Proceedings of the 2015 Winter Simulation Conference
L. Yilmaz, W. K. V. Chan, I. Moon, T. M. K. Roeder, C. Macal, and M. D. Rossetti, eds.

DISCRETE-EVENT SIMULATION USING R

Barry Lawson

Department of Mathematics and Computer Science
University of Richmond
Richmond, VA 23173, USA

Lawrence M. Leemis

Department of Mathematics
The College of William & Mary
Williamsburg, VA 23187, USA

ABSTRACT

R is a free software package with extensive statistical capability, customizable graphics, and both imperative and vectorized programming capabilities. For use in an introductory simulation course, the capabilities of R for analyzing simulation statistics, and for generating corresponding graphics, aid in developing student intuition. R also provides flexibility in determining whether simulation and analysis should be done using simulation code that students implement from scratch, using skeleton code which students modify, or using completed code given as a black box. These aspects of R make it a unique platform for programming and analyzing discrete-event simulations. In this paper, we present an R function named `ssq` which we wrote to simulate a single-server queue, and we provide several illustrations showing its use as an exemplar for using R in an introductory simulation course. All of the code to analyze the output from `ssq` uses functions from the base distribution of R.

1 INTRODUCTION

The choice of computing platform for a first course in discrete-event simulation is a difficult one, which typically includes making trade-offs between ease-of-use, cost, and ability to customize. If the first course emphasizes modeling, then commercial packages that are available for purchase by simulation software vendors should be considered. However, if the first course emphasizes programming in a high-level language as well as the probabilistic and statistical aspects of simulation, then the R language should be considered.

The reasons that make R a good choice for a computing platform for a first course in discrete-event simulation include the following.

- R is a programming language that includes conditional execution, looping, and functions — analogous to C or any other imperative programming language.
- Random number generation and random variate generation functions are built into R.
- R has its roots in the statistical analysis of data, appropriate for input modeling and output analysis.
- R has high-level graphics capability that is easy to use and to customize.

The major downsides to using R in a first course are that it does not handle animation well, that more lines of programming are required relative to the commercial simulation languages, and, because R is interpreted, execution time can be slow compared to imperative languages. We believe that the upsides associated with R outweigh the downsides, and we will illustrate some of the capabilities of R for simulation and analysis in this paper.

Because the simulation of a single-server queueing model illustrates many of the important concepts to be covered in discrete-event simulation, we use that model here to illustrate the use of R in an introductory simulation course. We have written a simulation program in R named `ssq` which conducts a discrete-event simulation of a single-server queue. (The program is available from the authors.) We have used R for

several other types of simulation models, including multiple-server queues, inventory, and machine shop models, but for brevity present only `ssq`.

2 THE SSQ PROGRAM IN R

In the context of this paper, we have developed an R program named `ssq` that acts as a black box implementation of a single-server queueing model to be used for simulation analysis. Within an introductory simulation course, the instructor may choose to have students either (a) implement their own such program from scratch, (b) modify a skeleton version provided by the instructor, or (c) use a completed version provided by the instructor. Then, simulation analysis will follow many of the examples presented later in this paper.

Our implementation of the program `ssq` allows execution with default parameter values, which results in the simulation of an M/M/1 queue using the following assumptions.

- The system begins at time 0 with no customers in the system and an idle server.
- The times between arrivals are mutually independent and identically distributed exponential random variates with arrival rate $\lambda = 1$.
- The service times are mutually independent and identically distributed exponential random variates with service rate $\mu = 10/9$. The traffic intensity is $\lambda/\mu = 0.9$.
- The server does not take any breaks.
- A customer departs the system once the service is complete.

Our `ssq` program has 16 optional arguments described in Table 1. Those arguments whose description ends with a question mark are of the logical type in R.

Table 1: Optional arguments for `ssq`.

Argument	Default	Description
`setSeed`	`TRUE`	set a random number seed?
`seed`	`NULL`	random number seed
`arrivalFunction`	`defaultArrival`	random interarrival time generator
`serviceFunction`	`defaultService`	random service time generator
`maxTime`	`10000.0`	maximum simulation time
`maxArrivals`	`Inf`	maximum number of arrivals to the system
`maxDepartures`	`Inf`	maximum number of departures from the system
`saveAllData`	`FALSE`	save all seven categories of data below?
`saveArrivals`	`FALSE`	save the individual interarrival times?
`saveServices`	`FALSE`	save the individual service times?
`saveWaits`	`FALSE`	save the individual wait times?
`saveSojourns`	`FALSE`	save the individual sojourn times?
`saveNumberInQueue`	`FALSE`	save the number in the queue?
`saveNumberInSystem`	`FALSE`	save the number in the system?
`saveServerStatus`	`FALSE`	save the server status?
`showOutput`	`TRUE`	display simulation progress and output?

