

SIMULATION FUNDAMENTALS

Barry Lawson

Department of Mathematics and Computer Science
University of Richmond
Richmond, VA 23173-0001, U.S.A.

Lawrence Leemis

Department of Mathematics
The College of William & Mary
Williamsburg, VA 23187-8795, U.S.A.

ABSTRACT

This paper provides an introduction to simulation fundamentals via the Monte Carlo and next-event approaches to simulation, supported by two representative programs from the software suite written specifically for the Winter Simulation Conference's Simulation 101 workshop. The simulation libraries and functions associated with workshop are written using the R statistical language. R is suitable for an introduction to simulation because of its easy learning curve and its wide range of statistical procedures, built-in functions, and graphics capabilities. This paper begins with general instructions for downloading, compiling, and executing the software. This is followed by explanations of the Monte Carlo and next-event approaches, using two examples: `craps()` uses Monte Carlo simulation to estimate the probability of winning the dice game Craps, and `ssq3()` uses a next-event approach to estimate several measures of performance associated with a single-server queue.

1 INTRODUCTION

This paper provides an introduction to simulation fundamentals through the concepts of Monte Carlo and next-event simulation. The concepts are supported using examples from the simulation software provided with the Simulation 101 workshop and associated with the introductory simulation textbook by [Leemis and Park \(2006\)](#). For the workshop, the software is written entirely in R, using object-oriented design when appropriate. In this paper, we discuss the downloading, installation, and execution of R and the simulation software, followed by a discussion of the simulation concepts with supporting examples.

2 INSTALLING R AND THE SIMULATION SOFTWARE

R ([The R Foundation 2009](#)) is a programming language and software environment used primarily for statistical computing. It includes standard statistical tools, such as classical statistical tests and regression analysis, and more modern tools such as bootstrapping, survival analysis, and time-series analysis. R is also capable of producing publication-quality plots and has many built-in functions, such as `sort` for sorting the elements of a vector and `gamma` for evaluating the gamma function.

The primary advantage of using R instead of a more typical high-level language (HLL) for simulation work lies in R's affordability (it can be downloaded for free), easy-to-use statistical procedures, built-in functions, and graphics capabilities. The primary disadvantage of using R is that it is an interpreted language—in general, simulation programs will execute more slowly when written in R rather than another HLL (sometimes dramatically so). Nonetheless, because of its ease of use and its many built-in statistical capabilities, we have provided a suite simulation software in R for the Simulation 101 workshop, well-suited for an introduction to simulation.

2.1 Obtaining R

R can be downloaded via (<http://www.r-project.org/>). Click on the CRAN (Comprehensive R Archive Network) link and choose a mirror that is located near you. Then choose your platform (Linux, MacOS X, Windows), and the appropriate version of the software can be downloaded and installed according to the typical installation process for your platform.

2.2 Obtaining the R.oo Package

In addition to the base R software, there is an add-on package to R that is necessary to run the simulation software. The R.oo package (Bengtsson 2003) provides functionality for object-oriented programming and pass-by-reference capabilities in R.

The package can be obtained as follows. After starting R software, type

```
source("http://www.braju.com/R/hbLite.R")
```

which fetches and sources in R the functions necessary for installing R.oo. Next type

```
hbLite("R.oo")
```

which invokes a newly sourced function to install the R.oo package. The previous two steps are a one-time cost.

2.3 Obtaining the R Simulation Software

The simulation software for the workshop is available as a package for R. Visit the link <http://www.mathcs.richmond.edu/~blawson/Sim101/packages/> and choose the directory that corresponds to your platform, saving to disk the package file (ending in .tar.gz for Mac OS-X or Linux, .zip for Windows) in that directory. If you are using Windows, in R enter

```
install.packages(file.choose(), repos=NULL)
```

If you are using Mac OS-X or Linux, in R enter

```
install.packages(file.choose(), repos=NULL, type="source")
```

In the resulting pop-up window, navigate to and choose the Simulation 101 package that you downloaded above. (You may receive progress and/or warning messages in R, but, in the absence of a fatal error, these may be ignored.) The download and install steps are a one-time cost.

