* Dynamic Variables **********/

 // matrix of signal settings for each direction/lane
 matrix<int> Light(4, maxLanes, 0);  // all lanes initialized to red

 // matrix of state info for each direction/lane of traffic
 matrix<carQueue> Queue(4, maxLanes);

 // matrix of next departure time for each direction and lane
 matrix<double> NextDeparture(4, maxLanes, infty);


 // ******* simulation implementation variables ********* //
struct {
  double current;                 /* current time                        */
  double next;                    /* next (most imminent) event time     */
} t;
event_list event;
int        e;                     /* next event index                    */
int        l;
double     newTime;
int        newApproach;
int        newLaneIn;
int        newLaneOut;
bool       CarPlaced;
double     phaselength;


PlantSeeds(565789086);
t.current = 0.0;

// initialize event list
for (l = 0; l < 4; l++) {
```

```
    event[l].t = GetArrival(l);
    event[l].x = 1;
}

event[4].t = infty;       /* no pass thru allowed when no line */
event[4].x = 0;
event[5].t = phaselength+2.0;       /* time to first Change2Red */
event[5].x = 1;
event[6].t = infty;                 /* time to first Change2Green */
event[6].x = 0;

/* histogram type stats structure */
/* 1 row for each approach direction,
   1 column for each lane, plus 1 extra for all right turns. */
matrix<wait_times> TheWaitTimes(4, maxLanes+2);

// Set signal to Phase 1 greens
Change2Green(Light, phaselength);

double STOP = 30*hour;
int count = 0;
while (t.current < STOP) {
    e = NextEvent(event);
    t.current = event[e].t;

    if (e < 4) {                     /* process an arrival */
        event[e].t += GetArrival(e);  /* set next arrival in that dir*/
        // define the new car
        int turn = PickDirection(TurnProb, e);
        double length = PickSize(SizeProb, e);
        Car NewCar(t.current, turn, 0.0, length);

        // place the new car
        PickLane(Direction, Geometry, Queue, NewCar, e,
                 maxLanes, CarPlaced, newLaneIn);
        if (!CarPlaced)
            PlaceEmptyLane(Light, Queue, NewCar, e, newLaneIn,
                           maxLanes, t.current, TheWaitTimes);
```

```
} // end arrival
else
if (e == 4) {                    /* process a departure */
    PassThru(Queue, NextDeparture, Geometry, Direction,
             newApproach, newLaneOut, t.current,
             maxLanes, TheWaitTimes);
    newTime = t.current + infty;
    newApproach  = -1;
    newLaneOut = -1;

    /* search for next departure */
    for (int i = 0; i < 4; i++)              // ck each direction
        for (int j = 0; j < maxLanes; j++)   // ck each lane
            if (Light[i][j] == 1)            // if green, compare
                if (NextDeparture[i][j] < newTime) {
                    newApproach = i;
                    newLaneOut = j;
                    newTime = NextDeparture[i][j];
                }
    if (newTime < t.current + 60)
        event[4].t = newTime;   /* schedule next departure */
    else
        event[4].x = 0;         /* turn departure 'off' */
}  // end departure
else
    if (e == 5) {               /* process a Change2Red */
        // update signal settings
        Change2Red(Light);

        // turn off Change2Red Process
        event[5].x = 0;

        // schedule and turn on Change2Green Process
        event[6].x = 1;
        event[6].t = t.current + blockTime;
    } // end Change2Red
    else {                      /* process a Change2Green */
        // update signal settings
```

```
                Change2Green(Light, phaselength);

                // turn off Change2Green Process
                event[6].x = 0;

                // schedule and turn on Change2Red Process
                event[5].x = 1;
                event[5].t = t.current + phaselength;

                // check new green lights for imminent departures
                bool Waiting = UpdateDepartures(Light, Queue, NextDeparture,
                                                maxLanes, t.current, newApproach,
                                                newLaneOut);
                if (Waiting) {
                    event[4].t = t.current; /* next departure time is NOW*/
                    event[4].x = 1;         /* just to be safe */
                } // end if(Waiting)
            } // end Change2Green

    } // end while (sim still going)

