

ATIS at Rush Hour: Adaptation and Departure Time Coordination in Iterated Commuting

Terence Kelly
tpkelly@eecs.umich.edu
Artificial Intelligence Lab
University of Michigan
Ann Arbor, MI 48109 USA

May 18, 1997

Abstract

Morning commuters adjust their departure times in response to day-to-day changes in congestion. Advanced Traveler Information Systems (ATIS) may enable motorists to employ fundamentally new strategies when adapting their departure times to fluctuations in congestion. At the same time, new driver strategies will likely give rise to different road network behaviors. This paper explores the mutual feedback between driver strategy and traffic system performance through a simulation model of rush hour commuting. Motorists in this model choose departure times according to three adaptive strategies. When commuters apply adaptive strategies that require ATIS in the present model, outcomes for both individual motorists and the system as a whole are by several measures worse than when drivers use a simple strategy that does not require ATIS. These results largely agree with an earlier study of a nearly identical model of rush-hour commuting.

This document is available in HTML on the World Wide Web at
<http://www-personal.engin.umich.edu/~tpkelly/rh/>

1 Introduction

Intelligent Transportation Systems (ITS) technologies are exciting for a number of reasons. For motorists, Advanced Traveler Information Systems (ATIS) hold the promise of faster and more convenient driving through navigational aids. For urban planners and policymakers, ATIS may offer an opportunity to improve the performance of traffic systems through real-time route guidance. ATIS technologies add a new dimension to traffic modeling because they *render plausible* several assumptions of traditional economic and game-theoretic models, e.g. agent rationality and perfect information.

Well-known results from economics, game theory and traffic science tell us that fundamental conflicts may exist between the interests of individual motorists and the overall performance of traffic systems. Because the various ITS technologies empower both traffic engineers and individual drivers, these technologies require new and better models to help us make wise tradeoffs between user-level and system-level interests. In particular, we must understand how traffic systems respond when the information and computational power available to motorists is increased. Traffic science provides us with numerous paradoxical results which demonstrate that traffic systems seldom respond to policy intervention or technological advances in simple or intuitive ways. Before our society invests hundreds of dollars each on ATIS systems for tens of millions of vehicles, we would like some assurance that conditions on our roads will not worsen as a result.

In this work I use a simple simulation model to explore the system-level consequences of an abstract ATIS technology, under the assumption that motorists use the technology in reasonable ways. The model makes only modest and plausible assumptions about the information and computational capacity available to motorists. Briefly, we assume that every vehicle is equipped with a global positioning system (GPS) and a computer which together allow each simulated driver to record basic facts about its driving history and perform simple analyses of these data. We examine how traffic system performance evolves in response to the local application by each driver of simple adaptive strategies in an iterated “rush hour” commuting scenario. This is the same issue addressed in [27]. Here I attempt to reproduce the results reported in [27] using *exactly* the same model, with one important difference: whereas the earlier results were generated by a car-following simulation model, here the microsimulation is based on a link performance function. One goal of [27] was to offer qualitative suggestions regarding the wisdom of enabling or encouraging various driver strategies through ATIS technology. The goal of the present investigation is to see whether the conclusions of [27] are robust when the microsimulation model is changed.

2 Previous Work

2.1 Traffic Paradoxes

Traffic science abounds with counterintuitive results. Perhaps the most famous of these is Braess’s paradox, in which adding a new link to a congested network can increase travel times on all links (see [42] for an early English-language description of Braess’s

1968 German-language paper). In [59] the authors demonstrate that “under reasonable assumptions, Braess’s Paradox is not a curious anomaly but in fact might occur quite frequently.” Reasonable examples can furthermore be constructed where travel times in a network decrease in response to increased inflows [16]. Paradoxes can also occur when motorists choose departure times rather than routes as in Braess’s example [3].

2.2 Effects of ITS Technologies

Intuitively, we expect that ITS technologies such as ATIS can only improve the lot of drivers and the performance of traffic systems. For instance, one simple model of the benefits of dynamic route guidance yields a savings of 3–4% in total system travel time [26]. Many motorists are eager to receive traffic information and apply it to their driving decisions: one large-scale opinion survey of commuters suggests that a substantial fraction are willing to change departure time or route choice based on traffic information [5]. It is unwise to assume, however, that *any* implementation of ATIS is sure to bring about better outcomes for both users and systems, just as it is was unwise to assume that air bags and anti-lock brakes would result in simple, intuitive safety improvements. A substantial body of literature suggests that under certain circumstances ATIS can have little impact or even adverse effects on traffic system performance.

The authors of [10] consider an equilibrium model of road networks containing both unguided vehicles and vehicles which receive central guidance intended to reduce total network congestion. They find that “increasing the number of guided vehicles should increase congestion in the network.” A recent simulation study of the Ali-Scout dynamic route guidance system concludes that “the benefits of Ali-Scout are significant only when the level of market penetration is *below* a certain level” (emphasis added) and explains that “equipped vehicles are taking the same routes and causing their own congestion” [20]. A simulation study of the potential of ATIS technology to alleviate non-recurrent (incident) congestion similarly concludes that the additional benefits to ATIS-equipped drivers decrease rapidly as market penetration increases [14], and [1] reports that benefits to guided vehicles in a simulation study of incident congestion level off once a critical fraction of vehicles are guided. Hall argues in [23] that ATIS strategies for alleviating peak-period incident-caused congestion will yield little long-term benefit, and that ATIS should not be viewed as an alternative to increased capacity.

2.3 Rush Hour, Version One

In [27] I explored the effects of a variety of driver-level adaptive strategies on overall traffic system performance using a car-following simulation model of iterated “rush hour” commuting. As in the present model described in Section 3, drivers in my original rush hour world had knowledge of their own past commuting experiences and used this information to guide their departure time decisions. I found that “In the context of this model, increasingly sophisticated agent-level commuting strategies result in decreased system-level performance as measured by several criteria.”

This basic finding is similar in spirit to [2], where the authors present an analytic model of the rush hour in which drivers choose both departure times and routes. They find that

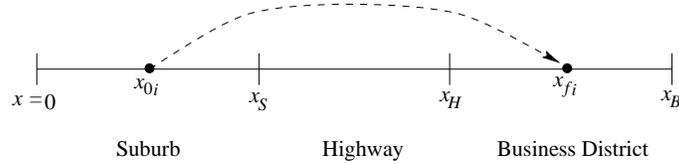


Figure 1: The Rush-Hour World

providing imperfect information on congestion and capacity conditions to all drivers “can cause drivers to change their departure times in such a way as to exacerbate congestion.” Influential work on morning commuting and departure time decisions can be found in [47, 46, 40] and the references therein.

The goal of the present project is to repeat exactly the experiments of [27] with one difference: the simulation model employs a link performance function rather than car-following logic.

3 Methods

3.1 The Rush-Hour World

The basic traffic scenario under study is nearly identical to that used in [27]. We have a roadway divided into three sections as shown in Figure 1: a “suburb” that extends from road location $x = 0$ to x_S along which are located houses from which commuters depart every morning, a “highway” between x_S and x_H containing neither origins nor destinations of trips, and a “business district” extending from x_H to x_B containing “offices” to which our commuters drive every morning. We require that $0 < x_S < x_H < x_B$.

Our N cars are evenly spaced along the suburb at locations $x_{0i} = i\Delta x$ where $\Delta x = x_S/(N + 1)$ and i indexes over cars, $i \in 1, \dots, N$. Each car furthermore has a destination or “office” to which it drives every simulated morning. These destinations x_{fi} are evenly spaced along the “business district”, $x_H < x_{fi} < x_B$. Cars are assigned to offices randomly. This is the *only* use of the random number generator. We say that a driver’s *ideal commuting time*, denoted t_{Ii} , is the time it would take that driver to reach work on a completely uncongested road. Unlike in [27], in this paper we have the simple relation $t_{Ii} = (x_{fi} - x_{0i})/\mu_f$ where μ_f is the free-flow speed or “speed limit.” A commuter’s *actual travel time* on a particular day j , denoted t_{Aij} , will depend on congestion.

The goal of every driver is to arrive at its office at exactly 9:00 a.m. every morning. In order to decide when to leave home on day j , each commuter computes a *predicted travel time*, denoted \hat{t}_{Aij} , based on past experience and leaves home at 9:00 a.m. minus this time. A driver’s *strategy* is the mapping from its past experience to \hat{t}_{Aij} , where past experience consists of t_{Ii} and $t_{Ai1} \dots t_{Ai(j-1)}$. The only information available to each driver is its own past commuting history, which we may imagine it obtains from a global positioning system and in-vehicle computer.

After motorists have selected their departure times, the day’s commute is entirely

deterministic. We use a link performance function, described in detail in Section 3.3, to determine how long each commuter will spend in each of the three segments of our roadway. The link performance function determines a travel time based on the distance to be traveled (e.g. $x_S - x_{0i}$ for the suburban part of the journey) and the number of cars already present on the entire link in question. This simple model, while markedly different from a traditional link performance function, allows us to completely avoid issues of car following and queueing, thus greatly simplifying the model implementation. The original rush hour simulation study in [27] used a relatively elaborate car-following model which allowed queueing delays at destinations to occur endogenously.

3.2 Driver Strategies

On the first simulated day of commuting, drivers have no past experience to guide their choice of departure time. Each driver therefore assumes that it will experience no congestion delay and leaves home at 9:00 minus t_{Ii} . On subsequent days, each motorist examines its past history and selects a departure time according to one of three adaptive strategies:

The yesterday rule: Assume that it will take as long to drive to work today as it did yesterday: $\hat{t}_{Ai(j+1)} = t_{Aij}$.

The mean rule: Compute the mean of previous commuting times and assume that today's commute will take that long.

The OLS rule: Perform an ordinary least-squares (OLS) linear regression on past commuting times and predict today's commuting time by extrapolation.

These simple rules are meant to represent the kinds of principles that real human drivers might employ, given time series data on their own commuting histories and simple computers. The adaptive strategies considered here do not include an elegant model described in [25]. In this model, predicted travel time is given by

$$\hat{t}_{Ai(j+1)} = \alpha + \beta(\hat{t}_{Aij} - t_{Aij}) + t_{Aij} + \epsilon \quad (1)$$

where α and β are parameters and ϵ is a random error term. This equation provides an intuitively plausible model of how an unaided human might update predicted travel time in response to new information, but it is not necessarily a good model of how a driver would use a GPS and computer. Furthermore, as the authors admit in [25], it is difficult to estimate parameters for this model, and the estimated models do not fit data from human subjects well.

As in [27] the agent population is always homogeneous with respect to strategy. During any given run of the rush hour simulation, all agents use the same adaptive strategy every day; there is never an instance when, for instance, an "OLS driver" shares the road with a "yesterday rule" driver.

3.3 Link Performance Function

The link performance function that we use is inspired by an equation attributed to Davidson [56, equation 13.21, page 358]. The function returns travel time on a link t_a as a function of flow on the link x_a , free-flow travel time t_a^0 , link capacity c_a and a model parameter J :

$$t_a = t_a^0 \left[1 + J \frac{x_a}{c_a - x_a} \right] \quad (2)$$

This function has a number of desirable properties that make it preferable to the widely-used U.S. Bureau of Public Roads link performance function: travel time t_a approaches infinity as flow on a link approaches capacity c_a , and it contains a single model parameter J where the B.P.R. function has two. I have devised a link performance function for the present investigation that is very similar to Davidson's except that it considers density k on a link *at the instant when a car enters the link* and jam density k_j rather than recent flow into the link:

$$t_a = t_a^0 \left[1 + K \frac{k}{k_j - k} \right] \quad (3)$$

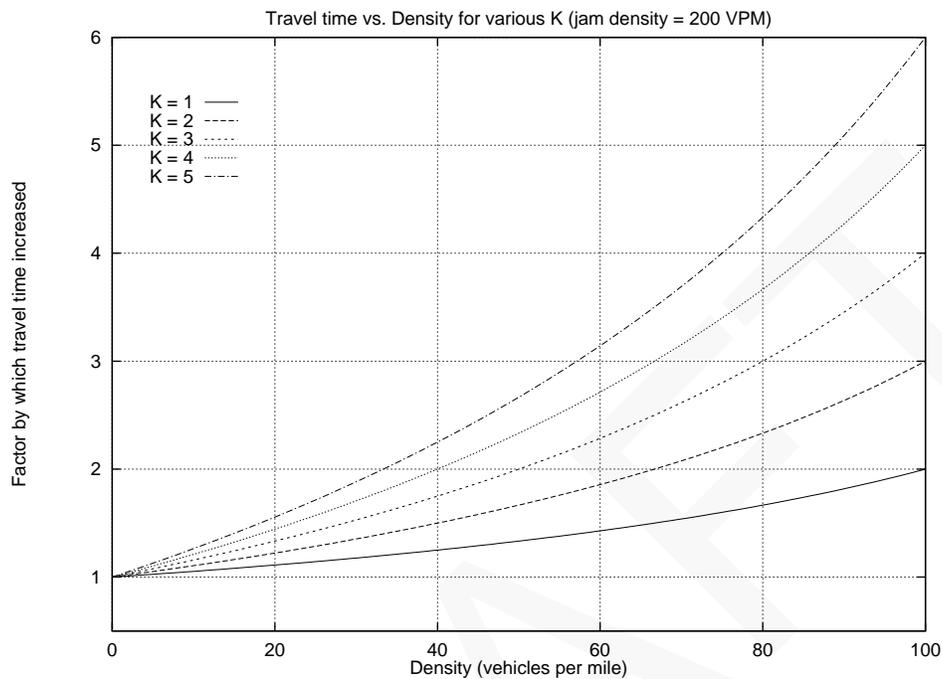
For convenience, the implementation of my link performance function uses as parameters a jam density k_j and a free-flow speed μ_f rather than a free-flow travel time per link. It computes a travel time t_a based on the density of traffic on a link and the distance along the link which a car will travel. It is straightforward to convert equation 3 into the desired form:

$$t_a = \left(\frac{d}{\mu_f} \right) \left[1 + K \frac{k}{k_j - k} \right] \quad (4)$$

where d is the distance to be traveled, μ_f is the free-flow speed, k and k_j are respectively link density and jam density and K is a model parameter. Note that this link performance function implies that a motorist entering an unoccupied link is guaranteed an unobstructed journey along that link at speed u_f . This creates a strong incentive for a motorist to "beat the rush" by departing before anyone else.

