Simple Random Sampling: Not So Simple

Kellie Ottoboni
with Philip B. Stark and Ron Rivest

Department of Statistics, UC Berkeley
Berkeley Institute for Data Science

Qualifying Exam
January 23, 2017

University of California, Berkeley
Department of Statistics

BERKELEY INSTITUTE
FOR DATA SCIENCE

WILEY

Permutation Tests for Complex Data

Theory, Applications and Software

Fortunato Pesarin • Luigi Salmaso

WILEY SERIES IN PROBABILITY AND STATISTICS

Simple Random Sampling

Simple random sampling: drawing k objects from a group of n in such a way that all $\binom{n}{k}$ possible subsets are equally likely.

In practice, it is difficult to draw truly random samples.

Instead, people tend to draw samples using

1. A **pseudorandom number generator** (PRNG) that produces sequences of bits, plus
2. A sampling algorithm that maps a set of pseudorandom numbers into a subset of the population

Most people take for granted that this procedure is a sufficient approximation to simple random sampling.

Using computers to sample

Social applications may require the PRNG to produce all possible samples – e.g. jury duty summons, gaming machines, lottery tickets.

Marsaglia [2003]

> The preceding examples indicate that social applications may require the [PRNG] is able to select from every possible outcome, a requirement that can be satisfied with RNGs having many random seed values... Thus, multiple-seed [PRNGs] seem desirable for some applications and mandatory for others.

But passing PRNs into a sampling algorithm adds an additional layer to the problem.

Simple Random Sampling

If PRNGs are unable to generate all simple random samples, the problem will be even worse for other methods: permutation, bootstrap samples, MCMC, Monte Carlo integration...

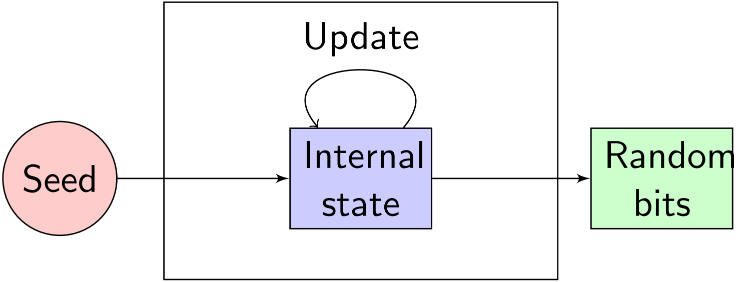
Number of possible samples for $n = 100$, $k = 50$

SRSs	$\binom{n}{k}$	$\binom{100}{50} \approx 10^{29}$
Bootstrap Samples	n^k	$100^{50} = 10^{100}$
Permutations	$n!$	$100! \approx 10^{158}$

Pseudorandom number generators (PRNGs)

A PRNG is a deterministic function with several components:

- A user-supplied **seed value** used to set the internal state
- A function that maps the **internal state to random bits**
- A function that **updates the internal state**



Pigeons and Pigeonholes

Theorem (Pigeonhole Principle)

If there are n pigeonholes and $m > n$ pigeons, then there exists at least one pigeonhole containing more than one pigeon.

(Wikipedia)

Pigeons and Pigeonholes

Theorem (Pigeonhole Principle)

If there are n pigeonholes and $m > n$ pigeons, then there exists at least one pigeonhole containing more than one pigeon.

(Wikipedia)

Corollary (Too few pigeons)

If $\binom{n}{k}$ is greater than the size of a PRNG's state space, then the PRNG cannot possibly generate all samples of size k from a population of n.

