Random Sampling: Practice Makes Imperfect

Philip B. Stark and Kellie Ottoboni

Abstract The pseudo-random number generators (PRNGs), sampling algorithms, and algorithms for generating random integers in some common statistical packages and programming languages are unnecessarily inaccurate. Most use PRNGs with state spaces that are too small for contemporary sampling problems and randomization-based methods such as the bootstrap and permutation tests, as simple pigeonhole arguments show. Some use a "textbook" sampling algorithm that involves generating a random permutation, overtaxing the PRNG even for modest population sizes. Some use a better algorithm involving generating random integers to select items; but of those, some generate random integers using a "textbook" method (multiply a random binary fraction or a float by a constant and take the floor) that can lead to extremely nonuniform sampling. Statistics packages and scientific programming languages should use cryptographically secure PRNGs by default, and offer weaker PRNGs only as an option. Software should not generate a random sample by randomly permuting the population and taking the first k items. Software should not generate random integers by multiplying a binary fraction or float by a constant and rounding the result or taking its floor.

> The difference between theory and practice is smaller in theory than it is in practice. – unknown
>
> In theory, there's no difference between theory and practice, but in practice, there is. – Jan L.A. van de Snepscheut

Philip B. Stark
University of California, Berkeley, e-mail: pbstark@berkeley.edu

Kellie Ottoboni
University of California, Berkeley e-mail: kellieotto@berkeley.edu

1 Introduction

Pseudo-random number generators (PRNGs) are central to the practice of statistics. They are used to draw random samples, allocate patients to treatment, implement the bootstrap, calibrate permutation tests, perform MCMC, approximate p-values, partition data into training and test sets, and countless other purposes.

Practitioners generally do not question whether standard statistical software is adequate for these tasks. Textbooks give algorithms for generating random integers, random samples, and IID random variates[1] that implicitly or explicitly assume that the PRNGs in common software packages can be substituted for true IID $U[0,1]$ variables without introducing material error.

We show here that at least one of those assumptions is incorrect for commonly used statistical packages, including MATLAB, R, SPSS, and Stata.

For example, whether software can in principle generate all samples of size k from a population of n items—much less generate them with equal probability—depends on the size of the problem and the internals of the software, including the underlying PRNG and the algorithm used to turn PRNG output into a sample. We show that even for datasets with hundreds of observations, many pseudo-random number generators cannot draw all subsets of size k, for modest values of k.

The choice of sampling algorithm—the mapping from PRNG output to a random sample—also matters: some algorithms put greater demands on the PRNG than others. Some involve permuting the data; these quickly run out of "headroom," because the maximum number of items the PRNGs can permute ranges from 13 to 2084, far smaller than many data sets. Others require uniformly distributed integers as input, but many software packages generate pseudo-random integers using a truncation method that does not give nonuniform outputs, even if the PRNG were uniformly distributed on k-bit binary integers.

As a result of the limitations of common PRNGs and sampling algorithms, the L_1 distance between the uniform distribution on samples of size k and the distribution induced by a particular PRNG and sampling algorithm can be nearly 2. It follows that there are bounded functions of random samples that have very different expectations with respect to those two distributions.

We consider three broad classes of PRNGs: those generally inadequate for quantitative work, those generally considered "adequate for statistics," and cryptographically secure PRNGs. This paper explores whether PRNGs generally considered adequate for statistical work really are adequate. Section 2 presents an overview of PRNGs and gives examples of better and worse ones. Section 3 shows that, for modest n and k, the state spaces of common PRNGs considered adequate for statistics are too small to generate all permutations of n things or all samples of k of n things. Section 4 discusses different sampling algorithms and shows that some are less demanding on the PRNG than others. Section 4.1 shows that a common, textbook procedure for generating pseudo-random integers using a PRNG can be

[1] Add citation! @Kellie: we found at least one, right?

quite inaccurate; unfortunately, this is essentially the method that R uses. Section 5 concludes with recommendations and best practices.

2 Pseudo-random number generators

A pseudo-random number generator (PRNG) is a deterministic algorithm that, starting with a "seed," produces a sequence of numbers that are supposed to behave like random numbers for some purposes. An ideal PRNG has output that is computationally indistinguishable from truly random, uniform, IID binary bits. (Cryptographically secure PRNGs approach this ideal, but common PRNGs do not.)

