# Solutions to Problems in Chapter 1 of *Simulation Modeling and Analysis*, 5th ed., 2015, McGraw-Hill, New York by Averill M. Law

1.1. (*a*) If it's small enough and feasible, then a physical experiment might be possible. Otherwise, simulate.

(*b*) Probably a simulation model would be used.

(*c*) A mathematical model (as opposed to a physical) model should be used. The complexity of the situation would probably prelude a valid analytical model being developed, resulting in the use of a simulation model.

(*d*) Since the stakes might not be too great in the short run, a physical model might be tried. If the operation is very large, then a simulation model would probably have to be used.

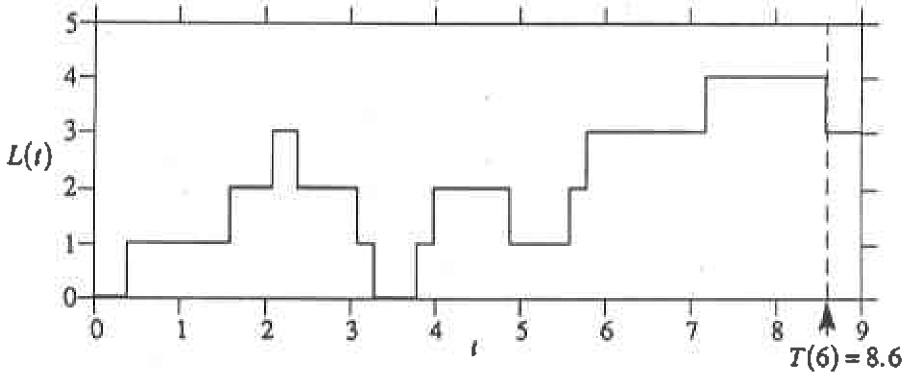
(*e*) A physical experiment might be tried if not too disruptive, but otherwise a simulation model.

(*f*) If a prototype communications network is available, then field tests could be performed for a limited number of scenarios. The results from these tests could be used to validate a simulation model, which could then be used to test other scenarios of interest.

1.2. All the systems would appear to require a dynamic model. Most of the systems would appear to be stochastic, so should be modeled as such unless the amount of system variability is very small. In most cases, a discrete model would seem to be in order. However, if the number of cars to be modeled in (*b*) is very large, then a continuous differential-equation model might be a useful (and faster to execute) approximation.

**1.3.** (*a*) No, since when the system is empty and the server is idle, $L(t) = Q(t) = 0$; as long as the server is busy, it *is* true that $L(t) = Q(t) + 1$. In any case, it is *always* true that $L(t) = Q(t) + B(t)$.

(*b*)



(*c*) Reading the areas of rectangles off the graph in (*b*),

$$\begin{aligned}\hat{\ell}(6) = {} & [(0.4-0)\times 0+(1.6-0.4)\times 1+(2.1-1.6)\times 2+(2.4-2.1)\times 3+\\ & (3.1-2.4)\times 2+(3.3-3.1)\times 1+(3.8-3.1)\times 0+(4.0-3.8)\times 1+\\ & (4.9-4.0)\times 2+(5.6-4.9)\times 1+(5.8-5.6)\times 2+(7.2-5.8)\times 3+\\ & (8.6-7.2)\times 4]/8.6\\ = {} & 17.6/8.6\\ = {} & 2.05\end{aligned}$$

which is an estimate of the expected time-average number of customers in the system over the time period needed to observe $n = 6$ delays in queue; this performance measure could be denoted by $\ell(6)$.

(*d*) A box in the "Statistical counters" area of each part of the figure would be added, keeping track of the accumulated area under $L(t)$; its values at the fourteen values of the simulation clock are:

| Clock | 0 | 0.4 | 1.6 | 2.1 | 2.4 | 3.1 | 3.3 | 3.8 | 4.0 | 4.9 | 5.6 | 5.8 | 7.2 | 8.6 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Area under $L(t)$ | 0 | 0.0 | 1.2 | 2.2 | 3.1 | 4.5 | 4.7 | 4.7 | 4.9 | 6.7 | 7.4 | 7.8 | 12.0 | 17.6 |

Note that at each clock value, the area under $L(t)$ is equal to the sum of the areas under $Q(t)$ and under $B(t)$, as expected since $L(t) = Q(t) + B(t)$ for all times $t$. When implementing this, it is important to compute the area of the rectangle at each event using the *old* value of $L(t)$, i.e., *before* the corresponding state variable is updated.