Pigeons and Pigeonholes

Period of 32-bit linear congruential generators (the most basic acceptable PRNG):

at most $2^{32} \approx 4 \times 10^9$

Samples of size 10 from 50:

$\binom{50}{10} \approx 10^{10}$

More than half of samples cannot be generated

Pigeons and Pigeonholes

Period of 32-bit linear congruential generators (the most basic acceptable PRNG):

at most $2^{32} \approx 4 \times 10^9$

Samples of size 10 from 50:

$\binom{50}{10} \approx 10^{10}$

More than half of samples cannot be generated

Period of Mersenne Twister (standard PRNG in Statistics):

$2^{32\times 624} \approx 2 \times 10^{6010}$

Permutations of 2084 objects:

$2084! \approx 3 \times 10^{6013}$

Less than 0.01% of permutations can be generated

Overview

- Some sampling algorithms are better than others – look under the hood of your software
- PRNGs for Statistical applications should be judged on how well they produce random samples when passed into a reasonable sampling algorithm

I will show

- New tests for pseudorandomness based on simple random sampling
- New PRNGs based on cryptographic hash functions

Contents

Sampling Algorithms

Given a sequence of (pseudo)random numbers, how do we use them to draw a SRS?

Two general strategies:

- "Shuffle the deck" and take the top k as the sample
- Number the population, select k random integers, and take the corresponding items

PIKK

Algorithm 1 PIKK: Permute indices and keep k

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

- Relies on assumption that all permutations are equally likely
- Inefficient: requires n PRNs and $O(n \log n)$ sorting operation
- Possibly the basis for the `sample` function in Stata

TO DO: WHO ELSE RECOMMENDS PIKK? DO A LIT SEARCH

Shuffling algorithms

- Knuth shuffle: requires $n-1$ random integers, but no sorting. This is what `np.random.choice` does.

Algorithm 2 Fisher-Yates-Knuth-Durstenfeld shuffle

for $i = 2, \ldots, n$ **do**
    $J \leftarrow$ random integer uniformly distributed on $1, \ldots, i$
    $(a[J], a[i]) \leftarrow (a[i], a[J])$
Take the first k to be the sample

Proof

- Reservoir algorithms: Algorithm R (Waterman, Knuth 1997), Algorithm Z (Vitter 1985) are related to shuffling and don't require knowing the population size a priori

Random indices

Algorithm 3 Uniform random indices

$\tilde{n} \leftarrow n$
Population indices $\leftarrow \{1, \ldots, n\}$
for $i = 1, \ldots, k$ **do**
    $w \leftarrow$ A random integer on $\{1, \ldots, \tilde{n}\}$
    $j \leftarrow$ The wth element in Population indices
    Sample indices $\leftarrow$ Sample indices $\cup \{j\}$
    Population indices $\leftarrow$ Put last remaining index in place w
    $\tilde{n} \leftarrow \tilde{n} - 1$
Take the items with selected Sample indices

- Method used by R `sample`, Python `random.sample`
- More efficient: uses only k PRNs and no sorting

Generating (non)uniform integers

- These algorithms depend on uniformly distributed integers
- A common way to obtain an integer in the range $\{1, \ldots, m\}$ using a PRN U on $[0, 1)$ is $\lfloor mU \rfloor + 1$.
- Unless m is a multiple of 2^w, $\lfloor mU \rfloor$ will not truly be uniform! (Knuth [1997])

Lemma

For $m < 2^w$, the ratio of the largest to smallest selection probability is, to first order, $1 + m2^{-w}$.

Proof

TO DO: THINK CAREFULLY ABOUT IMPORTANT VALUES OF m AND w

Generating (non)uniform integers

- A better way to generate integers on $\{1, \ldots, m\}$: Let

$$w = \begin{cases} \log_2(m) & \text{if } m \text{ is a power of 2} \\ \lfloor \log_2(m) \rfloor + 1 & \text{otherwise} \end{cases}$$

  Generate a w-bit integer J. If $J > m$, discard and repeat.

- Possibly slow: will discard nearly half of draws when m is close to 2^{w-1}
- Resulting integers will truly be uniform

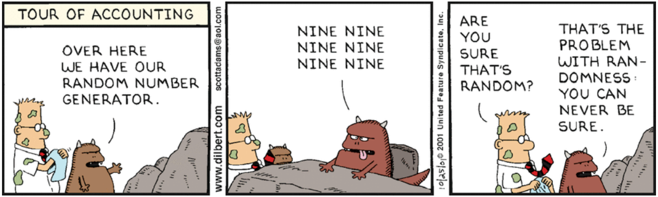
Sampling Algorithms

Package	Sampling algorithm	Random integer algorithm
R	random indices	variant of floor method
Python `random`	random indices	discard method
Numpy `random`	shuffle and keep k	discard method
Stata	?	?

