C27-28 Random Number Generation

David Housman

## Uniform Random Number Generator

Let $U$ be a random variable with a uniform continuous distribution on $[0,1)$. Here is R’s function to generate five values from $U$.

runif(5)

The R code below shows that R only generated pseudorandom numbers: specifying a seed for “random” number generation results in the same “random” numbers! This can actually be useful for code testing.

set.seed(17)
runif(5)
set.seed(17)
runif(5)

The linear congruential generator was a standard method used in computer systems because of its simplicity and speed. Starting with a seed $x\_{0}$, let $x\_{i}=ax\_{i-1}+c (mod m)$ where $a$, $c$, and $m$ are fixed integers called the multiplier, increment, and modulus, respectively. The $i$th random real in $[0,1)$ is $x\_{i}/m$.

x = 19570830 #the seed value
rlcg = function() {
 #updating the global variable
 x <<- (1103515245 \* x + 12345) %% 2^31
 x/2^31 #the random number in [0, 1]
}
tibble(x = replicate(10000, rlcg())) |>
 ggplot(aes(x = x)) +
 geom\_histogram(breaks = seq(0, 1, 0.1), color = "white")

By default, R uses a more modern and higher quality method called the Mersenne-Twister (although the user can choose among several algorithms). A more in depth examination of pseudo-random number generation would be appropriate in a second probability course or a number theory course.

## An Easy Continuous Random Number Generator

Suppose the random variable $X$ has the probability density function $f\left(x\right)=3/x^{4}$ for $1\leq x<\infty $.

1. Verify that $f$ is a pdf.
2. Determine the mean and standard deviation of $X$.
3. Define the pdf, cdf, inverse cdf, and random generator as R functions.
4. As we have seen, R was built to handle most computations with vectors as simply as with numbers. The function if is an exception, but the function ifelse is the vectorized form of the if function. R also treats -Inf and Inf as numbers when used in reasonable ways, e.g., 3/Inf = 0 and -2/0 = -Inf but cos(Inf) and -Inf + Infboth returnNaN, the reserved word for "not a number". If we could be sure that users of theeasy` function would not input “incorrect” values (i.e., asking for the pdf at a number below 1, the cdf for a number below 1, or an inverse cdf for a number below 0 or 1 or above), then the conditional code would not be needed. Check that the first three functions handle extreme cases.

deasy(c(-Inf, -3, Inf)) == c(0, 0, 0)
peasy(c(-Inf, -3, Inf)) == c(0, 0, 1)
qeasy(c(0, 1)) == c(1, Inf)
qeasy(c(-Inf, -3, 3, Inf)) #all NaN

1. Compare a simulation with theory via the means, standard deviations, pdfs, cdfs, and quantiles.

#Simulate
runs = 1000
simulation = tibble(x = reasy(runs))

#Compute the theoretical distribution
theory = tibble(
 x = seq(0, max(simulation$x), length.out=100),
 pdf = deasy(x),
 cdf = peasy(x)
)

#Compare simulation and theoretical statistics
rbind(
 simulation |> summarize(type = "simulation", mean = mean(x), sd = sd(x)),
 tibble(type = "theory", mean = 1.5, sd = sqrt(0.75))
)

#Compare simulation and theoretical pdfs
ggplot(simulation, aes(x = x, y = after\_stat(density))) +
 geom\_histogram(bins = 30, color = "white") +
 geom\_line(aes(x = x, y = pdf), theory, color = "red")

#Compare simulation and theoretical cdfs
ggplot(simulation, aes(x = x)) +
 stat\_ecdf() +
 geom\_line(aes(x = x, y = cdf), theory, color = "red")

#Compare simulation and theory with a quantile-quantile plot
ggplot(simulation, aes(sample = x)) +
 stat\_qq(distribution = qeasy, alpha = 0.2) +
 geom\_abline(slope = 1, intercept = 0, color = "red")

## The Precision of Simulations

How close do we expect a random number generator to be in comparison with theory? If theory says that the probability of generating numbers within a particular interval $[a,b)$ is $p$ and we generate $n$ random numbers, then the number of random numbers that are in $[a,b)$ is a random variable $X$ that has a binomial distribution with the parameters $n$ and $p$. Hence, the mean and standard deviation of $X$ is $μ\_{X}=np$ and $σ\_{X}=\sqrt{np\left(1-p\right)}$. The proportion of random numbers that are in $[a,b)$ is the random variable $Y=X/n$, which therefore has the mean and standard deviation $μ\_{Y}=p$ and $σ\_{Y}=\sqrt{p\left(1-p\right)/n}$. Thus, we expect that 95% of the time, the proportion of random numbers that are in $[a,b)$ will be within $2\sqrt{p\left(1-p\right)/n}$ of the theoretical probability $p$. Since $p\left(1-p\right)\leq 1/4$ for all $p\in \left[0,1\right]$, it follows that $2\sqrt{p\left(1-p\right)/n}\leq 1/\sqrt{x}$. The later formula is often called the *margin of error* formula. Since $1/\sqrt{1600}=0.025$, pollsters who have surveyed 1600 randomly selected people will state that with 95% confidence the reported percentages could have an error of $\pm 2.5\%$. Most pollsters use a stratified, rather than simple, random sample so that they can report meaningful results for subgroups of the population (e.g., by political party or gender), but this means that the margin of error for the full sample will be larger than indicated by the above formula.

Let’s apply this to the 1000 numbers generated by reasy (which were stored in the tibble simulation). The first line finds the 0.00, 0.20, 0.40, 0.60, 0.80, and 1.00 quantiles for the theoretical distribution. The second line creates a frequency distribution for the randomly generated numbers based upon the theoretical quantiles. We would expect about $1000/5=200$ in each bin. The third line finds the relative frequencies for each bin without the names. We would expect each number to be about $0.200$. The fourth line finds the deviations of the relative frequencies from the expected relative frequency. The fifth line finds the margin of error and the sixth line finds the more precise margin of error. We expect all or most of the deviations to be less than these numbers.

breaks = qeasy(seq(0,1,1/5)); breaks
freqdist = table(cut(simulation$x, breaks, right=FALSE)); freqdist
relfreq = unname(freqdist)/runs; relfreq
deviations = abs(relfreq-0.200); deviations
moe = 1/sqrt(runs); moe
pmoe = 2\*sqrt(0.200\*(1-0.200)/runs); pmoe

### A Hard Continuous Random Number Generator

Suppose the random variable $X$ has the probability density function $f\left(x\right)=\frac{2}{\sqrt{π}}e^{-x^{2}}$ for $0\leq x<\infty $. Carry out the same steps as were done for the easy random variable.

What makes this hard is that the cdf and inverse cdf cannot be expressed with simple formulas. Instead numerical approximation is used, but the integrate and uniroot functions only work with numbers, not vectors. R provides the Vectorize function so that we can write code that would normally work only for numeric input but can then be used with vector input. It was found empirically, that $F\left(3.86\right)=1$ to the level of machince precision.