Winter 3-13-2013

Simulation Insights Using R

Boyan Kostadinov
CUNY New York City College of Technology

SIMULATION INSIGHTS USING R

Boyan Kostadinov
Department of Mathematics
New York City College of Technology, CUNY
Brooklyn, NY 11201, USA
bkostadinov@citytech.cuny.edu

SIMULATION INSIGHTS USING R

Abstract: This article attempts to introduce the reader to computational thinking and solving problems involving randomness. The main technique being employed is the Monte Carlo method, using the freely available software R for Statistical Computing. The author illustrates the computer simulation approach by focusing on several problems of increasing difficulty. The simulation techniques and the specific problems discussed in this article would be of interest to STEM students and instructors, teaching courses in Monte Carlo simulations, stochastic modeling, probability and statistics. The R code for all problems is discussed in full detail so that the reader can get a taste of the functionality and unique simulation and visualization features that R offers.

Keywords: Monte Carlo simulations, R software, student projects, probability, sampling, central limit theorem, birthday problem

INTRODUCTION

Many important applied and pure research questions in science today involve computing as well as theory. Computing can often bring additional insight and understanding that theory alone cannot achieve.

We believe that computational thinking can be understood as a fundamental analytical skill in the 21st century that everyone can use to solve problems in all fields of science and education. In 2010, the National Research Council Committee on Computational Thinking published a report on the scope and nature of computational thinking, [4]. The study committee explored the idea that the ever increasing use of computational devices must be supported by the widespread promulgation of computational thinking.

This is a vision for the 21st century shared by many who believe that computational problem-solving is an essential skill in this technology age and should be introduced at the K-12 level, and further supported and enhanced by college curricula, in many different subjects. Some of the most vigorous supporters of this vision are the Center for Computational Thinking at Carnegie Mellon, organized by Jeannette Wing and funded by Microsoft, [10], SHODOR, a national resource for computational science education [8], and not surprisingly, Google is one of the big technology companies committed to promoting computational thinking as an essential 21st century skill, [1].

In this article, we explore the power of computational thinking and problem-solving by investigating five problems of increasing difficulty, using Monte Carlo simulations. We have used most of these problems as computational projects for the students in our applied mathematics program, mostly in courses exploring Monte Carlo simulations and stochastic modeling but the problems we discuss along with the simulation techniques are also suitable for courses in applied probability and statistics, as well as computer science courses.

The Monte Carlo method dates back to the 1940's and was developed by John von Neumann, Stanislaw Ulam and Nicholas Metropolis while working on the Manhattan Project at the Los Alamos National Lab. They were trying to solve very difficult problems of neutron diffusion, for which no analytical solutions were available, when the idea of using repeated random sampling to solving probabilistic problems came to Stanislaw Ulam, while playing a game of solitaire. See [3] for more details on the history of the Monte Carlo method. The essence of the method is repeated sampling of random numbers, from a given probability distribution, imposed by the problem. These sampled numbers are then used to find simulation solutions to problems of any complexity, involving randomness.

We program all computer simulations using the freely available software environment R for Statistical Computing. Visit the website [5] of the R Project for more information on how to download, install and use R on different platforms. They also offer many free tutorials and manuals.

The books [2] and [9] are good introductions to Monte Carlo simulations, using R and MATLAB, respectively, while the book [6] is a good introduction to the applications of statistics using R. Another useful resource worth mentioning is RStudio, which is a free and open source integrated development environment for R. You can run it on your desktop (Windows, Mac, or Linux). Visit the website [7] of RStudio for more information.

We illustrate the Monte Carlo simulation approach by designing and implementing in R programming procedures that help us gain a computational insight into the following problems:

- Estimating probabilities by Monte Carlo simulations.
- Estimating the probability of meeting a date at the movie theatre.
- Estimating integrals by Monte Carlo simulations.
- Simulation insight into the Birthday Problem.
- Simulation insight into the Central Limit Theorem.

COMPUTING PROBABILITIES BY SIMULATIONS

Let U_1 and U_2 be independent random variables, uniformly distributed on the interval $(0, 1)$. Using Monte Carlo simulations, we want to estimate the probability $P(U_1U_2 < 0.5)$.

Of course, this problem can be solved analytically, and the exact answer is $\frac{1+\ln(2)}{2} \approx 0.8466$. We leave the analytical solution as a challenge to the interested reader and we focus on a simulation approach using R.