1.4. We would no longer need to keep track of the times of arrival of the customers in queue, since the only thing they were used for was to compute the customers' delays in queue. We could thus eliminate the array holding these times of arrival, and delete the code in the subprogram for the departure event to move the queue up one customer whenever anyone leaves the queue and enters service; we could also delete the check in the subprogram for the arrival event to check whether there is enough space in the queue to hold the arriving customer. We would still need to have a state variable to hold the number of customers in queue, since that is used to get the time-average number of customers in queue, which we still want.

**1.5.** From the first nine interarrival times and the first six service times given at the beginning of Sec. 1.4.2 for the realization there, the most we can figure out in terms of the record of arrival times, service-starting times, departure times, and times in system of customers is this:

| Customer number ($i$) | Time of arrival | Time service starts | Time of departure | $W_i$ |
|---|---|---|---|---|
| 1 | 0.4 | 0.4 | 2.4 | 2.0 |
| 2 | 1.6 | 2.4 | 3.1 | 1.5 |
| 3 | 2.1 | 3.1 | 3.3 | 1.2 |
| 4 | 3.8 | 3.8 | 4.9 | 1.1 |
| 5 | 4.0 | 4.9 | 8.6 | 4.6 |
| 6 | 5.6 | 8.6 | 9.2 | 3.6 |
| 7 | 5.8 | 9.2 | | |
| 8 | 7.2 | | | |
| 9 | 9.1 | | | |

Thus, the average of the times in system of the first five customers to depart is (2.0 + 1.5 + 1.2 + 1.1 + 4.6)/5 = 2.08. But we must think of how to get this out of the simulation logic and code.

With the stopping rule for the simulation as given in the problem ($m$ customers have exited the system) rather than originally ($n$ customers have completed their delay in queue), we could get by without any new state variables, since all customers entering service during the simulation will leave service (and thus the system) before the simulation ends. The time in system for a customer could thus be computed at the time of his or her entry into service, as the sum of the delay in queue (just computed) plus the service time (generated now); this would be added into a counter to be divided by $m$ when the simulation ends. We would add a box in the "Statistical counters" area of each part of Fig. 1.7 to keep track of the accumulated times in system; its value is incremented whenever a service time starts (marked with a * below), and at the fourteen values of the simulation clock it is:

| Clock | 0 | 0.4* | 1.6 | 2.1 | 2.4* | 3.1* | 3.3 | 3.8* | 4.0 | 4.9* | 5.6 | 5.8 | 7.2 | 8.6* |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Total time in system | 0 | 2.0 | 2.0 | 2.0 | 3.5 | 4.7 | 4.7 | 5.8 | 5.8 | 10.4 | 10.4 | 10.4 | 10.4 | 14.0 |

(Although we are able to determine that $W_6 = 3.6$, this is not used in computing $\hat{w}(5) = 10.4 / 5 = 2.08$.) If the only performance measure were $\hat{w}(5)$, we could stop the simulation at time 4.9 since we know the value of $W_5$ at that time even though customer 5 does not leave until time 8.6. However, if we wanted other performance measures (such as the time-average number in queue) defined for the period required for 5 customers to leave, we would have to continue the simulation until customer 5 actually leaves, i.e., until time 8.6.

An alternative approach, which seems more natural but does involve introducing another state variable, is to wait until a customer actually finishes service and leaves before computing his or her time in system, and adding it into an accumulator. Here, we would need a state variable giving the time of arrival to the system of the customer currently in service (being undefined if the server is idle), which is subtracted from the clock when the customer leaves. While this produces the same result for this stopping rule, the times at which the new accumulator is incremented are different:

| Clock | 0 | 0.4 | 1.6 | 2.1 | 2.4* | 3.1* | 3.3* | 3.8 | 4.0 | 4.9* | 5.6 | 5.8 | 7.2 | 8.6* |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Total time in system | 0 | 0 | 0 | 0 | 2.0 | 3.5 | 4.7 | 4.7 | 4.7 | 5.8 | 5.8 | 5.8 | 5.8 | 14.0 |

again giving $\hat{w}(5) = 10.4 / 5 = 2.08$. Unlike the first approach, we would definitely have to run the simulation out to time 8.6 to compute even $\hat{w}(5)$, let alone other performance measures.