Note that the `setSeed` argument should be explicitly set to `FALSE` in cases when the user wishes to set the initial random number seed external to the `ssq` call, e.g., for replication.

There are three ways to end the simulation:

- when the system clock reaches `maxTime`, the simulation terminates immediately;
- when the number of departures reaches `maxDepartures`, the simulation terminates immediately;

- when the number of arrivals reaches `maxArrivals`, the input to the simulation ends and remaining customers are processed (as long `maxTime` and `maxDepartures` are not reached in the meantime);

whichever of the three occurs first.

We designed our program to facilitate entry-level use of `ssq`. All parameters to the program have default values, and a few basic statistics are given as output (see below). The user may explicitly request other data for performance measures, as will be illustrated subsequently. With the exception of the basic statistics given as output, all other statistical analysis must be performed as post-processing using native R functions. We chose this design approach so that introductory students could gain first-hand experience in the analysis of output, rather than have that analysis built into the black box of the simulation.

As an example, the R session shown below left calls `ssq`, indicating on output that, for the default `maxTime` value of 10000 time units, 9914 customers arrived and 9913 customers departed from the system, with an average sojourn time of 8.82 units. The random number seed was defaulted to use the system clock, hence this exact simulation will (likely) not be reproduced on the next call to `ssq`. (The dots displayed just after the call to `ssq` indicate the progress of the simulation during execution. This progress and output display can be suppressed—see Table 1. We will omit progress dots in all future examples.) A second call to `ssq` with all arguments defaulted is given in the R session shown below middle. As suggested, the previous output is not reproduced: there were 10097 arrivals and 10093 departures, which means that four customers were in the system when the simulation terminated.

One way to obtain reproducible simulation results is to select a random number stream, manifested by the choice of a specified initial seed. This is good practice in general because using the system clock results in a seed selected at random such that the simulation results can not be reproduced. A third call to `ssq` specifying a `seed` argument is given below right. This particular simulation ended with 10034 arrivals and 10004 departures, which means that thirty customers were in the system when the simulation terminated. If this same call to `ssq` were made again, the exact same results would occur. For the rest of the paper, we will specify an initial seed for the random number generator so that the reader can reproduce our results.

```
> ssq()
..................
customerArrivals
[1] 9914

customerDepartures
[1] 9913

$avgWait
[1] 7.9241

$avgSojourn
[1] 8.8236

$utilization
[1] 0.89105

$avgNumInSys
[1] 8.747

$avgNumInQueue
[1] 7.8559
```

```
> ssq()
..................
customerArrivals
[1] 10097

customerDepartures
[1] 10093

$avgWait
[1] 7.2437

$avgSojourn
[1] 8.1387

$utilization
[1] 0.90275

$avgNumInSys
[1] 8.2147

$avgNumInQueue
[1] 7.3119
```

```
> ssq(seed=8675309)
......................
customerArrivals
[1] 10034

customerDepartures
[1] 10004

$avgWait
[1] 9.0524

$avgSojourn
[1] 9.9666

$utilization
[1] 0.91627

$avgNumInSys
[1] 10.017

$avgNumInQueue
[1] 9.1004
```

3 ALTERING INPUT MODEL DISTRIBUTIONS

It is important for students to experiment with various arrival and service models, comparing the resulting impact on the system. Accordingly, we designed our `ssq` program to allow the user to easily specify their own arrival or service model with as little as one additional line of R code.

The default input models for our `ssq` program result in an M/M/1 queue: interarrival times are exponential with a mean of 1 while service times are exponential with a mean of 0.9. These distributions are defaulted within the implementation of `ssq` using the two R functions below:

```
defaultArrival <- function() return(rexp(1, 1.0))
defaultService <- function() return(rexp(1, 1.0 / 0.9))
```

The `rexp` function in R is used to generate a random exponential variate. The first argument to `rexp` corresponds to the number of random variates required and the second parameter corresponds to the rate.

Consider an alternative situation with Weibull time between arrivals and gamma service times, i.e., a G/G/1 queue. This can be easily accomplished by writing two new one-line functions in R, passed to `ssq` as shown below. (For brevity, only the first few lines of output from `ssq` are shown.)

```
> myArr <- function() return(rweibull(1, shape=2, scale=1))
> mySvc <- function() return(rgamma(1, shape=2, scale=1))

> ssq(seed=8675309, arrivalFunction=myArr, serviceFunction=mySvc)
customerArrivals
[1] 11267

customerDepartures
[1] 5053
```

A very different arrival and service pattern emerge from the use of the Weibull and gamma distributions. The results show that there are 11267 – 5053 = 6214 customers in the system when the simulation terminates at the default maximum time of 10000. Intuitively, this suggests to the student that this combination of arrival and service models results in a more congested queue compared to the earlier models.