2.4 Loading the Simulation Software at R Startup

Every time R is started, to use the simulation software you need to execute the following two commands in sequence:

```
library(R.oo)
library(Sim101)
```

The former command loads the add-on R.oo package (installed above) in your current R session; the latter loads the add-on Sim101 package. If the R prompt returns with no messages (or at most a version warning message), the Sim101 package has been successfully loaded.

3 LIBRARIES AND FUNCTIONS IN THE R SIMULATION SOFTWARE

Following is a list of the libraries and functions that are provided with the R version of the simulation software. These libraries and functions were written to be consistent with the corresponding C versions described in Leemis and Park (2006)—given the same initial conditions, both generate exactly the same random number sequences and can therefore produce exactly the same output. For more details about a specific library or function, use R's online help, e.g., `help(Rngs)`.

3.1 Libraries

The following random-number-generation libraries all use object-oriented (OO) design. These “libraries” are themselves used as objects in R, not as a typical R library you might load at startup. More specifically, instances of the random-number-generation classes are used within the simulation functions discussed below.

- **Rngs**: a multiple-stream Lehmer random number generator.
- **Rvgs**: a library used to generate random variates from six discrete (Bernoulli, binomial, “equilikely”, geometric, Pascal, and Poisson) and seven continuous (uniform, exponential, Erlang, normal, log-normal, chi-square, and student) distributions.
- **Rvms**: a library used to evaluate the probability density functions, cumulative distribution functions, and inverse distribution functions for the distributions provided in **Rvgs**.

3.2 Functions

In the R simulation software, we have also provided a collection of simulation functions (programs) and utilities, including three Monte Carlo simulations and three discrete-event simulations. In the R version, each of these is its own function.

- **craps**: produces a Monte Carlo estimate of the probability of winning the simple dice game Craps played with two fair dice.
- **galileo**: produces a Monte Carlo estimate of the probability of each sum 3,4,...,18 obtained when rolling three fair dice.
- **hat**: produces a Monte Carlo estimate of the probability that a hat check person will return all n hats to the wrong owners when returning n hats at random.
- **ssq1**: uses a process-interaction world view to implement the arrival and service processes of a trace-driven single-server queue.
- **ssq2**: an extension of **ssq1** that uses the **Rngs** library, implementing exponentially distributed interarrival times and uniformly distributed service times.
- **ssq3**: an extension of **ssq2** that uses the **Rngs** library, illustrating a next-event approach to the single-server queue.
- **cdh**: utility that plots a histogram of data drawn from a continuous population.
- **ddh**: utility that plots a histogram of data drawn from a discrete population.
- **estimate**: utility that computes a confidence interval estimate for a data set.

In addition, we have provided implementations of the inverse distribution (also known as *probability transformation*) functions for seven of the distributions provided in **Rvgs**. Each of these functions evaluates the corresponding inverse distribution and (optionally) displays an intuitive graphical representation:

- **idfBinomial**,
- **idfExponential**,
- **idfGeometric**,
- **idfLognormal**,
- **idfNormal**,
- **idfPascal**, and
- **idfUniform**.

For more details on any of these functions, use R’s online help (e.g., `help(idfPascal)`).

3.3 Viewing and Modifying R Simulation Source Code

The R code associated with the various simulation functions can be viewed by typing the name of the function. For example, typing `galileo` in R will display the source code for the function `galileo()`, the first few lines of which are shown in Figure 1. The first line of the function indicates that there are two parameters, `seed` (the random number generator seed) and `N` (the number of replications), which have default values 12345 and 1000 respectively.

To modify one of the simulation programs, you should first dump to file the source code of the related function, e.g.,

```
dump("galileo", file=file.choose(new=TRUE))
```

In the resulting pop-up window, if you name the file `galileo_mods.R`, you may then make modifications to the source code by editing the file `galileo_mods.R` using your favorite text editor. You can then overwrite the existing version of the function in your current R session by reading in your modified source code, e.g.,

```

> galileo
function (seed = 12345, N = 1000)
{
  if (!is.numeric(seed))
    stop("seed must be an integer > 0, ",
         "or < 0 for system-clock-supplied seed, ",
         "or 0 for interactively-supplied seed")
  if (!is.numeric(N) || N <= 0)
    stop("N must be an integer > 0")
  rng = Rng()

  :

```

Figure 1: The first few lines from the R source code for the `galileo()` function.

```
source(file.choose())
```

Any subsequent calls to the function in the current R session will exhibit the effects of your modifications. Note this will not supplant the implementation provided in the original `Sim101` library. If, on a subsequent restart of R, you load the `Sim101` library, the original version of the function will be used. If you want to use your modified version, you will need to read in your modified source code again.

