Proceedings of the 2017 Winter Simulation Conference
W. K. V. Chan, A. D'Ambrogio, G. Zacharewicz, N. Mustafee, G. Wainer, and E. Page, eds.

AN R PACKAGE FOR SIMULATION EDUCATION

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 free software for statistical computing, providing a variety of statistical and graphical functionality. For use in simulation education, R's capabilities help to develop student intuition. In this paper, we introduce the `simEd` package for R, written with a pedagogical focus. The package includes functions for generating discrete and continuous variates via inversion, with capabilities for independent streams and antithetic variates; for visualizing inversion in variate generation and the relationship to the pdf/pmf, cdf, and ecdf; for computing time-persistent statistics; for extensible single- and multiple-server queueing simulation; and includes data sets for input modeling and analysis. As we demonstrate using several illustrations, this package, along with native R functionality, provides a compelling case for using R in an introductory simulation course.

1 INTRODUCTION

R should be considered for a first course in discrete-event simulation that emphasizes programming in a high-level language as well as the probabilistic and statistical aspects of simulation. R includes traditional imperative programming language constructs, powerful statistical analysis functions and packages, and customizable high-level graphics capability. To date, a few R packages related to discrete-event simulation have appeared, focusing on simulation practice. As examples, consider `poisson`, which simulates homogeneous and non-homogeneous Poisson processes (Brock and Slade 2015); `simmer`, a process-oriented discrete-event simulation package designed to be a generic simulation framework (Ucar and Smeets 2017); and `rrepast`, which allows one to invoke Repast Simphony agent-based models directly from R (Garcia and Rodriguez–Paton 2016). None, however, have focused on simulation pedagogy.

In this paper, we introduce the `simEd` package for R, written with a focus on simulation pedagogy. Our package facilitates simulation education using R by introducing reorganized analogs of existing R functions as well as new simulation-related functions that do not exist in native R. The package includes:

- variate generation functions for two discrete and five continuous distributions (see Table 1);
- functions to visualize inversion for variate generation, including visual representations of inversion vs. the cumulative distribution function (cdf), of the histogram vs. the probability density/mass function (pdf/pmf), and of the empirical cdf (ecdf) vs. the cdf (see Table 2);
- functions implementing single- and multiple-server queueing models (see Table 3); and
- various utilities, including functions for analyzing time-persistent statistics and for sampling, as well as data sets for queueing input modeling (see Table 4).

The `simEd` package (Lawson and Leemis 2017) is available on the Comprehensive R Archive Network (CRAN) (The R Foundation 2017). The package can be installed (a one-time cost) using the R command `install.packages("simEd")`, and attached in each R session using `library(simEd)`.

Table 1: Variate generation functions (two discrete and five continuous distributions).

Distribution	Function Signature
Binomial	`vbinom(n, size, prob, stream = NULL, antithetic = FALSE)`
Geometric	`vgeom(n, prob, stream = NULL, antithetic = FALSE)`
Exponential	`vexp(n, rate = 1, stream = NULL, antithetic = FALSE)`
Gamma	`vgamma(n, shape, rate = 1, scale = 1/rate, stream = NULL, antithetic = FALSE)`
Normal	`vnorm(n, mean = 0, sd = 1, stream = NULL, antithetic = FALSE)`
Uniform	`vunif(n, min = 0, max = 1, stream = NULL, antithetic = FALSE)`
Weibull	`vweibull(n, shape, scale = 1, stream = NULL, antithetic = FALSE)`

Table 2: Functions for visualizing inversion of variate generation. (The last 10 parameters are consistent in all functions, indicated by ellipses.) A null value for `u` displays distribution plots only, without inversion.

Distribution	Function Signature

Binomial	`ibinom(u = runif(1), size, prob,`
	`minPlotQuantile = 0, maxPlotQuantile = 1, plot = TRUE,`
	`showCDF = TRUE, showPMF = FALSE, showECDF = FALSE, show = NULL,`
	`plotDelay = 0, maxPlotTime = 30, resetRowsMargins = TRUE)`

Geometric	`igeom(u = runif(1), prob, ... )`
Exponential	`iexp(u = runif(1), rate = 1, ... )`
Gamma	`igamma(u = runif(1), shape, rate = 1, scale = 1/rate, ... )`
Normal	`inorm(u = runif(1), mean = 0, sd = 1, ... )`
Uniform	`iunif(u = runif(1), min = 0, max = 1, ... )`
Weibull	`iweibull(u = runif(1), shape, scale = 1, ... )`

Table 3: Queueing simulation functions. (The last 14 parameters are identical, indicated by an ellipsis.)

Queue Model	Function Signature

Single-Server	`ssq(maxArrivals = Inf, seed = NA,`
	`interarrivalFcn = defaultInterarrival, serviceFcn = defaultService,`
	`maxTime = Inf, maxDepartures = Inf,`
	`saveAllStats = FALSE, saveInterarrivalTimes = FALSE,`
	`saveServiceTimes = FALSE, saveWaitTimes = FALSE, saveSojournTimes = FALSE,`
	`saveNumInQueue = FALSE, saveNumInSystem = FALSE, saveServerStatus = FALSE,`
	`showOutput = TRUE, showProgress = TRUE)`

Multi-Server	`msq(maxArrivals = Inf, seed = NA, numServers = 2,` `serverSelection = c("LRU", "LFU", "CYC", "RAN", "ORD"), ... )`

Table 4: Time-persistent statistics functions and other utilities.

Utility	Signature
Mean	`meanTPS(times, numbers)`
Std. Dev.	`sdTPS(times, numbers)`
Quantiles	`quantileTPS(times, numbers, probs = c(0, 0.25, 0.5, 0.75, 1.0))`
Sampling	`sample(x, size, replace = FALSE, prob = NULL, stream = NULL, antithetic = FALSE)`
Data Sets	`data(queueTrace)`
	`data(tylersGrill)`

To implement independent streams of random numbers for our variate generator functions, we use Josef Leydold's CRAN-available `rstream` package (Leydold 2015). The `rstream` package implements easy-to-use wrappers of Pierre L'Ecuyer's "mrg32k3a" random number generator (L'Ecuyer et al. 2002), and provides a source for independent streams of uniform random numbers. (We note that the "mrg32k3a" generator is available by default in R via `RNGkind` as "L'Ecuyer-CMRG", but only provides the basis for multiple streams as implemented by other packages.)