4 ANALYZING ARRIVAL PROCESSES

Returning to the M/M/1 queue, we now show how R can be used to analyze the system's arrival process using observations generated by the simulation. Our implementation allows the user to save the individual interarrival times of customers arriving to the system. For example, the commands below use `ssq` to create a list named `output` with eight components, which can be displayed using the `str` (structure) function:

```
> output = ssq(seed=8423089, saveArrivals=TRUE, showOutput=FALSE)
> str(output)
List of 8
 $ customerArrivals  : num 10260
 $ customerDepartures: num 10252
 $ avgWait           : num 8.38
 $ avgSojourn        : num 9.28
 $ utilization       : num 0.923
 $ avgNumInSys       : num 9.52
 $ avgNumInQueue     : num 8.59
 $ interarrivalTimes : num [1:10260] 0.175 1.143 1.303 2.723 0.703 ...
```

There were 10260 customer arrivals corresponding to 10260 interarrival times, which are 0.175, 1.143, etc.

The sample mean and sample standard deviation of the interarrival times can be calculated using the additional R commands

```
> mean(output$interarrivalTimes)
[1] 0.9745776
> sd(output$interarrivalTimes)
[1] 0.9681507
```

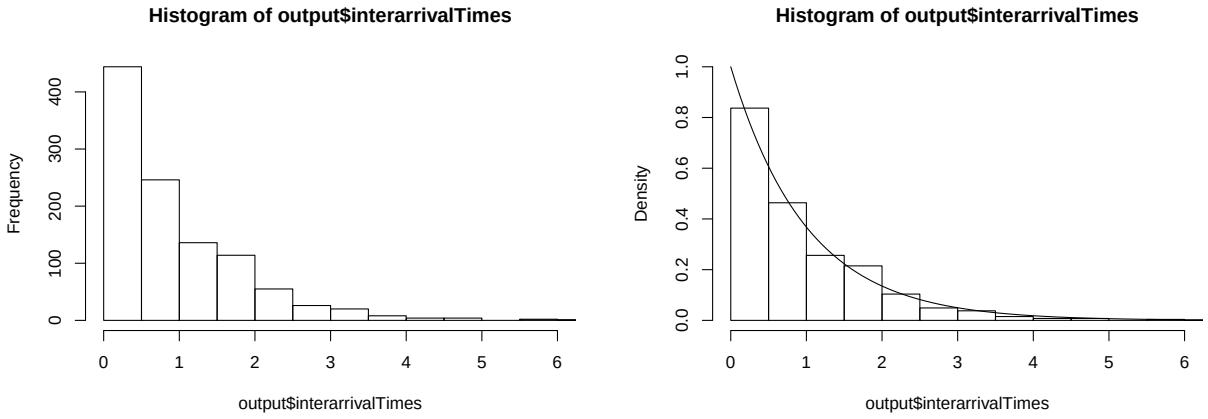
The sample interarrival time mean is just slightly below its population value of 1.0, indicating that this particular simulation run is slightly more congested than average. The sample standard deviation is also near its population value of 1.0.

The graphical capabilities of R can also be used to analyze the interarrival times. For example, as shown below, a histogram of the interarrival times can be generated using the `hist` function in R, depicted in Figure 1(a). (The `xlim` argument in the call to `hist` controls the limits on the horizontal axis.)

```
> hist(output$interarrivalTimes, xlim=c(0,6))
```

To show that the interarrival times closely mimic the population distribution, the population probability density function can be superimposed onto the histogram using the following R code. (Note that the `prob=TRUE` argument in the call to `hist` forces the area under the histogram to be 1.0 so that it can be easily compared with the population probability density function. The `add=TRUE` argument in the call to `curve` tells R to draw the function on top of the existing histogram.) Not surprisingly, the histogram provides a close approximation to the population distribution, as shown in Figure 1(b).

```
> hist(output$interarrivalTimes, xlim=c(0,6), ylim=c(0,1), prob=TRUE)
> curve(exp(-x), from=0, to=6, add=TRUE)
```



(a) (b)

Figure 1: (a) Histogram of interarrival times. (b) Histogram with population pdf superimposed.

Such a visual display, facilitated by R's easy use of graphics, is particularly compelling to those new to simulation, and leads to better intuition about the effects of different distributions. In addition, for courses including a focus on implementation and simulation "world views", providing the ability to save individual customer statistics allows for meaningful discussion of trade-offs between process-oriented (easy to collect individual statistics) versus event-oriented (additional data structures required) approaches.