Note that it is simple to construct departure time decisions that can result in passing when we use equation 4. One scenario in which passing occurs can be constructed as follows: let $N = 100$ and let cars 3 through 100 select departure times such that they all arrive at x_S at the same time T_S . Let car 2 depart very shortly before time T_S , and let car 1 depart shortly after T_S . Recall from Section 3.1 that car 1 is furthest from x_S and car N is nearest to x_S . Because k on the suburban section of our road is far higher when car 2 enters the road than when car 1 departs ($k = 98/x_S$ for car 2 and $k = 1/x_S$ for car 1), car 1 will arrive at x_S before car 2 for reasonable parameter values.

The C language implementation of the travel time function is shown below; see Appendix A for a complete listing of the remaining simulation code. Since u_f is expressed in miles per hour and density in vehicles per mile, equation 4 computes a travel time in hours, so we must multiply by 3600 to convert to seconds. We use global variables `g_k_j`

Figure 2: The effect of K on travel time.

and g_{μ_f} to respectively convey k_j and μ_f to the function. Global variable g_K contains the K model parameter.

```

6  <link performance function 6>≡
    static Time travel_time(const double dist, const double density) {
        Time t;
        assert(0.0 < dist);
        assert(0.0 <= density);
        assert(density < g_k_j);
        t = (dist / g_mu_f) *
            (1.0 + (g_K * density) / (g_k_j - density)) * 3600.0;
        assert(0.0 < t);
        return t;
    }

```

3.4 Parameter Selection

In the interest of “backward compatibility” all parameters used in [27] are also used in the experiments reported here; the goal is to vary nothing other than the microsimulation method. (Indeed, even the random number seeds remain unchanged; see sections 3.5 and C for details.) However, the K and k_j parameters discussed in Section 3.3 are new to this model (in [27] there was no link performance function and hence no K , and k_j was endogenous to the microsimulation). The choice of k_j for the present study was somewhat arbitrary. A 1959 study reported in [18, chapter 4, page 17] gives a figure equivalent to roughly 229 vehicles per mile (VPM). If we assume that jam density occurs at one vehicle per 25 feet, this implies $k_j = 211$ VPM. I opted for a lower value, $k_j = 200$ VPM, because the “suburb” and “business district” of our roadway contain many “driveways” where vehicles enter and exit the road, and I reason that k_j should be lower under such conditions than in ordinary traffic.

Choosing a value for K is a more complicated matter. The K parameter of the link performance function determines the rate at which travel time on a link approaches infinity as density approaches k_j . Figure 2 shows the factor by which travel time is extended as K is varied for $k_j = 200$ VPM. I settled on $K = 3$ because given the x_S , x_H , x_B and N parameters used here the maximum attainable k value is 100 VPM, and with $K = 3$ this increases travel time by a factor of four. For $u_f = 60$ MPH this implies a speed of 15 MPH.

3.5 Experiments

The data reported in Section 4 were obtained from sixty runs of my `sim` program. Parameter values were as follows: $x_S = 1$, $x_H = 4$, $x_B = 5$, number of cars $N = 100$, number of simulated days $M = 100$, number of offices = 10, free flow speed $\mu_f = 60$ miles per hour, $k_j = 200$ vehicles per mile, and link performance function parameter $K = 3$. The only other input parameters to the `sim` program are a random number seed and an adaptive rule. For each of the three adaptive rules I ran the program with each of twenty random number seeds. To see the parameters passed to the `sim` program, refer to Appendix C, which contains the shell script used to run the `sim` program and generate all results reported in this paper. A complete annotated listing of the C code behind the `sim` program is contained in Appendix A.

4 Results

Following [27], we will examine how a number of system-level performance metrics respond when we vary the adaptive strategy used by our driver-agents. In Figure 3 we see a time series of each day’s total commuting time normalized to the total ideal commuting time. More formally, for each day j we plot

$$\frac{\sum_{i=1}^N t_{Aij}}{\sum_{i=1}^N t_{Ii}} \quad (5)$$

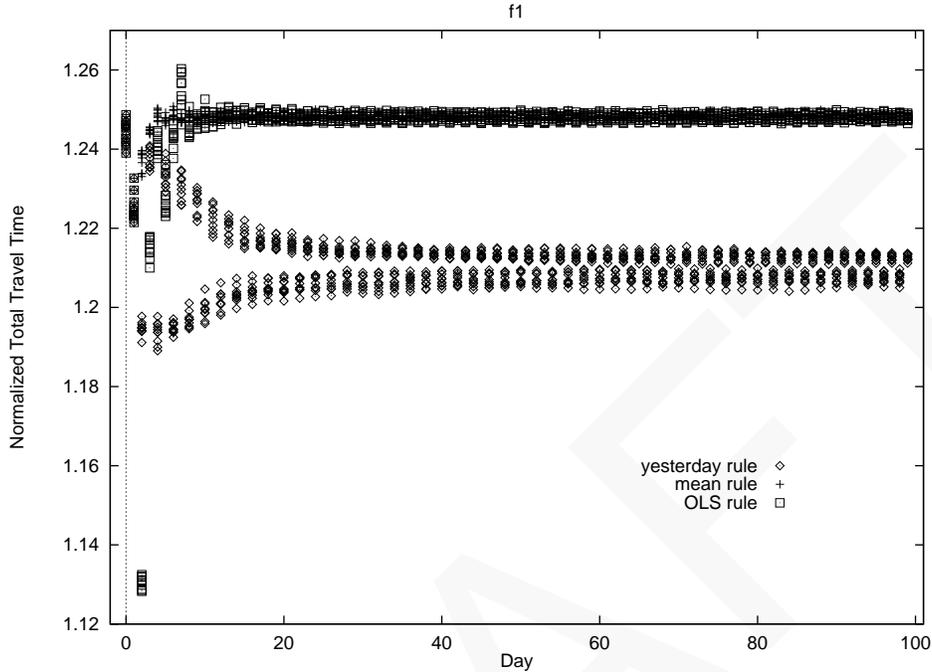


Figure 3: The effect of adaptive strategy on total travel time.

(If you inspect the C code that performs this computation in Appendix A, note that arrays are indexed from 0 to $N - 1$.) We see that when every driver follows the “yesterday rule” aggregate travel time tends toward a markedly lower level than when commuters use either of the other adaptive strategies.

In Figure 4 we plot mean lateness per day for each of our sixty simulation runs. Total lateness $L(j)$ on each day j is given by

$$L(j) = \sum_i \max(0, t_{Aij} - 9 \text{ a.m.}) \quad (6)$$

where the summation is over cars i . The figure shows $L(j)/N$. In this figure we see that when commuters follow the yesterday rule mean lateness is markedly higher than when they employ either of the other strategies. Note that $L(j)$ can never be negative: when a commuter arrives at work *early* this makes no contribution to $L(j)$.

A crude measure of the “fairness” of a road system on a given day is the standard deviation of all drivers’ *relative delays*, where the relative delay d_{Rij} of driver i on day j is the extent to which congestion extends its commute: $d_{Rij} = (t_{Aij} - t_{Ii})/t_{Ii}$. The standard deviation of the d_{Rij} on a given day j will be zero if all drivers’ commuting times are extended by the same factor. In Figure 5 we see the standard deviation of drivers’ relative delays. The “yesterday rule” appears to offer some advantages by this measure

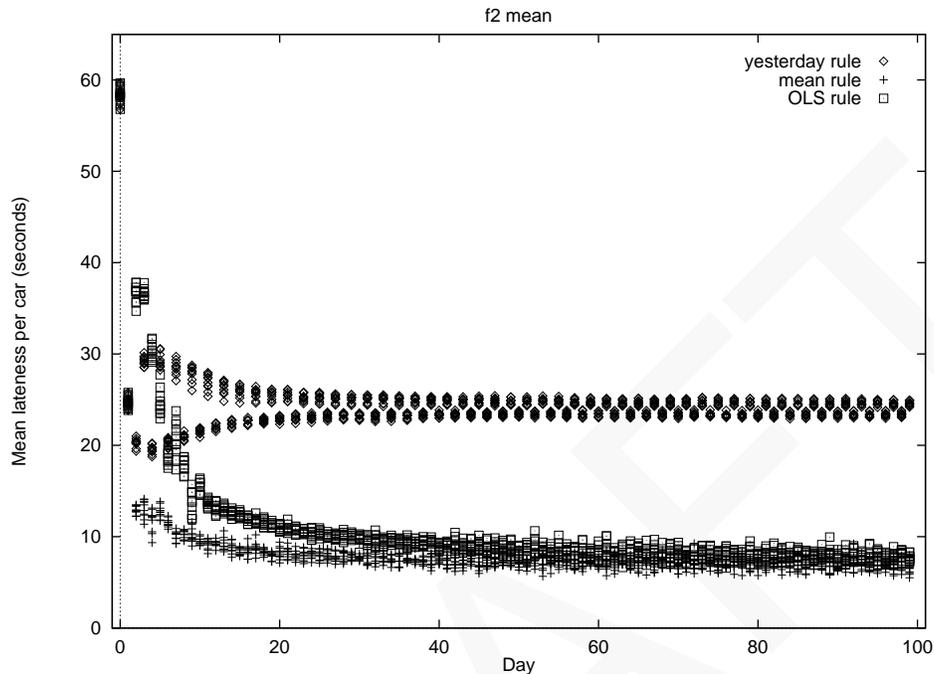


Figure 4: Mean of cars' latenesses.

up to around day 30, but after that time it appears that the “mean rule” yields more favorable outcomes.

Now we consider to what extent it is advantageous to live near one’s workplace. In Figure 6 we plot for each car number i the mean value of that car’s relative delays over 100 simulated days. Recall from Section 3.1 that cars are evenly spaced throughout the “suburb” of our roadway with car 1 located furthest from the “business district” and car N located nearest to it. A car’s number i therefore serves as a convenient scale-independent proxy for its origin location x_{0i} . We see from this figure that when all motorists follow the yesterday rule they tend to enjoy roughly the same relatively low levels of delay, regardless of whether they live near or far from work. By contrast, when they follow either of the other two adaptive strategies they experience more delay, and this hardship falls disproportionately on commuters who live near the middle of the suburb. This result is partly consistent with the finding reported in Figure 5 of [27].

Finally, we consider another measure of traffic system performance that real commuters care about, consistency. A commute that takes 30 ± 20 minutes is not necessarily preferable to one that takes 40 ± 5 minutes. We define the consistency of a car’s experience as the standard deviation of its absolute travel times. To normalize this measure we divide it by the car’s ideal travel time. Figure 7 shows the consistency of each car’s commuting experience. Among the three adaptive strategies considered, no clear winner is apparent in

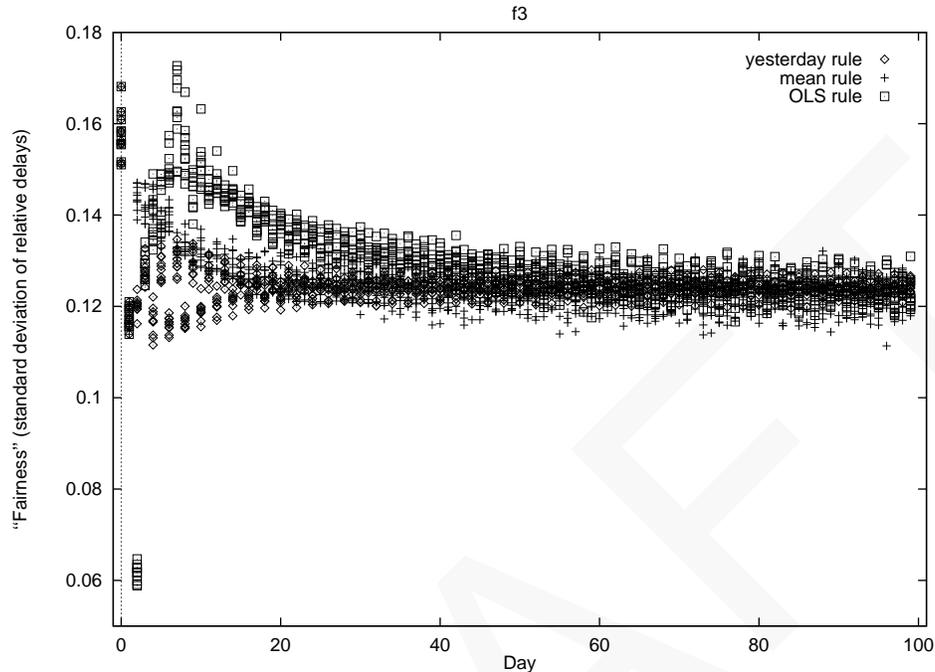


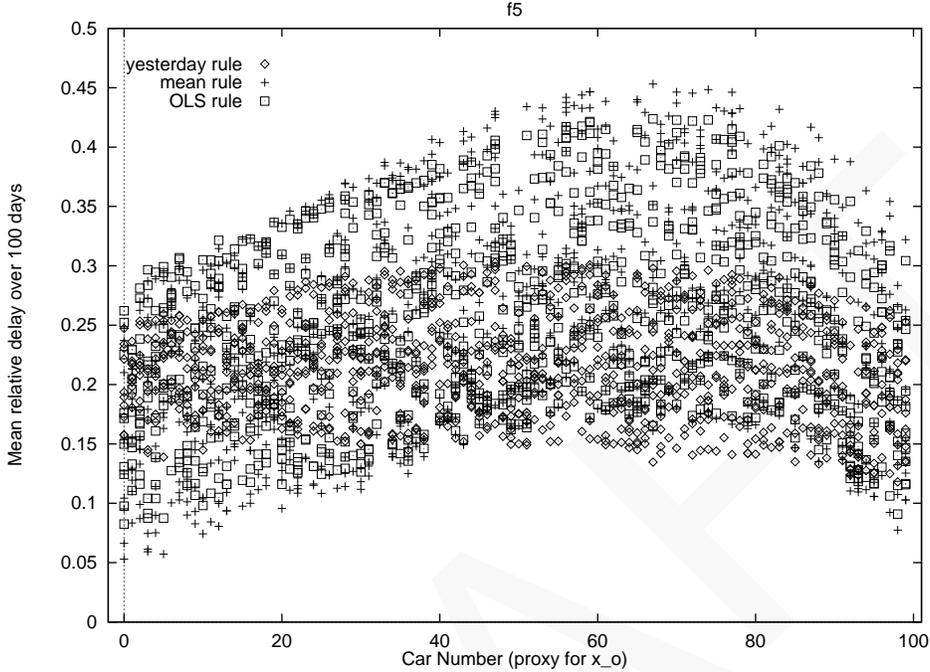
Figure 5: Fairness time series.

this figure, although it would seem that the mean rule yields the most favorable outcomes by the consistency measure.