A PRNG has several components: an internal *state*, initialized with a "seed"; a function that maps the current state to an integer output; and a function that updates the internal state.

If the state space is finite, the PRNG must eventually revisit its initial state. The *period* of a PRNG is the maximum, over initial states, of the number of states the PRNG goes through before returning to a state that has been visited. The period is at most the total number of possible states. If the period is equal to the total number of states, the PRNG is said to have *full period*. PRNGs for which the state and the output are the same have periods no larger than the number of possible outputs. Better PRNGs generally use a state space with much larger dimension than the dimension of the output.

Some PRNGs are sensitive to the initial state. Some (depending on the initial state) need many "burn-in" calls before the output behaves well.

2.1 Simple PRNGs

> Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin. -John von Neumann

Linear congruential generators (LCGs) have the form $X_{n+1} = (aX_n + c) \mod m$, for a *modulus* m, *multiplier* a, and *additive constant* c. LCGs are fast to compute and require little computer memory. The behavior of LCGs is well understood from fundamental number theory. For instance, the Hull-Dobell theorem [**?**] gives necessary and sufficient conditions for a LCG to have full period for all seeds.

To take advantage of hardware efficiencies, early computer systems implemented LCGs with moduli of the form $m = 2^b$, where b was the integer word size of the computer. Such LCGs cannot have full period because m is not prime. While better LCGs have been developed—and some are used in commercial statistical software packages—they are still generally considered to be inadequate for statistics because of their short periods (typically 2^{32}) and correlation among outputs.

Other types of PRNGs have been constructed using simple mathematical relations, including lagged Fibonacci sequences and xorshift operations TO DO: CITE.

More complex PRNGs can be built from multiple simpler ones. For instance, the KISS generator combines 4 generators of three types, and has a period greater than 2^{210}. Nonetheless, such PRNGs are predictable from a relatively small number of outputs. For example, one can find the LCG constants a, c, and m from only 3 outputs.

The Wichmann-Hill PRNG is a sum of three LCGs, used to produce random values on $[0,1)$. It is generally not considered adequate for statistics, but was (nominally) the PRNG in Excel for several generations. The generator in Excel had an implementation bug that persisted for several generations. Excel didn't allow the seed to be set so issues could not be replicated, but users reportedly generated negative numbers on occasion ([**?**]). As of 2014, IMF banking Stress tests use Excel [**?**].

2.2 Mersenne Twister (MT)

Mersenne Twister (MT) ([**?**]) is a "twisted" generalized feedback shift register, a complex sequence of bitwise and linear operations. Its state space is $19,937$ bits and it has an enormous period $2^{19937}-1$, a Mersenne prime. It is k-distributed to 32-bit accuracy for $k \leq 623$, meaning that output vectors of length up to 623 occur with equal frequency over the full period. An integer seed is used to set the state, a 624×32 binary matrix.

MT is the default PRNG in most common software packages, including GNU Octave, Maple, MATLAB, Mathematica, Python, R, Stata, and many more (see Table 2). We show below that it is not adequate for statistics. Moreover, MT can have slow "burn in," especially for seeds with many zeros. TO DO: CITE The outputs for close seeds can be similar, which can affect distributed computations.

2.3 Cryptographic hash functions

The PRNGs described above are based on simple mathematical operations. They are quick to compute but predictable due to their mathematical structure. Cryptographers have devoted a great deal of energy to inventing cryptographic primitives, functions for encrypting messages so that an adversary cannot efficiently decrypt the message. Cryptographic primitives can be easily turned into PRNGs, as the properties that make such functions cryptographically secure are properties of pseudo-randomness.

A cryptographic hash function H is a primitive with the following properties:

- H produces a fixed-length "digest" (hash) from arbitrarily long "message": $H : \{0,1\}^* \to \{0,1\}^L$.
- H is inexpensive to compute.

