Proceedings of the 2008 Winter Simulation Conference
S. J. Mason, R. R. Hill, L. Mönch, O. Rose, T. Jefferson, J. W. Fowler eds.

MONTE CARLO AND DISCRETE-EVENT SIMULATIONS IN C AND R

Barry Lawson

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

Lawrence Leemis

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

ABSTRACT

The Monte Carlo and discrete-event simulation code associated with the Simulation 101 pre-conference workshop (offered at the 2006, 2007, and 2008 Winter Simulation Conferences) is available in both C and R. This paper begins with general instructions for downloading, compiling, and executing the software. This is followed by detailed explanations of two programs that are representative of the software suite: `craps` uses Monte Carlo simulation to estimate the probability of winning the dice game Craps, and `ssq2` uses discrete-event simulation to estimate several measures of performance associated with a single-server queue.

1 INTRODUCTION

This paper discusses the use of the simulation software provided with the Simulation 101 workshop and associated with the introductory simulation textbook by Leemis and Park (2006). The complete suite of Monte Carlo and discrete-event simulation programs have been written in C and Java. A subset of the programs have been converted to R for use in the Simulation 101 workshop. We discuss only the C and R software in this paper.

2 THE C SOFTWARE

This section describes where to obtain and how to compile and execute the ANSI C version of the simulation software.

2.1 Obtaining the C Simulation Software

The C version of the software is freely available via ⟨`http://math.wm.edu/~leemis/DES_1e_SourceCode.zip`⟩ as a Windows-friendly zip file or via ⟨`http://math.wm.edu/~leemis/DES_1e_SourceCode.tgz`⟩ as a GNU-zip tar file. Download the file corresponding to the archive type of your choice. To extract the contents of the zip version, use any standard zip utility (e.g., WinZip for Windows, Stuffit Expander for Mac OS X, `zip` for Unix/Linux). To extract the contents of the GNU-zip tar file, execute the following command:

```
tar -xzvf DES_1e_SourceCode.tgz
```

The source code will be placed into a subdirectory named `DES_1e_SourceCode/` within the directory from which you initiate the extraction process.

2.2 Compiling the C Simulation Software

The software is ANSI C compliant — it be compiled by any compiler or development environment that supports ANSI C compilation.

We recommend using the GNU `gcc` compiler. The `Makefile` provided with the software assumes that `gcc` is the default compiler. If you use `gcc` and the associated `make` utility, then the following are commands of interest.

- `make`: Compiles and links all software, creating any of the executables that do not exist and executables for any of the source files that were modified since the last compile. The make utility creates executable files having the same names as the associated source files (e.g., the source file `galileo.c` results in the executable file `galileo`).
- `make clean`: Removes all executable and object files.
- `make` *programName*: Compiles and links only those files necessary for *programName*.

If you are not using `make`, then you will need to configure your compiler to appropriately link and compile the necessary software. When compiling any of the provided source files that include local versions of header (`.h`) file(s), you will need to include the corresponding `.c` file(s) in the compilation and linking process. For example, `ssq4.c`

978-1-4244-2708-6/08/$25.00 ©2008 IEEE

includes both `rngs.h` and `rvgs.h` and so the compilation and linking process must include `rngs.c` and `rvgs.c`:

```
gcc -o ssq4 ssq4.c rngs.c rvgs.c -lm
```

When compiling any source file (e.g., `ssq4.c`) that includes the math library header file `math.h`, you may need to explicitly signify to the compiler to include the math library in the linking process, typically with a `-lm` flag (as in the example above).

2.3 Executing the C Simulation Software

To execute any of the programs, simply provide the name of the executable as a command, e.g., `./ssq4`. (Note that some of the source files are to be used as libraries and do not directly create an executable.)

3 THE R SOFTWARE

R (The R Foundation 2008) 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 returning the gamma function.

The primary advantage of using R instead of C for simulation work lies in R's free-of-charge and easy-to-use statistical procedures, built-in functions, and graphics capabilities. The primary disadvantage of using R instead of C is that R is an interpreted language — in general, simulation programs will execute more slowly when written in R rather than C (sometimes dramatically so). Nonetheless, because of its ease of use and its many built-in statistical capabilities, we have written a subset of the simulation software in R for the current year's Simulation 101 workshop.