Although the second approach involves an additional state variable and might require running the simulation longer, it would allow us to change the form of the stopping rule, for instance to the original one in the text or to a fixed-time-of-stopping rule, and the average time in system would still be computed correctly, being based only on those customers who actually left the system during the simulation, however terminated. The first approach, on the other hand, could give incorrect results. For example, if our stopping rule in this realization were to run the simulation until (exactly) time $t = 9$, then five customers would complete their times in system, with an average of 10.4/5 = 2.08; the second approach would produce this result correctly, but the first approach, since customer six *began* (but did not complete) service before the simulation ends, would incorrectly produce 14.0/6 = 2.33.

1.6. The general mathematical expression would be

$$\max_{0 \le t \le T(n)} Q(t)$$

where $T(n)$ is the time the simulation ends (when the $n$th customer completes his or her delay in queue), and $Q(t)$ is the number of customers in queue at time $t$.

To compute this during the simulation, add a box to the "Statistical counters" area of each part of Fig. 1.7 to keep track of the maximum length the queue has attained so far. Before the state variables for each event have been updated, we would check to see if the current value of the number in queue (i.e., the value that has been in force since the last event) exceeds the largest value we had seen previously for it; if so, this largest value is replaced by the current (larger) value. At the fourteen values of the simulation clock in the realization of the hand simulation in Sec. 1.4.2, this would be

| Clock | 0 | 0.4 | 1.6 | 2.1* | 2.4* | 3.1 | 3.3 | 3.8 | 4.0 | 4.9 | 5.6 | 5.8 | 7.2 | 8.6* |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Maximum number in queue | 0 | 0 | 0 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 3 |

where the times that a new maximum is observed are marked with a *. The output performance measure is simply the final value in this box, being 3 in this case.

As an alternative approach, the check for a new maximum could be made *after* the state variables (including the queue length) are updated in the event. For this realization, we would thus get

| Clock | 0 | 0.4 | 1.6* | 2.1* | 2.4 | 3.1 | 3.3 | 3.8 | 4.0 | 4.9 | 5.6 | 5.8 | 7.2* | 8.6 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Maximum number in queue | 0 | 0 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 3 | 3 |

which again gives a value of 3.

Either of the above two approaches is valid for this stopping rule, since the simulation stops either when a customer leaves service and the $n$th customer is in queue and enters service (in which case the queue length falls by 1), or when an arrival (the $n$th one) finds the server idle (in which case the queue length remains at zero); in neither case can the queue length grow at termination, so nothing is missed in the first approach by doing the check before updating the state variables.

For a different type of stopping rule, though, the first approach could be wrong. For instance, if the simulation stopped when the 100th customer *arrives* to the system, it could be that this arrival would establish a new maximum queue length, which we would want to count since the time interval over which the maximum is taken includes the final time of the simulation; the first approach would miss this, and thus produce a result that is too small (by one). [On the other hand if we defined the maximum slightly differently to exclude the final time of the clock (i.e., a "<" rather than a "≤" in the second inequality under the "max" operator above), it would instead be the second approach that would be wrong.] The moral is to be very careful both in defining exactly what it is that is supposed to be computed, and in deciding the order in which to execute updates of state variables and statistical counters in the simulation code.

1.7. (*a*) We used the logic in Prob. 1.3, with a new state variable representing $L(t)$ = the total number of customers in the system at any time. Whenever a customer arrives this variable is incremented by 1, and whenever a customer finishes service and departs it is decremented by 1; the area update is done alongside the earlier code to compute the area under $Q(t)$.

(*b*) We used the second approach of Prob. 1.5, i.e., introduced a new state variable giving the time of arrival to the system of the customer currently in service, and subtracted this from the clock time of departure for each customer; the values were added into a new statistical accumulator. Since the stopping rule for the simulation is when $n = 1000$ *delays* have been completed, there will have been $n - 1 = 999$ times in service completed [regardless of whether the simulation ends with the $n$th arrival's seeing an idle server (and thus there were $n - 1$ departures), or whether the $n$th entry into service is from the queue upon the $(n - 1)$st departure], so the divisor is $n - 1$ rather than $n$.

(*c*) We used the second approach in Prob. 1.6, checking for a new maximum queue length *after* the state changes are made in each event.

