With Great Power Comes Great Responsibility: Multivariate Permutation Tests and their Numerical Implementation

Philip B. Stark and Kellie Ottoboni

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

Permutation tests are a class of nonparametric hypothesis tests for which the null distribution of the test statistic is calculated based on applying a group action to the observed data. The data X has distribution F_X and takes values in $\mathscr{X}$. Let $\mathscr{G}$ be a group of transformations from $\mathscr{X}$ to $\mathscr{X}$. There is a probability measure μ on $\mathscr{G}$ that is left and right invariant: If $G \subset \mathscr{G}$ is measurable, then $\forall g \in \mathscr{G}$, gG and Gg are measurable, and $\mu(G) = \mu(gG) = \mu(Gg)$.

Under the null hypothesis H_0, $X \sim gX$ for all $g \in \mathscr{G}$. $T : \mathscr{X} \to \mathbb{R}$ is a test statistic on $\mathscr{X}$; large values of T are considered evidence against H_0. We observe data $X = x$. The P-value of H_0, conditional on $X \in \mathscr{G}x$, is the measure of the set of objects in the orbit of $\mathscr{G}$ that are at least as large as $T(x)$:

$$p = \mu\{g \in \mathscr{G} : T(gx) \geq T(x)\}.$$

In practice, the orbit $\mathscr{G}$ is often too large to enumerate. Instead, the P-value is estimated by Monte Carlo simulation. To estimate p, first generate a random sample of group actions, $\{G_j\}$ independent and identically distributed with measure μ. The test statistic $1_{T(G_j x) \geq T x}$ is unbiased: it follows a Bernoulli distribution with mean p.

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

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

This simple procedure enables one to construct tests, including sequential tests, and upper confidence bounds for p.

However, this procedure crucially relies on the ability to generate independent and identically distributed $\{G_j\}$. In fact, standard software cannot do this in all cases. Whether it is possible depends on the size of the problem (orbit) and the internals of the software: the sampling algorithm and pseudo-random number generator. We show that even for datasets with hundreds of observations, simple pseudo-random number generators are incapable of drawing all simple random samples of a smaller size.

The paper is organized as follows. 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 talks about different sampling algorithms; some "use up" more pseudo-random numbers than others. Section 4.3 presents how the most straightforward algorithm for converting pseudo-random numbers into pseudo-random integers on an arbitrary rangegoes wrong. Section 5 concludes.

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 seed set by the user; a function which maps the state to a pseudo-random number (typically 32 bits, but sometimes more); and a function that updates the internal state.

The *period* of a PRNG is the maximum, over initial states, of the number of states visited before the state repeats. For simple PRNGs, the state is equal to the output, so the period is the same as the dimension of the output. Better PRNGs generally use a state space with much larger dimension than the dimension of the output. The period is at most the total number of states.

Some PRNGs are sensitive to the initial state. Many PRNGs don't do well if the seed has too many zeros. Some require many iterations before the output behaves well (i.e. indistinguishable from uniform random variates).

There are three general classes of PRNGs: really bad, "adequate for statistics," cryptographically secure. Are those deemed "adequate for statistics" really adequate for statistics?

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 memory. The behavior of LCGs is well-understood from fundamental number theory. For instance, the following theorem describes which LCGs have full period m.

Theorem 1 (Hull-Dobell Full Period Theorem). *The period of an LCG is m for all seeds X_0 if and only if*

- *m and c are relatively prime,*
- *$a-1$ is divisible by all prime factors of m, and*
- *$a-1$ is divisible by 4 if m is divisible by 4.*

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 violate the first principle of choosing good parameters, as no non-prime m can yield an LCG with 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. 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}. TO DO: MENTION LAGGED FIBONACCI SEQUENCES, XORSHIFT 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.

Lagged Fibonacci, KISS, xorshift family, PCG, ...

The Wichmann-Hill PRNG is a sum of three LCGs, used to produce random values on $[0,1)$. Given three seed values s_1, s_2, s_3, the next value r is given by

$$\begin{aligned} s_1 &= 171 s_1 \mod (30269) \\ s_2 &= 172 s_2 \mod (30307) \\ s_3 &= 170 s_3 \mod (30323) \\ r &= (s_1/30269 + s_2/30307 + s_3/30323) \mod (1) \end{aligned}$$

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: CITE 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 find a pattern to use to undo the function application. Cryptographic primitives can be easily turned into PRNGs, as all of the properties that make such functions cryptographically secure are properties of pseudorandomness. Cryptographic hash functions are one such primitive. A cryptographic hash function H has the following properties:

- H produces a fixed-length "digest" 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 message, where i counts how many PRNs have already been generated
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 2 (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, this means that if the number of possible random permutations or samples exceeds the state space of a PRNG, then that PRNG cannot generate them all. This is a problem: the statistical distributions approximated by randomization 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 permutations or samples to the state space. The number of permutations of n objects is $n!$ and the number of random samples of k objects from n, without replacement, is $\binom{n}{k}$. Despite these formulae, they are difficult to calculate because factorials grow quickly in n and k. We offer 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 attempts to put permutation pigeons into 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 and random sampling. 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	$\frac{2^{32\times 624}}{\binom{3.9\times 10^8}{1000}}$		$< 1.66 \times 10^{-6}$

3.1 L1 bounds

Suppose $\mathbb{P}_0$ and $\mathbb{P}_1$ are probability distributions on a common measurable space.