- H is "one-way," i.e., it is hard to find the pre-image of any hash except by exhaustive enumeration.
- H is collision-resistant, i.e. it is hard to find $M_1 \neq M_2$ such that $H(M_1) = H(M_2)$.
- small changes to input produce big changes to output, making it unpredictable
- outputs of H are equidistributed: bits of the hash are essentially random

These properties of H make it suitable as the basis of a PRNG: It is *as if* $H(M)$ is a random L-bit string assigned to M in a way that's essentially unique. We can construct a simple hash-based PRNG with the following procedure:

1. Generate a random string S of reasonable length, e.g., 20 digits.
2. Let "S,i" be the state, where i counts how many times the hash function has been called.
3. Set $X_i = \text{Hash}(S, i)$, interpreted as a (long) hexadecimal number.
4. Increment i and repeat to generate more PRNs.

Since a message may have arbitrary length, this PRNG has an unbounded state space.

3 Counting permutations and samples

We present simple mathematical theorems that show that any PRNG with a finite state space cannot be "adequate for statistics" for every statistical problem. We begin with an elementary counting theorem: the pigeonhole principle.

Theorem 1 (Pigeonhole principle). *If you put $N > n$ pigeons in n pigeonholes, at least one pigeonhole must contain more than one pigeon.*

Corollary 1. *At most n pigeons can be put in n pigeonholes if at most one pigeon is put in each hole.*

The corollary implies that a PRNG cannot generate more permutations or samples than the number of states the PRNG has (which is in turn an upper bound on the period of the PRNG). Of course, that does not mean that the permutations or samples a PRNG can generate are generated with approximately equal probability, whether the number of states is less than, equal to, or greater than the number of permutations or samples: that depends on the quality of the PRNG, not just the number of states it has.

The number of permutations of n objects is $n!$, the number of possible samples of k of n items with replacement is n^k, and the number of possible samples of k of n without replacement is $\binom{n}{k}$. These bounds are helpful:

- Stirling bounds: $en^{n+1/2}e^{-n} \geq n! \geq \sqrt{2\pi}n^{n+1/2}e^{-n}$.
- Entropy bounds: $\frac{2^{nH(k/n)}}{n+1} \leq \binom{n}{k} \leq 2^{nH(k/n)}$, where $H(q) \equiv -q\log_2(q) - (1-q)\log_2(1-q)$.

- Stirling combination bounds: for $\ell \geq 1$ and $m \geq 2$, $\binom{\ell m}{\ell} \geq \frac{m^{m(\ell-1)+1}}{\sqrt{\ell}(m-1)^{(m-1)(\ell-1)}}$.

Table 1 counts permutation pigeons and PRNG pigeonholes. For PRNGs with a small state space, even modest population sizes make it impossible to generate all possible randomizations. MT fails too: fewer than 1% of permutations of 2084 items are actually attainable.

Table 1 Illustration of the pigeonhole principle applied to PRNGs, samples, and permutations. For a PRNG of each size state space, we provide examples where not all samples or permutations are attainable.

Feature	Size	Full	Scientific notation
32-bit state space	2^{32}	4,294,967,296	4.29×10^9
Permutations of 13	13!	6,227,020,800	6.23×10^9
Samples of 10 out of 50	$\binom{50}{10}$	10,272,278,170	1.03×10^{10}
Fraction of attainable samples with 32-bit state space	$\frac{2^{32}}{\binom{50}{10}}$	0.418	
64-bit state space	2^{64}	18,446,744,073,709,551,616	1.84×10^{19}
Permutations of 21	21!	51,090,942,171,709,440,000	5.11×10^{19}
Samples of 10 out of 500	$\binom{500}{10}$		2.46×10^{20}
Fraction of attainable samples with 64-bit state space	$\frac{2^{64}}{\binom{500}{10}}$	0.075	
128-bit state space	2^{128}		3.40×10^{38}
Permutations of 35	35!		1.03×10^{40}
Samples of 25 out of 500	$\binom{500}{25}$		2.67×10^{42}
Fraction of attainable samples with 128-bit state space	$\frac{2^{128}}{\binom{500}{25}}$	0.0003	
MT state space	$2^{32\times624}$		9.27×10^{6010}
Permutations of 2084	2084!		3.73×10^{6013}
Samples of 1000 out of 390 million	$\binom{3.9\times10^8}{1000}$		$> 10^{6016}$
Fraction of attainable samples	$\frac{2^{32\times624}}{\binom{3.9\times10^8}{1000}}$		$< 1.66 \times 10^{-6}$