4.1 The Life and Times of Car Number 50

We gain better intuition into the behavior of the link performance version of the rush hour model by considering how the experience of a single motorist varies over time under the three adaptive strategies. Figure 8 summarizes the commuting experiences of car number 50, who lives roughly in the middle of the suburb of our world, when we use random number seed `1f50 1a03 f4b5`. The left-hand column of figures shows the departure and arrival times of car 50 on each of our 100 simulated days and the right-hand column shows the actual travel time of car 50 on each day. The adaptive rule used to generate the top row was the yesterday rule. The center row results from the mean rule, and the bottom row the OLS rule. Note that 9:00 a.m., the time at which our driver-agents are due at work, is equivalent to 32,400 seconds after midnight.

We see that in all three cases car 50's departure time (the lower of the two series in the left-hand figures) rapidly converges to a relatively constant value, or a simple oscillation around a constant in the case of the yesterday rule. Most of the fluctuation in travel time is due to fluctuations in arrival time rather than departure time. The car's absolute travel time appears to converge to roughly 300 seconds in the yesterday rule run ($t_{I50} = 258.8$

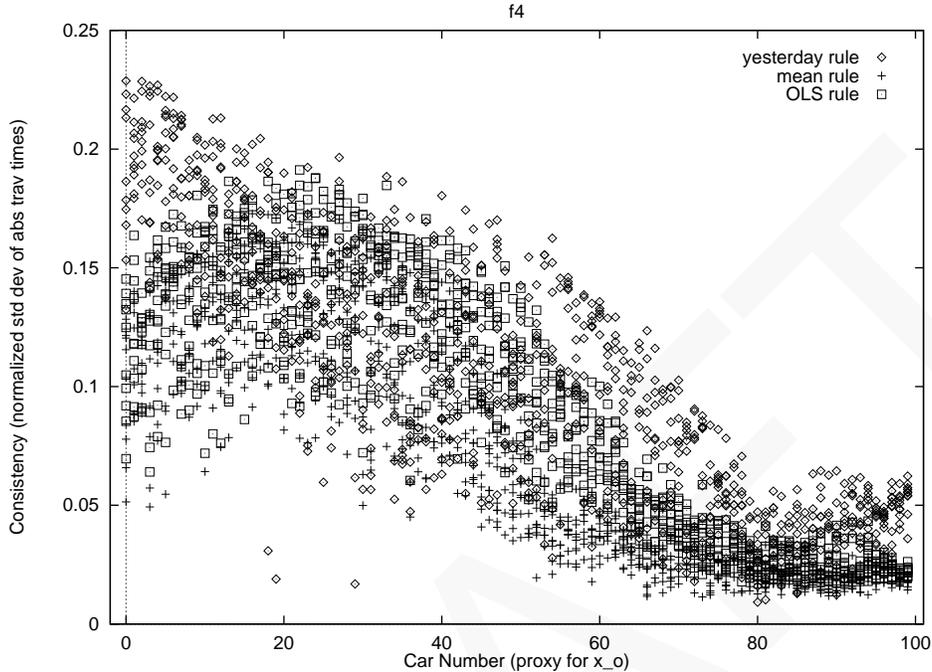
Figure 6: Mean relative delay vs. x_0 .

seconds for the random number seed used). When the mean rule is applied, the t_{A50j} time series oscillates with a larger amplitude and does not appear to converge toward any value. The OLS rule figure is still more interesting: the amplitude with which absolute travel time fluctuates itself varies with time.

The top row of Figure 8 provides an explanation for why the yesterday rule leads to lower travel times yet higher lateness: the variance in arrival times is higher when drivers follow the yesterday rule, and when we compute lateness we consider only positive lateness; in equation 6 no credit is given to drivers who arrive at work early. Overall, the figure suggests that the yesterday rule leads to better driver-level outcomes than either of the two other rules. In particular, it appears that individual motorists in the yesterday rule scenario experience increasingly consistent commuting times, although a more detailed analysis would be required in order to substantiate this impression.

5 Discussion

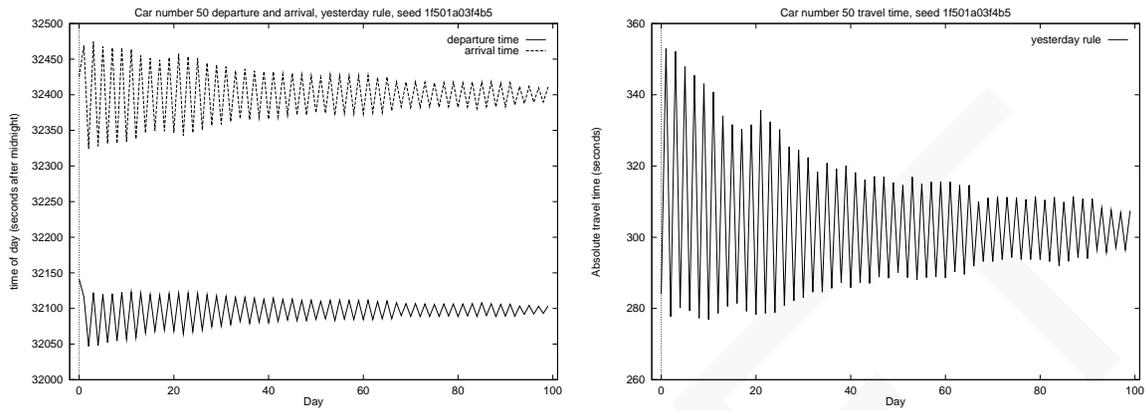
The most basic goal of this project was to repeat the experiment of [27] varying only the microsimulation method in order to test the robustness of the earlier reported results. Although the present investigation accomplishes this goal, the qualitative conclusions drawn from the earlier model are not strongly supported by these latest results. In [27] I found

Figure 7: Consistency vs. x_0 .

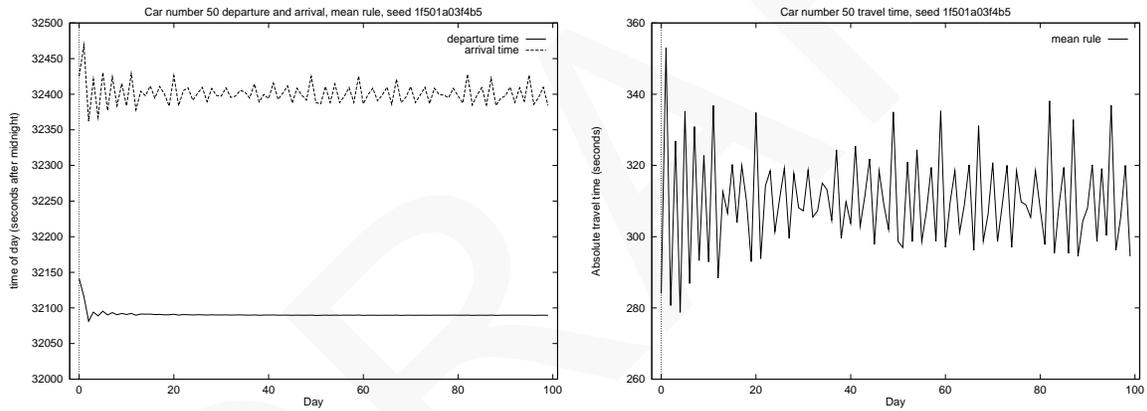
that the yesterday rule resulted in markedly better system-level outcomes by each of the performance metrics considered (aggregate lateness, fairness, consistency and mean relative delay). The yesterday rule is so simple that any motorist with a wristwatch can follow it, yet it led to better driver- and system-level outcomes than more sophisticated strategies requiring ATIS-like technology. The results of this paper are less clear. As in the earlier work, the yesterday rule leads to far better total travel times and mean relative delays than the other rules, but unlike in [27] it results in increased lateness. By and large, however, the yesterday rule performs quite well as compared with the more sophisticated strategies, particularly when we consider the experience of individual motorists in the model. If we accept the yesterday rule as a baseline strategy used by motorists *without* any kind of ATIS — after all, it requires only a wristwatch and a good memory — then it is not clear that ATIS systems which empower drivers to perform more sophisticated time-series analyses yield any benefit whatsoever.

It is hardly surprising that the results of [27] and the present model do not completely agree, given that they differ so fundamentally at the level of microsimulation. Indeed, it is remarkable that the results from the two models are comparable at all. The question is not which model is correct, because in neither case was the model intended to predict the behavior of an actual road network. The question is whether the conclusions we draw from these models lead us to the right traffic engineering decisions. We can ask

yesterday rule



mean rule



OLS rule

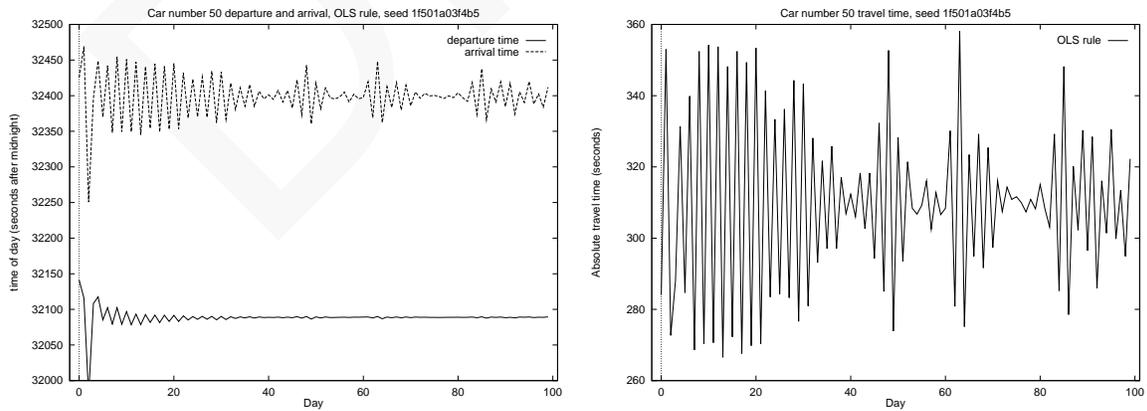


Figure 8: Departure, arrival and travel time data for car 50.

this same question about commercial traffic simulation packages. It might be interesting, for instance, to systematically compare version 1.5 of INTEGRATION, which uses link performance functions, with later releases, which use car-following logic.

Ambitious, large-scale traffic simulation models sometimes aspire to predictive power. This is the case, for instance, with the TRANSIMS simulation project currently underway at Los Alamos National Laboratories, which hopes to provide planners with a tool to assess the impact of changes to urban road networks. The model presented here is far more modest in its aims. It cannot offer quantitative predictions but merely qualitative suggestions concerning the likely effects of new technologies on traffic system performance. In [44] the authors cite a simulation model to argue that traffic systems are unstable when operating at maximum flow, and therefore Advanced Traffic Management Systems (ATMS) should not be used to push road networks toward maximal flow. As in [27], the argument here is that ATIS technologies may enable new classes of commuting strategies, and that some of these strategies may degrade traffic system performance. This conclusion is still very tentative but results obtained to date suggest that it should be taken seriously.

5.1 Future Work

Commuting under congested conditions is clearly a kind of non-zero-sum game, and models such as the one explored in this paper deserve game-theoretic analysis. The microsimulation logic computes a mapping from a vector of motorists' departure times to a vector of arrival times. If this function is sufficiently well-behaved it is straightforward to define concepts of user (i.e. Nash) equilibria, system optimality, and Pareto efficiency. However in the present model the mapping from departures to arrivals is not even continuous, and it is not clear how to analyze the "payoff matrix" embodied by this function. Other departure-to-arrival mappings might be considered, but finding a mapping that is both analytically tractable and plausible as a model of road system behavior is a formidable challenge.

We can also view the problem of rush-hour commuting as a coordination problem. Indeed, many parallels exist between my iterated commuting scenario and economist Brian Arthur's "El Farol Problem" [4, 8]. Whereas Arthur's agents must decide whether or not visit a particular location (Arthur's favorite pub in Santa Fe, the El Farol), my motorists must decide *when* to depart for their destinations. Like Arthur's agents, my drivers suffer if they fail to stagger their decisions appropriately. It may be worthwhile to further consider "the Canyon Road Problem" (the El Farol is located on Canyon Road). The interesting question is whether there exist agent-level adaptive strategies which lead the system to globally favorable outcomes, as Arthur discovered in a simulation of the El Farol Problem. If such strategies do exist, then the policy implication is clear: we should prefer ATIS systems which facilitate them.