From the File menu in the R Graphical User Interface (GUI), we first create a new document (script) in the editor and we can then write our R code there. It only takes a couple of lines to estimate the required probability by simulation:

```
u1=runif(1e6); u2=runif(1e6) # two samples from U(0,1)   (1)
mean(u1*u2<0.5) # an estimate of P(U1U2<0.5)             (2)
```

Note that a hashtag symbol # marks the beginning of a comment in the code that is ignored by R. The first line of code, (1), generates two inde-

pendent samples of size one million from $U(0,1)$, the standard uniform distribution. This is accomplished by the R command `runif(1e6)`.

This leads to an obvious question: What if we had used not a million but fewer simulations? The underlying mathematical analysis shows that the error of the Monte Carlo simulation method decreases as $N^{-1/2}$, where N is the number of simulations. For example, going from $N = 10^4$ to $N = 10^6$ should reduce the error by a factor of about 10. That is why, in all problems with a single straightforward simulation, we use 10^6 simulations in order to achieve high accuracy. However, some problems require one simulation on the top of another, which is the case of simulating the sampling distribution of a random variable of interest. In this case, we would typically use 10^4 simulations in order to speed up the calculations and get a result within a few minutes. We should also mention that for some visualizations, we would use instead 10^5 simulations in order to have smoother graphs and thus improve the quality of the plots, but this comes at a computational cost and it could be time-consuming, depending on the speed of the computer being used.

If we want to find more information and examples about a given R function, just type `?name` at the prompt, where `name` stands for the name of the function. For example, type `?runif` at the R prompt for help on this function, which explains the optional parameters and their default values. In this command, `r` stands for *random* and `unif` stands for *uniform*. In general, the R command for any other random number generator follows the same convention, namely `rname(n,p1,p2,...)`, where `name` ranges over `unif`, `binom`, `geom`, `hyper`, `pois`, `exp`, `chisq`, `gamma`, `norm`, `t` etc., `n` is the number of requested observations, and `p1,p2,...` are the parameters that define the given distribution.

The code in line (2), estimates the required probability by computing the relative frequency of the event $(U_1U_2 < 0.5)$, that is, the fraction of times the event occurs. The R code `u1*u2<0.5` returns a vector of logical values, `TRUE` or `FALSE`, depending on whether the inequality is satisfied or not. In R, the multiplication of two vectors of the same size, like `u1*u2`, is performed entry-wise, and the result is a vector of the same size as the original vectors. Thus, the vector `u1*u2` represents a

sample of U_1U_2. In R, we can perform any arithmetic operations on a vector of logical values, in which case, R coerces `TRUE` into the number 1 and `FALSE` into the number 0. In particular, taking the mean of a vector of logical values is equivalent to computing the fraction of `TRUE` values. This way, we obtain the relative frequency of the event that the inequality `u1*u2<0.5` is satisfied, which serves as an estimate for the required probability.

We can execute the lines of R code we have written in the script by first selecting these lines and then pressing *Command+Return* on a Mac or *Ctrl+R* on a Windows PC. The result of this simulation, from line (2), is given at the prompt of the R GUI window, in the form:

```
[1] 0.846609
```

MEETING A DATE AT THE CINEMA

Suppose you invite a date to see a movie together but you both realize that you will arrive at the movie theatre at some random time between 8:00 p.m. and 8:30 p.m.. Assume that the arrival time, with respect to 8:00 p.m., for each of you, is uniformly distributed between 0 and 30 minutes. However, you both agree that whoever arrives first will wait for 10 minutes only and then leave. What is the probability that you will actually meet your date at the movie theatre? The goal is to estimate this probability using computer simulations.

The two arrival times, X and Y, relative to 8:00 p.m., are independent and uniformly distributed random variables on the interval $[0, 30]$. The probability that the distance between the two arrival times is at most 10, $P(|X - Y| \leq 10)$, can be estimated in a single line of R code.

Here is the entire simulation solution in R that gives a *point estimate* of the required probability:

```
(1)[1] x = runif(1e6,0,30); y = runif(1e6,0,30)
(2) mean(abs(x-y)<=10)
```

[1]Before you run the given R code, make sure you remove the line numbering.