- Stata blogs recommend people use PIKK when coding up sampling themselves. But Stata's sort function is randomized by default. Not reproducible! (Schumm [2006])
- R creates random integers the "wrong" way – working to submit a bug report

Contents

Pseudorandomness

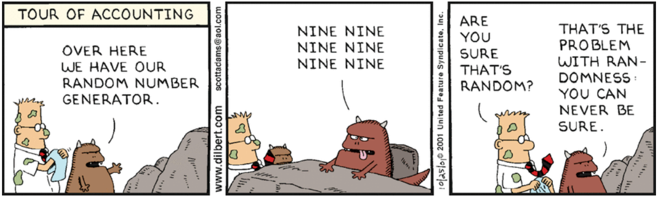


Dilbert

Pseudorandom: deterministic, but having the same relevant statistical properties as if random

- Uniformity: values and sequences of values should be equiprobable
- Independence: lack of serial correlation, unpredictable

Pseudorandomness



Dilbert

Pseudorandom: deterministic, but having the same relevant statistical properties as if random

- Uniformity: values and sequences of values should be equiprobable
- Independence: lack of serial correlation, unpredictable

PRNGs can't do this perfectly.

- They are **deterministic**: knowing input tells you the output.
- Most are **periodic**: they eventually produce the same sequence of values.
- They have some predictable mathematical structure.

What makes a PRNG

- Mimics a random sequence (statistically indistinguishable)
- Unpredictable. This is different from random - if it's deterministic, then it's predictable to some degree
- Fast and memory efficient
- Desirable, but not essential:
  - Jump-ahead feature to efficiently skip through random numbers, generate multiple streams for parallel applications
  - Easy seeding: should be simple to set the state from the seed, robust to value supplied

Testing PRNGs

- “Uniform” and “independent” are broad criteria – many ways to define and check for these properties
  - Uniformity at varying levels of granularity
  - Independence within and between subsequences
- Test batteries:
  - Diehard battery (Marsaglia [1995])
  - NIST Statistical Test Suite (Soto [1999], Rukhin et al. [2010])
  - TestU01 suite (L’Ecuyer and Simard [2007])

Testing for uniformity

1. Kolmogorov-Smirnov test: values should appear IID $U[0,1]$

Testing for uniformity

1. Kolmogorov-Smirnov test: values should appear IID $U[0, 1]$
2. Chi-squared test: break values or sequences of values into categories with known frequencies under the null

Testing for uniformity

1. Kolmogorov-Smirnov test: values should appear IID $U[0,1]$
2. Chi-squared test: break values or sequences of values into categories with known frequencies under the null
3. **New proposal:** Range test
   Break values into equally probable categories and compute the range of observed frequencies

$$R = \max_i O_i - \min_i O_i$$

   R has a complicated distribution based on multinomial distribution... use asymptotic approximation from Young [1962]:

$$\mathbb{P}(R \leq r) \approx P(W_N \leq (r - (2B)^{-1})(N/B)^{1/2})$$

   where W_N denotes the sample range of N independent standard normal random variables.

Testing for independence

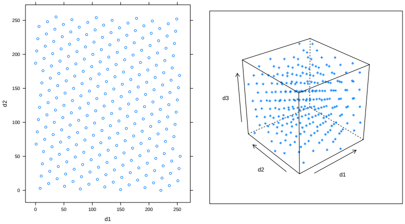
1. Gap test: count length of sequence between values in a given range
2. Permutation test: look at ordering of values in subsequences of length t
3. Serial correlation test: correlation between consecutive pairs of values
4. Many more possibilities!