If there is some set S for which $\mathbb{P}_0 = \varepsilon$ 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

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

- If PRNG has n states and want to generate $N > n$ equally likely outcomes, at least $N - n$ outcomes will have probability zero instead of $1/N$.
- $\|\text{true} - \text{desired}\|_1 \geq 2 \times \frac{N-n}{N}$

4 Sampling algorithms

Given a good source of randomness, many ways to draw a simple random sample.

One basic approach is like shuffling a deck of n cards, then dealing the top k: permute the population at random, then take the first k elements of the permutation to be the sample. There are a number of standard ways to generate a random permutation – i.e., to shuffle the deck. If we had a way to generate independent, identically distributed (iid) $U[0,1]$ random numbers, we could sample k out of n as follows:

Algorithm 1 PIKK: Permute indices and keep k

Require: $n \geq k \geq 0$

1: Assign IID uniform values on $[0,1]$ to the n elements of the population
2: Sort the population according to these values (break ties randomly)
3: Take the top k to be the sample

This amounts to generating a random permutation of the population and throwing out all but the first k. If the numbers really are independent and identically distributed, every permutation is equally likely, and it follows that the first k are an SRS. If permutations are not equiprobable, then samples generated using this algorithm may 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.

There are more efficient ways to generate a random permutation than assigning a number to each element and sorting. One example is the"Fisher-Yates shuffle" or "Knuth shuffle" (Knuth attributes it to Durstenfeld).

Algorithm 2 Fisher-Yates-Knuth-Durstenfeld shuffle (backwards version)

1: **for** $i = 1, \ldots, n-1$ **do**
2: $J \leftarrow$ random integer uniformly distributed on $i, \ldots, n$
3: $a[J], a[i] \leftarrow a[i], a[J]$
4: **end for**

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.

Algorithm 3 Fisher-Yates-Knuth-Durstenfeld shuffle (streaming version)

$i \leftarrow 0$
$a \leftarrow []$
while there are records left **do**
    $i \leftarrow i+1$
    $J \leftarrow$ random integer uniformly distributed on $\{1,\ldots,i\}$
    if $J < i$ **then**
        $a[i] \leftarrow a[J]$
        $a[J] \leftarrow$ next record
    else
        $a[i] \leftarrow$ next record
    end if
end while
return a

4.1 `Random_Sample`

This algorithm is attributed to [Cormen(2009)]. It is a recursive algorithm that requires only k random integers and does not require sorting.

Algorithm 4 *Random_Sample*

Require: $n \geq k \geq 0$

function RANDOM_SAMPLE(n,k)
    if k is 0 **then return** the empty set
    else
        $S \leftarrow$ `Random_Sample`$(n-1,k-1)$
        $i \leftarrow$ random integer uniformly distributed on $\{1,\ldots,n\}$
        if i is in S **then**
            $S \leftarrow S \cup \{n\}$
        else
            $S \leftarrow S \cup \{i\}$
        end if, **return** S
    end if
end function

4.2 Reservoir algorithms

The previous algorithms require n to be known. There are *reservoir* algorithms that do not. Moreover, the algorithms are suitable for streaming (aka *online*) use: items are examined sequentially and either enter into the reservoir, or, if not, are never revisited.

Algorithm 'R', Waterman (per Knuth, 1997) + Put first k items into the reservoir

+ when item $k+j$ is examined, either skip it (with probability $j/(k+j)$) or swap for a uniformly selected item in the reservoir (with probability $k/(k+j)$)

+ naive version requires at most $n-k$ pseudo-random numbers

+ closely related to FYKD shuffle

Algorithm Z, Vitter (1985)

Much more efficient than Algorithm 'R', using random skips. Essentially linear in k.

Note: Vitter proposes using the (inaccurate) $J = \lfloor mU \rfloor$ to generate a random integer between 0 and m in both algorithm 'R' and algorithm 'Z'. Pervasive!

4.3 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.

If $m > 2^w$, at least $m - 2^w$ values will have probability 0 instead of probability $1/m$. Unless m is a power of 2, the distribution of Y isn't uniform on $\{1,\ldots,m\}$. 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 like `sample`.

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. 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 function `numpy.random.randint()` (Version 1.14) generates pseudorandom integers.[1]

[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.

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": they are unable to generate all possible samples or permutations of sufficiently large populations. The assumption of uniform random samples needed for valid permutation inference 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 that 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?

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 have some recommendations for 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.
- 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.

+ Building library for permutation tests in Python http://statlab.github.io/permute/
- Includes NPC
+ Smaller library of examples in R to go with Pesarin Salmaso's text https://github.com/statlab/permuter
+ Working on plug-in CS-PRNG replacement for MT in Python https://github.com/statlab/cryptorandom
- Bottleneck in type casting in Python
Recommendations
+ Replace the standard PRNGs in R and Python with PRNGs with unbounded state spaces, and cryptographic or near-cryptographic quality
- Consider using AES in counter mode, since Intel chips have hardware support for AES
+ Replace 'floor(1+nU)' in R's 'sample()' with bit mask algorithm

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.

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