Proceedings of the 2021 Winter Simulation Conference
S. Kim, B. Feng, K. Smith, S. Masoud, Z. Zheng, C. Szabo, and M. Loper, eds.

STRUCTURING A SIMULATION COURSE AROUND THE `simEd` PACKAGE FOR R

Barry Lawson

Digital and Computational Studies
Bates College
Lewiston, ME 04240, USA

Lawrence M. Leemis

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

ABSTRACT

The `simEd` package in R provides a number of functions — including text-based, visualization, and animation functions — that can be useful in teaching a course or a module in discrete-event simulation. This paper suggests topics and several sample exercises for such a course, demonstrating how the `simEd` package facilitates delivery of those topics.

1 INTRODUCTION

The first discrete-event simulation course that students take is typically one of two types. The first type is a simulation *modeling* course in which the student learns the syntax of a high-level simulation language and models complex problems using the language. This type of course often involves a term project where the student models a real-world system using the simulation language. The second type is a simulation *analysis* course involving fundamental concepts in simulation modeling such as random number and random variate generation, input modeling, output analysis, etc. This paper is aimed for instructors (rather than students) of the second type of course, and we heartily encourage use of the examples and code herein.

Most introductory simulation courses are taught in computer science (CS) departments, IE/OR departments, and business schools. The topics and emphasis chosen in these three settings vary. Simulation courses in a CS department emphasize algorithms, data structures, software design, etc. Simulation courses in an IE/OR department emphasize the statistical theory that lies behind simulation modeling. Simulation courses in a business school emphasize methodology and applications. There is considerable overlap between topics in all three of these settings, and the examples investigated here are appropriate for all of these settings.

In a simulation analysis course, the instructor would like to avoid spending too much class time having the students learn the nuances of a simulation modeling language. It is beneficial to get to simulating as quickly as possible so as to devote as much time as possible to covering the long list of simulation topics. In order to minimize the time to learn language syntax, we have developed the `simEd` package in R which can be used to allow students to simulate with minimal overhead in terms of learning the syntax associated with a simulation package. The suite of functions in the `simEd` package allows a user to run elementary Monte Carlo and discrete-event simulations with a single call to a function, and then use built-in R graphical and statistical tools for analysis.

2 CONTEXT AND RELATED WORK

This is the fourth in a series of papers concerning the use of R for teaching simulation, and as this paper focuses on structuring a course around the `simEd` package rather than introducing any new software, the interested reader is referred to the other three for more details about related works, approaches, and tools. One of the key functions offered is the `ssq` function (Lawson and Leemis 2015), which simulates a single server queue. Subsequently introduced, the `simEd` package for R (Lawson and Leemis 2017) includes

978-1-6654-3311-2/21/$31.00 ©2021 IEEE

select Monte Carlo simulation functions, data sets, and various random variate generation functions that mirror those in the base R language. These random variate generation functions differ from those in the base R language in two important respects. First, they include an optional visualization capability, which can be useful for classroom presentations. Second, these variate generation functions support random number streams and antithetic variates, which can be useful in implementing certain variance reduction techniques. Finally, Kudlay, et al. (2020) introduced a number of animation and visualization functions in `simEd` which are helpful for developing intuition in classroom demonstrations and homework exercises. These include animations for a Lehmer random number generator, for acceptance–rejection, for generating a non-homogeneous Poisson process via thinning, and for event-driven details of a single-server queue model. This paper does not introduce new functionality in `simEd`, but instead suggests a potential structure for an introductory simulation course, suggesting topics and exercises that leverage functionality from `simEd`.

3 LOADING THE `simEd` PACKAGE

To install R, visit https://www.r-project.org, select the desired platform (Linux, Mac, or Windows), and follow the prompts to download R. All of the code given here works in native R or in the popular IDE RStudio. Once R is installed, initiate an R session (via its startup icon, or by typing `R` in a terminal window), and then install and load the `simEd` package in R with the two commands below. The first command is a one-time cost; the second must be executed with each new R session.

```
install.packages("simEd")
library(simEd)
```

The remainder of this paper has section headings that describe topics that are typically covered in an introductory simulation course. These sections will illustrate how the functions in the `simEd` package can be used as classroom demonstrations or homework exercises to help convey understanding of the topic.

4 MONTE CARLO SIMULATION

We believe that a week spent early in a simulation analysis course on Monte Carlo simulation will pay dividends later in the course. Monte Carlo simulation experiments bring up all of the sampling and convergence issues that arise in discrete-event simulation, but without all of the moving parts that are present in a discrete-event simulation. This allows the students to absorb the random sampling variability and convergence concepts without having to simultaneously absorb event calendars, autocorrelation, etc.