Other obvious directions for future work involve other microsimulation techniques. Now that car-following and link performance functions have been explored it might be worthwhile to implement a "particle hopping" (i.e. cellular automata) model [44, 43, 54]. Another dimension along which the model might be varied is agent strategy. A variety of adaptive rules other than those used here, for instance that of equation 1, might be

used. Or the mean rule and OLS rule could be modified to examine only a recent window of a driver's past experience. It would be interesting to explore a mixed simulation involving heterogeneous driver populations where, for instance, yesterday rule motorists share the road with OLS rule commuters. Finally, a detailed parameter-sensitivity analysis of various microsimulation models would shed a different kind of light on the robustness trustworthiness of the models.

DRAFT

A Simulation Code Listing

WARNING: The appendices of this report are in very rough form. The code listing below is complete, but the typeset comments surrounding the code are not yet finished. Later drafts of this paper will contain more thorough explanations of the code.

Thanks to the magic of literate programming, all of the C source code used to generate the simulation results described in this paper is listed somewhere within the document itself. The particular tool used is Norman Ramsey's `noweb`. See [52] and [51] for details. See Appendix B of this paper for a brief introduction to the concept of literate programming.

A.1 Main Program

We begin with a basic outline of the program:

```
16a <main.c* 16a>≡
    <warning and ID 16b>
    <file includes 17a>
    <named constants 17b>
    <data types 18a>
    <global variables 18b>
    <function definitions 20>
    <main function 22a>
```

This code is written to file `main.c`.

Now let's look at each of these sections in detail. We begin by warning folks not to edit the `.c` file but rather the `noweb` source file, and provide information about the latter:

```
16b <warning and ID 16b>≡
/*
 * WARNING: DO NOT EDIT THIS FILE!
 * Edit only the noweb source file from which it is derived.
 * $Id: project.nw,v 1.58 1997/05/18 08:42:04 tpkelly Exp tpkelly $
*/
```

The `Id` string is automatically generated by RCS, the Unix Revision Control System. It contains information about the current version of the `noweb` source file, the date on which the file was last modified, and the owner of the file. Read `man rcsintro` and `man rcs` for more. Now for the real code, first we'll include a number of standard header files. We use the `assert()` macro defined in `assert.h` to verify run-time conditions, test loop invariants and the like. We must define a special preprocessor token in order to obtain our random number generator function prototypes from `stdlib.h`.

```
17a  (file includes 17a)≡
      #include <assert.h>
      #include <ctype.h>
      #include <errno.h>
      #include <math.h>
      #include <stdio.h>
      #define __EXTENSIONS__
      #include <stdlib.h>
      #undef __EXTENSIONS__
      #include <string.h>
      #include <sys/types.h>
      #include <sys/utsname.h>
      #include <time.h>
      #include <unistd.h>
      #include "fpe.h"
      #include "main.h"
```

Some limitations of this implementation of the model and rather arbitrary numbers. Units of distance are miles, units of time are seconds.

```
17b  (named constants 17b)≡
      #define MAX_CARS      100
      #define MAX_OFFICES  20
      #define MAX_DAYS     100
      #define MAX_X        1000.0
      #define TARGET_TIME  32400.0 /* 9:00 a.m. in seconds */
```

Now for some data types:

```
18a  <data types 18a>≡
      typedef double Time;
      typedef enum {
          ADAPT_TOLOW,
          ADAPT_YESTERDAY,
          ADAPT_MEAN,
          ADAPT_OLS,
          ADAPT_TOOHIGH
      } Adaptation_type;
      typedef enum {
          EVENT_TOLOW,
          ENTER_ROAD,
          ARRIVE_Xs,
          ARRIVE_Xh,
          EXIT_ROAD,
          EVENT_TOOHIGH
      } Event_type;
      struct Event {
          Time time;
          Event_type type;
          int car;
          struct Event *next;
      };
```

The schedule of events is implemented as a simple linked list of **Event** structures. The pointer to the head of the list is a global variable. Other global variables store model parameters as described in Section 3. The use of a linked list as a schedule data structure isn't very efficient but for the present purpose simplicity is more important. I use prefix "g_" to indicate global variables.

```
18b  <global variables 18b>≡
      static struct Event *g_Schedule = NULL;
      static double g_mu_f;
      static double g_k_j;
      static double g_K;
      static char *g_adapt_str[] = { NULL, "yesterday rule", "mean rule",
                                     "OLS rule", NULL };
      static char *g_event_str[] = { NULL, "nr", "Xs", "Xh", "xr" , NULL };
      char *g_argv_0 = NULL;
```

Some global variables are available to both modules of the program and are therefore placed in a header file. The `g_argv_0` pointer is set equal to `argv[0]` in the `main()` function to facilitate diagnostic error messages containing the name of the program. The `ERROR_FORMAT` string is used in error messages to specify the formatting of the coordinates (i.e. program name, file name and line number) where the error occurred.

```
19 <main.h* 19>≡
    #ifndef MAIN_H_INCLUDED
    #define MAIN_H_INCLUDED
    #define ERROR_FORMAT "%s:%s:%d: "
    /* verbose alternative:
    #define ERROR_FORMAT "program '%s' file '%s' line %d: "
    */
    extern char          *g_argv_0;
    #endif /* #ifndef MAIN_H_INCLUDED */
```

This code is written to file `main.h`.

Function to insert a new event. Talk about memory allocation and how we must check return value of `malloc()`. NOT appropriate to use assertions for this. Note that the memory allocated here must be freed elsewhere. Quote LCLint, every allocation creates an obligation to de-allocate. Normally we'd wrap `calloc()` within a wrapper function & macro to centralize error checking, but since it only occurs once in this program we'll perform the check here. If it fails we report detailed diagnostic and dump core file. Note that autopsy of core file is easy.

```
20 (function definitions 20)≡
    static void insert_event(const Time t, const Event_type ty,
                           const int car) {
        struct Event *new_ev, *p, **q;
        assert(EVENT_TOOHIGH > ty);
        assert(0.0 < t);
        assert(0 <= car);
        new_ev = (struct Event *)calloc((size_t)1, sizeof(*new_ev));
        if (NULL != new_ev) {
            new_ev->time = t;
            new_ev->type = ty;
            new_ev->car = car;
            for (q = &g_Schedule, p = g_Schedule; NULL != p && p->time < t;
                 q = &(p->next), p = p->next) {
                if (t == p->time) {
                    (void)fprintf(stderr, ERROR_FORMAT "warning: two events "
                                   "have same time\n", g_argv_0, __FILE__,
                                   __LINE__);
                }
            }
            /* p now points to next event after time t and *q points to
             * event immediately prior to t, so we splice the new event in
             */
            new_ev->next = p;
            *q = new_ev;
        } else {
            (void)fprintf(stderr, ERROR_FORMAT "calloc returned NULL\n"
                           "  errno == %d: %s\n Aborting...\n", g_argv_0,
                           __FILE__, __LINE__, errno, strerror(errno));
            abort();
            /*NOTREACHED*/
        }
    }
}
```

The following function removes the next event from the schedule and returns a pointer to it, or **NULL** if no events remain. It is the responsibility of the caller of this function to de-allocate storage associated with events.

```
21a  (function definitions 20)+≡
      static struct Event *get_next_event(void) {
          struct Event *p;
          p = g_Schedule;
          if (NULL != p) {
              g_Schedule = g_Schedule->next;
              p->next = NULL;
          }
          return p;
      }
```

If we compile with a “-DPRINT_EVENT_QUEUES” flag, the contents of the event queue will be printed when control reaches certain points in the code.

```
21b  (function definitions 20)+≡
      #ifdef PRINT_EVENT_QUEUES
      static void print_event_queue(void) {
          struct Event *p;
          (void)printf("\nContents of event queue:\n\n"
                      "   Time (sec):  Type:  car:\n\n");
          for (p = g_Schedule; NULL != p; p = p->next) {
              (void)printf("   %9.2f  %5d  %4d\n", p->time, p->type, p->car);
          }
      }
      #endif
```

Now we need a function to compute time required to travel a certain distance on a link as a function of density on the link. This function is described mathematically, and listed, in Section 3.3.

```
21c  (function definitions 20)+≡
      (link performance function 6)
```

Now we're ready for the main function. The call to `fpe_set()` arranges for floating-point exceptions (e.g. division by zero, overflow, underflow) to be detected at run time. Unfortunately this is not default behavior on Solaris. See Appendix A.2 for details.

```
22a  (<main function 22a>≡
      int main (int argc, char *argv[]) {
          <variable declarations 22b>
          g_argv_0 = argv[0];          /* to aid diagnostic output */
          fpe_set();                  /* detect FP exceptions */
          <process command line 24>
          <check input 26>
          <seed random number generator 28a>
          <initialize Xo and Xf 28b>
          <print output header 29a>
          <print origin and destination information 31a>
          <main loop 31b>
          <print results 36a>
          return 0;
      }
```

The following static variables store quantities like x_{0i} and t_{Ii} . The purpose of each should be fairly obvious from the names. Note that the ANSI C standard requires that static variables be initialized to zero. The non-static variables below will serve a variety of miscellaneous purposes. Integer variables `Np`, `Ns`, `Nh`, `Nb`, `Na` will track the number of cars in each segment of our roadway during each day's commute. `Np` and `Na` respectively store the number of cars that have not yet departed ("parked") and the number that have arrived at their destinations.

```
22b  (<variable declarations 22b>≡
      static double      Xo [MAX_CARS];
      static double      Xf [MAX_CARS];
      static Time        ideal [MAX_CARS];
      static Time        depart [MAX_CARS] [MAX_DAYS];
      static Time        arrive [MAX_CARS] [MAX_DAYS];
      static Time        mean_speed [MAX_CARS] [MAX_DAYS];
      static Time        abs_lateness [MAX_CARS] [MAX_DAYS];
      static Time        abs_delay [MAX_CARS] [MAX_DAYS];
      static Time        rel_delay [MAX_CARS] [MAX_DAYS];
      int day, car, office, num_events_today, c, i;
      int Np, Ns, Nh, Nb, Na;
      int print_events_flag = 0, print_daily_flag = 0;
      struct Event *ep;
      Time linktime, yesterday_time, total_time, trav_time;
      double deltaX;
      struct utsname struct_utsname;
      time_t tod; /* "time of day" */
      uid_t userid;
      <model parameters 23>
```

To aid debugging and input checking we initialize all user-supplied variables to impossible values. Later we will use conditional tests to verify that the user has specified valid parameter values to override these defaults, and we use assertions to detect that none of these variables have their initial values due to programming errors.

23 *(model parameters 23)*≡
Adaptation_type Adapt = ADAPT_TOOLOW;
double Xs = -1.0, Xh = -1.0, Xb = -1.0;
int Numcars = -1, Numdays = -1, Numoffices = -1;
char *Random_seed = NULL;
unsigned short seed16v[3] = {0, 0, 0};

Now we process the command line using the standard `getopt(3C)` function. Most user-supplied command line arguments set model parameter values like x_S and μ_f . But the `D` and `S` flags request more detailed output, and the `-F` flag when used with `-s 0` should cause a division by zero floating exception to occur, allowing us to verify that the FPE detection code is working properly.

```

24  {process command line 24}≡
    {check errno 25}
    while (EOF != (c=getopt(argc, argv, "a:s:h:b:c:d:o:r:u:j:K:FSD"))) {
        switch (c) {
            case 'a':
                Adapt      = atoi(optarg);    break;
            case 's':
                Xs          = atof(optarg);    break;
            case 'h':
                Xh          = atof(optarg);    break;
            case 'b':
                Xb          = atof(optarg);    break;
            case 'c':
                Numcars     = atoi(optarg);    break;
            case 'd':
                Numdays    = atoi(optarg);    break;
            case 'o':
                Numoffices  = atoi(optarg);    break;
            case 'r':
                Random_seed = optarg;         break;
            case 'u':
                g_mu_f      = atof(optarg);    break;
            case 'j':
                g_k_j       = atof(optarg);    break;
            case 'K':
                g_K         = atof(optarg);    break;
            case 'F':
                (void)printf("%f\n", 1.0/Xs);  break; /* FPE if Xs = 0 */
            case 'S':
                print_events_flag = 1;        break;
            case 'D':
                print_daily_flag = 1;         break;
            case '?:
                /*FALLTHRU*/
            default:
                (void)fprintf(stderr, ERROR_FORMAT "unrecognized option '%c'\n"
                    "Exiting...\n",
                    g_argv_0, __FILE__, __LINE__, c);
                exit(EXIT_FAILURE);
                /*NOTREACHED*/
                break;
        }
    }
}

```

<check errno 25>

We check the global `errno` variable both before and after converting command line and warn the user if it is nonzero. One of the evil features of the standard C library is that functions often hide error codes in their return values. If the `atof()` function is passed a string that would cause underflow, for instance, `errno` would be set to `ERANGE` and the function would return zero [28, page 251]. So `atof()` returns zero when it is given an input string like "0" *and* when an underflow has occurred. The only way to be sure that errors in input conversion have not occurred is to test `errno`. Note that the procedure used here differs from the expert advice given in [30, page 73]. The advice given there does not apply to cases where a function hides an error code in a legitimate return value.

```
25 <check errno 25>≡
    if (0 != errno) {
        (void)fprintf(stderr, ERROR_FORMAT "warning: errno == %d: %s\n",
                    g_argv_0, __FILE__, __LINE__, errno,
                    strerror(errno));
        errno = 0;
    }
```