3.1 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.

3.2 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.)

Finally, now (and every time R is started and you need `R.oo` functionality) execute the command

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

to load the add-on `R.oo` package in your current session of R.

3.3 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.

Finally, now (and every time R is started and you need the simulation software functionality) execute the command

```
library(Sim101)
```

to load the add-on `Sim101` package in your current session of R. If the R prompt returns with no messages (or at most a warning message about the package being built using

a newer version of R), the `Sim101` package has been successfully loaded.

3.4 Libraries/Functions in the R Simulation Software

The R version of the simulation software was written with the goal of remaining consistent with the C version of the software while simultaneously leveraging R's advantages. For this reason, we have written in R object-oriented versions of the same random number generation libraries that are present in the C version. (Note these "libraries" in R 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.) In this way, the R and C versions of the simulation software are capable of using the same random number sequences for their stochastic components, thereby producing exactly the same output. For each library, the functions implemented in the C version are also implemented in the R version. Use R's online help (e.g., `help(Rng)`) for more details.

Rng: an object-oriented implementation of a single-stream Lehmer random number generator.

Rngs: an object-oriented implementation of a multiple-stream Lehmer random number generator.

Rvgs: an object-oriented implementation for generating random variates from six discrete and seven continuous distributions.

Rvms: an object-oriented implementation for evaluating the probability density functions, cumulative distribution functions, and inverse distribution functions for the distributions provided in **Rvgs**.

For the workshop, we have also provided in R a subset of the programs available in C, including three Monte Carlo simulations, three discrete-event simulations, and three utilities. 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 $3, 4, \ldots, 18$ 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: plots a histogram of data drawn from a continuous population.

ddh: plots a histogram of data drawn from a discrete population.

estimate: computes a confidence interval estimate for a data set.

In addition, we have provided implementations of the inverse distribution functions for seven distributions: **idfBinomial**, **idfExponential**, **idfGeometric**, **idfLognormal**, **idfNormal**, **idfPascal**, and **idfUniform**. Each of these functions evaluates the corresponding inverse distribution and displays an intuitive graphical representation. For more details on any of these functions, use R's online help (e.g., `help(idfPascal)`).

3.5 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
```

displays the source code for the function `galileo()`. The first line of the function indicates that there are two parameters, `seed` (which defines the random number generator seed) and `N` (which defines the number of Monte Carlo replications), which have default values `12345` and `1000` respectively.

In order to modify one of the simulation programs, you should first dump to a file the source code of the corresponding 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.,

```
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.)

3.6 Executing the R Simulation Software

In order to execute a simulation program, type the corresponding function name with associated arguments. 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. If subsequent operations on the estimates are required, they can be placed into a vector, e.g.,

```
x = galileo()
```

Subsequent function calls from the R language (e.g., `sum(x)` to confirm that this is 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 from the simulation software (e.g., `cdh(x)` to produce a continuous data histogram of the probabilities) can be applied to `x`.

3.7 Interfacing R with C Simulation Software

Because the simulation software will execute more slowly in R (an interpreted language) than in C, especially for long simulations, users may be interested in having R interface with the C simulation programs. This is accomplished by writing for each simulation program an R function that accepts arguments from the user, invokes the corresponding C program passing in those arguments, and returns 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 2008).

4 MONTE CARLO SIMULATION

We illustrate some of the details associated with the Monte Carlo approach using the `craps()` function in R.

The gambling game known as "craps" involves tossing a pair of fair dice one or more times and observing sum of the two up faces (i.e., the total number of pips showing). 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.