Example: Via Monte Carlo simulation, estimate the probability of rolling a sum of ten using three fair six-sided dice. Although this example can be easily coded using base R functions, `simEd` has a function named `galileo` which can also be used to run the simulation. This is helpful for students who might not be familiar with R at the beginning of the course. The statements below use the `set.seed` function to establish a random number seed and the `galileo` function to run a Monte Carlo simulation with 10,000 replications.

```
library(simEd)
set.seed(3)
galileo(10000)
```

Letting X be the number of pips showing on the up faces of the three dice, these R statements return $\hat{P}(X = 10) = 0.1254$. This provides a good opportunity to highlight why just one run of a simulation is not a good idea in practice. Five successive runs of this simulation without resetting the seed yield the following estimates of $P(X = 10)$.

0.1254 0.1234 0.1213 0.1245 0.1204.

These results should be compared with the analytic result $P(X = 10) = 27/216 = 1/8 = 0.125$, and it should be noted that four values fall below the analytic result and one falls above. This is also an appropriate time to let students know that Monte Carlo simulation experiments *support*, but do not *confirm* analytic solutions. A good follow-on exercise to this one is to assess the impact of the number of simulation replications on the precision of the estimates. Alternatively or subsequently, the simple game of craps can be introduced as a problem using Monte Carlo simulation, via the `craps` function in the `simEd` package.

5 SINGLE SERVER QUEUE

A reasonable first discrete-event simulation model for a student to encounter in a simulation analysis course is the model of a single-server queue, implemented in the `ssq` function. More specifically, the M/M/1 queue is appealing in that students might have encountered it in a course on stochastic processes. An instructor's dilemma involves whether to treat `ssq` simply as a function to be executed, or to give some notion of the algorithm being implemented. We are strong proponents of the latter approach, and give an exercise below which encourages students to think about the mechanics behind a discrete-event simulation. For those inclined to the former approach, an exploration of the effect of sample size for the default M/M/1, e.g., `ssq(`n`, seed = 5551212)` with $n = 10, 100, 1000$, etc., serves as a very easy-to-use starter example.

Example: Investigate the event-driven algorithm used in the `ssq` function. The `ssq` function consists of hundreds of lines of R code, but much of the code involves error checking, collecting statistics, dynamic storage allocation, etc. Some key statements from the "engine" associated with the arrival and the departure of customers from the system (without collecting statistics) are given by the R code below.

```
library(simEd)
t.current   <- 0.0
numInSystem <- 0
maxTime     <- 1000

arrivalsCal <- vexp(1, 1, stream = 1)
serverCal   <- Inf

while (t.current < maxTime) {
   t.current <- min(arrivalsCal, serverCal)
   if (t.current == arrivalsCal) {
      numInSystem  <- numInSystem + 1
      arrivalsCal  <- t.current + vexp(1, 1, stream = 1)
      if (numInSystem == 1)
         serverCal <- t.current + vexp(1, 10 / 9, stream = 2)
      print(paste("Arrive @", t.current, "numInSystem =", numInSystem))
   }
   else
   {
      numInSystem  <- numInSystem - 1
      if (numInSystem > 0)
         serverCal <- t.current + vexp(1, 10 / 9, stream = 2)
      else
         serverCal <- Inf
      print(paste("Depart @", t.current, "numInSystem =", numInSystem))
   }
}
```

This code simulates a single-server queue for 1000 time units. The interarrival time is assumed to be exponentially distributed with a mean of 1 time unit. The service time is also assumed to be exponentially

distributed, but the mean service time is $9/10$ of a time unit. So this is an example of an $M/M/1$ queue, with a traffic intensity $\rho = \lambda/\mu = 1/(10/9) = 9/10$. The server takes no breaks, never interrupts service while working on a customer, and begins service on a waiting customer immediately upon completion of service for the previous customer. Students should type in this code and then run the program. Students should then answer the following questions concerning the implementation.

1. What variable holds the *simulation clock*?
2. What variables comprise the *calendar* (or *event list*)?
3. What are the two events in the simulation?
4. How many customers are in the system at time zero?
5. What is the status (busy or idle) of the server at time zero?
6. Are arrivals scheduled all at once, or are they scheduled on the fly as the simulation progresses?
7. What is the purpose of the statement `t.current <- min(arrivalsCal, serverCal)`?
8. What is the purpose of `arrivalsCal <- t.current + vexp(1, 1, stream = 1)`?
9. What is the purpose of `serverCal <- t.current + vexp(1, 10 / 9, stream = 2)`?

These questions are designed to guide the students to an understanding of what happens behind the scenes in the implementation of a discrete-event simulation model. An instructor could also ask the student to modify this code, for example, to calculate statistics on a particular measure of performance, such as the average sojourn time. The use of `vexp`, rather than R's native `rexp`, also facilitates discussion of streams for different stochastic components.