The numerical value returned by line (2) is 0.5565, which is an estimate of the required probability. Note the similarity between the mathematical expression and the R code:

$$P(|X - Y| \leq 10) \longleftrightarrow \texttt{mean(abs(x-y)<=10)}$$

The code in line (1) generates two independent samples of size one million (n = `1e6`) from the uniform distribution on $[0, 30]$, using the R function `runif(1e6,0,30)`. In line (2), we use the built-in function `abs()`, which takes the absolute value of a vector, entry-wise. Applying the comparison operator `<=` forms a logical expression, `abs(x-y)<=10`, and the result is a vector of size equal to the size of the `x` and `y` vectors, having only the logical values `TRUE`, when the inequality is satisfied, and `FALSE` otherwise. As discussed earlier, R coerces `FALSE` into 0 and `TRUE` into 1, so the 0's are corresponding to the cases where the inequality `|x-y|<=10` is not satisfied and the 1's are corresponding to the cases where the inequality is satisfied. The R function `mean()` computes the arithmetic average of a vector. In line (2), taking the mean of a vector of logical values, `mean(abs(x-y)<=10)`, returns the *relative frequency* ($= \frac{1}{n}\sum 1$'s) of the event that `|x-y|<=10` is satisfied, which is an estimate of the probability that the two will meet.

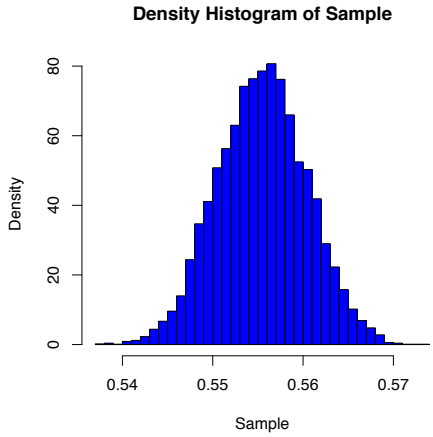

Figure 1. Sampling distribution of the probability estimator.

In fact, we can easily generate the entire *sampling distribution* of this probability estimator by packaging the two lines of code in (1) and (2) into a custom R function `f()` with no arguments:

```
f<-function(){
      x=runif(1e4,0,30); y=runif(1e4,0,30)
      return(mean(abs(x-y)<=10))}
```

A better point estimate of the required probability is obtained by simulating its sampling distribution and computing its mean:

```
Sample = replicate(1e4,f())
hist(Sample,50,freq=FALSE,col='blue')
mean(Sample)
```

The numerical value, returned by `mean(Sample)` is 0.5555. The R code `Sample=replicate(1e4,f())` generates the `Sample` vector of size 10^4, as the function `f()` is called 10^4 times by the built-in function `replicate()` and the results collected in the `Sample`. We can visualize the sampling distribution of this probability estimator by plotting the *density histogram* (with 50 bins) of the `Sample`, using the R function `hist()` with the optional argument `freq` set to `FALSE`, see Figure 1.

To better appreciate the one-line simulation estimate using R, let us find the exact answer by using the standard analytical methods of probability. This is a two-dimensional probability problem for the two independent random variables, $X, Y \sim U[0, 30]$, whose ranges form the sample space of the problem, the 30-by-30 square. The conditions $|X - Y| \leq 10,\ X, Y \in [0, 30]$ specifying a meeting, if satisfied, can be represented geometrically by the dark shaded region $ABCDEF$, inside the square. Since X and Y are independent, the probability to fall into the dark shaded region is the fraction of its area from the total area of the 30-by-30 square. The area of the dark shaded region is the area of the quadrilateral $GCHF$ minus the sum of the areas of the two triangles $\triangle ABG$ and $\triangle EDH$ (see Figure 2). Since the two triangles have the same areas and the area of the quadrilateral is the length of the side

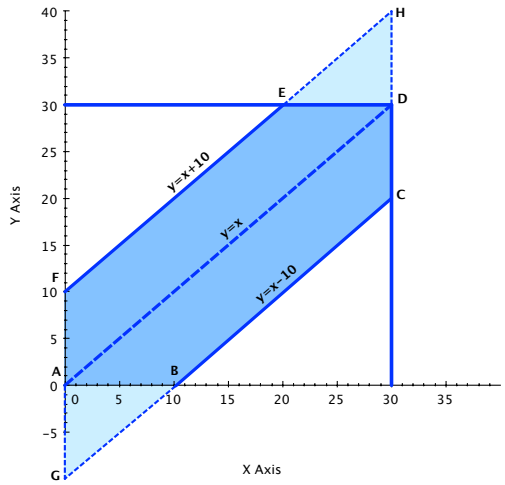

Figure 2. The dark shaded region $|X - Y| \leq 10$ in the sample space.