To ease in transitioning to the `simEd` package, we chose function names and parameter conventions to be as consistent as possible with existing R functions. We also mask the `set.seed` function in the `base` package by providing our own version of `set.seed` that will appear first in the search path (and therefore be the version called) whenever the `simEd` package is attached. The `simEd` version of `set.seed` performs two tasks: first, it explicitly calls `base::set.seed` so that `stats` functions will act as expected; second, it appropriately seeds the `rstream` random number streams used by the `simEd` package. In this way, the user can call `set.seed` as usual, with no need for two different functions to set the initial seed for `stats` and `simEd` functions. (Although not discussed here, we also mask the `sample` function from the `base` package to allow for independent streams and antithetic variates—see Table 4.)

2 VARIATE GENERATION

Our `simEd` package includes variate generation functions for two discrete distributions (binomial and geometric) and five continuous distributions (exponential, gamma, normal, uniform, and Weibull), as outlined in Table 1. We have chosen naming and parameter conventions similar to the variate generation functions available by default in R via the `stats` package, but replacing the leading 'r' in each function name with 'v' (for variate) instead. For example, the `stats` functions for generating variates from two discrete distributions are named `rbinom` and `rgeom`; the corresponding `simEd` functions are named `vbinom` and `vgeom`. Moreover, the `simEd` versions have parameters similar to the `stats` versions, as shown in the R examples below (the left uses `stats` functions, the right uses `simEd` functions).

```
> set.seed(8675309)
> rexp(n = 3, rate = 2)
[1] 0.8311 0.6116 0.3837
> rgamma(n = 3, shape = 2, scale = 1)
[1] 2.099 3.011 1.336
```

```
> set.seed(8675309)
> vexp(n = 3, rate = 2)
[1] 0.08687 0.32522 0.72366
> vgamma(n = 3, shape = 2, scale = 1)
[1] 2.805 1.012 2.317
```

Each of our variate generation functions first generates n $U(0,1)$ variates (more below on how these are generated) and then inverts those variates using the appropriate quantile functions available in `stats` (i.e., `qbinom`, `qgeom`, etc.). Although certain of our variate generator functions may be slow in comparison to the corresponding `stats` generator functions, our functions are both synchronized and monotone, which is important for certain variance reduction techniques. Moreover, the inversion approach used here is entirely consistent with the functions for visualizing inversion for variate generation discussed in Section 3.

The `simEd` variate generation functions provide the capability for independent streams of random numbers via a `stream` parameter. By default, the `stream` parameter has a value of `NULL`, in which case our inversion approach uses the `stats::runif` function to generate $U(0,1)$ variates for inverting. In other words, if the `stream` argument is `NULL`, the left and right examples below are identical in output (but differ from the `rexp` output above left, since `stats` functions do not necessarily use inversion).

```
> set.seed(8675309)
> u <- runif(n = 3)
> qexp(u, rate = 2)
[1] 0.08687 0.32522 0.72366
```

```
> set.seed(8675309)
> vexp(n = 3, rate = 2)
[1] 0.08687 0.32522 0.72366
```

If, however, the `stream` argument is not `NULL`, but rather is an integer in the range $\{1,2,\ldots,25\}$ (where 25 is the maximum number of streams implemented in the `simEd` package), then our functions use the `rstream::rstream_sample` function associated with one of the 25 independent streams to generate

$U(0,1)$ variates for inverting. This capability to use independent streams permits the user to employ appropriate variance reduction techniques (see Section 3). For example, the following R code demonstrates the ability to use streams for isolating stochastic components of a simulation. We begin by defining an exponential interarrival-time function with rate $\lambda = 1$ and an exponential service-time function with rate $\mu = 10/9$, each using separate streams. As shown on the left and right below, calls to the functions can be interleaved without disturbing the order in which variates are generated from each stream.

```
> myArr <- function() { vexp(n = 1, rate = 1,    stream = 1) }
> mySvc <- function() { vexp(n = 1, rate = 10/9, stream = 2) }
```

```
> set.seed(8675309)
> for (i in 1:4)  cat(myArr(), ' ')
0.2335   1.078   0.7856  0.3531
> for (i in 1:4)  cat(mySvc(), ' ')
0.03059  0.1224  1.425   4.551
```

```
> set.seed(8675309)
> for (i in 1:4)
    cat(myArr(), ' ', mySvc(), ' ')
0.2335   0.03059  1.078   0.1224  0.7856 ...
```

Additionally, the `simEd` variate generation functions include an `antithetic` parameter, which provides the capability to produce antithetic variates. The `rstream` package can produce antithetic variates directly, but given that speed is not our primary concern, we compute the antithetic versions of the $U(0,1)$ variates ourselves, whether produced using `stats::runif` or `rstream::rstream_sample`. For example, the following R code demonstrates the use of antithetic $N(0,1)$ variates using the `vnorm` function. We begin by explicitly using `stats::runif` and `stats::qnorm` to demonstrate the production of antithetic variates, and then show that our `vnorm` function produces the same behavior simply by changing the `antithetic` argument. The results below are identical because of the default `NULL` value for the `stream` argument, which causes `vnorm` to use `stats::runif` to generate $U(0,1)$ variates for inverting. Were the `stream` parameter to have an integer argument, the function output would be different since `vnorm` would generate $U(0,1)$ variates using `rstream::rstream_sample` instead.

```
> set.seed(8675309)
> u <- runif(n = 3)
> qnorm(u, mean = 0, sd = 1)
[1] -0.9966 -0.0547  0.7218
> qnorm(1 - u, mean = 0, sd = 1)
[1]  0.9966  0.0547 -0.7218
```

```
> set.seed(8675309)
> vnorm(3, mean = 0, sd = 1, antithetic = F)
[1] -0.9966 -0.0547  0.7218
> set.seed(8675309)
> vnorm(3, mean = 0, sd = 1, antithetic = T)
[1]  0.9966  0.0547 -0.7218
```