(*d*) A new statistical counter, the maximum delay in queue observed so far, is created and initialized to 0. Each time a delay in queue is observed, it is checked to see if it is a new maximum; if so, it becomes the current maximum.

(*e*) A new statistical counter, the maximum time in system observed so far, is created and initialized to 0. Each time a time in system is observed, it is checked to see if it is a new maximum; if so, it becomes the current maximum.

(*f*) A new statistical counter, the number of delays in excess of 1 minute, is created and initialized to 0. Each time a (nonzero) delay in queue is observed, this new statistical counter is incremented by 1 if the delay is > 1; it is divided by $n = 1000$ at the end of the simulation to get the desired proportion.

Here are the new variables:

| Definition | Variable |
|---|---|
| Input parameter: | |
| Definition of "long" delay (=1.0 here) | long_delay_def |
| Modeling variables: | |
| Number of customers now in system | num_in_sys |
| Time of arrival of customer in service | time_arrival_servee |
| Maximum queue length so far | max_num_in_q |
| Area under $L(t)$ so far | area_num_in_sys |
| Total of times in system so far | total_time_in_sys |
| Maximum delay so far | max_delay |
| Maximum time in system so far | max_time_in_sys |
| Number of delays > 1 minute | num_long_delays |
| Time in system of a customer | time_in_sys |

In the code below, lines where there were changes or additions to the file in Sec. 1.4.4 are marked with vertical lines on the left, with the actual changes' being in italics.

