Simple Random Sampling: Not So Simple

Kellie Ottoboni and Philip B. Stark

Draft August 3, 2018

Abstract

The algorithm that R (Version 3.5.0) uses to generate uniform random integers is inaccurate. This paper describes why the algorithm gives unequal weight to different values. The magnitude of the problem can be disastrous for certain ranges of integers: in the worst case, some integers may be sampled twice as frequently than others. We present another algorithm for generating uniform random integers that does not have this bias.

1 Introduction

Sampling is a foundational idea in Statistics that is critical for the theory of surveys, permutation testing, the bootstrap, MCMC, and many more methods. The theory and implementation of these methods depend on the ability to draw random samples with specific probabilities. In particular, many methods require *uniform* random sampling, i.e. with equal probability of occurrence.

The algorithm that R (Version 3.5.0) [R Core Team, 2018] uses to generate uniform random integers from its PRNGs is inaccurate. Table 1 shows that in the worst

case, some integers are twice as likely to be selected as others. These biases are passed into the functions to draw random samples, making certain samples more likely than others. Statistical methods that rely on the ability to draw random samples uniformly will therefore be biased. Sampling using the command `sample(1:m, k, replace=FALSE)` may not weight all m values equally.

Section 2 describes the problematic algorithm to convert pseudorandom numbers to integers on a particular interval and presents a better way that is guaranteed to produce integers with equal probability. Section 3 shows the magnitude of the problem for varying ranges of integers.

2 Algorithms for generating random integers

Pseudorandom number generators (PRNGs) are deterministic algorithms that produce sequences of numbers that are statistically indistinguishable from truly random uniform numbers. PRNGs typically produce w-bit integers, which are normalized to floats $U \in \{0, 2^{-w}, 2 \times 2^{-w}, ..., 1 - 2^{-w}\}$ between 0 and 1. (Some PRNGs in R have $w = 25$, but most have $w = 32$.)

R calls the function `unif_rand` to generate pseudorandom numbers with word size at most $w = 32$. To generate integers with a larger word size, it is necessary to augment their precision. This can be done by combining two w-bit integers to obtain a $(2w-1)$-bit integer. The `ru` function combines two pseudorandom numbers with word length w to produce one with word length $2w - 1$. This output has word length at least $w = 50$ bits and at most $w = 53$ bits (depending the chosen PRNG). We will revisit these functions in Section 3.

Many algorithms for drawing a random sample of size k out of m items rely on independent random integers uniformly distributed on $\{1, \ldots, m\}$. Additional

algorithms are needed to convert w-bit integers output by PRNGs to integers on the range $\{1, \ldots, m\}$.

2.1 Problematic algorithm

A standard way to generate a random integer on the range $\{1, \ldots, m\}$ is to start with $X \sim U[0, 1)$ and define $Y \equiv 1 + \lfloor mX \rfloor$. In theory, that is fine when X is truly uniform on the interval $[0, 1)$. In practice, X has a discrete uniform distribution, derived by normalizing a pseudorandom number that is uniform on w-bit integers $\{0, 1, \ldots, 2^w - 1\}$. Then, unless m is a power of 2, the distribution of Y differs from uniform on $\{1, \ldots, m\}$.

The following theorem gives an approximate upper bound on the largest ratio of selection probabilities produced by this algorithm. If Y were perfectly uniform, then this ratio would be 1.

Theorem 2.1 (Knuth [1997])**.** *There exists $m < 2^w$ such that, to first order, the ratio of the largest selection probability to the smallest selection probability is $1 + m2^{-w+1}$.*

2.2 A better algorithm

A more accurate 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 at most 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 may be inefficient, as

[1] See Knuth [1997] p. 114.

it can potentially require throwing out half of draws if m is close to a power of 2, but 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 $\mu = \log_2(m-1)$.
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 Y.
4. Define y to be the bitwise and of y and the mask.
5. If $y \leq m-1$, output $x = y+1$; otherwise, return to step 3.

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

3 Random sampling in R

The sampling algorithms implemented in R rely on uniformly distributed random integers on the set $\{1, 2, \ldots, m\}$, for arbitrary positive m. R uses the flawed method of generating pseudorandom integers from Section 2.1. There exist values of m for which some integers occur more frequently than others.

In R, one would generally use the function `sample(1:m, k, replace=FALSE)` to draw k pseudo-random integers between 1 and m, inclusive, without replacement.

[2]However, Python's built-in `random.choice()` (Versions 2.7 through 3.6) does something else that's biased: it finds the closest integer to mX.

More generally, to draw a simple random sample of size k from a list of m values, the syntax is `sample(values, k, replace=FALSE)`.

The `sample` function behaves differently for two cases, depending on the level of numerical precision needed to generate large enough integers. It uses the function `ru` when $m >= 2^{31}$ and the function `rand_unif` when $m < 2^{31}$.[3] The issue of nonuniformity is worst when m is just below the cutoff 2^{31}.

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

The problem is worst for large population sizes just below the threshold 2^{31}. In this 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.

This error in random integer selection probabilities feeds back into the algorithm for sampling. If certain integers have higher probability, then certain samples or permutations will also occur with higher probability.

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.

[3]There is an additional function, `sample2`, which gets called when $m > 10^7$ and $k < m/2$. It also uses this flawed method of generating pseudo-random integers, so we will not treat it separately.

References

Thomas H. Cormen. *Introduction to Algorithms*. MIT Press, July 2009.

Donald E. Knuth. *Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley Professional, Reading, Mass, 3 edition edition, November 1997. ISBN 978-0-201-89684-8.

R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2018. URL `https://www.R-project.org`.

Jeffrey S. Vitter. Random Sampling with a Reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, March 1985. ISSN 0098-3500. doi: 10.1145/3147.3165. URL `http://doi.acm.org/10.1145/3147.3165`.