3 VISUALIZING INVERSION FOR VARIATE GENERATION

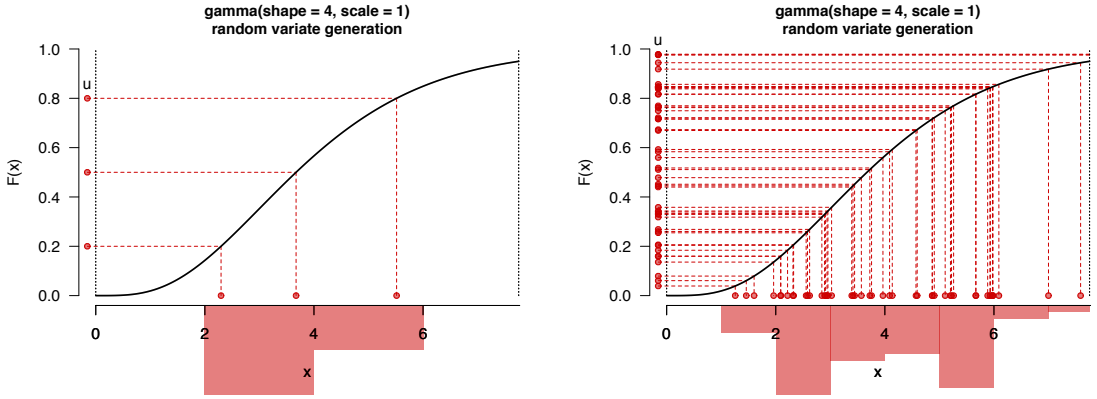
Our `simEd` package also includes functions to visualize distributions and inversion for variate generation for the seven distributions discussed in Section 2. Again, we have chosen naming and parameter conventions similar to the `stats` variate generation functions, but replacing the leading 'r' in the function name with 'i' (for inversion). Each of these functions accepts zero or more $U(0,1)$ variates, followed by distribution-specific parameter values, and then an optional sequence of plotting-command arguments (see Table 2). By default, each function produces custom graphics (which can optionally be suppressed) to visualize the inversion process associated with generating variates from the corresponding distribution, and returns the values of the generated variates. Several examples are given below. Moreover, if the value of the `u` parameter is `NULL`, the functions will display only two plots — pdf/pmf and cdf — allowing the user to visualize distributions and their parameter values independent of variate inversion.

The example of R code below left, along with Figure 1a, shows the result of executing the `igamma` function given user-specified values for $U(0,1)$ variates to invert. The example below right, along with Figure 1b, shows the result of executing `igamma` using 50 randomly-generated $U(0,1)$ variates. In Figure 1, red points on the vertical axis correspond to the values of the $U(0,1)$ variates to be inverted, and red points on the horizontal axis represent the distribution-specific variates generated. A vertically-inverted histogram of the generated variates is displayed beneath the horizontal axis. Note the `plotDelay` argument in the

second example — on execution, there will be a 0.1-second delay between plotting each variate inversion, with the $U(0,1)$ variates inverted in the order they are generated/supplied.

```
> u <- c(0.2, 0.5, 0.8)
> igamma(u, shape = 4, scale = 1)
[1] 2.297 3.672 5.515
```

```
> set.seed(8675309)
> u <- runif(n = 50)
> igamma(u, shape = 4, scale = 1, plotDelay = 0.1)
[1] 2.090 3.569 5.222 5.261 2.621 4.592 9.001 ...
```

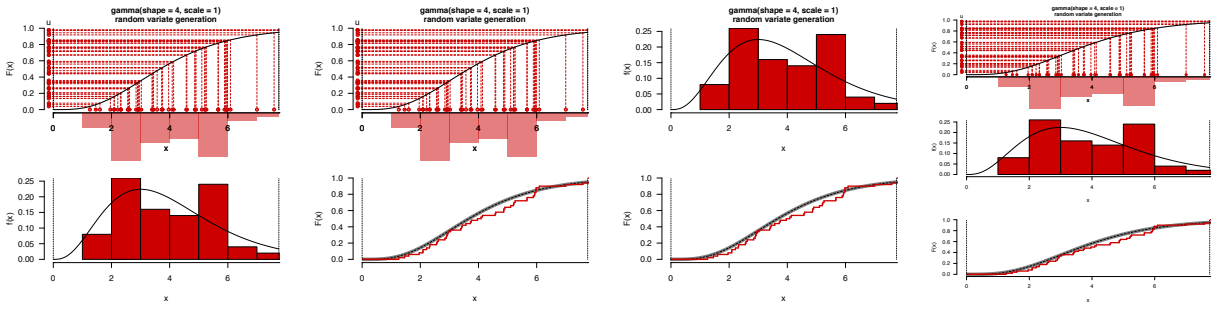


(a) (b)

Figure 1: Function `igamma` displaying (a) inversion of $u = 0.2$, 0.5, and 0.8; (b) inversion of 50 randomly-generated $U(0,1)$ variates. Note the vertically-inverted histogram of variates beneath the horizontal axis.