Below is the code, omitting functions timing and expon, which are exactly as in Figures 1.13 and 1.18, respectively.

```
/* External definitions for Prob. 1.7. */
#include <stdio.h>
#include <math.h>
#include "rand.h"     /* Header file for random-number generator. */
#define Q_LIMIT 100  /* Limit on queue length. */
#define BUSY      1  /* Mnemonics for server's being busy */
#define IDLE      0  /* and idle. */
int   next_event_type, num_custs_delayed, num_delays_required,
      num_events, num_in_q, server_status, num_in_sys, max_num_in_q;
float area_num_in_q, area_server_status, mean_interarrival,
      mean_service, time, time_arrival[Q_LIMIT + 1],
      time_last_event, time_next_event[3], total_of_delays,
      area_num_in_sys, time_arrival_servee, total_time_in_sys,
      max_delay, max_time_in_sys, num_long_delays, long_delay_def;
FILE  *infile, *outfile;
void  initialize(void);
void  timing(void);
void  arrive(void);
void  depart(void);
void  report(void);
void  update_time_avg_stats(void);
float expon(float mean);
```

```
main()  /* Main function. */
{
    /* Open input and output files. */
    infile  = fopen("problp7.in",  "r");
    outfile = fopen("problp7.out", "w");
    /* Specify the number of events for the timing function. */
    num_events = 2;
    /* Read input parameters. */
    fscanf(infile, "%f %f %d %f", &mean_interarrival, &mean_service,
           &num_delays_required, &long_delay_def);
    /* Write report heading and input parameters. */
    fprintf(outfile, "Problem 1.7\n\n");
    fprintf(outfile, "Mean interarrival time%11.3f minutes\n\n",
            mean_interarrival);
    fprintf(outfile, "Mean service time%16.3f minutes\n\n",
            mean_service);
    fprintf(outfile, "Number of customers%14d\n\n",
            num_delays_required);
    fprintf(outfile, "Definition of long delay%9.3f minutes\n\n",
            long_delay_def);
    /* Initialize the simulation. */
    initialize();
    /* Run the simulation while more delays are still needed. */
    while (num_custs_delayed < num_delays_required) {
        /* Determine the next event. */
        timing();
        /* Update time-average statistical accumulators. */
        update_time_avg_stats();
        /* Invoke the appropriate event function. */
        switch (next_event_type) {
            case 1:
                arrive();
                break;
            case 2:
                depart();
                break;
        }
        /* Check to see if a new maximum queue length was just
           established. */
        if (num_in_q > max_num_in_q) max_num_in_q = num_in_q;
    }
    /* Invoke the report generator and end the simulation. */
    report();
    fclose(infile);
    fclose(outfile);
    return 0;
}
```

```
void initialize(void)  /* Initialization function. */
{
    /* Initialize the simulation clock. */
    time = 0.0;
    /* Initialize the state variables. */
    server_status   = IDLE;
    num_in_q        = 0;
    time_last_event = 0.0;
    num_in_sys      = 0;
    /* Initialize the statistical counters. */
    num_custs_delayed  = 0;
    total_of_delays    = 0.0;
    area_num_in_q      = 0.0;
    area_server_status = 0.0;
    area_num_in_sys    = 0.0;
    total_time_in_sys  = 0.0;
    max_num_in_q       = 0;
    max_delay          = 0.0;
    max_time_in_sys    = 0.0;
    num_long_delays    = 0.0;
    /* Initialize event list.  Since no customers are present, the
       departure (service completion) event is eliminated from
       consideration. */
    time_next_event[1] = time + expon(mean_interarrival);
    time_next_event[2] = 1.0e+30;
}
```

```
void arrive(void)  /* Arrival event function. */
{
    float delay;
    /* Increment number in system. */
    ++num_in_sys;
    /* Schedule next arrival. */
    time_next_event[1] = time + expon(mean_interarrival);
    /* Check to see whether server is busy. */
    if (server_status == BUSY) {
        /* Server is busy, so increment number of customers in queue.
           */
        ++num_in_q;
        /* Check to see whether an overflow condition exists. */
        if (num_in_q > Q_LIMIT) {
            /* The queue has overflowed, so stop the simulation. */
            fprintf(outfile, "\nOverflow of the array time_arrival at");
            fprintf(outfile, " time %f", time);
            exit(2);
        }
        /* There is still room in the queue, so store the time of
           arrival of the arriving customer at the (new) end of
           time_arrival. */
        time_arrival[num_in_q] = time;
    }
    else {
        /* Server is idle, so arriving customer has a delay of zero.
           (The following two statements are for program clarity and do
           not affect the results of the simulation.) */
        delay            = 0.0;
        total_of_delays += delay;
        /* Increment the number of customers delayed, and make server
           busy. */
        ++num_custs_delayed;
        server_status = BUSY;
        /* Set time of arrival of servee. */
        time_arrival_servee = time;
        /* Schedule a departure (service completion). */
        time_next_event[2] = time + expon(mean_service);
    }
}
```

```
void depart(void)  /* Departure event function. */
{
    int   i;
    float delay, time_in_sys;
    /* Decrement number in system. */
    --num_in_sys;
    /* Compute the time in system of the departing customer and update
       the total time in system accumulator. */
    time_in_sys = time - time_arrival_servee;
    total_time_in_sys += time_in_sys;
    /* Check to see if this was a new maximum time in system. */
    if (time_in_sys > max_time_in_sys) max_time_in_sys = time_in_sys;
    /* Check to see whether the queue is empty. */
    if (num_in_q == 0) {
        /* The queue is empty so make the server idle and eliminate the
           departure (service completion) event from consideration. */
        server_status      = IDLE;
        time_next_event[2] = 1.0e+30;
    }
    else {
        /* The queue is nonempty, so decrement the number of customers
           in queue. */
        --num_in_q;
        /* Compute the delay of the customer who is beginning service
           and update the total delay accumulator. */
        delay            = time - time_arrival[1];
        total_of_delays += delay;
        /* Check to see if this was a new maximum delay. */
        if (delay > max_delay) max_delay = delay;
        /* Check to see if this was a long delay. */
        if (delay > long_delay_def) ++ num_long_delays;
        /* Increment the number of customers delayed, and schedule
           departure. */
        ++num_custs_delayed;
        time_next_event[2] = time + expon(mean_service);
        /* Set time of arrival of servee. */
        time_arrival_servee = time_arrival[1];
        /* Move each customer in queue (if any) up one place. */
        for (i = 1; i <= num_in_q; ++i)
            time_arrival[i] = time_arrival[i + 1];
    }
}
```

```
void report(void)  /* Report generator function. */
{
    /* Compute and write estimates of desired measures of performance.
       */
    fprintf(outfile, "\n\nAverage delay in queue%11.3f minutes\n\n",
            total_of_delays / num_custs_delayed);
    fprintf(outfile, "Average number in queue%10.3f\n\n",
            area_num_in_q / time);
    fprintf(outfile, "Server utilization%15.3f\n\n",
            area_server_status / time);
    fprintf(outfile, "Average number in system%9.3f\n\n",
            area_num_in_sys / time);
    fprintf(outfile, "Average time in system%11.3f minutes\n\n",
            total_time_in_sys / (num_custs_delayed - 1));
    fprintf(outfile, "Maximum number in queue%10d\n\n", max_num_in_q);
    fprintf(outfile, "Maximum delay in queue%11.3f minutes\n\n",
            max_delay);
    fprintf(outfile, "Maximum time in system%11.3f minutes\n\n",
            max_time_in_sys);
    fprintf(outfile, "Proportion of long delays%8.3f\n\n",
            num_long_delays / num_custs_delayed);
    fprintf(outfile, "Time simulation ended%12.3f minutes", time);
}

void update_time_avg_stats(void)  /* Update area accumulators for
                                     time-average statistics. */
{
    float time_since_last_event;
    /* Compute time since last event, and update last-event-time
       marker. */
    time_since_last_event = time - time_last_event;
    time_last_event       = time;
    /* Update area under number-in-queue function. */
    area_num_in_q      += num_in_q * time_since_last_event;
    /* Update area under server-busy indicator function. */
    area_server_status += server_status * time_since_last_event;
    /* Update area under number-in-system function. */
    area_num_in_sys    += num_in_sys * time_since_last_event;
}
```