5 ANALYZING SERVICE PROCESSES

We now show how R can be used to analyze the service process of a system using observations generated by the simulation. Similar to that for the arrival process, our implementation allows the user to save the individual service times of customers. Below we use `ssq` (with immediate output omitted), limiting the number of arrivals, and hence the number of service times, to 50. Another example of R's statistical capabilities is given by the `fivenum` function, which displays Tukey's five-number summary of the service times: minimum, estimated 25th percentile, sample median, estimated 75th percentile, and maximum.

```
> output = ssq(seed=3, maxArrivals=50, saveServices=TRUE, showOutput=FALSE)
> fivenum(output$serviceTimes)
[1] 0.133141 0.484717 0.712196 1.170141 2.461797
```

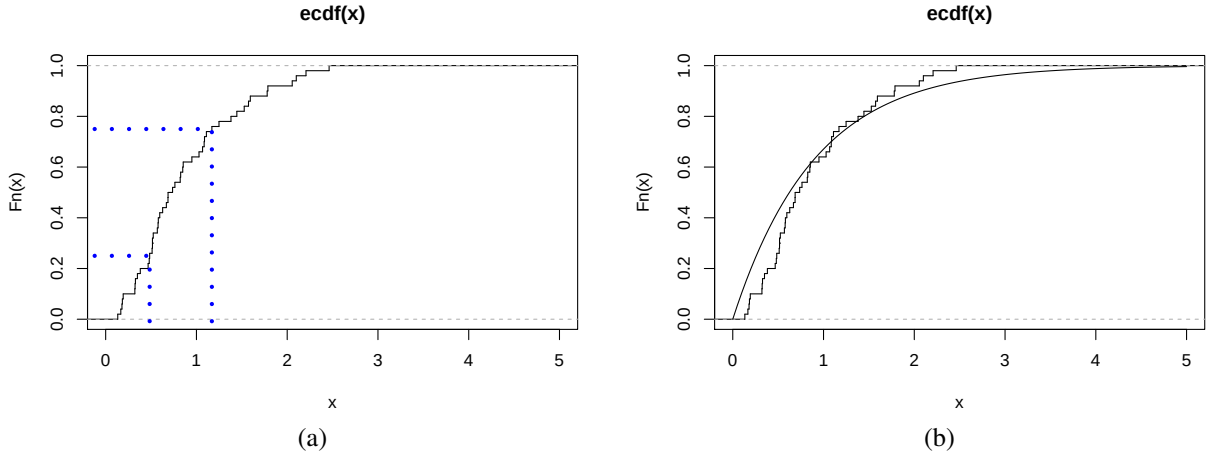
Then a plot of the empirical cumulative distribution function, generated using the `plot.ecdf` function, can be used to visualize those statistics in the context of the overall distribution of service times. For example, the 25th and 75th percentile estimates from Tukey's five-number summary can be superimposed using heavy dotted blue line segments on the ecdf plot. The resulting plot in Figure 2(a) clearly indicates that the majority of the 50 service times are roughly one time unit or less.

```
> plot.ecdf(output$serviceTimes, xlim=c(0,5), verticals=TRUE, pch="")
> xvals.25 = c(-0.5,0.485,0.485,0.485); yvals.25 = c(0.25,0.25,0.25,-0.5)
> xvals.75 = c(-0.5,1.170,1.170,1.170); yvals.75 = c(0.75,0.75,0.75,-0.5)
> lines(x=xvals.25, y=yvals.25, col="blue", lwd=4, lty="dotted")
> lines(x=xvals.75, y=yvals.75, col="blue", lwd=4, lty="dotted")
```

As before, the `curve` function can also be used to superimpose the population cumulative distribution function. The resulting plot, given in Figure 2(b), shows that there are fewer observations in both tails of the distribution than expected, resulting in 50 service times that are compressed relative to the population values. These service times, which are expressing less variability than usual, will have an impact on the wait times and sojourn times for customers.

```
> plot.ecdf(output$serviceTimes, xlim=c(0,5), verticals=TRUE, pch="")
> curve(1 - exp(-10 * x / 9), from=0, to=5, add=TRUE)
```

Again, R's easy-to-use graphics capabilities allow the novice user to intuit aspects of the underlying models in the simulation.



(a) (b)

Figure 2: Empirical cdf with superimposed: (a) 25th and 75th percentile estimates; (b) population pdf.

6 ANALYZING OBSERVATION-PERSISTENT STATISTICS

A very important concept to discuss in an introductory simulation course is the generation and analysis of observation-persistent statistics versus time-persistent statistics. In this section, we show the power of R for observation-persistent statistics; we discuss time-persistent statistics in the next section.

In a queueing model, the sojourn time (sum of wait time and service time) of each customer is an interesting quantity to observe in the aggregate. As an example, we show below the use of `ssq` to save the sojourn times for the first 8000 customers into the list `output`, followed by Tukey's five-number summary of those 8000 sojourn times.

```
> output = ssq(seed=8675309, maxArrivals=8000, saveSojourns=T, showOutput=F)
> fivenum(output$sojournTimes)
[1]  0.001658612  3.131279795  7.121564269 13.382116960 49.428772754
```

The minimum time in the system is 0.0017, the median time in the system is 7.1216, and the maximum time in the system is 49.4288.