The graphics are customizable in that the user can choose which, if any, among three plots to display: the cdf with inverted $U(0,1)$ variates; the pdf/pmf superimposed on a histogram of the generated variates; and the cdf superimposed on an empirical cdf of the generated variates. Displaying of these plots is controlled via `showCDF`, `showPDF` (`showPMF` for discrete distributions), and `showECDF` parameters. (The `show` parameter, with a length-three binary vector for its argument, can instead be used to control the plots.) Figure 1b shows an example of the default behavior, with only `showCDF` being true, resulting in a graphic with only the cdf and inverted $U(0,1)$ variates. The additional R code below, along with Figure 2, shows four other examples using variates identical to those in Figure 1b but with different plot selections.

```
> variates <- igamma(u, shape = 4, scale = 1, showPDF  = TRUE)
> variates <- igamma(u, shape = 4, scale = 1, showECDF = TRUE)
> variates <- igamma(u, shape = 4, scale = 1, show = c(0, 1, 1))  # show PDF, ECDF
> variates <- igamma(u, shape = 4, scale = 1, show = c(1, 1, 1))  # show all three
```

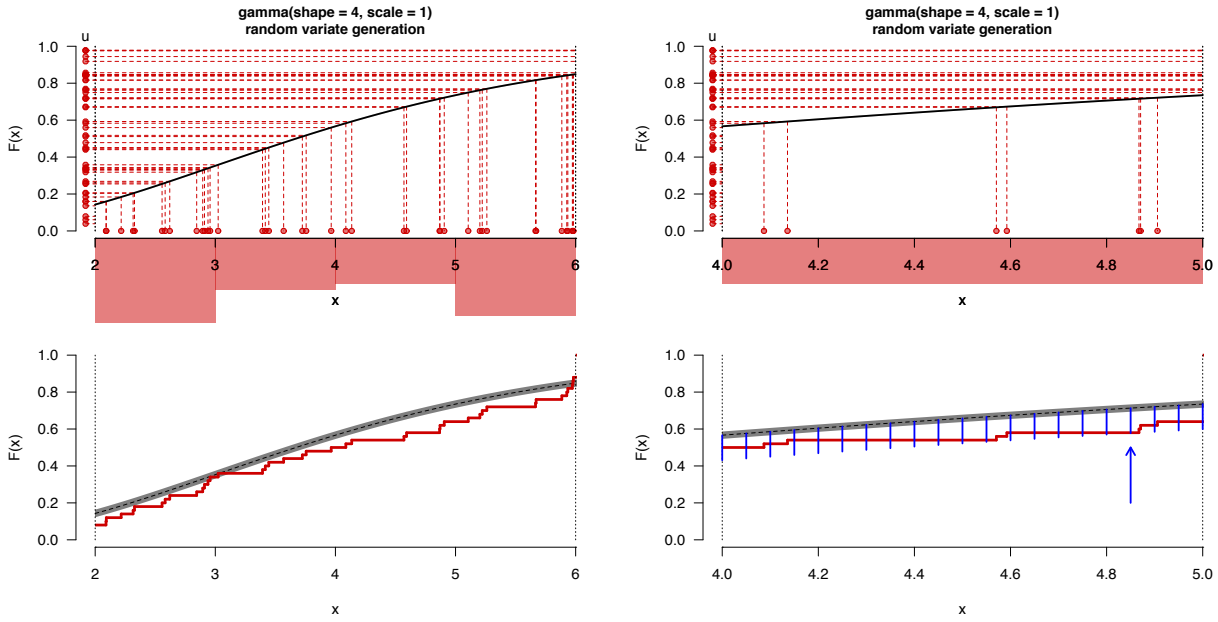


(a) (b) (c) (d)

Figure 2: Variate generation visualization function `igamma` displaying (a) cdf with inversion and pdf, (b) cdf with inversion and ecdf, (c) pdf and ecdf, (d) all three plots.

The graphics are also customizable via the `minPlotQuantile` and `maxPlotQuantile` parameters. In this way, the user can focus on a portion of the quantile space, as shown in the R code below and in the corresponding plots in Figure 3a. The vertical axes will maintain their default scales, but the horizontal axes will display only the portion corresponding to the indicated quantile sub-space. In particular, the inverted-variates plot will display points corresponding to *all* of the $U(0,1)$ variates on the vertical axis; but on the horizontal axis, only points corresponding to inverted values that fall within the quantile limits are displayed. As confirmed by the code below and Figure 3a, 40 of the 50 inverted variates fall in [2,6].

```
> qLo <- pgamma(q = 2, shape = 4, scale = 1)
> qHi <- pgamma(q = 6, shape = 4, scale = 1)
> variates <- igamma(u, shape = 4, scale = 1, showECDF = TRUE,
                     minPlotQuantile = qLo, maxPlotQuantile = qHi)
> length(variates[ variates >= 2 & variates <= 6 ])
[1] 40
```



(a) (b)

Figure 3: Function `igamma` displaying (a) subset of the quantile space, and (b) focused subset of the quantile space with superimposed vertical lines to visualize Kolmogorov–Smirnov test statistic.