6 TIME PERSISTENT STATISTICS

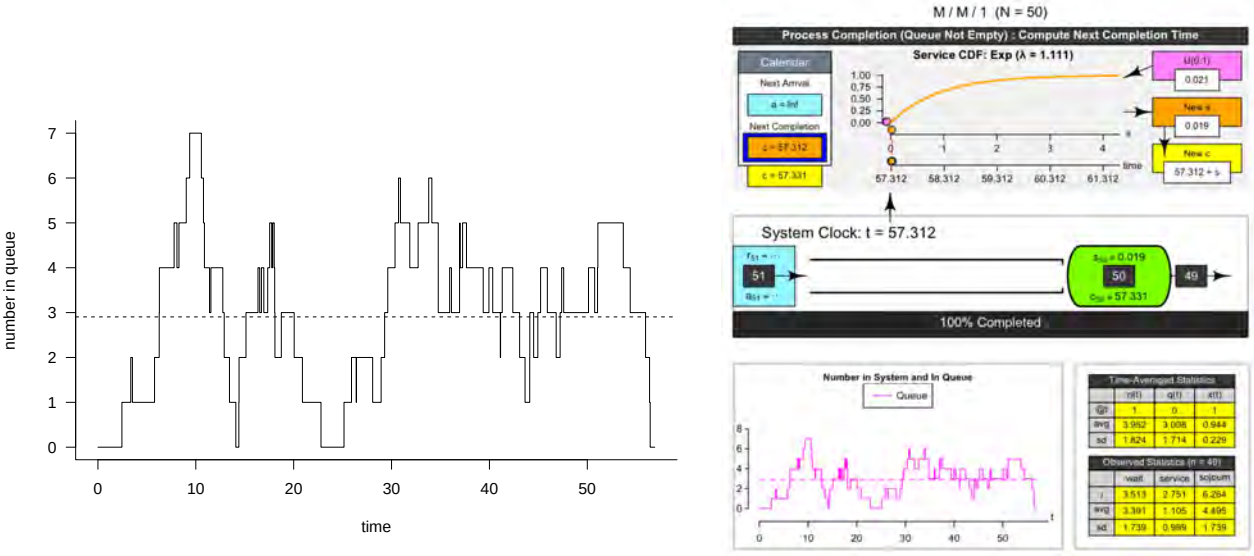
An important concept that routinely appears in discrete-event simulation is that of *time-persistent statistics*. Most students have not seen the concept of a time-persistent statistic in their introductory statistics course, so this tends to be new material that they are seeing for the first time. Because of this, a graphical presentation is helpful for understanding and developing intuition. The `ssq` function can be used to visualize and analyze time-persistent statistics.

Example: Run the `ssq` function to capture the number in queue for the first 50 customers. Make a plot of the number in queue over time and add a horizontal line at the sample mean. In this case, the `ssq` function is called with the `saveNumInQueue` argument set to `TRUE` so as to save two vectors, `output$numInQueueT`, with the 101 time values when changes occurred to the number in the queue, and the corresponding vector `output$numberInQueueN` contains the associated number of customers in the queue at each change. A visualization of the number in queue over time is generated using the R function `plot`, with the `type = "s"` argument indicating that a "step" function plot should be graphed. The `simEd` function `meanTPS` is used to calculate the sample mean. The R function `abline` is used to plot the sample mean as a dashed horizontal line.

```
library(simEd)
output <- ssq(maxArrivals = 50, seed = 8675309, saveNumInQueue = TRUE)
plot(output$numInQueueT, output$numInQueueN, type = "s",
     xlab = "time", ylab = "number in queue", las = 1, bty = "l")
xbar <- meanTPS(output$numInQueueT, output$numInQueueN)
abline(h = xbar, lty = "dashed")
```

The resulting "skyline" graph is shown in Figure 1(a). This graph provides students with the ability to translate their notions of central tendency and dispersion from the realm of statistics based on observations to time-persistent statistics.

The animation function `ssqvis` can then be used to visualize step-by-step details of the "engine" for this particular simulation, reinforcing the ideas presented in Section 5, which also includes the step-by-step generation of that same skyline function across time. This is depicted in Figure 1(b).



(a) (b)

Figure 1: (a) "Skyline" graph of number in queue versus time for the first 50 customers. (b) Animation from `ssqvis` of the same 50-customer simulation animating step-by-step simulation engine details.

.

7 CONVERGENCE

Once the `ssq` and `ssqvis` functions have been introduced and the students have some sense of how those work, it is valuable for them to see that the same notions of convergence that were applied to Monte Carlo simulation also apply to discrete-event simulation.

