Simple Random Sampling: Not So Simple

Kellie Ottoboni and Philip B. Stark

Draft August 27, 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. Quantization effects inherent in this approach result in a non-uniform distribution: different integers between 1 and m generally have different probabilities. The difference 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, to use random bits to construct rather than random floats. That is the strategy taken in Python's `numpy.random.randint()` function.

1 Introduction

A textbook way to generate a random integer on the range $\{1, \ldots, m\}$ is to start with $X \sim U[0, 1)$ and define $Y \equiv 1 + \lfloor mX \rfloor$. If X is truly uniform on the interval $[0, 1)$, Y is then uniform on $\{1, \ldots, m\}$. However, if X has a discrete distribution derived by scaling a pseudorandom binary integer, the resulting distribution is not uniformly distributed on $\{1, \ldots, m\}$ even if the underlying PRNG is perfect (unless m is a power of 2).

Theorem 1.1 (?). *Suppose X is uniformly distributed on w-bit binary numbers, and let $Y_m \equiv 1 + \lfloor mX \rfloor$. Let $p_+(m) = \max_{1 \leq k \leq m} \Pr\{Y_m = k\}$ and $p_-(m) = \min_{1 \leq k \leq m} \Pr\{Y_m = k\}$. There exists $m < 2^w$ such that, to first order, $p_+(m)/p_-(m) = 1 + m2^{-w+1}$.*

The algorithm that R (Version 3.5.0) [?] uses to generate uniform random integers has this issue (albeit in a slightly more complicated form, because, depending on m, R starts with pseudorandom binary integers of different lengths). Because `sample()` relies on random integers, it inherits the problem.

R uses `unif_rand` to generate pseudorandom numbers with word size at most $w = 32$. To generate integers with a larger word size, R extends the precision by combining two w-bit integers to obtain an integer with 50 to 53 bits (depending the chosen PRNG). It then generates

A better way to generate random integers on $\{1, \ldots, m\}$ is to use pseudorandom bits directly. The integer m can be represented with $\mu = \lceil \log_2(m) \rceil$ bits. To generate a pseudorandom integer between 1 and m, first generate μ pseudorandom bits (for instance, by taking the most significant μ bits from the PRNG output). If that binary number is larger than μ, then discard it and repeat until getting μ bits that represent an integer less than or equal to m.[1] This procedure might discard almost half the draws if m is slightly larger than a power of 2, but if the input bits are IID Bernoulli(1/2), the resulting integers will actually be uniformly distributed.

To summarize the steps for generating a pseudorandom integer on $\{1, \ldots m\}$ from a pseudorandom w-bit integer:

1. Set $\mu = \log_2(m - 1)$.

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

2. Define a w-bit mask consisting of the first μ bits set to 1 and the remaining $(w - \mu)$ bits set to zero.

3. Generate a random w-bit integer X.

4. Define Y to be the bitwise and of X and the mask.

5. If $Y \leq m - 1$, output $Y + 1$; otherwise, return to step 3.

This is how the Python function `numpy.random.randint()` (Version 1.14) generates pseudorandom integers.[2]

The R `sample()` function has a branch in its logic depending on the number of elements in the population to be sampled. It uses the function `ru` when $m >= 2^{31}$ and the function `rand_unif` when $m < 2^{31}$.[3] The nonuniformity of selection probabilities is largest when m is just below the cutoff 2^{31}. In that case, `sample` calls `unif_rand`, which gives outputs with word size $w = 32$. 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.

When $m \geq 2^{31}$, R calls `ru()` to produce a pseudorandom number with word length at least $w = 50$ bits and at most $w = 53$ bits (depending the chosen PRNG). In that branch of the logic, ratio of selection probabilities only becomes large (on the order of $1 + 10^{-3}$) for large population sizes, say $m > 2^{40} \approx 10^{12}$.

[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.

[3] A different function, `sample2`, is called when $m > 10^7$ and $k < m/2$. It uses the same flawed method of generating pseudo-random integers.

Population size (m)	Word length (w)	Ratio of selection probabilities
10^6	32	1.0004
10^9	32	1.466
$2^{31} - \epsilon$	32	2
$2^{31} + \epsilon$	53	$1 + 2.3 \times 10^{-7}$
10^{12}	53	1.0001
10^{15}	53	1.11

Table 1: Maximum ratio of selection probabilities for different population sizes.