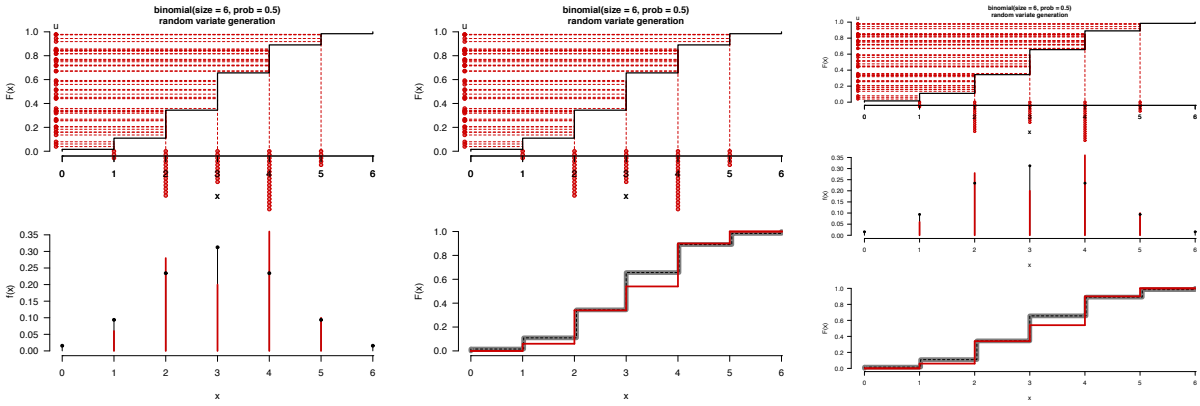
The graphics can be further customized through use of the `resetRowsMargins` parameter. In creating the visualization plots, our functions explicitly modify the R-default plot margins and number of rows, which are reset by default before exiting. On occasion, the user may wish to superimpose additional graphics or text onto the final plot. As an example, inspection of Figure 3a suggests that the largest vertical distance between the cdf and ecdf occurs somewhere between 4 and 5 on the horizontal axis. As shown in the R code below and in the corresponding Figure 3b, we can use the Kolmogorov–Smirnov test (using `ks.test` from the `stats` package) to quantify that largest vertical distance D, and then, by setting `resetRowsMargins` to false, visually explore the location of D by superimposing line segments on the bottommost plot. (If more than one plot is displayed and `resetRowsMargins` is true, any subsequent superimposition will appear relative to the entire graphics area, not to the axes of the bottom plot.)

```
> qLo <- pgamma(q = 4, shape = 4, scale = 1)
> qHi <- pgamma(q = 5, shape = 4, scale = 1)
```

```
> variates <- igamma(u, shape = 4, scale = 1, resetRowsMargins = FALSE,
                     showECDF = TRUE, minPlotQuantile = qLo, maxPlotQuantile = qHi)
> D <- ks.test(variates, "pgamma", shape = 4, scale = 1)$statistic
> for (x in seq(4, 5, by = 0.05)) {
    y <- pgamma(x, shape = 4, scale = 1)
    segments(x, y - D, x, y, lwd = 2, col = "blue")  }
> arrows(4.85, 0.2, 4.85, 0.5, lwd = 2, length = 0.1, col = "blue")
```

All of the visualization functions given in Table 2 work similarly, differing only in distribution-specific parameters. Examples for a discrete distribution are given in the R code below and in Figure 4. In the discrete case, spikes represent the estimated pmf (red, without dots) and theoretical pmf (black, with dots).

```
> set.seed(8675309)
> u <- runif(n = 50)
> variates <- ibinom(u, size = 6, prob = 0.5, showPMF  = TRUE)
> variates <- ibinom(u, size = 6, prob = 0.5, showECDF = TRUE)
> variates <- ibinom(u, size = 6, prob = 0.5, showPMF  = TRUE, showECDF = TRUE)
```



(a) (b) (c)

Figure 4: Variate generation visualization function `ibinom` displaying (a) cdf with inversion and pmf, (b) cdf with inversion and ecdf, (c) cdf with inversion, pmf, and ecdf.

4 QUEUEING FUNCTIONS AND UTILITIES

The `simEd` package includes two functions simulating different queueing models: a function `ssq` for a single-server queue model, and a function `msq` for a multiple-server queue model. Both functions are very flexible through use of the parameters shown in Table 3, and are extensible by permitting user-provided arrival and service functions. Both `ssq` and `msq` use an event-oriented approach in their implementation.

The `ssq` function is discussed at length in Lawson and Leemis (2015), to which we refer the interested reader for details of `ssq` and corresponding pedagogical examples. Relative to that previous work, the current version of `ssq` has been modified slightly: some of the parameters have been renamed and reordered for clarity and ease of use; and the default arrival and service functions now use streams as implemented by the `vexp` function discussed in Section 2. Otherwise, using default parameter values, the `ssq` function still simulates an $M/M/1$ queue, having arrival rate $\lambda = 1$ and service rate $\mu = 10/9$. Here, we provide examples using: `ssq` with our time-persistent statistics functions; `ssq` with our variate generator functions (see Section 2) demonstrating variance reduction techniques; and `msq` employing multiple service models.

Time-persistent statistics: For a time-persistent statistic $x(t)$ observed over the time interval $[0,T]$,

$$\bar{x} = \frac{1}{T}\int_0^T x(t)\,dt \qquad \text{and} \qquad s^2 = \frac{1}{T}\int_0^T x^2(t)\,dt - \bar{x}^2.$$

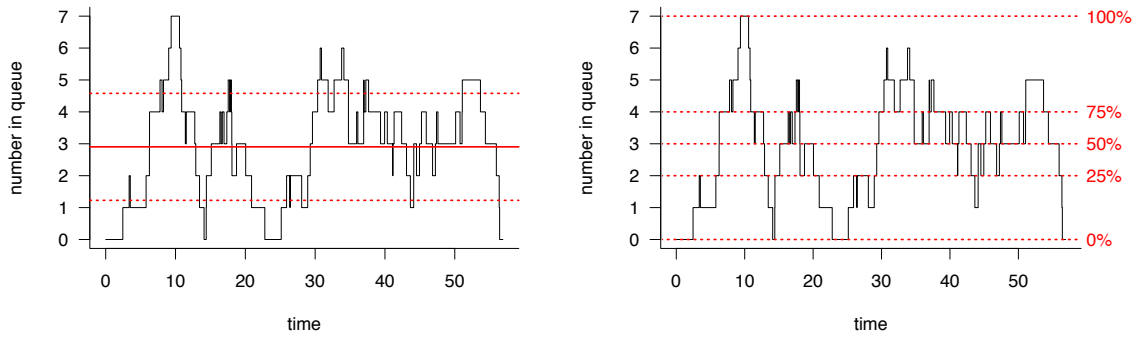
are the mean and variance, respectively. The notion of time-persistent statistics is unique to simulation and is often challenging for students new to simulation. For this reason, we provide three functions for computing time-persistent statistics given simulation output: `meanTPS`, `sdTPS`, and `quantileTPS`.