Now we perform a great many careful checks on user-supplied input parameters. Nowadays “garbage in, garbage out” is an unacceptable programming philosophy. The rule enforced in my code is “garbage in, error message out.”

```

26 (<check input 26>≡
    if (!(ADAPT_TOLOW < Adapt && ADAPT_TOOHIGH > Adapt)) {
        (void)fprintf(stderr, ERROR_FORMAT "bad adapt value: %d\n"
            "Exiting...\n",
            g_argv_0, __FILE__, __LINE__, Adapt);
        exit(EXIT_FAILURE);
        /*NOTREACHED*/
    }
    if (!(0.0 < Xs && Xs < Xh && Xh < Xb && Xb < MAX_X)) {
        (void)fprintf(stderr, ERROR_FORMAT "must have "
            "0 < Xs < Xh < Xb < %f\n"
            "Exiting...\n",
            g_argv_0, __FILE__, __LINE__, MAX_X);
        exit(EXIT_FAILURE);
        /*NOTREACHED*/
    }
    if (!(0 < Numcars && 0 < Numdays && 0 < Numoffices)) {
        (void)fprintf(stderr, ERROR_FORMAT "must have "
            "Num[cars, days, offices] > 0\n"
            "Exiting...\n",
            g_argv_0, __FILE__, __LINE__);
        exit(EXIT_FAILURE);
        /*NOTREACHED*/
    }
    if (MAX_CARS < Numcars || MAX_OFFICES < Numoffices ||
        MAX_DAYS < Numdays) {
        (void)fprintf(stderr, ERROR_FORMAT "must have "
            "Numcars <= %d, Numoffices <= %d, Numdays <= %d\n"
            "Exiting...\n", g_argv_0, __FILE__, __LINE__,
            MAX_CARS, MAX_OFFICES, MAX_DAYS);
        exit(EXIT_FAILURE);
        /*NOTREACHED*/
    }
}

```

We are particularly careful about checking the user-supplied random number seed. If an error occurs in converting this parameter we are likely to lose reproducibility of results.

```

27 (<check input 26>)+≡
    if (NULL == Random_seed) {
        (void)fprintf(stderr, ERROR_FORMAT "missing random seed\n"
            "Exiting...\n",
            g_argv_0, __FILE__, __LINE__);
        exit(EXIT_FAILURE);
        /*NOTREACHED*/
    }
    if (12 != strlen(Random_seed)) {
        (void)fprintf(stderr, ERROR_FORMAT "random seed '%s' invalid\n"
            "Exiting...\n",
            g_argv_0, __FILE__, __LINE__, Random_seed);
        exit(EXIT_FAILURE);
        /*NOTREACHED*/
    }
    for (i=0; i < 12; i++) {
        if (!isxdigit(Random_seed[i])) {
            (void)fprintf(stderr, ERROR_FORMAT "random seed character "
                "%d = '%c' is not a hex digit\n"
                "Exiting...\n",
                g_argv_0, __FILE__, __LINE__, i+1, Random_seed[i]);
            exit(EXIT_FAILURE);
            /*NOTREACHED*/
        }
    }
    if (0.0 >= g_mu_f || 0.0 >= g_k_j || 0.0 >= g_K) {
        (void)fprintf(stderr, ERROR_FORMAT "must have "
            "\\mu_f > 0, k_j > 0, K > 0\n"
            "Exiting...\n",
            g_argv_0, __FILE__, __LINE__);
        exit(EXIT_FAILURE);
        /*NOTREACHED*/
    }

```

The standard `rand()` generator available on most Unix systems is evil for two reasons: first of all, on many systems its output is very far from random. On SunOS, for instance, the low bit toggles between 0 and 1 on successive calls. Furthermore it is not portable and is implemented differently on different vendors' systems. The `rand48(3C)` family of generators is of reasonably high quality and is available on many different Unix systems. The man pages describe the mathematical behavior of these generators in great detail, and I believe that their behavior is identical on different vendors' systems (I have not verified this). There are reasonable alternatives to `rand48(3C)`, such as `random(3C)`. Whereas `rand48(3C)` maintains a 48-bit state vector, the `random(3C)` available on my Solaris system maintains 256 *bytes* of state. For a good introduction to the important issue of random number generation, see [49].

```
28a (seed random number generator 28a)≡
    if (3 != sscanf(Random_seed, "%4hx%4hx%4hx",
                    &(seed16v[0]), &(seed16v[1]), &(seed16v[2]))) {
        (void)fprintf(stderr, ERROR_FORMAT "error converting random "
                    "seed '%s'\nAborting...\n", g_argv_0,
                    __FILE__, __LINE__, Random_seed);

        abort();
        /*NOTREACHED*/
    }
    if (0 == seed16v[0] && 0 == seed16v[1] && 0 == seed16v[2]) {
        (void)fprintf(stderr, ERROR_FORMAT "cannot have all zero random "
                    "seed.\nExiting...\n", g_argv_0, __FILE__, __LINE__);
        exit(EXIT_FAILURE);
        /*NOTREACHED*/
    }
    (void)seed48(seed16v);
```

Now we assign x_0 and x_f to the cars. The x_0 are uniformly spaced along $[0, x_S]$, and the destinations x_f are uniformly distributed along $[x_H, x_B]$. Each car is randomly assigned a destination. As in [27] this is the *only* use of the random number generator in the entire program.

```
28b (initialize Xo and Xf 28b)≡
    for (car = 0; car < Numcars; car++) {
        deltaX = Xs / ((double)(Numcars + 1));
        assert(0.0 < deltaX);
        Xo[car] = ((double)(car + 1)) * deltaX;
        assert(0.0 < Xo[car] && Xo[car] < Xs);
        deltaX = (Xb - Xh) / ((double)(Numoffices + 1));
        assert(0.0 < deltaX);
        office = lrand48() % Numoffices;
        Xf[car] = Xh + ((double)(office + 1)) * deltaX;
        assert(Xh < Xf[car] && Xf[car] < Xb);
        ideal[car] = travel_time(Xf[car] - Xo[car], 0.0);
    }
```

Now we print a maximally informative output header. The purpose of this header is to establish an “audit trail” to aid in the tracking of bugs and the correlation of versions of the source code with program outputs.

```
29a  (<print output header 29a>≡
      (void)printf("#\n");
      (void)printf("# compile-time information:\n");
      (void)printf("#  source file:      " __FILE__ "\n");
      (void)printf("#  RCS revision:      $Revision: 1.58 $\n");
      (void)printf("#  compilation time:  " __DATE__ " " __TIME__ "\n");
      #ifdef __STDC__
      (void)printf("#  standard C version: %ld\n", __STDC_VERSION__);
      #endif
      #ifdef __GNUC__
      (void)printf("#  compiled with:      GNU C version "
                  __VERSION__ "\n");
      #endif
      (void)printf("#  strict ANSI?:      ");
      #ifdef __STRICT_ANSI__
      (void)printf("YES\n");
      #else
      (void)printf("NO\n");
      #endif
      (void)printf("#\n");
```

Next, we print a summary of run-time information, e.g. the time at which the program was run. Note that according to Purify the call to `localtime()` on my system results in seven blocks of memory in use at exit. It appears that `localtime()` calls `malloc()` indirectly and this memory is never freed. Repeated calls to `localtime()` do not, however, result in memory leaks.

```
29b  (<print output header 29a>+≡
      (<get runtime information 30b>
      (void)printf("# run-time information:\n");
      (void)printf("#  local time:  %s",  asctime(localtime(&tod)));
      (void)printf("#  UTC/GMT time: %s",  asctime(gmtime(&tod)));
      (void)printf("#  sysname:    %s\n",  struct_utsname.sysname);
      (void)printf("#  nodename:   %s\n",  struct_utsname.nodename);
      (void)printf("#  release:    %s\n",  struct_utsname.release);
      (void)printf("#  version:    %s\n",  struct_utsname.version);
      (void)printf("#  machine:    %s\n",  struct_utsname.machine);
      (void)printf("#  userid:     %ld\n",  userid);
      (void)printf("#\n");
```

Now we print the user-supplied parameters both before and after conversions from string to integer and floating-point number have been performed.

```
30a (<print output header 29a>)+≡
(void)printf("# command line:\n");
(void)printf("# ");
for (i=0; NULL != argv[i]; i++) {
    (void)printf(" %s", argv[i]);
}
(void)printf("\n");
(void)printf("#\n");
(void)printf("# converted parameters:\n");
(void)printf("# Adaptive strategy: %d => %s\n",
    (int)Adapt, g_adapt_str[(int)Adapt]);
(void)printf("# Xs: %f\n", Xs);
(void)printf("# Xh: %f\n", Xh);
(void)printf("# Xb: %f\n", Xb);
(void)printf("# Numcars: %d\n", Numcars);
(void)printf("# Numdays: %d\n", Numdays);
(void)printf("# Numoffices: %d\n", Numoffices);
(void)printf("# Random seed: %04hx %04hx %04hx\n",
    seed16v[0], seed16v[1], seed16v[2]);
(void)printf("# \mu_f: %f\n", g_mu_f);
(void)printf("# k_j: %f\n", g_k_j);
(void)printf("# K: %f\n", g_K);
(void)printf("#\n");
```

The code used to obtain some run-time information is probably not completely portable, but this information is sufficiently important to our audit trail to justify compromising portability.

```
30b (<get runtime information 30b>)+≡
if (0 > uname(&struct_utsname)) {
    (void)fprintf(stderr, ERROR_FORMAT "call to uname() failed\n"
        "errno == %d: %s\nAborting...\n", g_argv_0,
        __FILE__, __LINE__, errno, strerror(errno));
    abort();
    /*NOTREACHED*/
}
userid = getuid();
if (0 > (tod = time(NULL))) {
    (void)fprintf(stderr, ERROR_FORMAT "call to time() failed\n"
        "errno == %d: %s\nAborting...\n", g_argv_0,
        __FILE__, __LINE__, errno, strerror(errno));
    abort();
    /*NOTREACHED*/
}
```

Now we print each car's origin and destination and ideal travel time:

```
31a <print origin and destination information 31a>≡
(void)printf("#\n# Origin & destination info:\n#\n");
(void)printf("# car:      Xo:      Xf:      ideal:\n");
(void)printf("#\n");
for (car = 0; car < Numcars; car++) {
    (void)printf("c  %4d  %10.6f  %10.6f  %10.6f\n",
                car, Xo[car], Xf[car], ideal[car]);
}
(void)printf("#\n# End O-D information\n#\n");
```

The main loop iterates once for each simulated commuting day. It consists of setting the departure times of all cars, inserting these departure events into the schedule, and then calling an inner loop which processes the day's events until no more remain.

```
31b <main loop 31b>≡
for (day = 0; day < Numdays; day++) {
    assert(NULL == g_Schedule);
    Np = Numcars;
    Ns = Nh = Nb = Na = 0;
    num_events_today = 0;
    <set departure times 32a>
    <process events of day 34>
    assert(4 * Numcars == num_events_today);
}
(void)printf("#\n# END OF SIMULATION\n#\n");
#ifdef PRINT_EVENT_QUEUES
print_event_queue();
#endif
```

As described in Section 3, on the first simulated day the departure time of each motorist is set in such a way that the driver would arrive at work exactly on time in the absence of congestion. On subsequent days motorists adjust their departure times according to one of several “adaptive strategies” described in Section 3 above. Note that while we’re at it we check the day’s arrival times to ensure they’re zero! Note that on day 1 both the mean rule and the OLS rule degenerate to the yesterday rule.

```

32a  (<set departure times 32a>≡
      for (car = 0; car < Numcars; car++) {
        if (0 == day) {
          depart[car][day] = TARGET_TIME - ideal[car];
        } else {
          switch (Adapt) {
            case ADAPT_YESTERDAY:
              (<apply yesterday rule 32b>
              break;
            case ADAPT_MEAN:
              (<apply mean rule 33a>
              break;
            case ADAPT_OLS:
              if (1 == day) {
                (<apply yesterday rule 32b>
              } else {
                (<apply OLS rule 33c>
              }
              break;
            default:
              (void)fprintf(stderr, ERROR_FORMAT "internal error: "
                "unrecognized adaptation type: %d\n"
                "Aborting...\n", g_argv_0, __FILE__,
                __LINE__, Adapt);

              abort();
              /*NOTREACHED*/
              break;
          }
        }
        assert(0.0 == arrive[car][day]);
        insert_event(depart[car][day], ENTER_ROAD, car);
#ifdef PRINT_EVENT_QUEUE
        (void)printf("after inserting car %d:\n", car);
        print_event_queue();
#endif
      }

```

The code which applies adaptive rules is relatively straightforward.

```

32b  (<apply yesterday rule 32b>≡
      yesterday_time = arrive[car][day-1] - depart[car][day-1];
      depart[car][day] = TARGET_TIME - yesterday_time;

```

```

33a  <apply mean rule 33a>≡
      total_time = 0.0;
      for (i = 0; i < day; i++) {
          total_time += arrive[car][i] - depart[car][i];
      }
      depart[car][day] = TARGET_TIME - (total_time / ((double)day));

```

We use a formula for OLS regression taken from [67]. Note that the method used here is straightforward but not efficient. To make the discussion clearer we'll introduce a few temporary variables.

```

33b  <variable declarations 22b>+≡
      double sum_x, sum_y, mean_x, mean_y, dev_x, dev_y;
      double sum_dev_xy, sum_dev_x2, a, b, extrapolation;

33c  <apply OLS rule 33c>≡
      assert(1 < day);
      sum_x = sum_y = mean_x = mean_y = sum_dev_xy = sum_dev_x2 = 0.0;
      for (i = 0; i < day; i++) {
          sum_x += (double)i;
          sum_y += arrive[car][i] - depart[car][i];
      }
      mean_x = sum_x / ((double)day);
      mean_y = sum_y / ((double)day);
      for (i = 0; i < day; i++) {
          dev_x = ((double)i) - mean_x;
          dev_y = (arrive[car][i] - depart[car][i]) - mean_y;
          sum_dev_xy += dev_x * dev_y;
          sum_dev_x2 += dev_x * dev_x;
      }
      b = sum_dev_xy / sum_dev_x2;
      a = mean_y - b * mean_x;
      extrapolation = a + b * ((double)day);
      depart[car][day] = TARGET_TIME - extrapolation;
      #ifdef PRINT_EXTRAPOLATION
      for (i = 0; i < day; i++) {
          (void)printf("car %4d day %4d delay: %10.6f\n", car, i,
                      arrive[car][i] - depart[car][i]);
      }
      (void)printf("car %4d day %4d extra: %10.6f\n", car, day,
                  extrapolation);
      (void)printf("car %4d day %4d: a = %10.6f b = %9.6f "
                  "extr = %10.6f\n", car, day, a, b, extrapolation);
      #endif

