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 don't produce uniform randomness. 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 an intuitive, but simplistic, 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 frequently taught 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 (not for their security properties, but for their statistical ones), and offer weaker PRNGs only as an option. Software should not use inaccurate methods, such as generating a random sample by randomly permuting the population and taking the first *k* items and generating random integers by multiplying a binary fraction or float by a constant and rounding the result, when more accurate methods are available.

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, perform 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 software is adequate for these tasks. Textbooks give algorithms for generating random integers, random samples, and independent and identically distributed (IID) random variates 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 [19, 7, 2, 16, 15].

We show here that this assumption is incorrect for algorithms in many 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 PRNGs 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 through a full period and begin recycling values because the maximum number of items that common PRNGs can permute ranges from 13 to 2084, far smaller than many data sets. Others require uniformly distributed integers (as opposed to the approximately $U[0,1)$ PRNG outputs) 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 exist bounded functions of random samples whose expectations with respect to those two distributions differ substantially.

We divide PRNGs into three broad classes: 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 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 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. An ideal PRNG has output that is statistically indistinguishable from random, uniform, IID bits. Cryptographically secure PRNGs approach this ideal—the bits are (or seem to be) computationally indistinguishable from IID uniform bits—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 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 visits before returning to a state already 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 dimension much larger 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 number theory. For instance, the Hull-Dobell theorem [5] gives necessary and sufficient conditions for a LCG to have full period for all seeds, and there are upper bounds on the number of hyperplanes of dimension k that contain all k-tuples of outputs, as a function of m [10]. A smaller number of hyperplanes containing all k-tuples indicates greater correlation among the PRNG outputs.

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. This led to wide propagation of a particularly terrible choice, RANDU, originally introduced on IBM mainframes [6, 9]. LCGs of this form cannot have full period because m is not prime. Better LCGs have been developed—and some are used in commercial statistical software packages—but they are still generally considered inadequate for statistics because of their short periods (typically $\leq 2^{32}$) and correlation among outputs.

The Wichmann-Hill PRNG is a sum of three normalized LCGs; its output is in $[0,1)$. 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 reported that the PRNG occasionally gave a negative output [12]. As of 2014, IMF banking Stress tests used Excel simulations [13].

Other approaches to generating pseudo-random numbers have been proposed, and PRNGs can be built by combining simpler ones (carefully–see [6] on "randomly" combining PRNGs). For instance, the KISS generator combines four 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 determine the LCG constants a, c, and m from only 3 outputs.

2.2 Mersenne Twister (MT)

Mersenne Twister (MT) [11] is a "twisted" generalized feedback shift register, a 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-equidistributed 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. The state is a 624×32 binary matrix.

MT is the default PRNG in common languages and software packages, including Python, R, Stata, GNU Octave, Maple, MATLAB, Mathematica, and many more (see Table 2). We show below that it is not adequate for statistical analysis of modern data sets. Moreover, MT can have slow "burn in," especially for seeds with many zeros [18]. The outputs for close seeds can be similar, which can affect distributed computations.

2.3 Cryptographic hash functions

The PRNGs described above are quick to compute but predictable, and their outputs are easy to distinguish from actual random bits [8]. Cryptographers have devoted a great deal of energy to inventing cryptographic hash functions, which can be easily used to create PRNGs, as the properties that make such functions cryptographically secure are properties of pseudo-randomness.

A *cryptographic hash function H* is a function 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 (this is the basis of hashcash “proof of work” for Bitcoin and some other distributed ledgers)
- 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 M produce unpredictable, big changes to $H(M)$.
- outputs of H are equidistributed: bits of the hash are essentially IID 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. We can construct a simple hash-based PRNG with the following procedure, which we first learned about from Ronald L. Rivest:

1. Generate a random string S with a substantial amount of entropy, e.g., 20 rolls of a 10-sided die.
2. Set $i = 0$. (i is the number of values generated so far.)
3. Let “S,i” be the state.
4. Set $X_i = \text{Hash}(S,i)$, interpreted as a (long) hexadecimal number.
5. Increment i and return to step 4 to generate the next output.

Since a message can be arbitrarily long, this PRNG has an unbounded state space.