The R code example below shows the output produced by these three functions using simulation output from a 50-customer $M/M/1$ queue simulation having arrival rate $\lambda = 1$ and service rate $\mu = 10/9$. In particular, we compute the time-averaged mean, standard deviation, and (default) quantiles of the number of customers in the queue across time. The vector named `output$numInQueueT` contains the time values when changes occurred to the number in the queue, and the vector `output$numInQueueN` contains each associated number in the queue at those times. Figure 5a depicts the number in queue versus time for this simulation, with the time-averaged mean (2.904) superimposed as a solid horizontal line and one standard deviation away from the mean (2.904 ± 1.676) superimposed as dotted horizontal lines. Figure 5b also depicts the number in queue versus time, but with time-averaged quantiles superimposed as dotted lines. Visual depictions of these statistics aid in understanding, and are easy to produce using our software.

```
> output <- ssq(maxArrivals = 50, seed = 8675309, saveNumInQueue = T, showOutput = F)
> mean   <- meanTPS(output$numInQueueT, output$numInQueueN)
> sd     <- sdTPS(output$numInQueueT, output$numInQueueN)
> quant  <- quantileTPS(output$numInQueueT, output$numInQueueN)
```

```
> plot(output$numInQueueT, output$numInQueueN, type = "s",
       xlab = "time", ylab = "number in queue", las = 1, bty = "l")
> abline(h = mean, lwd = 2, col = "red")
> abline(h = c(mean - sd, mean + sd), lty = "dotted", lwd = 2, col = "red")
```

```
> plot(output$numInQueueT, output$numInQueueN, type = "s",
       xlab = "time", ylab = "number in queue", las = 1, bty = "l")
> abline(h = quant, lty = "dotted", lwd = 2, col = "red")
> mtext(c("0%","25%","50%","75%","100%"), side = 4, at = quant, las = 1, col = "red")
```



(a) (b)

Figure 5: Number in the queue versus time for the first 50 customers using `ssq`, with (a) time-averaged mean and one standard deviation superimposed and (b) time-averaged quantiles superimposed.

Common random numbers: Using our `ssq` function, the example below demonstrates the use of our streams-capable variate generators for implementing the variance reduction technique of common random numbers. We begin by defining two different arrival functions having different rates, each using stream 1. The subsequent calls to `ssq` each use one of the two different arrival functions (highlighted in bold) but the same default service function (which uses stream 2). The corresponding printed output demonstrates that different interarrival times, but exactly the same service times, are seen by each simulation. (The differences in interarrival times and in service times are also highlighted in bold.) Our streams-capable variate generators allow for easy separation of stochastic elements in a simulation.

```
> myArr1  <- function() { vexp(1, rate = 1,    stream = 1) }
> myArr2  <- function() { vexp(1, rate = 10/9, stream = 1) }
> output1 <- ssq(maxArrivals = 1000, seed = 8675309, showOutput = FALSE,
                 interarrivalFcn = myArr1, saveAllStats = TRUE)
> output2 <- ssq(maxArrivals = 1000, seed = 8675309, showOutput = FALSE,
                 interarrivalFcn = myArr2, saveAllStats = TRUE)
> sum(output1$interarrivalTimes != output2$interarrivalTimes)
[1] 1000
> sum(output1$serviceTimes != output2$serviceTimes)
[1] 0
```

Antithetic Variates: We will now use our software to demonstrate the use of antithetic variates as a variance reduction technique in discrete-event simulation. We begin with some general notation. Let X_1 be the estimate of a particular measure of performance for a first run of some discrete-event simulation model. Let X_2 be an estimate of that same measure of performance for a second run of the same model. Then, $\bar{X} = (X_1 + X_2)/2$ will be used to estimate the measure of performance from the two runs. We consider two approaches to estimating the measure: "*raw simulation*", in which X_1 and X_2 are independent, resulting from independent sequences of random numbers used in the two separate simulation runs; and "*simulation with antithetic variates*", in which X_1 and X_2 are dependent. In the latter case, the run resulting in X_1 uses a sequence of random numbers $u_1, u_2, u_3, \ldots$, while the run resulting in X_2 uses the antithetic random numbers $1-u_1, 1-u_2, 1-u_3, \ldots$ at the analogous positions. Hence, in the case of simulation with antithetic variates, X_1 and X_2 are dependent because of the induced correlation. The variance of $\bar{X}$ is

$$V[\bar{X}] = V\left[\frac{X_1 + X_2}{2}\right] = \frac{1}{4}\left[V[X_1] + V[X_2] + 2\mathrm{Cov}(X_1, X_2)\right].$$

In the case of raw simulation, X_1 and X_2 are independent, so $\mathrm{Cov}(X_1, X_2) = 0$. But when antithetic variates are employed, then typically $\mathrm{Cov}(X_1, X_2) < 0$, which will reduce $V[\bar{X}]$, the essence of antithetic variates.