It is often the case in a queueing application that the interest is in customers with long sojourn times, because a long wait can result in customer dissatisfaction. The 95th percentile, for example, can be estimated with either of the two following R commands.

```
> sort(output$sojournTimes)[0.95 * 8000]
[1] 24.7899
> quantile(output$sojournTimes, 0.95)
     95%
24.7902
```

The estimate for the 95th percentile of the distribution of the sojourn times is 24.79, whether obtained by extracting the 95th percentile value from the sorted sojourn times or by using the (slightly more precise) `quantile` function provided by R. Such a dual approach provides an important consistency check, and helps the student with intuition about those values returned by functions like `quantile`.

A discussion of autocorrelation, a very important concept when considering simulation statistics, is facilitated by R's powerful statistical and graphical capabilities. As an example, a plot of the customer number (on the horizontal axis) versus the sojourn time (on the vertical axis) is easily accomplished using the R command

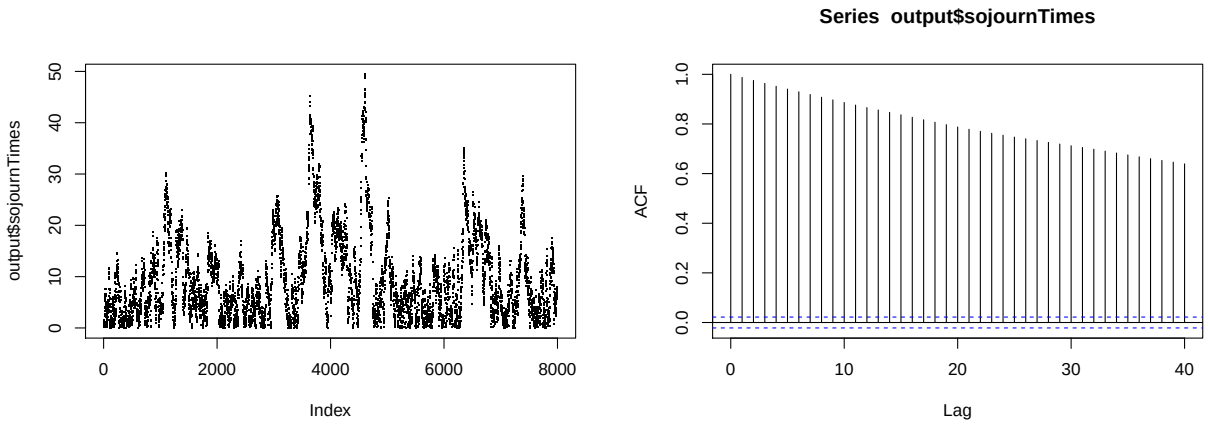
```
> plot(output$sojournTimes, pch=".")
```

(The `pch`, or plotting character, argument places a small dot for each sojourn time.) As shown in Figure 3(a), it is clear that the queue rises and falls in a cyclic fashion, corresponding to a high degree of autocorrelation between the sojourn times.

Further demonstrating the power of R in this context, the first 40 lags of the sample autocorrelation function (that is, the correlogram) can be plotted simply by using the additional R command

```
> acf(output$sojournTimes, lag.max=40)
```

As shown in Figure 3(b), the correlogram shows a strong statistically significant positive autocorrelation that extends well beyond the first 40 lags. The dashed lines on either side of the horizontal axis are 95% confidence bounds which are helpful in determining statistical significance.



(a) (b)

Figure 3: (a) Individual sojourn times plotted in order. (b) Autocorrelation function of the sojourn times.

Another plot of interest associated with the sojourn times is the spectral density function, that is, the periodogram. The additional R command to compute and plot the spectral density function is simply

```
> spectrum(output$sojournTimes)
```

The resulting spectral density function shown in Figure 4 indicates that there is more contribution to the variability of the sojourn times from lower frequencies than from higher frequencies, consistent with the data values shown in Figure 3(a). In other words, if the plotted sojourn times exhibited high frequency oscillation (e.g., one observation low, the next one high, the next one low, etc.), the spectral density function would have more weight on the high frequency end. However, sojourn times have long cycles that are associated with the queue backing up for a series of customers, then emptying out, then filling up for another series of customers. This corresponds to low frequency variations, as identified by Figure 4.