FG ($= 20$) times the length of the altitude to this side ($= 30$), we get:

$$\text{Area}(ABCDEF) = \text{Area}(GCHF) - 2\text{Area}(\triangle ABG) =$$
$$= 20 \times 30 - 2(10^2)/2 = 500$$

Finally, the exact answer for the required probability is $\frac{500}{30\times 30} = 0.5555$, which is in a good agreement with the value estimated by simulation.

MONTE CARLO INTEGRATION

Monte Carlo simulations can be used to estimate deterministic integrals. Consider, for example, the following integral:

$$I = \int_{-1}^{1} e^{-x^2+x-1} dx$$

Of course, we can evaluate this integral numerically but we can also estimate I using a Monte Carlo approach. The idea is to represent I as an expectation. Let $X \sim U(-1, 1)$ be a uniform random variable on $(-1, 1)$. The density function of X is $f_X(x) = \frac{1}{2}$ and if $g(x)$ is any integrable function, then the expected value $E[g(X)] = \int_{-1}^{1} f_X(x)g(x)dx$. The key

observation is that we can rewrite I in terms of an expectation:

$$I = 2\int_{-1}^{1} \frac{1}{2} e^{-x^2+x-1} dx = 2\int_{-1}^{1} f_X(x) e^{-x^2+x-1} dx = 2E[e^{-X^2+X-1}]$$

Now, we can estimate the expectation by Monte Carlo simulations. First, we simulate a sample x of size one million from $X \sim U(-1, 1)$ and then we can estimate the expected value $E[g(X)]$ by averaging over all values of the function g applied to the given sample. The entire simulation in R that gives an estimate of the integral I is given here:

```
> x=runif(1e6,-1,1); 2*mean(exp(-x^2+x-1))
```

The estimate obtained this way is `0.62263`. Note that the value of I obtained by a deterministic numerical technique using Mathematica or Wolfram Alpha is 0.62233. Keep in mind that the simulated estimate, represents a random variable since it is based on random samples. We can further improve the accuracy of our point estimate by simulating the sampling distribution of our estimator. We just need to use the built-in function `replicate()`, as we done before, and compute a sample of estimated I values, by replicating our simulation many times, and computing the mean of this sample (the variance would give the error).

In general, the Monte Carlo integration technique applied to the integral $I = \int_a^b g(x)dx$ leads to estimating an expectation by simulation:

$$I = (b-a)E[g(X)], \quad X \sim U(a, b),$$

Thus, if $x_1, x_2, \ldots, x_n$ is a particular sample of the random variable X, then our estimate $\hat{I}$ is obtained by simple averaging:

$$\hat{I} = \frac{1}{n}\sum_{k=1}^{n} g(x_k)(b-a).$$

The accuracy of Monte Carlo integration is independent of dimension, therefore this simulation technique becomes preferable to other deterministic numerical techniques in higher dimensions.

SIMULATION INSIGHT INTO THE BIRTHDAY PROBLEM

The classical birthday problem investigates the least number of persons required if the probability exceeds 0.5 that two or more of them have

the same birthday. Year of birth need not match, only the day and month are relevant. We assume that February 29 is ignored as a possible birthday and that the other 365 days are regarded as equally likely and independent birth dates. Clearly, if we have 366 people, then at least two will have the same birthday, for sure. We shall implement in R a simulation solution, which shows that even with 57 people, the probability of having at least two common birthdays is already around 99%, and having just 23 people implies that the probability exceeds 50% that two or more of them have the same birthday.