```

The inner loop consists of repeatedly processing the next event in the schedule, checking a loop invariant, and de-allocating memory. This continues until no more events remain. One of my rules of C coding is that if a pointer does not refer to valid, in-use memory, it must be set to `NULL`. This simple rule ensures that if a pointer to freed memory is ever dereferenced the Unix memory management hardware will likely detect the bug. I also think it's a good idea to use assertions to check loop invariants.

```
34 (process events of day 34)≡
    #ifdef PRINT_EVENT_QUEUES
    print_event_queue();
    #endif
    while (NULL != (ep = get_next_event())) {
        num_events_today++;
        (process event 35a)
        #ifdef PRINT_EVENT_QUEUES
        print_event_queue();
        #endif
        ep->time = -1.0;
        ep->type = EVENT_TOOHIGH;
        ep->car = -1;
        free(ep);
        ep = NULL;
        assert(Np + Ns + Nh + Mb + Na == Numcars);
    }
```

The events we process are arrivals of vehicles at the milestones along our roadway, namely x_0 , x_S , x_H , and x_f .

```

35a  (process event 35a)≡
      switch (ep->type) {
        case ENTER_ROAD:
          Np--;
          linktime = travel_time(Xs - Xo[ep->car], ((double)Ns)/Xs);
          insert_event(linktime + ep->time, ARRIVE_Xs, ep->car);
          Ns++;
          break;
        case ARRIVE_Xs:
          Ns--;
          linktime = travel_time(Xh - Xs, ((double)Nh)/Xh);
          insert_event(linktime + ep->time, ARRIVE_Xh, ep->car);
          Nh++;
          break;
        case ARRIVE_Xh:
          Nh--;
          linktime = travel_time(Xf[ep->car] - Xh, ((double)Nb)/Xb);
          insert_event(linktime + ep->time, EXIT_ROAD, ep->car);
          Nb++;
          break;
        case EXIT_ROAD:
          Nb--;
          assert(0.0 == arrive[ep->car][day]);
          arrive[ep->car][day] = ep->time;
          Na++;
          break;
        default:
          (void)fprintf(stderr, ERROR_FORMAT "internal error: "
                      "invalid event type: %d\n", g_argv_0,
                      __FILE__, __LINE__, ep->type);
          abort();
          /*NOTREACHED*/
          break;
      }
      (print road state 35b)

```

After processing each event we print a record of the state of the roadway after the event has been processed, i.e. the numbers of cars on each segment of the road. User can turn this on with a command-line flag.

```

35b  (print road state 35b)≡
      if (0 != print_events_flag) {
        (void)printf("s %4d %10.2f %4d %3s %4d %4d %4d %4d %4d\n",
                   day, ep->time, ep->car, g_event_str[ep->type],
                   Np, Ns, Nh, Nb, Na);
      }

```

Now we output the simulation results. First we print out each car's daily experience, if the user has requested it via the appropriate command-line flag:

```
36a (print results 36a)≡
    if (0 != print_daily_flag) {
        (void)printf("#                travel");
        (void)printf("          abs      rel      abs      mean\n");
        (void)printf("# day:  car:  depart:  arrive:  time:");
        (void)printf("      delay:  delay:      late:  speed:\n");
        (void)printf("#\n");
    }
    for (day = 0; day < Numdays; day++) {
        for (car = 0; car < Numcars; car++) {
            trav_time = arrive[car][day] - depart[car][day];
            abs_lateness[car][day] = arrive[car][day] - TARGET_TIME;
            abs_delay[car][day] = trav_time - ideal[car];
            rel_delay[car][day] = abs_delay[car][day] / ideal[car];
            mean_speed[car][day] = (Xf[car] - Xo[car]) / trav_time;
            mean_speed[car][day] *= 3600.0; /* convert to MPH */
            if (0 != print_daily_flag) {
                (void)printf("d %4d %4d %8.2f %8.2f "
                    "%8.2f % 11.5f % 8.5f "
                    "% 9.4f %7.3f\n",
                    day, car, depart[car][day], arrive[car][day],
                    trav_time, abs_delay[car][day], rel_delay[car][day],
                    abs_lateness[car][day], mean_speed[car][day]);
            }
        }
    }
}
```

Now let's add to this some summary data that will be used to produce figures:

```
36b (print results 36a)+≡
    (void)printf("#\n");
    (void)printf("# SUMMARY DATA:\n");
    (void)printf("#\n");
```

Note that absolute total travel time on a given day is simply sum of all drivers' absolute delay on that day. The relative total travel time is the absolute figure divided by the sum of all drivers' ideal times.

```
36c (variable declarations 22b)+≡
    double tot_ideal, tot_daily_lateness;
```

```

37a  {print results 36a)+≡
      tot_ideal = 0.0;
      for (car = 0; car < Numcars; car++) {
        tot_ideal += ideal[car];
      }
      (void)printf("#\n");
      (void)printf("# Total travel time per day\n");
      (void)printf("#\n");
      (void)printf("#          abs          rel\n");
      (void)printf("#          total          total\n");
      (void)printf("#          travel          travel\n");
      (void)printf("# day:          time:          time:\n");
      (void)printf("#\n");
      for (day = 0; day < Numdays; day++) {
        (void)printf("f1 %4d", day);
        total_time = 0.0;
        for (car = 0; car < Numcars; car++) {
          total_time += arrive[car][day] - depart[car][day];
        }
        (void)printf(" %11.3f %11.9f\n",
                    total_time, total_time / tot_ideal);
        assert(1.0 <= total_time/tot_ideal);
      }

```

Now we print out aggregate lateness data.

```

37b  {print results 36a)+≡
      (void)printf("#\n");
      (void)printf("# Aggregate lateness per day\n");
      (void)printf("#\n");
      (void)printf("#          total          mean\n");
      (void)printf("# day: lateness: lateness:\n");
      (void)printf("#\n");
      for (day = 0; day < Numdays; day++) {
        tot_daily_lateness = 0.0;
        for (car = 0; car < Numcars; car++) {
          if (0.0 < abs_lateness[car][day]) {
            tot_daily_lateness += abs_lateness[car][day];
          }
        }
        (void)printf("f2 %4d %11.3f %11.9f\n", day, tot_daily_lateness,
                    tot_daily_lateness / ((double)Numcars));
      }

```

And now for the "fairness" data.

```

37c  {variable declarations 22b)+≡
      double tot_daily_rel_del, mean_daily_rel_del, sum2_dev_rel_del;

```

```

38a  <print results 36a>+≡
      (void)printf("#\n");
      (void)printf("# Fairness (std dev of rel delays) per day\n");
      (void)printf("#\n");
      (void)printf("#          std dev\n");
      (void)printf("#          of rel\n");
      (void)printf("# day:      delays:\n");
      (void)printf("#\n");
      for (day = 0; day < Numdays; day++) {
          tot_daily_rel_del = 0;
          sum2_dev_rel_del = 0;
          for (car = 0; car < Numcars; car++) {
              tot_daily_rel_del += rel_delay[car][day];
          }
          mean_daily_rel_del = tot_daily_rel_del / ((double)Numcars);
          for (car = 0; car < Numcars; car++) {
              sum2_dev_rel_del +=
                  (rel_delay[car][day] - mean_daily_rel_del)
                  * (rel_delay[car][day] - mean_daily_rel_del);
          }
          (void)printf("f3 %4d %11.9f\n", day,
                      sqrt(sum2_dev_rel_del / ((double)(Numcars-1))));
      }

```

And now consistency data:

```

38b  <variable declarations 22b>+≡
      double tot_trav_time, mean_trav_time, sum2_dev_trav_time;
      double std_dev_trav_time;

```

```

39a  <print results 36a>+≡
      (void)printf("#\n");
      (void)printf("# Consistency as function of car number\n");
      (void)printf("#\n");
      (void)printf("#          normalized\n");
      (void)printf("#          std dev of\n");
      (void)printf("#          abs trav\n");
      (void)printf("# car:          times:\n");
      (void)printf("#\n");
      for (car = 0; car < Numcars; car++) {
          tot_trav_time = 0.0;
          sum2_dev_trav_time = 0.0;
          for (day = 0; day < Numdays; day++) {
              tot_trav_time += arrive[car][day] - depart[car][day];
          }
          mean_trav_time = tot_trav_time / ((double)Numdays);
          for (day = 0; day < Numdays; day++) {
              trav_time = arrive[car][day] - depart[car][day];
              sum2_dev_trav_time +=
                  (trav_time - mean_trav_time)
                  * (trav_time - mean_trav_time);
          }
          std_dev_trav_time = sqrt(sum2_dev_trav_time / ((double)Numdays-1));
          (void)printf("f4 %4d %11.9f\n",
                      car, std_dev_trav_time / ideal[car]);
      }

      Mean relative delay data:
39b  <variable declarations 22b>+≡
      double tot_rel_del;

39c  <print results 36a>+≡
      (void)printf("#\n");
      (void)printf("# Mean relative delay as function of car number\n");
      (void)printf("#\n");
      (void)printf("#          mean\n");
      (void)printf("#          relative\n");
      (void)printf("# car:          delay:\n");
      (void)printf("#\n");
      for (car = 0; car < Numcars; car++) {
          tot_rel_del = 0.0;
          for (day = 0; day < Numdays; day++) {
              tot_rel_del += rel_delay[car][day];
          }
          (void)printf("f5 %4d %11.9f\n",
                      car, tot_rel_del / ((double)Numdays));
      }

```

Now we're done.

```
40a <print results 36a>+≡
    if (0 > (tod = time(NULL))) {
        (void)fprintf(stderr, ERROR_FORMAT "call to time() failed\n"
            "errno == %d: %s\nAborting...\n", g_argv_0,
            __FILE__, __LINE__, errno, strerror(errno));
        abort();
        /*NOTREACHED*/
    }
    (void)printf("#\n");
    (void)printf("# Run complete\n");
    (void)printf("# local time: %s", asctime(localtime(&tod)));
    (void)printf("# UTC/GMT time: %s", asctime(gmtime(&tod)));
    (void)printf("#\n");
```

A.2 Floating Point Exception Handler Module

In a sane world, this module would not exist. It turns out that on my Solaris system if your program performs a floating point division by zero, or if overflow or underflow occurs, the default behavior is for absolutely nothing to happen. A quick survey of man pages on other systems suggests that the situation is largely the same on HP and IBM systems. In order to arrange for a trap to occur in response to a floating exception you must explicitly change the default behavior either by using code similar to that provided below or by setting a flag on a vendor-supplied compiler. Since I use `gcc` I must use the former method. See the man pages for `sigfpe(3)` and `fpsetmask(3C)` on a Solaris system for more. The only reason why this module is separated from the main module is that some Solaris header files won't compile with ANSI strictness enabled.

The header file for this module exports a single function. After the main program calls this functions, floating exceptions will cause the program to issue an error message and abort with a core dump.

```
40b <fpe.h* 40b>≡
    #ifndef FPE_H_INCLUDED
    #define FPE_H_INCLUDED
    extern void fpe_set(void);
    #endif /* #ifndef FPE_H_INCLUDED */
```

This code is written to file `fpe.h`.

Now for the implementation of the module. First we must include a few header files:

```
40c <fpe.c* 40c>≡
    <fpe header files 41a>
    <fpe static functions 41b>
    <fpe exported functions 41c>
```

This code is written to file `fpe.c`.

```
41a <fpe header files 41a>≡
#include <ieeefp.h>
#include <floatingpoint.h>
#include <siginfo.h>
#include <stdio.h>
#include <stdlib.h>
#include <ucontext.h>
#include "fpe.h"
#include "main.h"
```

Now we define a function borrowed almost verbatim from the `sigfpe(3)` man pages. When a floating exception occurs control will pass to this function, which will print an error message and then terminate. We want to dump core in this case in order to facilitate debugging.

```
41b <fpe static functions 41b>≡
/*ARGSUSED*/
static void fpe_handler(int sig, siginfo_t *sip, ucontext_t *uap) {
    char *label;
    switch (sip->si_code) {
        case FPE_FLTINV: label = "invalid operand"; break;
        case FPE_FLTRES: label = "inexact"; break;
        case FPE_FLTDIV: label = "division-by-zero"; break;
        case FPE_FLTUND: label = "underflow"; break;
        case FPE_FLTOVF: label = "overflow"; break;
        default: label = "???"; break;
    }
    (void)fprintf(stderr, ERROR_FORMAT "FP exception "
                "%s (0x%x) at address %p.\n received signal %d\n"
                "Aborting...\n",
                g_argv_0, __FILE__, __LINE__,
                label, sip->si_code, (void *)sip->si_addr, sig);
    abort();
    /*NOTREACHED*/
}
```

Now we enable detection of floating point exceptions. Note that if you enable `FP_X_IMP` detection virtually every floating operation causes one, so we do not trap on imprecision.

```
41c <fpe exported functions 41c>≡
extern void fpe_set(void) {
    (void)fpsetmask(FP_X_INV | FP_X_OFL | FP_X_UFL | FP_X_DZ);
    (void)sigfpe(FPE_FLTRES, (sigfpe_handler_type)fpe_handler);
    (void)sigfpe(FPE_FLTDIV, (sigfpe_handler_type)fpe_handler);
    (void)sigfpe(FPE_FLTUND, (sigfpe_handler_type)fpe_handler);
    (void)sigfpe(FPE_FLTOVF, (sigfpe_handler_type)fpe_handler);
    (void)sigfpe(FPE_FLTINV, (sigfpe_handler_type)fpe_handler);
}
```

B Literate Programming

The basic idea of literate programming is that programs should be written for the benefit of human readers rather than for compilers. The idea was first popularized by Don Knuth in [29] who used a tool called “Web” to document his \TeX typesetting program. The literate programming tool used to document the code presented in Appendix A is Norman Ramsey’s `noweb` system [52, 51]. This tool allows me to write my code and documentation in a single file. Filters are used to separate out the C code for the benefit of the compiler, and other filters process the `noweb` input file that I edit into a form suitable for typesetting by \LaTeX [32]. A literate program consists of documentation, such as the text you’re reading now, interspersed with “code chunks” containing pieces of computer programs. The code chunks may themselves contain references to other code chunks. In typeset documentation, an alphanumeric identifier appears in the margin to the side of every code chunk. This identifier consists of the page number on which the chunk appears followed by a letter to indicate whether it is the first, second, etc. chunk on the page. So chunk 5b refers to the second chunk on page 5.