3.4 Executing the R Simulation Software

To execute a simulation program in R, simply type the corresponding function name with associated arguments. For example, because `galileo` has default parameters, it can be executed using

```
galileo()
```

which displays a vector of 16 elements that are Monte Carlo estimates of the probabilities of rolling a 3, 4, ..., 18 when rolling three dice. The default parameters can be changed by supplying the parameters in the function call, as shown in the following examples:

```
galileo(8675309, 10000)
galileo(seed = 8675309, N = 10000)
galileo(N = 10000)
```

If subsequent operations on the estimates are required, the estimates can be placed into a vector, e.g.,

```
x = galileo()
```

Subsequent functions can be applied to `x`, whether native R functions (e.g., `sum(x)` to confirm that the elements of `x` form a legitimate probability density function, or `sum(3:18 * x)` to calculate the mean of the total number of pips showing on the dice in the experiments) or functions provided by the simulation software (e.g., `cdh(x)` to produce a continuous data histogram of the probabilities).

As another example, the `idfExponential()` function can be used to evaluate the inverse distribution function of the exponential distribution at $u = 0.2$, $u = 0.5$, and $u = 0.999$ as shown below, with a resulting graphic as shown in Figure 2.

```

> idfExponential(mean = 2.0, u = c(0.2, 0.5, 0.999))
[1] 0.4462871 1.3862944 13.8155106

```

Because the simulation software will execute more slowly in R (an interpreted language) than in some other HLLs, especially for long simulations, users may be interested in having R interface with C or Java. This is accomplished by writing simulation source code in C or Java, and then constructing an R function to accept arguments from the user, invoke the

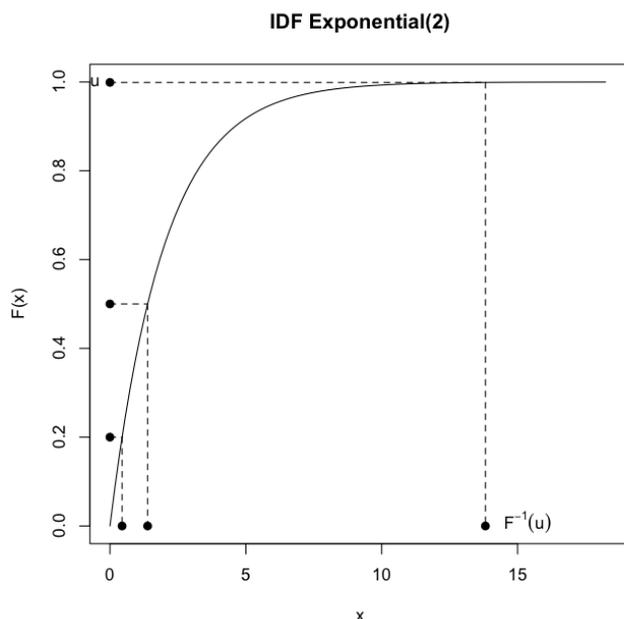


Figure 2: Graphical representation of an *Exponential(2)* idf evaluated at $u = 0.2, 0.5,$ and 0.999 .

corresponding C or Java program, and return the simulation results to the user in an R-accessible format (e.g., R vector or list). In this way, the user will experience much shorter execution times for long simulations while maintaining accessibility to the built-in statistical and graphical routines of R. The interested reader may contact the authors of this paper for existing implementations. For the details of the interfacing procedure, the interested reader should refer to the documentation on writing R extensions ([R Development Core Team 2009](#)).

4 A BRIEF INTRODUCTION TO SIMULATION

Here we consider two basic types of simulation models: Monte Carlo simulation and discrete-event simulation. A Monte Carlo simulation model contains one or more stochastic (random) components for which time evolution does not play a significant role. A discrete-event simulation model also contains one or more stochastic components but, unlike Monte Carlo, the passage of time plays a significant role. As implied by the name, discrete-event simulation involves state variable(s) that naturally change values only at discrete points in time. Other types of simulation models (e.g., stochastic continuous-time) are beyond the scope of this introduction. We illustrate some of the details associated with Monte Carlo and discrete-event models using examples from the R simulation software below.