To transition to a specific model, we will use the `ssq` function to investigate an $M/M/1$ queue with arrival rate $\lambda = 1$ and service rate $\mu = 10/9$. The measure of performance that will be estimated is the average sojourn time for the first 25 customers in the queue, assuming an initial state of empty and idle. (To five digits accuracy, the true mean sojourn time of the first 25 customers is 3.0245.) The R code for the experiment is given below. We begin by defining two pairs of interarrival and service functions, each using independent streams for the interarrival and service processes, but only the second pair of functions using antithetic variates. (The capability of our variate generators to produce independent streams and antithetic variates is needed for this experiment.) The raw simulation code then appears on the left, while the code for simulation with antithetic variates appears on the right. The vector `x1` will contain the average sojourn times for the first 10,000 replications. (Note that the initial seed is used for the first replication, whereas the previous state of the generator is used in subsequent replications.) In the case of raw simulation, the vector `x2` will contain the average sojourn times for the next 10,000 replications, where, for each replication, the underlying sequences of random numbers are independent from those used in the first 10,000 replications. In the case of simulation with antithetic variates, the vector `x2` will contain the average sojourn times for the next 10,000 replications, where, for each replication, the underlying sequences of random numbers are the antithetic versions of the sequences used in the first 10,000 replications. Code differences are in bold.

```
myArr1 <- function() { vexp(1, rate = 1,      stream = 1, antithetic = FALSE) }
mySvc1 <- function() { vexp(1, rate = 10 / 9, stream = 2, antithetic = FALSE) }

myArr2 <- function() { vexp(1, rate = 1,      stream = 1, antithetic = TRUE)  }
mySvc2 <- function() { vexp(1, rate = 10 / 9, stream = 2, antithetic = TRUE)  }

nrep <- 10000
```

```
# raw discrete-event simulation
x1   <- rep(0, nrep)
seed <- 5551212
for (i in 1:nrep) {
  if (i == 2) seed <- NA
  output <- ssq(maxArrivals = 25,
      seed             = seed,
      interarrivalFcn  = myArr1,
      serviceFcn       = mySvc1,
      saveSojournTimes = TRUE)
  x1[i] <- mean(output$sojournTimes)
}


x2   <- rep(0, nrep)
seed <- NA  # use previous state of RNG
for (i in 1:nrep) {
  output <- ssq(maxArrivals = 25,
      seed             = seed,
      interarrivalFcn  = myArr1,
      serviceFcn       = mySvc1,
      saveSojournTimes = TRUE)
  x2[i] <- mean(output$sojournTimes)
}
xBar <- (x1 + x2) / 2

cat(mean(xBar), var(xBar), cor(x1,x2))
```

```
# simulation with antithetic variates
x1   <- rep(0, nrep)
seed <- 5551212
for (i in 1:nrep) {
  if (i == 2) seed <- NA
  output <- ssq(maxArrivals = 25,
      seed             = seed,
      interarrivalFcn  = myArr1,
      serviceFcn       = mySvc1,
      saveSojournTimes = TRUE)
  x1[i] <- mean(output$sojournTimes)
}

x2   <- rep(0, nrep)
seed <- 5551212
for (i in 1:nrep) {
  if (i == 2) seed <- NA
  output <- ssq(maxArrivals = 25,
      seed             = seed,
      interarrivalFcn  = myArr2,
      serviceFcn       = mySvc2,
      saveSojournTimes = TRUE)
  x2[i] <- mean(output$sojournTimes)
}
xBar <- (x1 + x2) / 2

cat(mean(xBar), var(xBar), cor(x1,x2))
```

The results of the raw simulation are given in columns 2–4 of Table 5, comparing the results using three different initial seeds. For each, the mean of the 10,000 average sojourn times is close to the analytic value 3.0245, and since all of the runs are independent, the correlation is close to zero as expected. In particular, note the variance of the point estimate, which in the raw simulation case averages approximately 1.7. Similarly, the results of simulating with antithetic variates are given in columns 5–7 of Table 5. Again, the point estimators are close to the theoretical value 3.0245. The bottom right of Table 5 also shows that the correlation between the initial and antithetic replications is indeed negative, so we expect that the antithetic variates will be effective in reducing the variance of the point estimator of the mean sojourn time. Here, the variance of the sojourn time is roughly 1.0, so there is about a $[(1.7-1.0)/1.7]\cdot 100\% = 41.7\%$ reduction in variance by using antithetic variates. Our software makes implementing this experiment straightforward.

Table 5: Results of simulating an $M/M/1$ queue using raw simulation and using simulation with antithetic variates, estimating mean sojourn time of the first 25 customers.

	raw simulation			*simulation w/ antithetic variates*		
initial seed	5551212	8675309	1234567	5551212	8675309	1234567
mean of sojourn times	3.0091	3.0422	3.0292	3.0220	3.0293	3.0314
variance of sojourn times	1.7098	1.7474	1.7089	0.9893	1.0090	1.0333
correlation between means	0.0096	0.0103	−0.0051	−0.4147	−0.4135	−0.4035

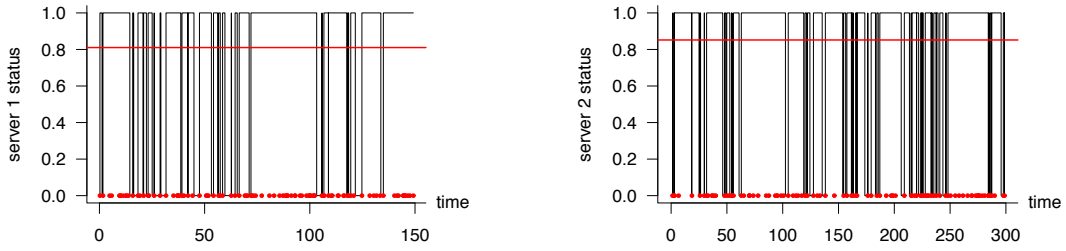
Multiple-Server Queue: Using default arrival and service processes, function `msq` simulates an $M/M/k$ queue (default $k = 2$), having arrival rate $\lambda = 1$ and service rate $\mu = 10/(9k)$ per server. Functionality of `msq` is similar to that of `ssq` with the exception of two additional parameters: `numServers`, indicating the number of servers k; and `serverSelection`, which takes one of five values corresponding to a server selection criteria (least recently used; least frequently used; cyclic, starting from the successor of

the last server engaged; random; and in-order, starting from the first — see Table 3). In addition, the `serviceFcn` parameter value can be a single function, in which case the same service process is used for all k servers, or can be an R list of functions, allowing the servers to use different service processes.