Code chunks can be documented in arbitrary order. This means that I can present the code in whatever order is most suitable to the discussion rather than in the order required by the compiler. For instance, I can show you my version of the prototypical C program:

42a `<hello.c 42a>≡`
`<hello includes 42b>`
`int main (void) {`
`(void)printf("hello world\n");`
`return 0;`
`}`

This code is written to file `hello.c`.

I can list the files included anywhere, e.g. in an appendix.

42b `<hello includes 42b>≡`
`#include <stdio.h>`

The `noweb` system guarantees that `hello.c` will be re-assembled into the proper order, so the compiler will see the `#include` line before the `main()` function, even though you see them in the opposite order here.

I chose to document my simulation code with `noweb` for several reasons. The most important motive was to facilitate the review of my code by other programmers. If we claim to be scientists, we must ensure that our results are open to scrutiny and reproducible at reasonable cost. By making available a well-documented listing of the code I used to generate my results I make it feasible for a skeptical reader to verify that the code is free of bugs and accurately implements the formal simulation model I'm using. It is very difficult to read and understand even simple and well-written C code. Comments in the code make the task easier, but many concepts are difficult to express in ASCII. By allowing the programmer to freely intersperse figures and typeset equations (as in Section 3.3 above), literate programming makes it easier for the writer of code to convey his intentions to the reader. It also makes it easier for the reader to verify that the code faithfully implements the programmer's intentions. In [24] an entire library of high-quality C functions is presented in this way.

An added benefit of literate programming is that code and documentation are kept together in a single file. This increases the likelihood that they will be updated together. Without literate programming it is much more likely that the documentation will fall out of sync with the code.

C Reproducing my Results

Here's how I got my results. First, I have a file of random number seeds, the first three of which were used to generate the results reported in [27]:

```
43 (<Seeds 43>≡  
    c8fab1cdc397  
    52fdb1ec3e68  
    1f501a03f4b5  
    6935292cac97  
    b48325a78e67  
    82b57b585639  
    d5e75358be18  
    492de5f37816  
    748326178fa8  
    e65a7cc1df67
```

This code is written to file `Seeds`.

I run my program with the following shell program. Except for the adaptive rule and random seed, all other parameters are held fixed, and are identical to the values used in [27].

```
44 <Run.sh 44>≡
#!/usr/um/bin/tcsh -f
set params="-s 1 -h 4 -b 5 -c 100 -d 100 -o 10 -u 60 -j 200 -K 3"
echo "" ; echo BEGIN ; echo ""
foreach seed ('cat Seeds')
  foreach a (1 2 3)
    sim $params -r $seed -a $a > Output/a${a}_s${seed}.out
  end
end
echo "" ; echo END ; echo ""
```

This code is written to file Run.sh.

D About the Simulation Code

D.1 Conditions of Use

The source code used to generate the findings reported in this paper is made available to the public in order to facilitate the reproduction of my results and to allow scientific scrutiny of my findings. The code is distributed without any warranty. Use it at your own risk.

Anyone may read, compile and run the simulation code without restriction. If you wish to modify and then re-distribute it, the changes that you have made must be clearly identifiable as yours. I strongly recommend the use of a revision control tool such as RCS or SCCS to track any changes you make in the code.

I took great pains to ensure that the simulation code compiles and lints as cleanly as possible. I request that if you modify the code and re-distribute it you take reasonable precautions to preserve this important property of the code. See Section D.3 for details on the precautions I take to ensure high-quality code.

D.2 Design Decisions

The primary goal of the code was clarity and close correspondence between the formal model and the implementation of the simulation code. The model itself is intended to be simple and plausible, and I want the mapping of formal model to C code to be straightforward and obvious to the reader. To achieve this, generality, modularity and clean interfaces were sacrificed. I did not use off-the-shelf discrete event simulation code such as that supplied with [64].

However, code portability is an issue orthogonal to modularity and generality, and considerable care was taken to ensure that the code be standard ANSI C. The C language is underspecified in many respects and it is therefore *very* difficult to write truly portable code (see Section D.4 for a review of this and other problems with C and its dialects). I believe that my simulation code will behave reasonably on any system, provided it compiles cleanly, although I cannot certify that its behavior will be identical on all platforms or on all inputs. In Section D.5 I describe the steps I have taken to ensure that the code is reliable.

D.3 Coding Standards

Writing correct code is difficult under any circumstances, and simulation modeling of multi-agent systems is prone to pitfalls not present in other programming problems. If simulation code models a physical system, e.g. a pendulum, it is possible to perform various “sanity checks” to test for bugs. For instance, we can verify that energy conservation is not violated. In the case of multi-agent system modeling we normally have few hard and fast rules about what kinds of outputs are absolutely impossible, so we can perform fewer sanity checks. Because it is easier to mistake erroneous output for legitimate scientific results, it is more important to use every tool and method at our disposal to ensure correct code when we write multi-agent simulation models than in almost any other scientific programming

domain. Furthermore, if we use the C programming language, we must be on guard against the special deficiencies of this language. In Section D.4 I briefly review some of the problems of the C language, then in Section D.5 I describe the techniques I use to guard against bugs. The general approach is to use automatic tools to obtain guarantees that entire classes of bugs are completely absent from the code. Following the advice of [35] I try to detect bugs as early as possible in order to minimize the cost of correcting them.

D.4 The Evils of C

Few C programmers other than compiler writers are aware of all of the undesirable properties of C. What follows is a brief summary of some of the defects in C that are particularly relevant to multi-agent simulation modeling. For an extreme viewpoint on the subject see [17]. A more balanced and informative account is available in [63].

We begin with the language itself. A desirable property of a language is that the behavior of a program written in the language should be entirely determined by the source code. The C language and its various dialects do not have this property. Many aspects of a C program's behavior are underspecified by the language standard. These "implementation defined" aspects include things as simple as arithmetic operators. Integer division and modulus, for instance, are implementation defined. In other words, the ANSI C standard does not say whether $-13/5$ is equal to -2 or -3 . If your C program performs division or modulus on numbers with opposite signs, there's no way of knowing what the program will do by inspecting the code, and the code is *guaranteed* to be non-portable. This comes as a surprise to even experienced C programmers. See [24, page 16] for a detailed treatment of this issue.

Although division and modulus are perhaps the most striking examples of semantic holes in the language, they are not the only ones. In the words of an expert compiler writer, "C's semantics are riddled with such holes" [24, page 18] and in order to deal with this sort of problem completely it is necessary to "encapsulate" underspecified language features in functions with well-defined semantics. Another case where encapsulation is strongly recommended are the standard C library functions. These functions often hide error codes in return values. The `malloc()` function, for instance, indicates a failure by returning a `NULL` pointer. This is not a particularly pernicious behavior because as long as you test the return value you know for sure whether the call has succeeded or not. The case is different with functions which return legitimate values to indicate errors. The `atof()` function, which converts strings to double-precision floating point numbers, returns zero when given a string like "0". That's reasonable enough. But it also returns zero when given a string that would cause underflow. So if the function returns zero you must check the global `errno` variable to see whether a conversion error has in fact occurred.

For a striking demonstration that what you see is not what you get, consider the effects of trailing whitespace, i.e. space characters and tab characters that occur immediately before a newline in C source code. Most programmers mistakenly believe that trailing whitespace is insignificant and cannot affect the behavior of a C program. This is not true, as we see from the following program:

```
#include <stdio.h>
int main(void) {
    char *p = "no";
    #define F00 \
    p = "yes";
    printf("%s\n", p);
    return 0;
}
```

This code is perfectly legal C and compiles cleanly with or without trailing whitespace on any or all lines. But what does it do? That depends on whether the `#define` line contains trailing whitespace. If it does, then the program prints **yes**, otherwise it prints **no**. This sort of bug is of course very difficult to find because it's invisible [63, page 53].

The C compiler traditionally performs only syntactic checks on its input, with perhaps a few relatively weak semantic checks. Stronger semantic checks are performed by lint. The practical result of the decision to separate the two types of checks into two different programs is that nobody performs semantic checks because nobody uses lint. This is a problem because only a lint can detect certain classes of bugs, e.g. uninitialized memory errors that span two or more files — no compiler can detect such a bug, because the compiler inspects only one source file at a time and does not consider interactions between modules. More generally, the ANSI standard does not require the C compiler to warn of code that is almost certainly erroneous. My favorite example is given in [63, page xx]. The programmer meant to write `x=2;` on a line by itself, but accidentally wrote `x==2;`. Both of these are perfectly legal, but any reasonable person inspecting the code will realize that the latter can't possibly be right. Nearly every lint will warn you about this sort of error, but compilers often don't. The scary thing about this bug is the heading of the section devoted to it: "The \$20 Million Bug."

Unfortunately, C++ isn't much better than C as far as the above problems are concerned. In the words of its designer, "The language is not called D, because it is an extension of C and does not attempt to remedy problems by removing features" [61]. Another important motive for choosing C rather than C++ for this project is that the rich set of Unix tools for C development are in many cases not yet available for C++. On my Solaris system, for instance, lint handles C but not C++. This alone is reason enough to favor C. Finally, the features of C++ that distinguish it most clearly from C are irrelevant to the problem at hand. C++ would add complication to the code without adding value to it.

D.5 Enforcing Fundamental Properties

In this section I discuss the steps that I take to ensure high-quality code.

D.6 Style Points

Programmers are often full of religious opinions on matters of coding style. I'm no exception. In this section I briefly outline some of the coding style guidelines I follow.

I never write very long lines of code. 70 characters is my current upper bound on line length. Longer lines tend to be unreadable, and when code is listed on paper they are often truncated or wrapped in confusing ways. Lines longer than around 70 characters are completely unnecessary and are absent from listings of experts' code (see, for instance, [24]).

My indentations normally consist of two spaces. Much more than this actually makes it harder to read the code. I never use tab characters in my code. When code is re-indented or sent back and forth between programmers via e-mail the tab characters result in the code being visually mangled. Regarding placement of curly braces, I ignore the rule that one should use the One True Brace Style found in [28]. I should probably change my ways in this regard. At the moment I try to be consistent.

When comparing a variable with a constant I always write "`5 == x`" rather than "`x == 5`." It is too easy to write an assignment statement like "`x = 5`" when you intend a comparison. If you always write the constant on the left the compiler will always detect the error.

References

- [1] Haitham Al-Deek and Adib Kanafani. Modeling the benefits of advanced traveler information systems in corridors with incidents. *Transportation Research C*, 1(4):303–324, 1993. Explores effects of ATIS through a simulation model. From the abstract: “It is found that the benefits to guided traffic decrease when the proportion of guided traffic exceeds a critical value.... The system benefits also level off once the critical value is exceeded.”.
- [2] Richard Arnott, Andre de Palma, and Robin Lindsey. Does providing information to drivers reduce traffic congestion? *Transportation Research*, 25A(5):309–318, 1991. Presents simple model of morning rush hour. Questions presumption that route guidance and information systems necessarily reduce congestion. Points out the need to consider general equilibrium effects of information. Suggests that congestion is an uninternalized externality and that information may cause drivers to change departure times in such a way as to exacerbate congestion.
- [3] Richard Arnott, Andre de Palma, and Robin Lindsey. Properties of dynamic traffic equilibrium involving bottlenecks, including a paradox and metering. *Transportation Science*, 27(2):148–160, May 1993.
- [4] Brian Arthur. Inductive reasoning, bounded rationality, and the bar problem. *American Economic Association Papers and Proceedings*, 84:406–411, 1994. Introduces the “El Farol Problem.” See [8] for a readable account of the problem and its relation to complex systems studies.
- [5] Woodrow Barfield, Mark Haselkorn, Jan Spyridakis, and Loveday Conquest. Integrating commuter information needs in the design of a motorist information system. *Transportation Research A*, 25(2/3):71–78, 1991. Presents results of large opinion survey of motorists’ opinions on information systems.
- [6] Moshe Ben-Akiva and Andre de Palma. Some circumstances in which vehicles will reach their destinations earlier by starting later: Revisited. *Transportation Science*, 20(1):52–55, February 1986. Argues that under more realistic assumptions Smeed’s 1967 argument that later departures can result in earlier arrivals does not hold.
- [7] Moshe Ben-Akiva, Andre de Palma, and Isam Kaysi. Dynamic network models and driver information systems. *Transportation Research A*, 25A(5):251–266, 1991.
- [8] John Casti. Seeing the light at el farol. *Complexity*, 1(5):7–10. A readable account of Brian Arthur’s “El Farol Problem” [4] suitable for general audiences.
- [9] John Casti. *Would-be Worlds*. John Wiley & Sons, Inc., 1997. Chapter 4 contains a rather detailed account of the TRANSIMS project at Los Alamos National Labs, suitable for a general audience. ISBN 0-471-12308-0.
- [10] Stefano Catoni and Stefano Pallottino. Traffic equilibrium paradoxes. *Transportation Science*, 25(3):240–244, August 1991.