Example: Investigate the behavior of the average sojourn time in an M/M/1 queue for varying number of customers. The sojourn time is the wait time plus the service time. There will be 20 total calls to the `ssq` function (5 per each value for number of customers) in this simulation experiment. In the code below, the `ssq` function is called in two nested `for` loops. The outer loop runs over the number of customer arrivals: 100, 1000, 10000, and 100000. The inner loop runs over five replications of the simulation for a particular fixed number of customers established by the outer loop index. In each call to `ssq`, the average sojourn time is computed. A plot of the number of customers (on the horizontal axis with a logarithmic scale) versus the average sojourn time (on the vertical axis) is made for each of the 20 simulation runs. In addition, a horizontal line is superimposed on the plot that depicts the steady-state average sojourn time, which is

$$\frac{1}{\mu-\lambda}=\frac{1}{10/9-1}=\frac{1}{1/9}=9$$

for a traffic intensity satisfying $\rho=\lambda/\mu<1$ (Hillier and Lieberman, 2001).

```
library(simEd)
set.seed(3)
ncust <- c(100, 1000, 10000, 100000)  # varying number of customers

plot(NULL, axes = FALSE, log = "x",
     xlab = "number of customers", ylab = "average sojourn time",
     xlim = c(100, 100000), ylim = c(0, 11))
axis(side = 1, at = ncust)
axis(side = 2, at = 0:11, las = 1)
```

```
for (n in ncust) {
   for (i in 1:5) {
      result <- ssq(maxArrivals = n, showOutput = FALSE)$avgSojourn
      points(n, result)  # add computed avg sojourn to existing plot
   }
}
abline(h = 9)  # steady-state average sojourn time
```

The resulting graphic is shown in Figure 2. There are three conclusions that can be drawn from the graph drawn by this code. First, there is some initialization bias that is apparent for smaller numbers of customers based on the fact that the system begins in an empty and idle state. Second, the average sojourn times seem to be converging to the steady-state value of 9 time units as the number of customers increases. Third, the variability of the average sojourn times seems to decrease as the number of customers increases.

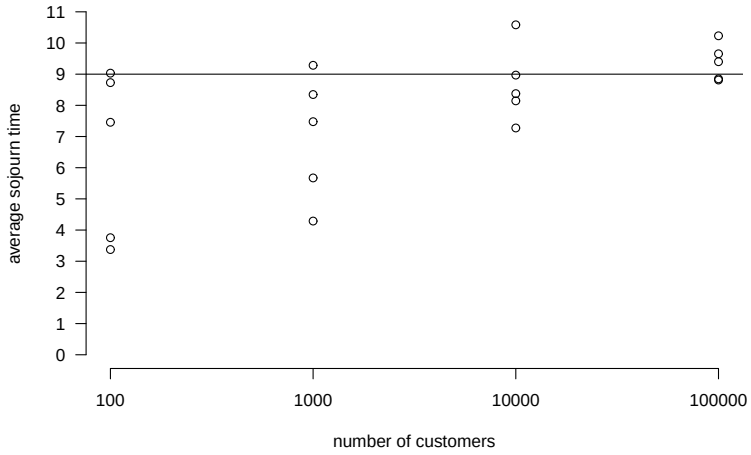


Figure 2: Average sojourn time as a function of number of customers.

8 AUTOCORRELATION

Most students who have taken an introductory statistics course are used to seeing data values drawn from a random sample, yielding independent and identically distributed observations. Autocorrelation will be a new concept to most of these students. The presence of autocorrelation in simulation output, as well as motivation for addressing autocorrelation, can easily be illustrated.

Motivating Example: Use `ssq` with default arguments and the base R function `t.test` to generate 100 different 95% confidence intervals, each using the service times for $n = 200$ customers, discarding the first 1000 times to account for the initial transient. Then repeat, but using sojourn times instead.

The R code below implements the motivating example using service times. Changes required to use sojourn times are noted in red comments.

```
library(simEd)
num_int <- 100
warmup  <- 1000
ci      <- list(lo = rep(NA, num_int), hi = rep(NA, num_int))

for (i in 1:num_int) {
   seed  <- if (i == 1) 8675309 else NA
```

```
   out   <- ssq(200 + warmup, seed, showOutput = FALSE, saveAllStats = TRUE)
   times <- out$serviceTimes[-(1:warmup)]  # change: out$sojournTimes
   interval <- t.test(times, conf.level = 0.95)
   ci$lo[i] <- interval$conf.int[1];   ci$hi[i] <- interval$conf.int[2]
}

# initially-empty plot using CI limits;  then overlay colored segments
plot(NA, NA, xlim = c(1, num_int), ylim = c(min(ci$lo), max(ci$hi)),
     las = 1, bty = "n", xaxt = "n",
     xlab = "", ylab = "Avg Service") # change: "Avg Sojourn"

theo <- 0.9  # for M/M/1 with TI (1.0 / 0.9): service = 0.9, sojourn = 9.0
abline(h = theo, lty = "dashed")
for (i in 1:num_int) {
   color <- if (ci$lo[i] <= theo && theo <= ci$hi[i]) "black" else "red"
   segments(i, ci$lo[i], i, ci$hi[i], col = color)
}
```