An *Equilikely*$(1,6)$ random variate is used to model the roll of a single fair die. The algorithm that follows 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);
```

The algorithm has been implemented in the C program `craps` and in the R function `craps()`. We discuss the R version here. The function `Roll` returns the outcome when a pair of dice is tossed by summing the results of two calls to *Equilikely*$(1,6)$ (available in R using `Rvgs` from the `Sim101` library). The `switch` statement is used to identify the result of the game based on the outcome of the first roll.

The function `craps()` as implemented in R has two arguments: the number of Monte Carlo simulation replications defaults to `N = 1000` and the random number generator seed defaults to `seed = 12345`. So calling `craps` with no arguments amounts to simply typing

```
craps()
```

which returns the probability estimate as a scalar

```
[1] 0.522
```

These particular 1000 replications might lead someone to believe that Craps is a game that can be played for long-term profit.

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 initial seed). In order to get a different set of Monte Carlo replications, the seed must be changed. For example, the statements

```
craps(seed = 987654321)
craps(seed = 54321)
```

yield winning probabilities 0.510 and 0.530. This rather surprising streak of three values above 0.500 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. The statements

```
craps(N = 10000, seed = 987654321)
craps(N = 10000, seed = 123456789)
craps(N = 10000, seed = 555555555)
```

yield winning probabilities 0.497, 0.485, and 0.502, which are scattered about the theoretical value 0.493, as expected. Our three earlier unexpectedly high results with `N = 1000` were simply a function of random sampling variability (often the case when the number of replications is too low).

5 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. The function `ssq2()` in the `Sim101` library for R is the implementation of a model for a single server queue with exponentially distributed interarrival times and uniformly distributed service times. The assumptions in this model are:

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

A number of measures of performance will be captured for this particular system, such as the average delay in the queue and the utilization of the server. The `ssq2()` function has five arguments. The arguments and their default values are

- `seed = 123456789`, the random number seed,
- `MaxJobs = 10000`, the number of jobs 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.

If the default 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 is executed with the default parameters using the command

```
ssq2()
```

which returns a list (a commonly used data structure in R) as shown below.

```
$jobsProcessed
[1] 10000

$averageInterarrival
[1] 2.0161

$averageService
[1] 1.4981

$averageDelay
[1] 2.3616

$averageWait
[1] 3.8597

$averageNumberInNode
[1] 1.9143

$averageNumberInQueue
[1] 1.1713

$utilization
[1] 0.743
```

The fact that the average interarrival time (2.0161) is slightly above the theoretical value (2.0) means that this particular run of the simulation was slightly less congested than average. The average wait (queuing delay time plus service time) in the queue node is 3.8597, and this is approximately 1.5 time units (the theoretical average service time) more than the average delay in the queue, as expected. The fact that the average service time (1.4981) is just slightly smaller than the theoretical value (1.5) means that the service was slightly faster than expected for this particular run of the simulation. As an important simulation consistency

check, notice that the average wait in the system (3.8597) is the sum of the average delay in the queue (2.3616) and the average service time (1.4981). The final three lines display time-averaged statistics: the average number in the queueing node (1.9143), the average number in the queue (1.1713) and the utilization of the server (74.3%) over the course of the simulation.

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 = ssq2()
w_bar = l$averageWait
s_bar = l$averageService
x_bar = l$utilization
```

Subsequent function calls from the R language or from the simulation software can be applied to the list `l` or to the scalars `w_bar`, `s_bar`, and `x_bar` (similar to that discussed in Section 3.6).

6 SUMMARY

The Monte Carlo and discrete-event simulation programs associated with the Simulation 101 workshop are available in ANSI C, Java, R, and in C interfacing with R. Only the ANSI C and R versions are described in this paper. The programs described are not general purpose, but are intended to be instructional tools for learning the concepts of simulation. The R version of the software was written to maintain consistency with the C version while leveraging the statistical and graphical capabilities of R. Both versions are capable of producing exactly the same results. The main disadvantage is that, because R is an interpreted language, the R versions of the simulation programs will execute more slowly than the corresponding C programs. Because of the capabilities and user-friendly nature of R, the R version will be used in the Simulation 101 workshop.

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. M., and S. K. Park. 2006. *Discrete-event simulation: A first course*. Upper Saddle River, NJ: Pearson Prentice Hall.

R Development Core Team 2008, August. Writing R extensions. Available via ⟨http://cran.r-project.org/manuals.html⟩ [accessed August 1, 2008].

The R Foundation 2008. The R project for statistical computing. Available via ⟨http://www.r-project.org/⟩ [accessed August 1, 2008].

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, parallel and distributed computing, 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, IEEE, IEEE Computer Society, and IEEE Systems, Man, and Cybernetics Society (IEEE/SMC). 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⟩.