Random Sampling: Practice Makes Imperfect

Philip B. Stark and Kellie Ottoboni

Abstract 1) Pigeonhole principle shows that PRNGs cannot be adequate 2) Some sampling algorithms are better than others: ones that use only k PRNs to sample are better than those that use n when $k \ll n$ 3) Common software has these issues.

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

1 Introduction

Resampling statistics, including permutation tests and bootstrapping, crucially rely on software to draw equally likely random samples of data. TO DO: FILL OUT SAMPLING BACKGROUND

In fact, standard software cannot always do this. Whether it is possible depends on the size of the problem and the internals of the software: the sampling algorithm and pseudo-random number generator. The pseudo-random number generator supplies "randomness" and is passed into the sampling algorithm, which uses these pseudo-random numbers to determine which items are included in the sample.

We show mathematically that even for datasets with hundreds of observations, many pseudo-random number generators are incapable of drawing all simple random samples of a smaller size. We apply the pigeonhole principle to random sample "pigeons" and pseudo-random number generator "pigeonholes" to estimate the frac-

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

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

tion of random samples that cannot be attained. There exist statistics that are biased when estimated from these empirical sampling distributions.

Moreover, the choice of sampling algorithm matters: some algorithms put greater demands on the PRNG than others. Algorithms that require more random numbers “use up” a PRNG more quickly than algorithms that depend on fewer. The algorithms themselves assume that their inputs are uniformly random; many software packages use a truncation method of obtaining pseudo-random integers that renders them nonuniform. Even if the PRNG were perfectly uniform, this problem would continue to affect the uniformity of random samples.

There are three general classes of pseudo-random number generators: really bad, “adequate for statistics,” cryptographically secure. Are those deemed “adequate for statistics” really adequate for statistics? This paper explores the question. Section 2 defines pseudo-randomness and gives examples of good and bad pseudo-random number generators. Section 3 gives several mathematical arguments that demonstrate the limited capability of pseudo-random number generators with finite state spaces. Section 4 discusses different sampling algorithms; some “use up” more pseudo-random numbers than others. Section 4.1 presents how the most straightforward algorithm for converting pseudo-random numbers into pseudo-random integers on an arbitrary range goes wrong. Section 5 concludes with recommendations and best practices.

2 Pseudo-random number generators

A pseudo-random number generator (PRNG) is a deterministic algorithm that produces numeric output that is statistically indistinguishable from truly random numbers. A PRNG has several components: an internal *state*, which is typically initialized with a numeric seed set by the user; a function that maps the state to a pseudo-random number (typically 32 bits, but sometimes more); and a function that updates the internal state.

If the state space has finite dimension, then the PRNG must eventually revisit its initial state. The *period* of a PRNG is the maximum, over initial states, of the number of states visited before the state repeats. The period is at most the total number of states. If the period is equal to the total number of states, the PRNG is said to have full period. Many simple PRNGs have their state equal to the output, so the period is at most the dimension of the output. 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. Many PRNGs quickly repeat states if the seed has too many zeros. Some require many iterations before the output behaves like uniform random variates.

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 an additive parameter c. LCGs are attractive because they 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 determines which LCGs have full period m based on the parameters a, c, and m.

Programmers took advantage of early computer hardware and developed LCGs using moduli of the form $m = 2^b$, where b was the integer word size of the computer, in order to facilitate computing multiplication and modulus operations. Such LCGs are automatically defective: only LCGs with a prime m have a full period. Better LCGs have been developed, but they are generally considered to be insufficiently random for use in statistics owing to their short periods (typically 2^{32}) and correlation structure between successive outputs.

LCGs are the simplest type of PRNG that is used in practice. 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. The KISS generator combines 4 generators of three types: two multiply-with-carry generators, the 3-shift register SHR3 and the congruential generator CONG. KISS has a period length over 2^{210}. These PRNGs are all predictable after observation of a small number of outputs. For example, one can backsolve for the LCG constants a, c, and m after only 3 observations.

The Wichmann-Hill PRNG is a sum of three LCGs, used to produce random values on $[0,1)$. The Wichmann-Hill generator is generally not considered adequate for statistics, but was (nominally) the PRNG in Excel for several generations. Excel did not allow the seed to be set, so analyses were not reproducible. Moreover, 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 ([McCullough(2008)]). As of 2014, the IMF Stress tests use Excel; we hope that the PRNG has been upgraded. TO DO: @PHILIP: IS THERE A CITATION FOR THIS?