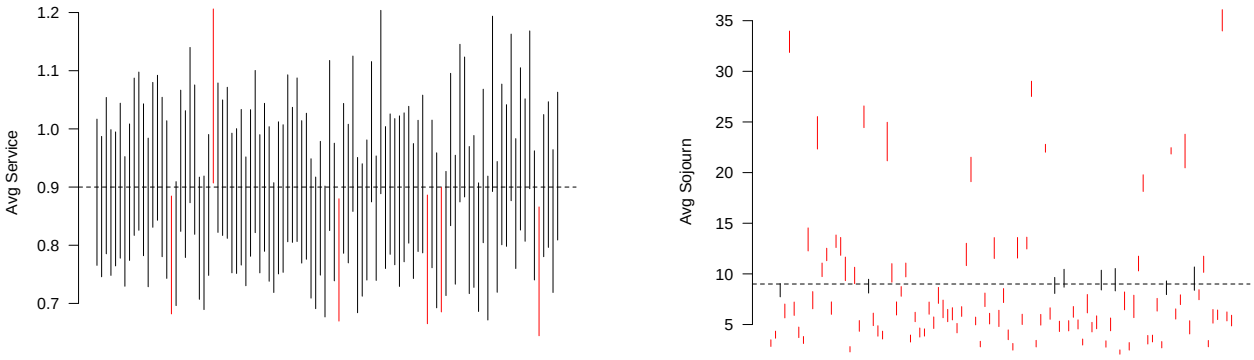


Figure 3: 100 95% confidence intervals using `t.test` naïvely on $n = 200$ service times (left) versus sojourn times (right) from `ssq`. Red intervals do not contain the true theoretical values (dashed lines).

The rightmost plot demonstrates that a naïve approach may produce intervals that clearly do not provide the claimed coverage, suggesting further exploration — first of autocorrelation, and then batch means.

Example: Run `ssq` with default arguments again, using only the initial seed that produces the first confidence interval in Figure 3. Again discard the first 1000 sojourn times to minimize the effect of the initial transient, and plot the remaining 200 sojourn times, the sample autocorrelation function, and the sample partial autocorrelation function.

```
library(simEd)
warmup   <- 1000
output   <- ssq(maxArrivals = 200 + warmup, seed = 8675309,
                saveAllStats = TRUE)
sojourns <- output$sojournTimes[-(1:warmup)]

layout(matrix(c(1, 1, 2, 3), 2, 2, byrow = TRUE))
plot(sojourns, type = "l")               # line plot the sojourn times
points(sojourns, pch = 16, cex = 0.6)    # overlay points onto line plot
acf(sojourns)                            # plot acf in lower left
pacf(sojourns)                           # plot pacf in lower right
```

The R function `layout` is used in the code above to allow the one plot of the sojourn times to span the two plots of the sample autocorrelation and the sample partial autocorrelation functions. The `plot` and `points` functions are used for the graph of the sojourn times as a time series, and the `acf` and `pacf` functions plot the sample autocorrelation and sample partial autocorrelation functions.

The graphs are shown in Figure 4. The sojourn times exhibit significant autocorrelation at low lags, as reflected in the plot of the sample autocorrelation function. The sample partial autocorrelation function has only one significant spike. More information on time series modeling is in Box and Jenkins (1970).