// declare file to write data in
     const char
         OutputFileName[] = "data1.out";

     ofstream
         OutStream(OutputFileName, ios::out);

// output matrix of count data for use in histogram creation
    for (l = 0; l < 4; l++) {                  // for each direction
        for (int j = 0; j < maxLanes+2; j++) {   // for each lane + 1
            for (int m = 0; m < maxTime; m++)
                OutStream << TheWaitTimes[l][j].count[m] << "  ";

            OutStream << "\n";
        }
    }
```

```
    return (0);
}
```

Bibliography

[1] Asante, S.A., Ardekani, S.A., and Williams, J.C., "A Simulation Study of the Operational Performance of Left-Turn Phasing and Indication Sequences," *Transportation Science*, Vol. 30, No. 2, pp. 112-119, 1996.

[2] Black, J., and Wanat, J., "Traffic Signals and Control, Ramp Metering, and Lane Control Systems," `<http://www.path.berkeley.edu/~leap/TTM/Traffic_Control/control.html>`, Updated 08 June, 1998, Accessed 29 August, 2001.

[3] Bynum, B., *Computer Organization and Programming Languages*, College of William and Mary, Computer Science Department, 2001.

[4] Carrano, F.M., Helman, P., and Veroff, R., *Data Abstraction and Problem Solving with C++*, Addison-Wesley, Inc., Reading, MA, 1998.

[5] Hagen, L.T., and Courage, K.G., "Comparison of Macroscopic Models for Signalized Intersection Analysis," *Transportation Research Record 1225*, TRB, National Research Council, Washington, D.C., pp. 33-44, 1989.

[6] Hurdle, V.F., "Signalized Intersection Delay Models - A Primer for the Uninitiated," *Transportation Research Record 971*, TRB, National Research Council, Washington, D.C., pp. 96-105, 1984.

[7] Law, A.M., and Kelton, W.D., *Simulation Modeling and Analysis*, McGraw-Hill, Inc., New York, 2000.

[8] Leemis, L., *Reliability: Probabilistic Models and Statistical Methods*, Prentice-Hall, Englewood Cliffs, N.J., 1995.

[9] May, A.D., Jr., "Gap Acceptance Studies," *Highway Research Board Record 72*, HRB, Washington, D.C., 1965.

[10] Meneguzzer, C., "Review of Models Combining Traffic Assignment and Signal Control," *Journal of Transportation Engineering*, Vol. 123, pp. 148-155, 1997.

[11] Park, B. B., Personal Correspondence, University of Virginia, Department of Civil Engineering, 2001-2002.

[12] Park, S., and Leemis, L., *Discrete-Event Simulation: A First Course*, The College of William and Mary, Computer Science Department, 2000.

[13] Pegden, C.D., Shannon, R.E., and Sadowski, R.P., *Introduction to Simulation Using SIMAN*, McGraw-Hill, Inc., New York, 1995.

[14] Reints, C., Personal Correspondence, Virginia Department of Transportation, 2001-2002.

[15] Sen, S., and Head, K.L., "Controlled Optimization of Phases at an Intersection," *Transportation Science*, Vol. 31, No. 1, pp. 5-17, 1997.

[16] Statistical Sciences, *S-PLUS Programmer's Manual, Version 3.2*, StatSci, a division of MathSoft, Inc., Seattle, 1993.

[17] Tolle, J.E., "The Lognormal Distribution Model," *Traffic Engineering and Control*, Vol. 13, No. 1, 1971.

[18] Webster, V.F., *Traffic Signal Settings*, Her Majesty's Stationary Office, London, England, 1958.

[19] Weiss, M.A., *Data Structures & Algorithm Analysis in C++*, Addison-Wesley, Inc., Reading, MA, 1999.

[20] Weiss, P., "Stop-and-Go Science: Traffic Gridlock in the 21st Century," *Science News*, Vol. 156, No. 1, p. 8, 1999.

[21] Yang, Q., and Koutsopoulos, H.N., "A Microscopic Traffic Simulator for Evaluation of Dynamic Traffic Management Systems," *Transportation Research Part C*, Vol. 4, Is. 3, pp 113-129, 1996.