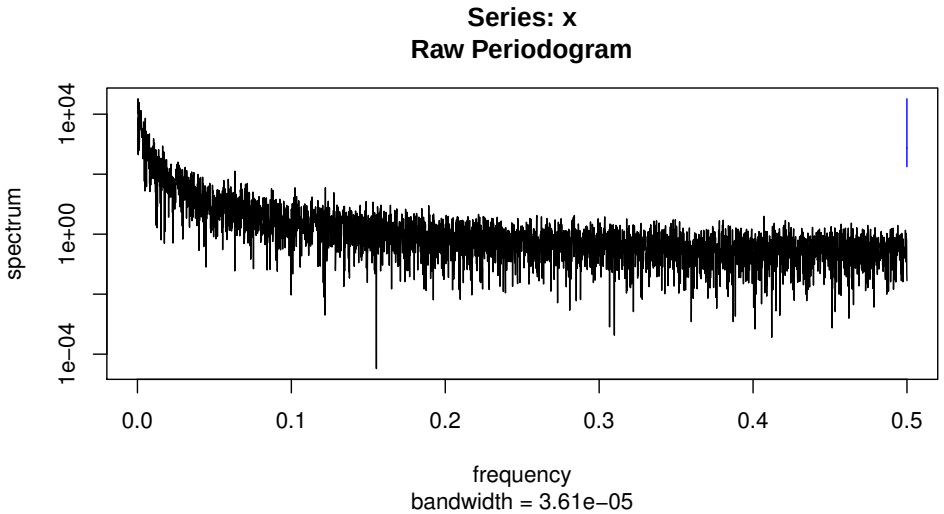


Figure 4: Spectral density function of the sojourn times.

Furthermore, after discarding the first 2000 sojourn times to account for a warmup period, a point estimate of the mean sojourn time along with a 95% confidence interval (using the classical confidence interval procedure) can be calculated with the additional R commands

```
> truncated.times = output$sojournTimes[2001:8000]  # discard first 2000 times
> mean(truncated.times)
[1] 9.86094
> t.test(truncated.times)$conf.int
[1] 9.64135 10.08052
```

The 95% confidence interval $9.64 < \mu < 10.08$ is returned, where μ is the mean population sojourn time. However, using individual observations and the sample autocorrelation function given in Figure 3b make the normality and independence assumptions associated with this confidence interval suspect. A batch means procedure (Law 2015) involving n batch means can be implemented using a simple R function like that shown below.

```
batch.means = function(values, n) {
  z = matrix(values, n)               # divide values into n batches
  t.test(apply(z, 1, mean))$conf.int  # t-test on means of batches
}
```

Once again discarding the first 2000 sojourn times to allow the system to warm up, and dividing the remaining 6000 sojourn times into 12 batches of 500 observations each, the additional R command

```
> batch.means(truncated.times, 12)
[1] 9.82541 9.89646
```

calculates the 95% confidence interval $9.83 < \mu < 9.90$, which is significantly narrower than the 95% confidence interval given by classical methods.

Another experiment can be conducted using the sojourn times, although this experiment involves the first three sojourn times rather than the first 8000 sojourn times. Kaczynski et al. (2012) give the 3×3 variance–covariance matrix of the first three sojourn times for an M/M/1 queue with arrival rate λ and service rate μ as

$$\Sigma = \begin{bmatrix} \dfrac{1}{\mu^2} & \dfrac{\lambda(2\mu+\lambda)}{(\lambda+\mu)^2\mu^2} & \dfrac{\lambda^2(\lambda^2+4\lambda\mu+5\mu^2)}{(\lambda+\mu)^4\mu^2} \\ \bullet & \dfrac{2\lambda^2+4\lambda\mu+\mu^2}{(\lambda+\mu)^2\mu^2} & \dfrac{\lambda(2\lambda^2+8\lambda^2\mu+11\lambda\mu^2+2\mu^3)}{(\lambda+\mu)^4\mu^2} \\ \bullet & \bullet & \dfrac{3\lambda^6+18\lambda^5\mu+45\lambda^4\mu^2+54\lambda^3\mu^3+30\lambda^2\mu^4+8\lambda\mu^5+\mu^6}{(\lambda+\mu)^6\mu^2} \end{bmatrix}$$

where the $\bullet$'s correspond to symmetric values about the diagonal. Using the default parameters $\lambda = 1$ and $\mu = 10/9$ reduces this matrix to

$$\Sigma = \begin{bmatrix} \dfrac{81}{100} & \dfrac{21141}{36100} & \dfrac{6173901}{13032100} \\ \dfrac{21141}{36100} & \dfrac{25191}{18050} & \dfrac{7230951}{6516050} \\ \dfrac{6173901}{13032100} & \dfrac{7230951}{6516050} & \dfrac{9202705083}{4704588100} \end{bmatrix} \approx \begin{bmatrix} 0.8100 & 0.5856 & 0.4737 \\ 0.5856 & 1.3956 & 1.1097 \\ 0.4737 & 1.1097 & 1.9561 \end{bmatrix}.$$