2.2 Mersenne Twister (MT)

Mersenne Twister (MT) ([Matsumoto and Nishimura(1998)]) 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 considered to be a good enough PRNG for doing statistics. It is the default PRNG in most common software packages, including GNU Octave, Maple, MATLAB, Mathematica, Python, R, Stata, and many more (see Table 2). However, it is not without problems. MT can have slow "burn in," where the first few outputs it produces don't appear statistically random, especially for seeds with many zeros. TO DO: CITE The outputs for close seeds can be close to each other, which can be problematic when running simulations in parallel. Moreover, MT cannot be used for secure applications: its behavior is perfectly predictable from observing 624 successive outputs.

2.3 *Cryptographic hash functions*

The PRNGs described above are based on mathematical relations. 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": $H : \{0,1\}^* \rightarrow \{0,1\}^L$.
- 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, then 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.*

In the context of PRNGs, the corollary states that a PRNG cannot generate all possible random permutations or samples if the number of such items exceeds the size of the PRNG’s state space. The statistical distributions approximated by random sampling will differ from the true distributions, in potentially biased ways, if some items are systematically omitted.

To determine whether a PRNG has sufficiently large state space for a problem, one can compare the number of possible permutations or samples to the state space. (The size of the state space is an upper bound on the period.) The number of permutations of n objects is $n!$ and the number of random samples of k objects from n, without replacement, is $\binom{n}{k}$. These formulae are difficult to apply because factorials grow quickly in n and k. We provide several bounds instead:

- 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.

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

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

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	$2^{32\times 624}$		9.27×10^{6010}
Permutations of 2084	2084!		3.73×10^{6013}
Samples of 1000 out of 390 million	$\binom{3.9\times 10^{8}}{1000}$		$> 10^{6016}$
Fraction of attainable samples	$\frac{2^{32\times 624}}{\binom{3.9\times 10^{8}}{1000}}$		$< 1.66 \times 10^{-6}$

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) [Knuth(1997)]. 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 only 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. [Cormen(2009)] 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 maximum recursion depth is 2000, so this algorithm cannot draw samples of greater than 2000 items.)

The previous algorithms require n to be known. *Reservoir* algorithms, such as Waterman's Algorithm 'R', do not [Knuth(1997)]. 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 [Vitter(1985)].

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, $1 + m2^{-w+1}$ [Knuth(1997)].

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) [R Core Team(2018)] 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.[1]

5 Discussion

We've presented mathematical arguments that uncover issues with common PRNGs. In particular, any PRNG with a finite state space suffers from the "pigeonhole problem": it unable to generate all possible samples or permutations of sufficiently large populations. The assumption of uniform random samples needed to approximate sampling distributions is violated.

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 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. We know from MT's equidistribution property that large ensemble frequencies will be right, but we expect to see issues of 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?

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 PRNG at least as good as the Mersenne Twister, and preferably a cryptographically secure PRNG. Consider the PCG family.
- Avoid standard linear congruential generators and the Wichmann-Hill generator.

[1] 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.

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

- 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.
- 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, we hope that R will replace the truncation algorithm to generate random integers in the `sample` function with the more precise bit masking algorithm. 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.

Acknowledgements If you want to include acknowledgments of assistance and the like at the end of an individual chapter please use the `acknowledgement` environment – it will automatically render Springer's preferred layout.

References

[Cormen(2009)] Thomas H. Cormen. *Introduction to Algorithms*. MIT Press, July 2009.

[Knuth(1997)] 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.

[Matsumoto and Nishimura(1998)] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transac-*

tions on Modeling and Computer Simulation, 8(1):3–30, January 1998. ISSN 10493301. doi: 10.1145/272991.272995. URL http://portal.acm.org/citation.cfm?doid=272991.272995.

[McCullough(2008)] B. D. McCullough. Microsoft Excel's 'Not The Wichmann–Hill' random number generators. *Computational Statistics & Data Analysis*, 52(10):4587–4593, June 2008. ISSN 0167-9473. doi: 10.1016/j.csda.2008.03.006. URL http://www.sciencedirect.com/science/article/pii/S016794730800162X.

[R Core Team(2018)] 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.

[Vitter(1985)] 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.