4.1 Monte Carlo Simulation

Monte Carlo simulation is often used in the estimation of one or more probabilities based on the *frequency theory of probability*. More specifically, if a random experiment is repeated N times and N_a is the number of times an event \mathcal{A} occurs ($N_a \leq N$), then the frequency theory of probability asserts that the relative frequency N_a/N of event \mathcal{A} converges to the probability of \mathcal{A} :

$$\Pr(\mathcal{A}) = \lim_{N \rightarrow \infty} \frac{N_a}{N}.$$

Consider the gambling game “Craps”. The game involves tossing a pair of fair dice one or more times and observing the sum of the two up faces (i.e., the total number of pips showing on the up faces). The rules are as follows:

- If a 7 or 11 is tossed on the first roll, the player wins immediately.
- If a 2, 3, or 12 is tossed on the first roll, the player loses immediately.

- If any other number is tossed on the first roll, this number is called the “point.” The dice are rolled repeatedly until the point is tossed (in which case the player wins) or a 7 is tossed (in which case the player loses).

The goal here is to find the probability that the player wins. This example is naturally modeled using Monte Carlo simulation—the evolution of time plays no role in the probability of winning. As the number of replications N of the game increases, the resulting probability estimate will tend to the true probability of winning at Craps.

To construct a Monte Carlo model, an *Equilikely*(1,6) random variate is used to model the roll of a single fair die (i.e., any of the integer values 1 through 6 is equally likely to occur). The algorithm shown in Figure 3 uses N for the number of replications of the game of Craps. The variable *wins* counts the number of wins and the `do-while` loop is used to simulate the player attempting to make the point when more than one roll is necessary to complete the game.

```

wins = 0;
for (i = 1; i ≤ N; i++) {
  roll = Equilikely(1, 6) + Equilikely(1, 6);
  if (roll == 7 or roll == 11)
    wins++;
  else if (roll != 2 and roll != 3 and roll != 12) {
    point = roll;
    do {
      roll = Equilikely(1, 6) + Equilikely(1, 6);
      if (roll == point) wins++;
    } while (roll != point and roll != 7)
  }
}
return (wins / N);

```

Figure 3: Monte Carlo algorithm for estimating the probability of winning at Craps.

The algorithm has been implemented in the R function `craps()` (type `craps` in R to see the source code corresponding to the algorithm above). The function has two arguments: the number of Monte Carlo simulation replications N (default is $N = 1000$) and the random number generator seed (default is `seed = 12345`). Invoking `craps()` to use the default arguments returns the probability estimate as a scalar, as shown below:

```

> craps()
[1] 0.522

```

If `craps()` is executed again in this fashion, an identical result will occur because the same sequence of random numbers will be used, manifested by the choice of (default) initial seed. In order to get a different set of Monte Carlo replications (and, therefore, different probability estimates), the seed must be changed as shown in the following two examples.

```

> craps(seed = 987654321)
[1] 0.51
> craps(seed = 54321)
[1] 0.53

```

This rather surprising streak of three estimates greater than one-half makes one wonder if this is a game with odds tilted toward the player. Fortunately, in this case the problem also has an analytic solution. (This will not be the case for the discrete-event simulation in the next section.) The solution using the axiomatic approach is $244/495 \cong 0.493$.

The three simulated values that exceeded the analytic value are the probabilistic equivalent of tossing three consecutive heads with a fair coin. This is evidence that the coin may be biased or double-headed, but certainly is not conclusive. It is a worthwhile investigation to increase the number of simulation replications in order to confirm the correctness of the analytic solution, as shown in the following three examples with probability estimates scattered about the true theoretical value 0.493 as expected.

```

> craps(N = 10000, seed = 987654321)
[1] 0.497

```

```
> craps(N = 10000, seed = 123456789)
[1] 0.4855
> craps(N = 10000, seed = 555555555)
[1] 0.5021
```

