Simple Random Sampling: Not So Simple

Kellie Ottoboni and Philip B. Stark

September 15, 2018

Abstract

R (Version 3.5.0) generates random integers between 1 and m by multiplying random floats by m, taking the floor, and adding 1 to the result. It is well known that quantization effects inherent in this approach result in a non-uniform distribution on $\{1, \ldots, m\}$. The difference, which depends on m, can be substantial. Because the `sample` function in R relies on generating random integers, random sampling in R is also biased. There is an easy fix, namely, construct random integers directly from random bits, rather than multiplying a random float by m. That is the strategy taken in Python's `numpy.random.randint()` function, among others.

A textbook way to generate a random integer on $\{1, \ldots, m\}$ is to start with $X \sim U[0, 1)$ and define $Y \equiv 1 + \lfloor mX \rfloor$. If X is truly uniform on $[0, 1)$, Y is then uniform on $\{1, \ldots, m\}$. However, if X has a discrete distribution derived by scaling a pseudorandom binary integer or floating-point number, the resulting distribution is, in general, not uniformly distributed on $\{1, \ldots, m\}$ even if the underlying pseudorandom number generator (PRNG) is perfect. Theorem 0.1 illustrates the problem.

Theorem 0.1 (**?**). *Suppose X is uniformly distributed on w-bit binary fractions, and let $Y_m \equiv 1 + \lfloor mX \rfloor$. Let $p_+(m) = \max_{1 \le k \le m} \Pr\{Y_m = k\}$ and $p_-(m) = \min_{1 \le k \le m} \Pr\{Y_m = k\}$. There exists $m < 2^w$ such that, to first order, $p_+(m)/p_-(m) = 1 + m2^{-w+1}$.*

A better way to generate random elements of $\{1, \ldots, m\}$ is to use pseudorandom bits directly, avoiding floating-point representation, multiplication, and the floor operator. Integers between 0 and $m-1$ can be represented with $\mu(m) \equiv \lceil \log_2(m-1) \rceil$ bits. To generate a pseudorandom integer between 1 and m, first generate $\mu(m)$ pseudorandom bits (for instance, by taking the most significant $\mu(m)$ bits from the PRNG output, if $w \ge \mu(m)$, or by concatenating successive outputs of the PRNG and taking the first $\mu(m)$ bits of the result, if $w < \mu(m)$). Cast the result as a binary integer M. If $M > m-1$, discard it and draw another $\mu(m)$ bits; otherwise, return $M+1$.[1] Unless $m = 2^{\mu(m)}$, this procedure is expected to discard some random draws—up to almost half the draws if $m = 2^p + 1$ for some integer p. But if the input bits are IID Bernoulli(1/2), the output will be uniformly distributed on $\{1, \ldots, m\}$. This is how the Python function `numpy.random.randint()` (Version 1.14) generates pseudorandom integers.[2]

The algorithm that R (Version 3.5.0) [**?**] uses to generate random integers has the issue pointed out in Theorem 0.1 in a more complicated form, because R uses a pseudo-random float at an intermediate step, rather than multiplying a binary fraction by m. The way the float is constructed depends on m. Because `sample` relies on random integers, it inherits the problem.

When m is small, R uses `unif_rand` to generate pseudorandom floating-point

[1] See **?**, p.114

[2] However, Python's built-in `random.choice()` (Versions 2.7 through 3.6) does something else biased: it finds the closest integer to mX, where X is a binary fraction between 0 and 1.

numbers X on $[0,1)$ starting from a w-bit random integer. Typically, $w = 32$; for simplicity of exposition, we shall write as if that is universally true. The range of `unif_rand` contains (at most) 2^w values, which are approximately equi-spaced (but for the vagaries of converting a binary fraction into a floating-point number [**?**]).