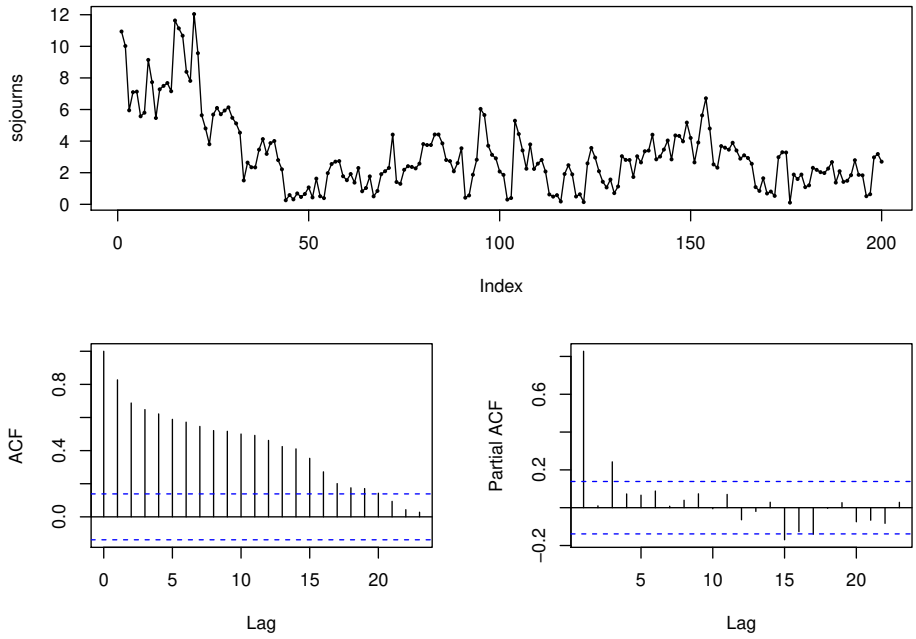


Figure 4: Sojourn times and autocorrelation and partial autocorrelation functions.

9 OUTPUT ANALYSIS

Once autocorrelation has been identified as a complicating factor in the analysis of simulation output, the next step is to introduce techniques that are used to minimize its impact. The example below uses the `ssq` function to estimate the actual coverage for two output analysis techniques: classical statistics based on the assumption of mutual independence and batch means.

Example: Simulate an M/M/1 queue using the `ssq` function using the default arrival rate and service rate. Store the first $1000 + 2^{16}$ customer sojourn times. The goal is to calculate point and interval estimates at the steady-state mean sojourn time and evaluate the actual coverage associated with confidence intervals generated by several output analysis techniques. Delete the first 1000 observations to account for the initial transient, which leaves 2^{16} observations. The point estimate for the steady-state population mean sojourn time is the sample mean of the 2^{16} sojourn times. Use the following techniques to construct a 95% confidence interval for the steady-state population mean:

1. classical statistics (that is, a traditional confidence interval based on the t distribution),
2. batch means with batches of size 2, 4, 8, ..., 2^{15}.

Estimate the actual coverage of the various confidence interval procedures.

The R code below implements the confidence interval using classical statistics via the `t.test` function and the batch means procedure via the `rowMeans` and `t.test` functions.

```
library(simEd)

nrep  <- 100
ncust <- 1000 + 2 ^ 16
theo  <- 9  # theoretical steady-state average sojourn
count <- numeric(16)

set.seed(3)
for (i in 1:nrep) {
   output   <- ssq(maxArrivals = ncust, showOutput = FALSE,
                   showProgress = FALSE, saveSojournTimes = TRUE)
   sojourns <- output$sojournTimes[1001:ncust]
   #  classical
   ci <- t.test(sojourns)$conf.int;   lo <- ci[1];   hi <- ci[2]
   if (lo < theo && theo < hi)  count[1] <- count[1] + 1
   #  batch means
   for (j in 1:15) {
      batchsize <- 2 ^ j
      batchmeans <- rowMeans(
                       matrix(sojourns, ncol = batchsize, byrow = TRUE) )
      ci <- t.test(batchmeans)$conf.int;   lo <- ci[1];   hi <- ci[2]
      if (lo < theo && theo < hi)  count[j + 1] <- count[j + 1] + 1
   }
}
print(count / nrep)
```

The estimated actual coverage using classical statistics is 0.07, which falls way short of the nominal 0.95. The positive autocorrelation between customer sojourn times has resulted in confidence intervals which are way too narrow. The results for the various batch sizes are given in Table 1. Entries set in boldface do not differ significantly from $1-\alpha = 0.95$. This exercise allows students to see the effects of autocorrelation on simulation output and techniques for overcoming these effects.

Table 1: Estimated actual coverage of a 95% CI using batch means (batch size over corresponding coverage).

2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16,384	32,768
0.09	0.15	0.24	0.28	0.42	0.56	0.73	0.82	0.89	**0.93**	**0.94**	**0.94**	**0.94**	**0.94**	**0.95**

10 UNCERTAINTY QUANTIFICATION

Output analysis techniques, such as batch means, assume that all of the parameters in the simulation model are known with certainty. This is seldom the case in practice. Uncertainty quantification takes into account the fact that the parameters in the simulation are estimated from data. The sample question below considers an M/M/1 queue in which the arrival rate and service rate are estimated from data.

Example: Call the `ssq` queueing simulation with default parameters to output the average of the first 25 sojourn times for 1000 simulation replications. Draw a histogram of the mean sojourn times. Next, re-run these simulations so that the mean interarrival time for each replication is the average of $n = 3$ unit exponential random variates, and the mean service time for each replication is the average of $m = 5$ exponential random variates with mean $9/10$. Draw a histogram of these mean sojourn times.