Our three earlier unexpectedly high estimates using $N = 1000$ were simply a result of random sampling variability (often the case when the number of replications is too low). If we use the same three seeds as above but increase the number of replications even more, we expect the probability estimates to be even closer to the theoretical value, as confirmed by the following three examples:

```
> craps(N = 100000, seed = 987654321)
[1] 0.49264
> craps(N = 100000, seed = 123456789)
[1] 0.4913
> craps(N = 100000, seed = 555555555)
[1] 0.49339
```

Keep in mind that for many (most) problems for which the Monte Carlo approach applies, a true theoretical probability may be difficult or even impossible to obtain. In those cases, one can use consistency checks to provide confidence that the simulation-derived estimates are accurate (e.g., as N increases, do the estimates tend to one another?). For more details, consult [Leemis and Park \(2006\)](#) or any other simulation text.

4.2 Discrete-Event Simulation

Monte Carlo simulation is appropriate for *static* systems that do not involve the passage of time. Discrete-event simulation is appropriate for *dynamic* systems where the passage of time plays a significant role. We describe one instance of a discrete-event simulation model in this section.

Queueing models are one of the common applications of discrete-event simulation. Consider a single-server queue: there is a single entity that provides service to jobs that arrive, with jobs waiting in an unlimited-length queue if necessary. The assumptions in this model are:

- The queue discipline is first-come, first-served.
- There is no time delay between jobs (customers).
- The server does not take any breaks.

In this example, the *state* of the system can be described by a single variable: the number of jobs in the node. Note that the number of jobs in service (0 or 1) and the number of jobs in the queue can be determined from the number in the entire node. It should be clear that, unlike the Monte Carlo simulation of Craps, time plays a significant role here. Changes to the state of the system occur at *discrete* points in time—whenever a new job arrives or a job in service is completed.

The most commonly-used approach for discrete-event simulation is a *next-event* approach. Given a collection of variable(s) that provide a complete description of system state, to construct a next-event model one must define:

- a simulation clock;
- the types of events that may change the system state;
- a data structure to store events yet to occur (i.e., the *event list*); and
- algorithms to define state changes for each type of event.

The simulation clock is easily defined as a real-valued variable t (with, say, maximum value T). The two types of events that may change the state of this system are *arrival* of a job and *completion of service*. Then after initializing the simulation clock and event list appropriately, the basic next-event algorithm proceeds as shown in Figure 4. While the simulated time is less than the maximum, the next event to occur in simulated time is retrieved from the event list, that event is handled (updating the system state) according to its type, and then the next event of that same type is generated and inserted into the event list.

The function `ssq3()` in the `Sim101` library for R is the implementation of a next-event simulation model for a single-server queue with exponentially distributed interarrival times and uniformly distributed service times. In addition,

```

while (t < T) {
  e = GetNextEvent();
  t = e.time;
  if (e.type == ARRIVAL)
    numberInNode++;
    schedule next arrival;
  else
    numberInNode--;
    schedule next completion of service;
}

```

Figure 4: Basic next-event algorithm for a single-server queue model.

`ssq3()` adds a third event type—feedback of a job into the queue for additional service. Several measures of performance can be captured for this particular system, such as the average delay in the queue and the utilization of the server. The `ssq3()` function has seven arguments with default values:

- `seed = 123456789`, the random number seed,
- `MaxTime = NULL`, the maximum time for the simulation,
- `MaxJobs = NULL`, the number of jobs to be processed during the simulation run,
- `interarrivalMean = 2.0`, the mean interarrival time,
- `minService = 1.0`, the minimum service time,
- `maxService = 2.0`, the maximum service time, and
- `feedbackProb = 0.0`, the probability that a job feeds back for additional service.

Note that exactly one of the `MaxTime` and `MaxJobs` should be initialized to a value other than the default `NULL` value. If the default interarrival and service parameters are used, then the jobs arrive to the queue every 2.0 time units on average, and are serviced in $(1.0+2.0)/2 = 1.5$ time units on average. The time units in the simulation are arbitrary; it is equally sensible to think of them as minutes or hours.