When $m > 2^w - 1$, R calls `ru` instead of `unif_rand`.[3] `ru` combines two floating point numbers, R_1 and R_2, each generated from a 2^w-bit integer, to produce the floating-point number X, as follows: the first float is multiplied by $U = 2^{25} - 1$, added to the second float, and the result is divided by 2^w:

$$X = \frac{\lfloor UR_1 \rfloor + R_2}{U}.$$

The cardinality of the range of `ru` is certainly not larger than 2^{2w}. The range of `ru` is unevenly spaced on $[0,1)$ because of how floating-point representation works, as we explain in greater detail below. The inhomogeneity can make the probability that $X \in [x, x+\delta) \subset [0,1)$ vary widely with x.

For the way R generates random integers, the nonuniformity of the probabilities of $\{1, \ldots, m\}$ is largest when m is just below 2^{31}. The maximum ratio of selection probabilities approaches 2 as m increases to the cutoff 2^{31}, or about 2 billion. Even if m is close to 1 million, the ratio is about 1.0004.

To illustrate the issue, suppose that the PRNG generating r_1 and r_2 produces 4-bit integers and that floating point numbers are represented in base 2 with 4 bits of precision. Let $U = 2^4$. Suppose $R_1 = 0$, represented as 0.000×2^0. Then the possible values of `ru` are equally spaced on $[0, 1/U]$, separated by U^{-2}. Suppose that R_1 is the largest value that the PRNG can generate, represented as 1.000×2^{-1} in

[3] A different function, `sample2`, is called when $m > 10^7$ and $k < m/2$. `sample2` uses the same method to generate pseudorandom integers.

floating point. Multiplying by U changes the exponent from -1 to 3. In floating point arithmetic, adding $\lfloor UR_1 \rfloor$ to small values of R_2 will result in underflow: for $R_2 < 1.000 \times 2^0$, $\lfloor UR_1 \rfloor + R_2 = \lfloor UR_1 \rfloor$. Moreover, $\lfloor UR_1 \rfloor + 1.000 \times 2^0 = \lfloor UR_1 \rfloor + 1.001 \times 2^0 = \lfloor UR_1 \rfloor + 1.111 \times 2^0$ and so forth. In this case, `ru` will give equally spaced values separated by a distance of U^{-1}. TO DO: PBS. The way R does it, there won't be "complete" underflow unless R_2 is sufficiently small. There are 53 bits available to store what was originally 64 bits of information, but the conversion to floating point to produce each rand changes things, too. The fact that the possible values don't have equal probabilities doesn't directly translate into the uniformity/non-uniformity of the distribution on $[0, 1)$, because the spacing is non-uniform, too. I think we need to work through one example carefully. The probability from the "underflow" bits will accumulate in the more significant bits. It isn't obvious to me how all this flows down to the uniformity of the resulting distribution, measured, e.g., by the probability of an interval of width Δ.

Knuth's theorem no longer applies to the ratio of selection probabilities in the branch of the logic that calls `ru`, as values aren't uniformly spaced. In fact, the nonuniformity of pseudorandom numbers might exacerbate the pseudorandom integer generation problem further. This could be avoided by using pseudorandom bits directly to gain precision, for example by concatenating two 32-bit pseudorandom numbers or by using a 64-bit PRNG (**?**). Because the Mersenne Twister (the default PRNG in R) is a 64-bit PRNG (unless R used the 32-bit version instead), implementing this in R should be straightforward.

We recommend that the R developers replace the current algorithm for generating pseudorandom integers with the masking algorithm and use pseudorandom bits

Population size (m)	Word length (w)	Maximum ratio of selection probabilities
10^6	32	1.0004
10^9	32	1.466
$2^{31} - \epsilon$	32	2

Table 1: Maximum ratio of selection probabilities for different population sizes m. R constructs random integers using random binary integers with different numbers of bits (word lengths) depending on m: if $m < 2^{31}$, the word length is between 25 and 32. The maximum ratio of selection probabilities is based on Knuth's theorem; for $m > 2^{31}$, the maximum may be even larger due to nonuniformity introduced by the `ru` function.

directly to generate pseudorandom numbers with more than 32 bits of precision.