Our `ssq` function can be used to support this result by limiting the number of arrivals to three and then saving those three customer sojourn times. The `kaczynski` function below provides a good example of how students may write their own R code to support existing implementations and to assist in analysis. This function accepts as an argument the number of simulation replications. The second command defines `times` as a `numReps` $\times$ 3 matrix containing 0's as elements; each row will be used to hold the first three customer sojourn times in an M/M/1 queue. Within the loop, the first three sojourn times generated by `ssq` are placed in row `i` of `times`. Finally, the function returns the `times` matrix of sojourn times.

```
kaczynski <- function(numReps) {
  times = matrix(0, numReps, 3)     # a numReps X 3 matrix, all 0's
  set.seed(8423089)                 # replication: set seed once
  for (i in 1:nrep) {
    out = ssq(setSeed=F, maxArrivals=3, saveSojourns=T, showOutput=F)
    times[i, ] = out$sojournTimes  # ith row: first 3 sojourn times
  }
  return(times)
}
```

Using the function above with, say, 100 000 replications, the `var` function available in R can then be used to compute the sample variance–covariance matrix of the sojourn times, which is displayed as output below. This output, along with several more runs of the simulation using different initial seed values, agrees with the analytic result of Kaczynski et al. (2012).

```
> times = kaczynski(100000)
> var(times)
          [,1]      [,2]      [,3]
[1,] 0.8137554 0.5873098 0.4725895
[2,] 0.5873098 1.3894298 1.1035314
[3,] 0.4725895 1.1035314 1.9593896
```

Indeed, whether by using only native R functions or by writing additional code, the flexibility and power of R to assist in simulation experimentation and analysis is evident.

7 ANALYZING TIME-PERSISTENT STATISTICS

Another very important concept that routinely appears in simulation is that of time-persistent statistics. In an introductory course, the ability to visualize and analyze time-persistent data and their statistics is critical for student understanding. Again, R's power and ease of use are well suited in this context. Whereas in the previous section we used R for observation-persistent statistics, in this section we show the power of R for time-persistent statistics using the number in the queue over time as generated by our `ssq` program.

In the example below, we use `ssq` to run the simulation for the first 50 customers, saving the number in queue over time in the list `output`.

```
> output = ssq(seed=8675309, maxArrivals=50, saveNumberInQueue=T, showOutput=F)
```

Contained in the resulting list `output` is a vector named `output$numberInQueueT`, with the 68 time values when changes occurred to the number in the queue, displayed below. Correspondingly, the vector `output$numberInQueueN` contains the associated number in queue. (Note that for some of the times when the queue length is zero, customers arrived and departed, but the number in the queue did not change.)

```
> out$numberInQueueT
 [1]  0.0  3.9  4.4  4.6  4.6  5.0  5.1  5.2  5.6 15.2 15.8 25.1 25.6 25.6 25.9
[16] 26.3 27.2 27.4 28.1 28.1 28.3 29.5 30.3 31.1 31.1 31.1 31.3 31.4 32.9 33.0
[31] 33.7 34.1 35.8 35.9 36.0 37.6 38.8 39.5 39.7 39.7 39.8 39.9 40.2 40.2 40.9
[46] 41.2 42.0 42.3 42.3 42.4 42.8 44.0 44.6 45.7 45.8 46.2 47.1 48.9 49.4 49.5
[61] 50.3 52.1 52.3 52.4 53.0 53.1 54.9 55.3
> output$numberInQueueN
 [1] 0 1 2 1 2 3 2 1 0 1 0 1 2 3 4 3 4 3 4 5 4 3 4 5 4 5 4 3 4 5 6 5 6 7 6 5 4
 [38] 3 2 3 4 3 2 3 2 3 2 1 0 1 2 3 2 3 2 3 4 5 6 7 6 5 4 3 2 1 0 0
```

A visualization of the number in queue over time is generated using the R `plot` function as

```
> plot(output$numberInQueueT, output$numberInQueueN, type="s",
       xlab="time", ylab="number in queue", las=1, bty="l")
```

All of the R graphics parameters can be used in the usual R convention. The `type="s"` argument indicates that a "step" plot should be plotted. In this particular plot, custom labels are placed on the horizontal and vertical axes using the `xlab` and `ylab` arguments. The tick mark labels on the vertical axis are rotated 90° with the `las` argument, while the upper and right-hand axes are suppressed with the `bty` argument.

The resulting plot given in Figure 5 shows that the number in queue peaks at seven customers in line. Interestingly, the plot highlights two early periods of lengthy inactivity in the system, corresponding to two relatively large interarrival times between jobs. After viewing the plot, these interarrival times are easy to spot in the `output$numberInQueueT` results above.