The program can be executed using 10000 jobs and otherwise default parameters as in the following example. The system statistics are returned as an R list.

```

> ssq3(maxJobs = 10000)
$jobsProcessed
[1] 10000
$averageInterarrival
[1] 1.9954
$averageService
[1] 1.5034
$averageDelay
[1] 2.4149
$averageWait
[1] 3.9183
$averageNumberInNode
[1] 1.9635
$averageNumberInQueue
[1] 1.2101
$utilization
[1] 0.7533

```

The fact that the average interarrival time (1.9954) is slightly lower than the theoretical value (2.0) means that this particular run of the simulation was slightly more congested than average. The fact that the average service time (1.5034) is just slightly more than the theoretical value (1.5) means that the service was slightly slower than expected for this particular run of the simulation. Similar to that suggested for the Monte Carlo example, we can also look for consistency checks to

provide a sense of assurance that the results are correct. The average wait (queuing delay time plus service time) in the queue node is 3.9183, and this is approximately 1.5 time units (the theoretical average service time) more than the average delay in the queue, as expected. The average wait in the system (3.9183) is the sum of the average delay in the queue (2.4149) and the average service time (1.5034). The final three lines display time-averaged statistics: the average number in the queueing node (1.9635), the average number in the queue (1.2101) and the utilization of the server (75.33%) over the course of the simulation.

Similar to that shown in the Monte Carlo example, the default parameters can be modified to yield quantitatively different simulation results. If subsequent operations on any of the values in the list are required, the individual values can be easily pulled from the list and stored as scalars, e.g.,

```
> l = ssq3(maxJobs = 10000)
> w_bar = l$averageWait
> x_bar = l$utilization
```

Subsequent function calls can be applied to the list `l` or to the scalars `w_bar` and `x_bar` (see examples in Section 3.4).

5 SUMMARY

The R statistical language is suitable for an introduction to simulation fundamentals because R is easy to use and provides a wide range of statistical procedures, built-in functions, and graphics capabilities. As part of the Simulation 101 workshop, we have developed a collection of simulation tools and programs specifically for R. The programs are not general purpose, but are intended to be instructional tools for learning the concepts of simulation. The main disadvantage is that, because R is an interpreted language, the R versions of the simulation programs will execute more slowly than corresponding programs written in a compiled high-level language. This paper outlines the downloading, installation, and execution of R and the associated simulation software. In addition, this paper provides an introduction to simulation by briefly introducing Monte Carlo and next-event simulation both in concept and through software examples in R.

REFERENCES

- Bengtsson, H. 2003, March. The R.oo package – object-oriented programming with references using standard R code. In *Proceedings of the 3rd International Workshop on Distributed Statistical Computing (DSC 2003)*, ed. K. Hornik, F. Leisch, and A. Zeileis. Vienna, Austria.
- Leemis, L., and S. Park. 2006. *Discrete-event simulation: A first course*. Upper Saddle River, NJ: Pearson Prentice Hall.
- R Development Core Team 2009, June. Writing R extensions. Available via <http://cran.r-project.org/manuals.html> [accessed July 2009].
- The R Foundation 2009. The R project for statistical computing. Available via <http://www.r-project.org/> [accessed July 2009].

AUTHOR BIOGRAPHIES

BARRY LAWSON is an Associate Professor of Computer Science in the Department of Mathematics and Computer Science at University of Richmond. He received Ph.D. and M.S. degrees in Computer Science from The College of William & Mary, and a B.S. degree in Mathematics and Computer Information Systems from University of Virginia's College at Wise. His research interests include computer security, scheduling, performance evaluation, and simulation. He previously worked in the Simulation Systems Branch laboratory at NASA Langley in Hampton, VA. He is a member of ACM and IEEE. His email address is blawson@richmond.edu.

LAWRENCE LEEMIS is a professor in the Mathematics Department at The College of William & Mary. He received his B.S. and M.S. degrees in Mathematics and his Ph.D. in Industrial Engineering from Purdue University. He has also taught at Baylor University, The University of Oklahoma, and Purdue University. His consulting, short course, and research contract work includes contracts with Air Logistic Command, Argonne National Laboratory, AT&T, Delco Electronics, Department of Defense (Army, Navy), Federal Aviation Administration, ICASE, Komag, Leibherr, Magnetic Peripherals, NASA/Langley Research Center, Tinker Air Force Base, and Woodmizer. His research and teaching interests are in reliability and simulation. He is a member of ASA, IIE, and INFORMS. His email address is leemis@math.wm.edu.