The R code below runs the simulations for the two different experiments, each of which results in a histogram of the 1000 average sojourn times.

```
library(simEd)

ncust <- 25
nrep  <- 1000
xbar  <- numeric(nrep)

set.seed(3)
for (i in 1:nrep) {
   sojourns <- ssq(maxArrivals  = ncust,
                   showOutput   = FALSE,
                   showProgress = FALSE,
                   saveSojournTimes = TRUE)$sojournTimes
   xbar[i] <- mean(sojourns)
}

par(mfrow = c(1, 2), las = 1)
hist(xbar)

set.seed(3)
for (i in 1:nrep) {
   lambda       <- 1 / mean(vexp(3, 1, stream = 1))
   mu           <- 1 / mean(vexp(5, 10 / 9, stream = 2))
   interarrival <- function() vexp(1, lambda, stream = 1)
   service      <- function() vexp(1, mu, stream = 2)
   sojourns     <- ssq(maxArrivals  = ncust,
                       showOutput   = FALSE,
                       showProgress = FALSE,
                       interarrivalFcn  = interarrival,
                       serviceFcn       = service,
                       saveSojournTimes = TRUE)$sojournTimes
   xbar[i]      <- mean(sojourns)
}

hist(xbar)
```

The two histograms are displayed in Figure 5. The second histogram has more variability to the average sojourn times than the first (note the difference in horizontal plot ranges). This is because some of the simulations generate a large arrival rate and a small service rate, which results in a traffic intensity that exceeds 1. In this particular setting, there will be longer sojourn times, and hence the longer right-hand tail in the second histogram. The sample mean and variance of the 1000 values are 3.0 and 3.5 in the first simulation experiment and 4.4 and 20.9 in the second simulation experiment, respectively.

11 METAMODELING

Metamodels are designed to mimic the input–output behavior of a discrete-event simulation model (Barton, 2020). They are essentially models of models. The example given next uses a simple linear regression model as a tentative, simple, yet inappropriate model to describe the relationship (that is, the metamodel) between the mean service time and the average sojourn time. Because they smooth out random sampling variability, metamodels can be helpful to students working on a simulation optimization problem.

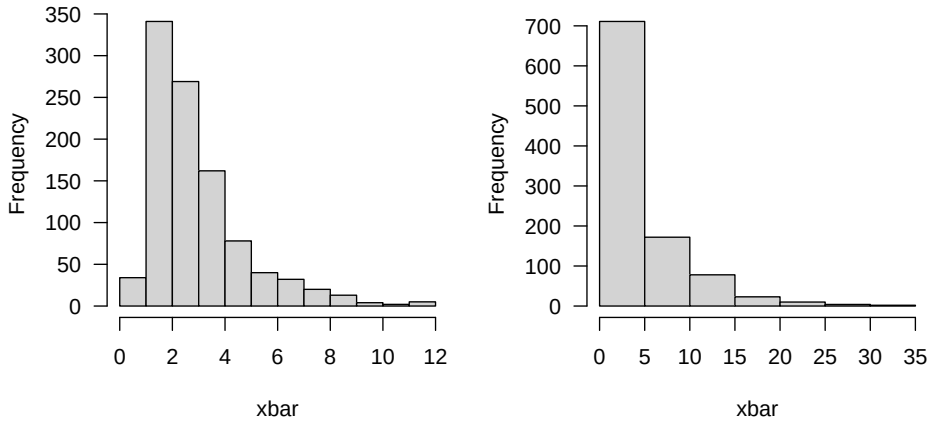


Figure 5: Average sojourn times for fixed versus random parameters.

Example: Fit a simple linear regression metamodel for mean service time (the independent variable) versus the mean sojourn time (the dependent variable) using three replications at each of the mean service times 0.70, 0.75, 0.80, 0.85, 0.90, and 0.95 for an M/M/1 queuing model with unit arrival rate. Warm up the simulation for 1000 customers and collect sojourn times for the next 1000 customers in each replication. Draw a graphic that shows the simulation results and the metamodel.

The code below stores the mean service times in the vector `mst`. The `service` function generates service times with the appropriate probability distribution using the `vexp` function. The first 2000 service times are stored in the vector `sojourns`, and the first 1000 sojourn times are deleted so that the steady-state values are used in the metamodel. The `lm` function fits a simple linear regression model to the results.

```
library(simEd)

set.seed(3)
mst  <- c(0.7, 0.75, 0.8, 0.85, 0.9, 0.95)
nrep <- 3
x <- NULL;   y <- NULL
for (i in mst) {
   for (j in 1:nrep) {
      service  <- function() vexp(1, rate = 1 / i, stream = 2)
      sojourns <- ssq(maxArrivals  = 2000,
                      serviceFcn   = service,
                      showOutput   = FALSE,
                      showProgress = FALSE,
                      saveSojournTimes = TRUE)$sojournTimes
      sojourns <- sojourns[1001:2000]
      x <- c(x, i)
      y <- c(y, mean(sojourns))
   }
}
plot(x, y)
fit <- lm(y ~ x)
abline(fit$coefficients)
```

The metamodel is shown in Figure 6. It is clear that both the linearity and homogeneity of variances are violated in this case. A plot of the residuals does not appear to be mutually independent normally-distributed errors. This example encourages the students to ask why this is the case and suggest alternative metamodels.