- [11] J. Cohen and F. Kelley. A paradox of congestion in a queueing network. *Journal of Applied Probability*, 27:730–734, 1990. Cited in [44] as evidence that adding a road to a road network can reduce capacity.
- [12] Ian F. Darwin. *Checking C Programs with lint*. O’Reilly & Associates, 1988. One of a very few books currently available on lint. Neither large nor recent, but provides a decent introduction to the basics. Also noteworthy for reprinting the Ten Commandments for C Programmers in an appendix. ISBN 0-937175-30-7.
- [13] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, April 1995. ISBN 0-201-51459-1.
- [14] Richard H. M. Emmerink, Kay W. Axhausen, Peter Nijkamp, and Piet Rietveld. The potential of information provision in a simulated road transport network with non-recurrent congestion. *Transportation Research C*, 3(5):293–309, 1995.
- [15] David Evans. *The LCLint User’s Guide*. MIT Laboratory for Computer Science, 2.2 edition, August 1996. LCLint is a powerful lint that uses special comments to convey the programmer’s intentions to the verifier. Read all about it on the Web at <http://larch-www.lcs.mit.edu:8001/larch/lclint/>.
- [16] Caroline Fisk. More paradoxes in the equilibrium assignment problem. *Transportation Research*, 13B:305–309, 1979. Presents examples where *decrease* in travel costs on a network results from *increase* in input flows.
- [17] Simson Garfinkel, Daniel Weise, and Steven Strassmann. *The UNIX-Hater’s Handbook*. IDG Books, 1994. A bitterly humorous look at the dark side of Unix, written by knowledgeable programmers. Worth the purchase price for the chapter subtitles alone, e.g. “power tools for power fools” and, for the C++ chapter, “the COBOL of the 90’s.” ISBN 1-56884-203-1.
- [18] Nathan Gartner, editor. *Traffic Flow Theory*. Transportation Research Board, 1997. This is a thoroughly updated version of TRB Special Report 165, “Traffic Flow Theory,” published in 1975. At the time of this writing it is still in draft form. Available on the Web at <http://stargate.ornl.gov/trb/tft.html>.
- [19] Denos C. Gazis, editor. *Traffic Science*. John Wiley & Sons, 1974. ISBN 0-471-29480-2.
- [20] A. Hadj-Alouane, N. Hadj-Alouane, O. Juma, G. Sarathy, and S. Underwood. The ali-scout route guidance simulation: Report on ali-scout simulation runs at various levels of market penetration. Technical Report Phase IIB #4 FT96-119, University of Michigan Intelligent Transportation Systems Lab, November 1996. FAST TRAC internal report. Discusses simulation of Ali-Scout dynamic route guidance system. From the introduction: “It appears that the benefits of Ali-Scout are significant only when the level of market penetration is below a certain level.” From the conclusion: “...equipped vehicles are taking the same routes and causing their own congestion”.

- [21] A. Hadj-Alouane, O. Juma, A. Okin, and S. Underwood. Traffic probe and corridor analysis. Technical Report Phase IIB #4/5, University of Michigan Intelligent Transportation Systems Lab, November 1995. FAST TRAC internal report.
- [22] A. Hadj-Alouane and Steven Underwood. Report on ali-scout simulation. Technical Report Phase IIA #5 FT95-005, University of Michigan Intelligent Transportation Systems Lab, March 1995. FAST TRAC internal report.
- [23] Randolph W. Hall. Non-recurrent congestion: How big is the problem? are traveler information systems the solution? *Transportation Research C*, 1(1):89–103, 1993. Suggests that ATIS cannot provide solution to peak-period non-recurrent congestion.
- [24] David R. Hanson. *C Interfaces and Implementations*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1997. An excellent book on modular and reusable software design by a true Jedi master of C (and a former professor of mine). All of the source code described in the book is available on the Web at <http://www.cs.princeton.edu/software/cii/>. The book itself is produced with the aid of Ramsey’s `noweb` literate programming tool [52, 51].
- [25] Yasunori Iida, Takamasa Akiyama, and Takashi Uchida. Experimental analysis of dynamic route choice behavior. *Transportation Research B*, 26B(1):17–32, 1992. Human subjects respond to questions about hypothetical route choices. From the abstract: “It is found that the traffic flow seems to be hard to converge to an equilibrium”.
- [26] A. Kanafani and H. Al-Deek. A simple model for route guidance benefits. *Transportation Research B*, 25B(4):191–201, 1991.
- [27] Terence Kelly. Driver strategy and traffic system performance. *Physica A*, 235(3–4):407–416, 1997. Explores the same basic question as the present paper using a car-following microsimulation model. A draft is available on the Web at <http://ai.eecs.umich.edu/people/tpkelly/papers/>.
- [28] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1988. See section 2.5, page 41, for the ugly truth about integer division and modulus involving negative operands. ISBN 0-13-110362-8.
- [29] Donald E. Knuth. *Literate Programming*. Center for the Study of Language and Information, Stanford, California, 1992. The 1995 printing is thought to be error-free. ISBN 0-937073-80-6. Details available on the Web at <http://www-cs-faculty.Stanford.EDU/~knuth/lp.html>.
- [30] Andrew Koenig. *C traps and Pitfalls*. Addison-Wesley, 1989. Includes a brief discussion of C portability issues. ISBN 0-201-17928-8.
- [31] Harris N. Koutsopoulos, Amalia Polydoropoulou, and Moshe Ben-Akiva. Travel simulators for data collection on driver behavior in the presence of information. *Transportation Research C*, 3(3):143–159, 1995.

- [32] Leslie Lamport. *LaTeX User's Guide and Reference Manual*. Addison-Wesley, second edition, 1994. ISBN 0-201-52983-1.
- [33] Jonathan Levine and Steven Underwood. Stakeholder preferences in urban transportation evaluation: A multiattribute analysis of goals for intelligent transportation system planning. Technical Report Phase IIA #7/8 FT95-007, University of Michigan Intelligent Transportation Systems Lab, January 1995. FAST TRAC internal report.
- [34] Jonathan C. Levine, Steven Underwood, and Gwo-Wei Torng. Cost analysis of its technologies: ATMS/ATIS integration. Technical Report Phase IIB #9 FT96-024, University of Michigan Intelligent Transportation Systems Lab, June 1996. FAST TRAC internal report.
- [35] Stephen A. Maguire. *Writing Solid Code*. Microsoft Press, 1993. Once you stop laughing at the idea that Microsoft is going to show us how to write good code, you'll learn a lot from this book. The basic argument is that software writers write too many bugs, and it's too expensive to catch and fix these bugs during testing. Therefore bugs must be caught as early as possible, ideally through automatic means. Maguire presents a series of steps that programmers can take to stop bugs as early as possible. Includes an entire chapter on the design and use of assertion macros. ISBN 1-55615-551-4.
- [36] Stephen A. Maguire. *Debugging The Development Process*. Microsoft Press, 1994. Whereas [35] is for individual programmers, this book is for small software team leaders. ISBN 1-55615-650-2.
- [37] H. Mahmassani and R. Herman. Dynamic user equilibrium departure time and route choice on idealized traffic arterials. *Transportation Science*, 18:362–384, 1984.
- [38] H. S. Mahmassani, G.-L. Chang, and R. Herman. Individual decisions and collective effects in a simulated traffic system. *Transportation Science*, 26:296, 1986.
- [39] H. S. Mahmassani and S. Peeta. Network performance under system optimal and user equilibrium assignments: Implications for advanced traveler information systems. *Transportation Research Record*, 1408:83–93, 1993.
- [40] Hani S. Mahmassani and Gang-Len Chang. Experiments with departure time choice dynamics of urban commuters. *Transportation Research B*, 20B(4):297–320, 1986. Addresses questions quite similar to [27].
- [41] Hani S. Mahmassani and R. Jayakrishnan. System performance and user response under real-time information in a congested traffic corridor. *Transportation Research A*, 25A(5):293–307, 1991.
- [42] J. D. Murchland. Braess' paradox of traffic flow. *Transportation Research*, 4:391–394, 1970.

- [43] Kai Nagel. Particle hopping models and traffic flow theory. *Physical Review E*, 53(5):4655–4672, May 1996. Describes how cellular automata or particle hopping models fit into the context of traffic flow theory.
- [44] Kai Nagel and Steen Rasmussen. Traffic at the edge of chaos. In Rodney A. Brooks and Pattie Maes, editors, *Artificial Life IV*, pages 222–235. MIT Press, 1994. This is the paper that got me interested in traffic modeling. Some time after seeing Nagel’s presentation at the Alife IV conference I wrote [27]. ISBN 0-262-52190-3.
- [45] Kai Nagel and M. Schreckenberg. A cellular automaton model for freeway traffic flow. *Journal of Physics I France*, 2:2221, 1992.
- [46] Gordon Newell. The morning commute for nonidentical travelers. *Transportation Science*, 21(2):74–88, May 1987.
- [47] Gordon Newell. Traffic flow for the morning commute. *Transportation Science*, 22(1):47–58, February 1988. Disagrees with some conclusions of [37].
- [48] Renfrey B. Potts and Robert M. Oliver. *Flows in Transportation Networks*, volume 90 of *Mathematics in Science and Engineering*. Academic Press, 1972. No ISBN number. Library of Congress number 70-182673.
- [49] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C*. Cambridge University Press, second edition, 1992. Chapter 7, pages 274–316, is devoted to random number generation. ISBN 0-521-43108-5. There’s also a *Numerical Recipes* Web page at <http://cfata2.harvard.edu/nr/nrhome.html>, where the most important books in the series are available in PostScript and PDF form.
- [50] Pure Software, 1309 S. Mary Ave., Sunnyvale CA 94087. *Purify User’s Guide*. An excellent introduction to an excellent tool. My edition is labelled part number PFY300-XPX-UGD. Visit the Pure Software Web site at <http://www.pureatria.com/>.
- [51] Norman Ramsey. *noweb(1), noweave(1), and notangle(1) manual pages*. University of Virginia. The *noweb* family of tools is installed on the University of Michigan CAEN system in [/afs/engin.umich.edu/u/t/p/tpkelly/bin/](http://afs.engin.umich.edu/u/t/p/tpkelly/bin/). The *noweb* tools are freely available on the Web at <http://www.cs.virginia.edu/~nr/noweb/>.
- [52] Norman Ramsey. Literate programming simplified. *IEEE Software*, 11(5):97–105, September 1994. Describes Ramsey’s *noweb(1)* family of simple literate programming tools.
- [53] A. Schadschneider and M. Schreckenberg. Cellular automaton models and traffic flow. *Journal of Physics A*, 26:L679, 1993.
- [54] M. Schreckenberg, A. Schadschneider, Kai Nagel, and N. Ito. Discrete stochastic models for traffic flow. *Physical Review E*, 51(4):2939–2949, April 1995. Investigates probabilistic cellular automata models of single-lane traffic flow.

- [55] Rachel Selk and Steven Underwood. Final focus group report: Traveler behavior. Technical Report Phase III #16A FT97-001, University of Michigan Intelligent Transportation Systems Lab, 1997. FAST TRAC internal report.
- [56] Yosef Sheffi. *Urban Transportation Networks*. Prentice Hall, 1985. ISBN 0-13-939729-9.
- [57] M. J. Smith. In a road network, increasing delay locally can reduce delay globally. *Transportation Research*, 12:419-422, 1978.
- [58] Richard M. Stallman and Roland McGrath. *GNU Make*. Free Software Foundation, 59 Temple Place, Suite 330, Boston MA 02111, 0.50 edition, March 1996. ISBN 1-882114-79-5.
- [59] Richard Steinberg and Willard I. Zangwill. The prevalence of braess' paradox. *Transportation Science*, 17(3):301-318, August 1983. Discusses necessary and sufficient conditions for Braess' Paradox to occur in transportation networks.
- [60] N. F. Stewart. Equilibrium vs system-optimal flow: Some examples. *Transportation Research*, 14A:81-84, 1980.
- [61] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, second edition, August 1995. ISBN 0-201-53992-6.
- [62] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, April 1995. ISBN 0-201-54330-3.
- [63] Peter van der Linden. *Expert C Programming: Deep C Secrets*. Prentice Hall, 1994. A highly readable and humorous look into the dark corners of C by a member of Sun's compiler and OS kernel group. Highly recommended. ISBN 0-13-177429-8. More information is available on the Web at <http://www.sun.com/books/books/vanderLinden/vanderLinden.html>.
- [64] Kevin Watkins. *Discrete Event Simulation in C*. McGraw-Hill International Series in Software Engineering. McGraw-Hill, 1993. Includes code on diskette, none of which was used in the project described in this paper. ISBN 0-07-707733-4.
- [65] David Watling and Tom van Vuren. The modeling of dynamic route guidance systems. *Transportation Research C*, 1(2):159-182, 1993. Survey of recent research into dynamic route guidance and overview of issues involved in modeling them.
- [66] Gray Watson. *Debug Malloc Library*, 3.2.0 edition, January 1997. Debug malloc is poor man's Purify. It's a drop-in replacement for the standard C library malloc. Detects many of the same errors as Purify (e.g. leaks, some forms of corruption). Available free on the Web at <ftp://ftp.letters.com/src/dmalloc/>.
- [67] Ronald J. Wonnacott and Thomas H. Wonnacott. *Introductory Statistics*. John Wiley & Sons, fourth edition, 1985. ISBN 0-471-86899-X.