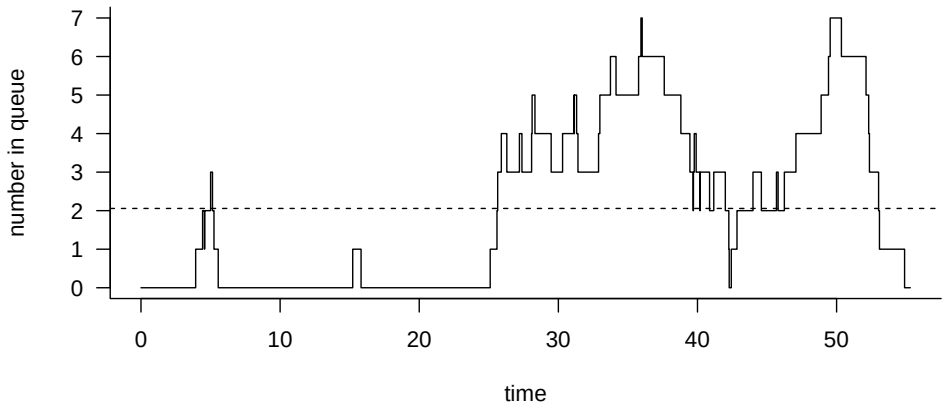


Figure 5: Number in queue for the first 50 customers, with time-averaged statistic superimposed.

The time-averaged number in the queue, which is the area under the curve in Figure 5 divided by the total simulation time, is easy to compute using R. The `meanTPS` function given below is another example of how students may supplement R by using their own code to assist in analysis and understanding. Within this function, the last three statements use R's vectorized programming capabilities to easily compute the normalized area under the curve, which is then returned. (An iterative approach could be used instead.)

```
meanTPS <- function(times, numbers) {
  obsPeriod     = times[length(times)] - times[1]
  timeIntervals = diff(times)
  areas         = c(timeIntervals, 0) * numbers
  return(sum(areas) / obsPeriod)
}
```

As shown below, our `meanTPS` function can be used to compute and display the time-averaged number in the queue by providing as arguments the times and numbers given by `ssq`. This value can then be superimposed onto the time plot as a dashed horizontal line using R's `abline` function, as depicted in Figure 5. Such visualization is extremely useful in helping students understand the meaning of time-persistent statistics.

```
> meanTPS(output$numberInQueueT, output$numberInQueueN)
[1] 2.05734
> abline(h=2.05734, lty="dashed")
```

As a final example, Little's formula (Little 1961), which relates certain observation-based statistics to time-persistent statistics, can also be verified using R. One form of Little's formula states that the sum of the wait times must equal the area under the function in Figure 5. We use `ssq` below to generate 50 customer arrivals to an M/M/1 queue, and then use R functions to demonstrate that the sum of the 50 wait times equals the area under that function.

```
> output = ssq(seed=8675309, maxArrivals=50, saveAllData=T, showOutput=F)
> sum(output$waitTimes)
[1] 113.767
> timeIntervals = diff(output$numberInQueueT)
> sum(c(timeIntervals,0) * output$numberInQueueN) # sum rectangle areas
[1] 113.767
```

8 CONCLUSIONS

The R language has received scant attention in the discrete-event simulation community. The goal of this paper was to introduce R in the context of a single-server queue simulation model, and to show how R can be used to easily conduct custom post-processing analysis. Despite weaknesses in animation and in execution speed due to being an interpreted language, R has many other strengths that make it an excellent computing platform for a variety of approaches to a first course in discrete-event simulation.

REFERENCES

Kaczynski, W. H., L. M. Leemis, and J. H. Drew. 2012. "Transient Queueing Analysis". *INFORMS Journal on Computing* 24 (1): 10–28.

Law, A. M. 2015. *Simulation Modeling and Analysis*. 5th ed. New York: McGraw-Hill.

Little, J. 1961. "A Simple Proof of $L = \lambda W$". *Operations Research* 9 (3): 383–387.

AUTHOR BIOGRAPHIES

BARRY LAWSON is Associate Professor of Computer Science at the University of Richmond. He received Ph.D. and M.S. degrees in Computer Science from The College of William & Mary, and his B.S. in Mathematics from UVA's College at Wise. His research interests are in agent-based simulation, with biological applications. He is a member of ACM, IEEE, and INFORMS. His email address is blawson@richmond.edu.

LAWRENCE M. LEEMIS is Professor in the Department of Mathematics at The College of William & Mary. He received B.S. and M.S. degrees in Mathematics and a Ph.D. in Industrial Engineering from Purdue University. His interests are in reliability and simulation, and has extensive consulting and research contract experience. He is a member of ASA and INFORMS. His email address is leemis@math.wm.edu.