3.1 L_1 bounds

Simple probability bounds demonstrate the extent of bias introduced by using a PRNG with insufficiently large state space to approximate the sampling distribution of a statistic. Suppose $\mathbb{P}_0$ and $\mathbb{P}_1$ are probability distributions on a common measurable space. If there is some set S for which $\mathbb{P}_0(S) = \varepsilon$ and $\mathbb{P}_1(S) = 0$, then $\|\mathbb{P}_0 - \mathbb{P}_1\|_1 \geq 2\varepsilon$.

Thus there is a function f with $|f| \leq 1$ such that

$$\mathbb{E}_{\mathbb{P}_0} f - \mathbb{E}_{\mathbb{P}_1} f \geq 2\varepsilon.$$

In this context, $\mathbb{P}_0$ is the true distribution of statistics across equally likely resamples of a population and $\mathbb{P}_1$ is the distribution of statistics attainable using a PRNG to resample from the population. If the PRNG has n states and we want to generate $N > n$ equally likely outcomes, at least $N - n$ outcomes will have probability zero instead of $1/N$. Some statistics will have bias of at least $2 \times \frac{N-n}{N}$.

4 Sampling algorithms

Given a good source of randomness, there are many ways to draw a simple random sample. One basic approach is like shuffling a deck of n cards, then dealing the top k: assign a pseudo-random number to each item, sort them, then take the first k elements of the permutation to be the sample. We call this algorithm PIKK: Permute indices and keep k.

If the numbers really are independent and identically distributed, then every permutation is equally likely, and it follows that the first k are a simple random sample. The algorithm assumes permutations are equiprobable; if not, then samples generated using this algorithm will not be either. Furthermore, this algorithm is inefficient: it requires the generation of n random numbers and then an $O(n \log n)$ sorting operation. Generating n pseudo-random numbers places more demand on a PRNG than other algorithms, discussed below, that only require k pseudo-random numbers.

There are a number of standard ways to generate a random permutation. PIKK uses one of the least efficient ways, assigning a number to each element and sorting. A more efficient method is the"Fisher-Yates shuffle" or "Knuth shuffle" (Knuth attributes it to Durstenfeld) [**?**]. This algorithm requires the ability to generate independent random integers on various ranges, but doesn't require sorting. There is also a version suitable for *streaming*, i.e. generating a random permutation of a list that has an (initially) unknown number of elements.

PIKK is not the best method for sampling k out of n items. Many other methods only require generating k random numbers, instead of the n required to permute the entire list. One simple method is to generate k random uniform integers between 1 and n, then take the items with those indices to be the sample. [**?**] provide a recursive algorithm to draw random samples of size k out of n. It is elegant in that it does not require sorting, but will not work if k is greater than the maximum recursion depth of the software. (In Python, the default maximum recursion depth is 2000, so this algorithm cannot draw samples of greater than 2000 items unless one increases the recursion limit.)

The previous algorithms require n to be known. *Reservoir* algorithms, such as Waterman's Algorithm 'R', do not [**?**]. Moreover, the algorithms are suitable for streaming use: items are examined sequentially and either enter into the reservoir, or,

if not, are never revisited. Vitter's Algorithm Z is even more efficient than Algorithm R and uses random skips to reduce run-time to be essentially linear in k [**?**].

4.1 Pseudo-random integers

Many of the above algorithms rely on the ability to generate pseudo-random *integers*. The output of a PRNG is typically a w-bit integer, so some method is needed to rescale it to the range $\{1,\ldots,m\}$.

The textbook way to generate an integer on the range $\{1,\ldots,m\}$ is to first draw a random $X \sim U[0,1)$ and then define $Y \equiv 1 + \lfloor mX \rfloor$. In practice, X from a PRNG is not really $U[0,1)$, as it is derived by normalizing a pseudo-random number that is (supposed to be) uniform on w-bit integers.