3 Counting permutations and samples

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: that depends on the quality of the PRNG, not just whether the number of states is less than, equal to, or greater than the number of permutations or samples.

Nonetheless, it follows that no PRNG with a finite state space can be “adequate for statistics” for every statistical problem.

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 for counting pigeonholes:

- 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 The pigeonhole principle applied to PRNGs, samples, and permutations. For a PRNG of each size state space, the table gives examples where some samples or permutations must be unobtainable.

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 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 $2 \times \frac{N-n}{N}$. As seen in Table 1, the fraction of attainable samples or permutations can be less than 1% in many cases, making the bias nearly 2.

4 Sampling algorithms

Given a source of randomness, there are many ways to draw a simple random sample. A common approach is like shuffling a deck of n cards, then dealing the top k: assign a (pseudo-)random number to each item, sort the items based on that number to produce a random permutation of the population, then take the first k elements of the permuted list to be the sample [19, 7, 2]. We call this algorithm PIKK: Permute indices and keep k.

If the random 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 that permutations are equiprobable; if not, then samples generated using this algorithm generally will not be either. (Of course, there's a possibility that even if the permutations are not equiprobable, the samples still are, but there's no reason to think they would be and certainly no proof.) Furthermore, this algorithm is inefficient: it requires generating n random numbers and then an $O(n \log n)$ sorting operation.

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) [6]. This algorithm involves generating independent random integers on various ranges, but does not require sorting. There is also a version suitable for *streaming*, i.e., permuting a list that has an (initially) unknown number of elements. Generating n pseudo-random numbers places more demand on a PRNG than some sampling algorithms discussed below, which only require k pseudo-random numbers.

One simple method to draw a random sample of size k from a population of size n is to draw k integers at random without replacement from $\{1, \ldots, n\}$, then take the items with those indices to be the sample. [1] provide an elegant recursive algorithm to draw random samples of size k out of n; it requires the software recursion limit to be at least k. (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 algorithms mentioned so far require n to be known. *Reservoir* algorithms, such as Waterman's Algorithm R, do not [6]. Moreover, reservoir algorithms are suitable for streaming: 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, using random skips to reduce runtime to be essentially linear in k [20].

4.1 Pseudo-random integers

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

A 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 $Y \equiv 1+\lfloor mX \rfloor$ ([16, 15]). In practice, PRNG outputs are not $U[0,1)$: they are derived by normalizing a value that is (supposed to be) uniformly distributed on w-bit integers.

The distribution of Y is not 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, $1+m2^{-w+1}$ [6].

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 3.5.1) [17] uses to generate pseudo-random integers, which eventually are used in the main sampling functions. Duncan Murdoch devised a simple simulation that shows how large the problem can be: for $m = (2/5)\times 2^{32} = 1,717,986,918$, the `sample()` function generates about 40% even numbers and about 60% odd numbers [1].

A more accurate way to generate random integers on $\{1,\ldots,m\}$ is to use pseudo-random bits directly. This is not a new idea; [4] describe essentially the same procedure to draw integers by hand from random digit tables. The integer $m-1$ can be represented with $\mu = \lceil \log_2(m-1) \rceil$ 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) and interpreting it as an integer. If the integer is larger than $m-1$, then discard it and draw another μ bits until the μ bits represent an integer less than or equal to $m-1$. When that occurs, return the integer, plus 1. This procedure potentially requires throwing out (in expectation) almost half the draws if $m-1$ is just below a power of 2, but the algorithm's output will be

[1] https://stat.ethz.ch/pipermail/r-devel/2018-September/076827.html, last visited 17 October 2018

uniformly distributed (if the input bits are). 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 items. The Mersenne Twister (MT) cannot generate all permutations of 2084 items.

Table 2 lists the PRNGs and sampling algorithms used in common statistical packages. Most use MT as their default PRNG; *is* MT adequate for statistics? Section 3.1 shows 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 k-equidistributed, we should expect that ensemble frequencies will be approximately what they should be. However, we expect dependence across samples. We have been searching for empirical problems that occur across seeds, large enough to matter in $O(10^5)$ replications or less. We have examined simple random sample frequencies, the frequency of derangements and partial derangements, the Spearman correlation between permutations, and other statistics; so far, we have not found a statistic with consistent bias large enough to be detected in $O(10^5)$ replications. MT must produce bias in some statistics, but which?