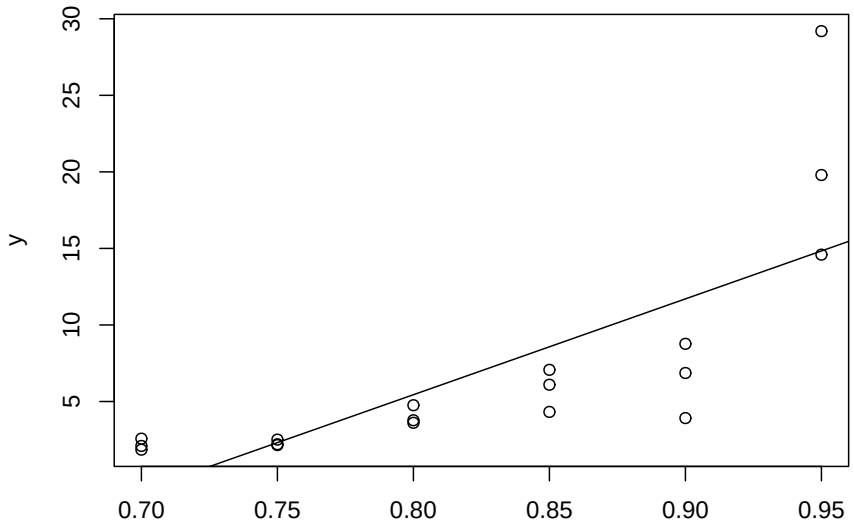


Figure 6: Simple yet inappropriate metamodel for the sojourn times.

12 CONCLUSIONS

The `simEd` package can be useful for simulation analysis courses and modules in order to minimize the amount of time spent on syntax. The `simEd` package is not a general-purpose simulation language and is not intended for use in that context. This paper has given several examples to illustrate how the functions in the `simEd` package can be used to introduce and explore several common topics that occur routinely in discrete-event simulation courses.

REFERENCES

Barton, R. R. 2020. "Metamodeling for Simulation". In *Proceedings of the 2020 Winter Simulation Conference*, edited by K.-H. Bae, B. Feng, S. Kim, S. Lazarova-Molnar, Z. Zheng, T. Roeder, and R. Thiesing, 1102-1116. Piscataway, NJ: Institute of Electrical and Electronics Engineers, Inc.

Box, G., and G. Jenkins. 1970. *Time Series Analysis: Forecasting and Control*. Holden-Day, San Francisco.

Hillier, F. S., and G. J. Lieberman. 2001. *Introduction to Operations Research*. 7th ed. New York: McGraw-Hill.

Kudlay, V., and B. Lawson and L. M. Leemis. 2020. "Animation for Simulation Education in R". In *Proceedings of the 2020 Winter Simulation Conference*, edited by K.-H. Bae, B. Feng, S. Kim, S. Lazarova-Molnar, Z. Zheng, T. Roeder, and R. Thiesing, 3260-3271. Piscataway, NJ: Institute of Electrical and Electronics Engineers, Inc.

Lawson, B., and L. M. Leemis. 2015. "Discrete-Event Simulation Using R". In *Proceedings of the 2015 Winter Simulation Conference*, edited by L. Yilmaz, V. W. K. Chan, I. Moon, T. M. K. Roeder, C. Macal, and M. D. Rosetti, 3502-3513. Piscataway, NJ: Institute of Electrical and Electronics Engineers, Inc.

Lawson, B., and L. M. Leemis. 2017. "An R Package for Simulation Education". In *Proceedings of the 2017 Winter Simulation Conference*, edited by V. W. K. Chan, A. D'Ambrogio, G. Zacharewicz, N. Mustafee, G. Wainer, and E. Page, 4175-4186. Piscataway, NJ: Institute of Electrical and Electronics Engineers, Inc.

Lehmer, D.H. 1951. "Mathematical Methods in Large-Scale Computing Units," In *Proceedings of the 2nd Symposium on Large-Scale Calculating Machinery*, Harvard University Press, Vol. 26, 141-146.

AUTHOR BIOGRAPHIES

BARRY LAWSON is the Colony Family Professor of Digital and Computational Studies at Bates College. He received Ph.D. and M.S. degrees in Computer Science from 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. His email address is blawson@bates.edu.

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