An example using different service functions per server is shown below. We first define two service functions with different rates, and then pass those functions in an R list to `msq` via the `serviceFcn` parameter. The two servers will have service rates $\mu_1 = 10/13$ and $\mu_2 = 10/23$ respectively. (The call to `msq` uses the default value of "`LRU`", least recently used, for `serverSelection`.) The corresponding printed output indicates that the first server has a lower utilization (0.81 vs. 0.85) despite receiving a significantly larger proportion of customers (63% vs. 37%). In addition, the estimated mean service times per server are consistent with their expected values of 1.3 and 2.3. Of interest in this context, the R code below, along with Figure 6, also shows how `msq` output can be used to plot each server's status versus time. For each server, we plot their first 100 serviced customers only, with overall utilization (all customers) superimposed as a horizontal line. Additionally, the beginning time of service for each of the 100 customers appears as a red dot on the horizontal axis. These plots are consistent with the `msq` output, in that the first server (higher service rate) processes 100 customers in approximately $t = 150$ time units, compared to approximately $t = 300$ time units for the second server to process the same number of customers.

```
> server1Svc <- function() { vexp(1, rate = 10 / 13, stream = 1) }
> server2Svc <- function() { vexp(1, rate = 10 / 23, stream = 2) }
> output <- msq(maxTime = 10000, seed = 8675309, numServers = 2,
     serviceFcn = list(server1Svc, server2Svc), saveAllStats = T, showOutput = F)
> for (s in 1:2) { cat( s, ':', output$utilization[s], '\t', output$serverShare[s],
                       '\t', mean(output$serviceTimesPerServer[[s]]), '\n') }
1 : 0.81038      0.62527     1.309
2 : 0.8519       0.37473     2.2958
```

```
> for (s in 1:2) {
    n <- output$serverStatusN[[s]] # each 1 entry: start service on another customer
    t <- output$serverStatusT[[s]]
    customerIndices <- which(cumsum(n) <= 100 & n == 1) # 100 starts-of-service
    plot(t[1:tail(customerIndices, 1)], n[1:tail(customerIndices, 1)], type = "s",
       xlab = "time", ylab = paste("server", s, "status"), las = 1, bty = "l")
    abline(h = output$utilization[s], lwd = 2, col = "red")
    points(t[customerIndices], rep(0,100), pch = 20, col = "red")
  }
```



(a) (b)

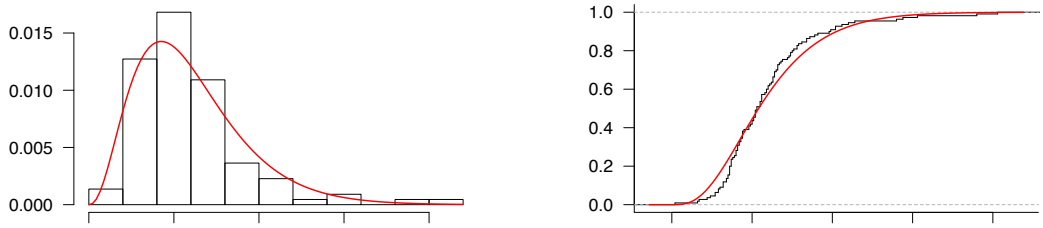
Figure 6: Server status versus time (first 100 customers) using `msq` with (a) $\mu_1 = 10/13$, (b) $\mu_2 = 10/23$.

Queueing Data Sets: Our package also includes two queueing data sets that can be used for input modeling and analysis. The first, `queueTrace`, contains two lists of 1000 arrival and service times (fabricated data) for a single-server queue. The second, `tylersGrill`, contains actual data collected on one business day at Tyler's Grill at the University of Richmond: its first list contains 1434 arrival times of all customers throughout the day; its second list contains 110 service times sampled throughout the day. The R code below, along with Figure 7, provides one example of using the `tylersGrill` data set for input modeling, specifically fitting a gamma distribution to the service times using the method of moments.

```
> svc  <- tylersGrill$serviceTimes
> aHat <- mean(svc) ^ 2 / var(svc)    # MOM estimator for gamma shape parameter
> bHat <- var(svc) / mean(svc)        # MOM estimator for gamma scale parameter
```

```
> x <- 1:max(svc)
> hist(svc, freq = FALSE, xlab = "", ylab = "")
> curve(dgamma(x, shape = aHat, scale = bhat),
    add = TRUE, col = "red", lwd = 2)
```

```
> plot.ecdf(svc, verticals = TRUE, pch = "",
     xlab = "", ylab = "")
> curve(pgamma(x, shape = aHat, scale = bhat),
     add = TRUE, col = "red", lwd = 2)
```



(a) (b)

Figure 7: Tyler's Grill service times fit with gamma using MOM: (a) histogram, (b) empirical CDF.

5 CONCLUSIONS

To date, no R package has focused on simulation pedagogy. This paper introduces the `simEd` package for R, which includes functions for variate generation with streams and antithetic capabilities, for visualizing inversion in variate generation, and for single- and multiple-server queueing simulation, as well as select utility functions and data sets. This package facilitates use of R for an introductory simulation course.

REFERENCES

Brock, K., and D. Slade. 2015. "poisson: Simulating Homogenous & Non-Homogenous Poisson Processes". https://cran.r-project.org/package=poisson.

Garcia, A., and A. Rodriguez–Paton. 2016. "rrepast: Invoke 'Repast Simphony' Simulation Models". https://cran.r-project.org/package=rrepast.

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

Lawson, B., and L. Leemis. 2017. "simEd: Simulation Education". https://cran.r-project.org/package=simEd.

L'Ecuyer, P., R. Simard, E. J. Chen, and D. W. Kelton. 2002. "An object-oriented random-number package with many long streams and substreams". *Operations Research* 50 (6): 1073–1075.

Leydold, J. 2015. "rstream: Streams of Random Numbers". https://cran.r-project.org/package=rstream.

The R Foundation 2017. "The Comprehensive R Archive Network". https://cran.r-project.org/.

Ucar, I., and B. Smeets. 2017. "simmer: Discrete-Event Simulation for R". http://r-simmer.org/.

AUTHOR BIOGRAPHIES

BARRY LAWSON is Professor of Computer Science at the University of Richmond. He received Ph.D. and M.S. degrees in Computer Science from William & Mary, and a B.S. in Mathematics from UVA's College at Wise. 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, simulation, and computational probability. He is a member of ASA and INFORMS. His email address is leemis@math.wm.edu.