Table 2 PRNGs and sampling algorithms used in common statistical and mathematical software packages. The trunc algorithm uses the flawed method to generate pseudo-random integers on a specified range. The mask algorithm uses the favored method to do so.

Package/Language	Default PRNG	Other	SRS Algorithm
SAS 9.2	MT	32-bit LCG	Floyd's ordered hash or [3]
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 + random indices
Python	MT		mask + random indices
MATLAB	MT		trunc + PIKK

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

[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.

- 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.
- Beware 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?

Moreover, we recommend that R and Python upgrade their algorithms to use best practices. We think 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 [14]. And we suggest R and Python use cryptographically secure PRNGs by default, with an option of using MT instead in case the difference in speed matters. We have developed a CS-PRNG prototype for Python at https://github.com/statlab/cryptorandom. The current implementation is unnecessarily slow, due to bottlenecks in the way Python data type conversions. We are developing a faster C implementation.

Acknowledgements We are grateful to Ronald L. Rivest for many helpful conversations.

References

1. Cormen, T.H.: Introduction to Algorithms. MIT Press (2009)
2. Dahlberg, L., McCaig, C.: Practical Research and Evaluation: A Start-to-finish Guide for Practitioners. Sage (2010)
3. Fan, C., Muller, M.E., Rezucha, I.: Development of sampling plans by using sequential (item by item) selection techniques and digital computers. Journal of the American Statistical Association **57**(298), 387–402 (1962)
4. Hodges Jr, J., Lehmann, E.: Basic Concepts of Probability and Statistics, vol. 48. SIAM (1970)
5. Hull, T., Dobell, A.: Random number generators. SIAM Review **4**(3), 230–254 (1962). DOI 10.1137/1004061
6. Knuth, D.E.: Art of Computer Programming, Volume 2: Seminumerical Algorithms, 3 edition edn. Addison-Wesley Professional, Reading, Mass (1997)
7. LeBlanc, D.C.: Statistics: Concepts and Applications for Science, vol. 2. Jones & Bartlett Learning (2004)
8. L'Ecuyer, P., Simard, R.: TestU01: A C Library for Empirical Testing of Random Number Generators (2007)
9. Markowsky, G.: The sad history of random bits. Journal of Cyber Security **3**, 126 (2014). DOI 10.13052/jcsm2245-1439.311

10. Marsaglia, G.: Random numbers fall mainly in the planes. Proceedings of the National Academy of Sciences of the United States of America **61**(1), 25–28 (1968)
11. Matsumoto, M., Nishimura, T.: Mersenne Twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Transactions on Modeling and Computer Simulation **8**(1), 3–30 (1998). DOI 10.1145/272991.272995. URL http://portal.acm.org/citation.cfm?doid=272991.272995
12. McCullough, B.D.: Microsoft Excel's 'Not The Wichmann–Hill' random number generators. Computational Statistics & Data Analysis **52**(10), 4587–4593 (2008). DOI 10.1016/j.csda.2008.03.006
13. Ong, L.: A Guide to IMF Stress Testing: Methods and Models. International Monetary Fund (2014). DOI 10.5089/9781484368589.071I
14. Ottoboni, K., Stark, P.: Random problems with R. https://arxiv.org/abs/1809.06520 (2018)
15. Peck, R., Olsen, C., Devore, J.: Introduction to Statistics and Data Analysis. Cengage Learning (2011)
16. Press, W., Flannery, B.P., Teukolsky, S., Vetterling, W.: Numerical Recipes in C: The Art of Scientific Computing. Cambridge University Press (1988)
17. 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
18. Saito, M., Matsumoto, M.: SIMD-Oriented Fast Mersenne Twister: a 128-bit Pseudorandom Number Generator. In: Monte Carlo and Quasi-Monte Carlo Methods 2006, pp. 607–622. Springer, Berlin, Heidelberg (2008)
19. Stine, R., Foster, D.: Statistics for Business: Decision Making and. Addison-Wesley SOFTWARE-JMP (2014)
20. Vitter, J.S.: Random Sampling with a Reservoir. ACM Trans. Math. Softw. **11**(1), 37–57 (1985). DOI 10.1145/3147.3165