The simulation approach is based on creating an R function `test(n)`, $n > 1$, which returns 1 or 0, depending on whether we have at least one birthday pairing among the n people or not. Here is the R code for the function: `test(n)`:

```
test<-function(n){ # n>1
        sample=sample(1:365,n,replace=TRUE)
        temp=sort(sample)
        flag=any(temp[1:(n-1)]-temp[2:n]==0)
        return(flag)}
```

This function simulates a single experiment by generating a single random sample of size n from the set $1, \ldots, 365$, with replacements. This is accomplished by the R command `sample(1:365,n,replace=TRUE)`. We then sort the sample using the R command `sort(sample)` and we hold the sorted sample in the variable `temp`. The essential line here is the one that defines the variable `flag`. This line implements the idea that if there are at least two common birthdays in the sample, after we sort it, we would have somewhere equal consecutive numbers. The question is how to detect the presence of such equal consecutive numbers. We can extract from the vector `temp` the first $n-1$ entries by specifying the range of indices `1:(n-1)` inside square brackets: `temp[1:(n-1)]`. Similarly, we can extract the shifted by 1 range of entries `2:n`, using `temp[2:n]`. Now, taking the difference of these sub-vectors of size $n-1$ will produce zeros only if the sorted vector `temp` has equal consecutive integers, somewhere. Using the comparison operator `==` (equal to) in

`temp[1:(n-1)]-temp[2:n]==0` returns a vector of size $n-1$ of logical values `TRUE` and `FALSE`, depending on whether the left-hand side is zero or not. The R command `any()` when applied to a vector of logical values, returns `TRUE` or `FALSE`, depending on whether the vector has at least one `TRUE` value or not. Thus, the function returns the value of `flag`, which is either `TRUE`, if there are at least two common birthdays or `FALSE` otherwise. If the value of the function `test(n)`, being a logical value, is later used in some algebraic expression then `TRUE` is coerced into the integer 1 and `FALSE` into 0. Once we have the ability to simulate a single random experiment by calling the function `test(n)`, for a given number of people `n`, we can easily replicate the random experiment `N` times by simply calling the R function `replicate()`. The R statement `indicator=replicate(N,test(n))` calls the function `test(n)`, `N` times and places the output of the function, which is either 1 or 0, in the vector `indicator`, of size `N`. Taking the mean of the `indicator` vector gives an estimate of the relative frequency of the event that there are at least two common birthdays among the given `n` people. It is now easy to build the probability curve of at least two common birthdays as a function of the number of people. We just need a `for loop`, which executes all statements in the body of the loop a given number of times.

For example, Figure 3 was created by building a probability curve with 69 points, corresponding to different number of people, $n = 2, \ldots, 70$, using a `for loop`. The choice for the number of simulations `nsim=1e5` was imposed by the requirement to generate a smooth enough curve. However, this choice along with the 69 points to be plotted requires 15-20 min to simulate, depending on the speed of the computer.

```
# Probability Curve Simulation
m=70 # number of people
nsim=1e5 # number of simulations to get a smooth curve
curve=rep(0,m-1) # create a vector of zeros of size m-1
# number of people n must be at least 2
for (n in 2:m){ # loop will be executed m-1 times, n=2,...,m
   indicator=replicate(nsim,test(n)) # vector of 0's and 1's
   curve[n-1]=mean(indicator) # populate m-1 entries of the curve
```

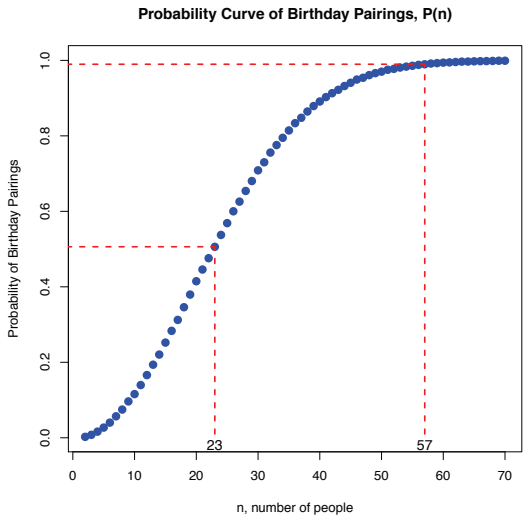

Figure 3. Probability curve of birthday pairings.

}

Finally, the curve can be plotted using the R command `plot()`, which has many optional parameters. Here is the main line of code that generated Figure 3:

```
plot(2:m,curve,type="p",lwd=3,pch=21,bg="blue",col="blue",xlab="n,
        number of people",ylab="Probability of Birthday Pairings",
        main="Probability Curve of Birthday Pairings, P(n)")
```

There are a couple of small details in Figure 3 that were generated using the R commands `segments()` and `text()`. In summary, the simulation gives that in a pool of 23 people, the probability that there are at least two common birthdays is 0.5061. It is worth noting here that the actual value, based on exact analytical calculations, is 0.5073.