Here are the results from running the program:

```
Problem 1.7

 Mean interarrival time      1.000 minutes

 Mean service time           0.500 minutes

 Number of customers          1000

 Definition of long delay    1.000 minutes


 Average delay in queue      0.430 minutes

 Average number in queue     0.418

 Server utilization          0.460

 Average number in system    0.878

 Average time in system      0.903 minutes

 Maximum number in queue        11

 Maximum delay in queue      4.128 minutes

 Maximum time in system      4.549 minutes

 Proportion of long delays   0.162

 Time simulation ended    1027.914 minutes
```

The new output values seem, at least, to be reasonable. Note that the average time in system (0.903) is close to the sum of the average delay in queue (0.430) and the expected service time (0.5), and that the time-average number in system (0.878) is exactly equal to the sum of the time-average number in queue (0.418) and the server utilization (0.460), as it must be. It is also interesting that the three maximum values are all a *lot* larger than their respective average values, indicating that there was evidently appreciable variability during the course of the simulation.

1.8. To restate the original algorithm in words, we take the natural logarithm of a U(0, 1) random variable (i.e., $U$) and multiply the result by $-\beta$. In the proof that this works, the only point where probability came up was when we made use of the fact that $U$ (what we take the logarithm of) has a U(0, 1) distribution. Now, consider the random variable $1 - U$. Clearly, it is between 0 and 1, and for $0 \leq x \leq 1$, its distribution function is

$$P(1 - U \leq x) = P(U \geq 1 - x) = 1 - P(U < 1 - x) = 1 - (1 - x) = x$$

so that $1 - U$ has a U(0, 1) distribution as well. So the formula $-\beta \ln(1 - U)$, in words, says to take the natural logarithm of a U(0, 1) random variable ($1 - U$ this time) and multiply the result by $-\beta$; this is probabilistically equivalent to the original statement. While it may seem ridiculous to use the new formula (it requires another operation, the subtraction of $U$ from 1, so would be slower), it does have merit for other reasons; see Chap. 8.

1.9. The following are the results from making ten independent replications of the single-server queueing system:

| Replication | Average delay in queue | Average number in queue | Server utilization | Time simulation ended |
|---|---|---|---|---|
| 1 | 0.430 | 0.418 | 0.460 | 1027.914 |
| 2 | 0.437 | 0.438 | 0.489 | 997.530 |
| 3 | 0.493 | 0.506 | 0.508 | 975.400 |
| 4 | 0.536 | 0.538 | 0.504 | 997.671 |
| 5 | 0.417 | 0.412 | 0.491 | 1012.722 |
| 6 | 0.411 | 0.416 | 0.504 | 990.372 |
| 7 | 0.515 | 0.520 | 0.506 | 990.208 |
| 8 | 0.394 | 0.378 | 0.465 | 1042.093 |
| 9 | 0.436 | 0.426 | 0.487 | 1023.938 |
| 10 | 0.549 | 0.588 | 0.514 | 940.969 |

Note the variation for a particular output measure across the replications; for example, average delay in queue ranges from 0.394 to 0.549. This variation is due to the fact that different random numbers were used to drive each replication, and indicates that just making a single run of a simulation is *not* enough to understand the meaning of the output. This subject is addressed in Chapters 9 through 12.