The distribution of Y isn't uniform on $\{1,\ldots,m\}$ unless m is a power of 2. If $m > 2^w$, at least $m - 2^w$ values will have probability 0 instead of probability $1/m$. For $m < 2^w$, the ratio of the largest to smallest selection probability is, to first order, $1 + m2^{-w+1}$ [**?**].

This ratio can grow large quickly: For $m = 10^9$ and $w = 32$, this bound is approximately 1.466. If $w = 32$, then for $m > 2^{32} = 4.24 \times 10^9$, some values will have probability 0. This is the algorithm that R (Version 3.5.0) [**?**] uses to generate pseudo-random integers, which eventually are used in the main sampling functions.

A more accurate way to generate random integers on $\{1,\ldots,m\}$ is to use pseudo-random bits directly. The integer m can be represented with $\mu = \lceil \log_2(m) \rceil$ bits. To generate a pseudo-random integer at most m, first generate μ pseudo-random 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. This procedure may be inefficient, as 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.This is how the Python package Numpy (Version 1.14) generates pseudo-random integers.[2]

5 Discussion

Any PRNG with a finite state space cannot generate all possible samples from or permutations of sufficiently large populations. That can matter. A PRNG with a 32-bit state space cannot generate all permutations of 13 things. The Mersenne Twister (MT) cannot generate all permutations of 2084 things.

Table 2 shows the PRNGs and sampling algorithms used in common statistical packages. Most use MT as their default PRNG; *is* MT adequate for statistics? We

[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.

have seen in Section 3.1 that for some statistics, the L_1 distance between the theoretical value and the attainable value using a given PRNG is big for even modest sampling and permutation problems. Because MT is equidistributed, we expect large ensemble frequencies will be right, but we expect to see dependence across samples. To detect these dependencies, we have been searching for empirical problems that occur across seeds, large enough to be visible in $O(10^5)$ replications. We have examined simple random sample frequencies, the frequency of derangements and partial derangements, the Spearman correlation between permutations, and other statistics; for everything we have tested so far, MT generates these close to their theoretical distributions. MT must introduce bias in certain statistics, but which ones?

Table 2 PRNGs and sampling algorithms used in common statistical and mathematical software packages.

Package/Language	Default PRNG	Other	SRS Algorithm
SAS 9.2	MT	32-bit LCG	Floyd's ordered hash or Fan et al. 1962
SPSS 20.0	32-bit LCG	MT1997ar	trunc + random indices
SPSS $\leq$ 12.0	32-bit LCG		
STATA 13	KISS 32		PIKK
STATA 14	MT		PIKK
R	MT		trunc + rand indices
Python	MT		mask + rand indices
MATLAB	MT		trunc + PIKK

We recommend the following best practices for using PRNGs to generate random samples and permutations:

- Use a source of real randomness to set the seed with a substantial amount of entropy, e.g., 20 rolls of 10-sided dice.
- Record the seed so your analysis is reproducible.
- Use a cryptographically secure PRNG unless you know that MT is adequate for your problem.
- Avoid standard linear congruential generators, the Wichmann-Hill generator, and PRNGs with small state spaces.
- Use open-source software, and record the version of the software.
- Use a sampling algorithm that does not "waste randomness." Avoid permuting the entire population: do not use PIKK.
- Be aware of discretization issues in the sampling algorithm; many methods assume the PRNG produces $U[0,1]$ or $U[0,1)$ random numbers, rather than (an approximation to) numbers that are uniform on w-bit binary integers.
- Consider the size of the problem: are your PRNG and sampling algorithm adequate?
- Avoid "tests of representativeness" and procedures that reject some samples. They alter the distribution of the sample.

Moreover, we recommend that R and Python upgrade their algorithms to use best practices. First, R should replace the truncation algorithm it uses to generate random integers in the `sample` function (and other functions) with the more precise bit masking algorithm, as discussed in [**?**]. Second, we suggest replacing MT with PRNGs with unbounded state spaces, and cryptographic or near-cryptographic quality. We have developed a CS-PRNG prototype for Python at https://github.com/statlab/cryptorandom. As of this writing, the implementation is slow due to bottlenecks in the way Python casts data types.