SIMULATION INSIGHT INTO THE CENTRAL LIMIT THEOREM

The central limit theorem is one of the most important results in probability theory. Intuitively, it says that a large and properly normalized

sum of independent and identically distributed (i.i.d.) random variables having finite mean and variance, will always have approximately a standard normal distribution.

More precisely, suppose that $X_1, X_2, \ldots$ is a sequence of i.i.d. random variables, having a common finite mean $E(X_k) = \mu$ and finite variance $Var(X_k) = \sigma^2 > 0$, for any k. Let $S_n = X_1 + \cdots + X_n$ be the partial sum, whose mean is $E(S_n) = n\mu$, by linearity of expectation, and variance is:

$$Var(S_n) = Var\left(\sum_{k=1}^{n} X_k\right) = \sum_{k=1}^{n} Var(X_k) = n\sigma^2,$$

where the second equality is valid because of independence. The central limit theorem is concerned with the distribution of the normalized sum:

$$Z_n = \frac{S_n - E(S_n)}{\sqrt{Var(S_n)}} = \frac{S_n - n\mu}{\sqrt{n}\sigma}, \quad \text{where } \sigma = \sqrt{\sigma^2}.$$

The normalization consists of transforming the random variable S_n into Z_n, the latter having mean zero and variance one:

$$E(Z_n) = 0, \quad Var(Z_n) = 1,$$

thanks to the standard properties of expectation and variance. This normalization is also known as *standardizing* a random variable, so that it has mean 0 and variance 1. Note that the random variable Z_n and the standard normal variable $Z \sim N(0, 1)$ have both mean 0 and variance 1. The central limit theorem shows that there is a much stronger relationship: In the limit, as $n \to \infty$, the sequence $\{Z_n\}$ converges in distribution to Z, i.e. for each fixed real x, we have:

$$\lim_{n\to\infty} P(Z_n \leq x) = P(Z \leq x) = \Phi(x),$$

where $\Phi(x)$ is the standard normal cumulative distribution function.

We now present a simulation approach to visualizing the central limit theorem, applied to a sequence $X_1, X_2, \ldots$ of i.i.d. standard uniform random variables. The R code that simulates the sampling distributions of Z_1, Z_2, Z_3 and Z_{10}, and then plots their density histograms in Figure 4, is given here in full detail, with comments:

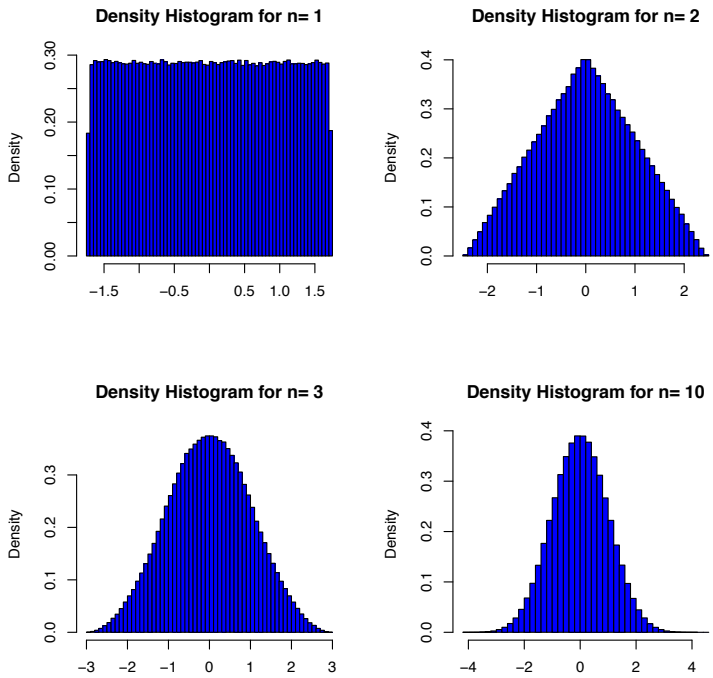

Figure 4. The density histogram of Z_n, for $n = 1, 2, 3, 10$.

```
# Simulations Visualizing the Central Limit Theorem
N=1e6 # number of sample values for each random variable
par(mfrow=c(2,2))# creates a 2x2 grid of plots filled by the loop
# n is the number of independent U(0,1) random variables
for(n in c(1,2,3,10)){  # n takes the values 1,2,3,10
  # body of the for loop
  mat=matrix(runif(n*N),nrow=n,ncol=N) # creates an nxN matrix
  S=apply(mat,2,sum)  # sum() applied to mat column-wise
  Zn=(S-n/2)/sqrt(n/12) # vector of size N, normalized sum
  hist(Zn,freq=FALSE,nclass=50,col='blue',ylab='Density',
      main=paste('Density Histogram for n=',n),xlab='') }
```

The key lines in this simulation code are the following three:

```
(1) mat=matrix(runif(n*N),nrow=n,ncol=N) # creates an nxN matrix
(2) S=apply(mat,2,sum)  # sum() applied to mat column-wise
```

```
(3) Zn=(S-n/2)/sqrt(n/12) # vector of size N, normalized sum
```

The first line, `(1)`, generates a matrix `mat` of size $n \times N$, where the n rows correspond to the n independent $U(0,1)$ random variables $X_1, \ldots, X_n$, each having N sample values. The matrix is populated by $n \times N$ random numbers sampled from the $U(0,1)$ by calling the function `runif(n*N)`. The second line, `(2)`, applies the built-in `sum()` function to each column of the matrix and returns a vector `S` of N values, representing the sampling distribution of the partial sum $S_n = X_1 + \cdots + X_n$. The visual representation of `apply(mat,2,sum)` is depicted in Figure 5.

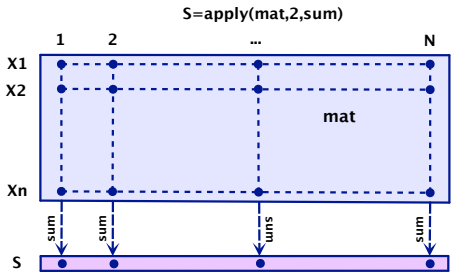

Figure 5. Visual representation of `S=apply(mat,2,sum)`

The third line of code, `(3)`, in the body of the `for loop` defines the variable `Zn`, which normalizes the partial sum `S` by subtracting the mean of the sum $E(S_n) = nE(X_1) = n/2$ and dividing the result by the standard deviation of the sum $\sqrt{Var(S_n)} = \sqrt{n/12}$, since $Var(X) = 1/12$ when $X \sim U(0,1)$. Finally, we create a density histogram of Z_n using the `hist()` function with 50 bins and frequency parameter set to `FALSE`. For more information on all arguments of the `hist()` function, just type `?hist` at the R prompt. The `for loop` repeats this procedure four times, and each time n takes on one of the values $1, 2, 3, 10$, held in the vector `c(1,2,3,10)`, created using the concatenation operator `c()`.

The result of this simulation, given in Figure 4, shows that the sample density of Z_n approaches the standard normal density very quickly, and it is almost indistinguishable from the density of $Z \sim N(0,1)$ when $n = 10$. In fact, the proportion of Z_{10} values in $(-\infty, 1.96]$,

for this simulation, equals 0.9747, and it can be obtained by using the sampling distribution of Z_n when $n = 10$ and the R command `mean(Zn<=1.96)`. On the other hand, this exact proportion for a standard normal distribution is 0.9750 and it is obtained using the R command `pnorm(1.96,mean=0,sd=1)`, which computes numerically the probability $P(Z \leq 1.96)$.

In general, the density function, the cumulative distribution function, the quantile function and random number generation for any distribution is implemented in R with the following general structure:

```
pdf: dname(x,p1,p2,...), cdf: pname(x,p1,p2,...),
quantile: qname(p,p1,p2,...), random numbers: rname(x,p1,p2,...),
```

where `x` is a single number or a vector of values or quantiles, `p` is a single number or a vector of probabilities and `p1,p2,` ... are the parameters that define a given distribution plus other optional parameters with default values. For example, the mean and the standard deviation are the two parameters that define the normal distribution, with default values of 0 and 1, respectively. The `name` should be specified for one of the many distributions in the R library: `norm`, `unif`, `binom`, `geom`, `hyper`, `pois`, `exp`, `chisq`, `gamma`, `t` and so on.

Looking back at Figure 4, it is worth mentioning that it is not a coincident the sample density of Z_2 is triangular. It turns out that the density function of the sum $S_2 = X_1 + X_2$ of two i.i.d. random variables is really the *convolution* of the density function $f(x)$ with itself: $(f * f)(x) = \int_{-\infty}^{\infty} f(x-u)f(u)du$. In our case, the distribution is $U(0,1)$ and the standard uniform density $f(x) = 1$ on $(0,1)$ and zero otherwise. The triangular density function seen in Figure 4 is a scaled and shifted version of $f * f$ to account for the normalization of the sum S_2 into Z_2. Similarly, the density function of $S_n = X_1 + \cdots + X_n$ is the convolution of the density function f with itself n times: $*^n f$, and again the density of Z_n one obtains by additional scaling and shifting that reflects the normalization of S_n into Z_n. Without going into any further details, continuing this analysis, by following a Fourier transform approach, can lead to an illuminating proof of the Central Limit Theorem, which can be

viewed as an application of Fourier Analysis rather than being a purely probabilistic result.

CONCLUSION

The main goal of this article is to illustrate the power of computational thinking and problem-solving with the help of Monte Carlo simulations, using the freely available software R for Statistical Computing.

The key ingredient in the Monte Carlo simulation approach is creating a custom R function `fun`, which simulates the random experiment of interest and returns a single realization of X, a random variable of interest. We then set `sample = replicate(N,fun)`, where the built-in `replicate()` command calls `fun` `N` times and generates a sample of size `N` from the distribution of X. We can then visualize the sampling distribution of X thus obtained with `hist(sample)`, and we can create either the density or frequency histogram by playing with the optional parameters of the built-in function `hist()`. This way, we can gain computational insight into problems with complicated analytical solutions or even problems for which no analytical solution is available and often we can program the general procedure above in just a few lines of R code. The key feature of R that allows us to create very succinct simulation programs is the ability to vectorize the R code.

ACKNOWLEDGEMENTS

I would like to thank Jai Mehta for his help in editing this article. I would also like to express my most sincere gratitude to the three anonymous referees, as well as the associate editor and the editor in chief, for their many helpful suggestions, which undoubtedly improved the quality of the paper.

REFERENCES

[1] *Exploring Computational Thinking throughout the K-12 curriculum*, `http://www.google.com/edu/computational-thinking/`

[2] Jones, O., R. Maillardet and A. Robinson, *Introduction to Scientific Programming and Simulation Using* R. CRC Press, 2009.

[3] Metropolis, N., *The Beginning of the Monte Carlo Method*, http://library.lanl.gov/cgi-bin/getfile?00326866.pdf

[4] National Research Council of the Academy of Sciences, *Report of a Workshop on the Scope and Nature of Computational Thinking*, http://books.nap.edu/catalog.php?record_id=12840

[5] R Project for Statistical Computing, http://www.r-project.org/

[6] Pruim, Randall, *Foundations and Applications of Statistics: An Introduction Using R*. AMS Vol. **13**, 2011. http://www.ams.org/bookstore-getitem/item=amstext-13

[7] RStudio, http://rstudio.org/

[8] SHODOR, a national resource for computational science education, http://www.shodor.org/

[9] Shonkwiler, R. and F. Mendivil, *Explorations in Monte Carlo Methods*. Springer Undergraduate Texts in Mathematics, 2009.

[10] Wing, Jeannette, *A Vision for the 21st Century: Computational Thinking*, CACM Vol. **49**, No. 3, March 2006, pp. 33-35, http://www.cs.cmu.edu/afs/cs/usr/wing/www/publications/Wing06.pdf
Center for Computational Thinking at Carnegie Mellon University: http://www.cs.cmu.edu/~CompThink/index.html

BIOGRAPHICAL SKETCH

Boyan Kostadinov is an Assistant Professor of Mathematics at the New York City College of Technology, a senior college of the City University of New York. He received the Robert Sorgenfrey Distinguished Teaching Award at UCLA in 2002 and earned his PhD from UCLA in 2005, under the supervision of Professor V.S.Varadarajan. He worked in the Financial Industry, in London and New York City, for a couple of years,

before happily returning to academia. He is mostly interested in applications of mathematics and his main fields of interest are Monte Carlo simulations, financial mathematics and applied Fourier calculus.