Geometric tests

- Some PRNGs generate points with geometric structure (more on this later)
- The **spectral test** in dimension d considers a set of parallel hyperplanes that covers all points $(x_n, \ldots, x_{n+d-1})$.

$$\nu_d = 1/\max \text{distance between parallel hyperplanes}$$

Idea: More information needed to describe structure $\iff$ less correlation between consecutive PRNs



$$X_{n+1} = (137 X_n + 187) \mod 256$$

New tests for PRNGs

- Some sampling algorithms use PRNGs in ways that are not covered by these tests
- E.g. Algorithm 3 for sampling by uniform random indices
  - To generate a SRS of size k from n, obtain PRNs $(U_1, \ldots, U_k)$ where U_j is uniform on $\{1, \ldots, n-j+1\}$
  - The sequences $(U_1, \ldots, U_k)$ should themselves be equiprobable

New tests for PRNGs

- Some sampling algorithms use PRNGs in ways that are not covered by these tests
- E.g. Algorithm 3 for sampling by uniform random indices
  - To generate a SRS of size k from n, obtain PRNs $(U_1, \ldots, U_k)$ where U_j is uniform on $\{1, \ldots, n-j+1\}$
  - The sequences $(U_1, \ldots, U_k)$ should themselves be equiprobable
- **Proposal:** test by generating a large "sample" of B SRSs

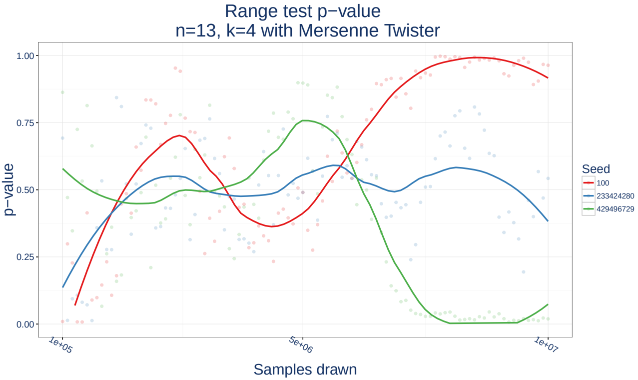
$$H_0 : \mathbb{P}(\text{SRS}_i) = \frac{1}{\binom{n}{k}} \text{ for all } \binom{n}{k} \text{ possible SRSs}$$

$$H_1 : \mathbb{P}(\text{SRS}_i) \neq \frac{1}{\binom{n}{k}} \text{ for some SRS}$$

- Under H_0, the number of times each SRS is observed follows a multinomial distribution with B trials and equal selection probabilities $1/\binom{n}{k}$

New tests for PRNGs

Chi-squared or range test: generate a fixed number of samples starting with seed $\mathcal{S}$ and look at sample frequencies.



Unsatisfactory: how do we choose number of samples?

Sequential probability ratio test

- A **sequential probability ratio test** is a hypothesis testing procedure that weighs the evidence of each observation as it comes in (Wald [1973])
- Tests $H_0 : (X_n) \sim f_0$ against $H_1 : (X_n) \sim f_1$ by checking the likelihood ratio (LR) after each observation

Algorithm 4 Sequential probability ratio test

```
α, β ← desired type I and II error rates
LR ← 1
while β/(1−α) < LR < (1−β)/α do
    X_m ← A new observation
    LR ← LR × f_1(X_m)/f_0(X_m)
if LR ≤ β/(1−α) then
    Fail to reject the null hypothesis; stop
if LR ≥ (1−β)/α then
    Reject the null hypothesis; stop
```

Sequential probability ratio test

Sequential test for multinomial random variables: reduce it to a comparison of Bernoullis

- Fix s and define

  $B(n) \equiv \mathbb{I}(n\text{th sample is among the } s \text{ most frequent samples before step } n)$

- $B(n) \sim \text{Bernoulli}(p)$. Then we may test

$$H_0 : p = p_0 = s\binom{n}{k}^{-1}$$

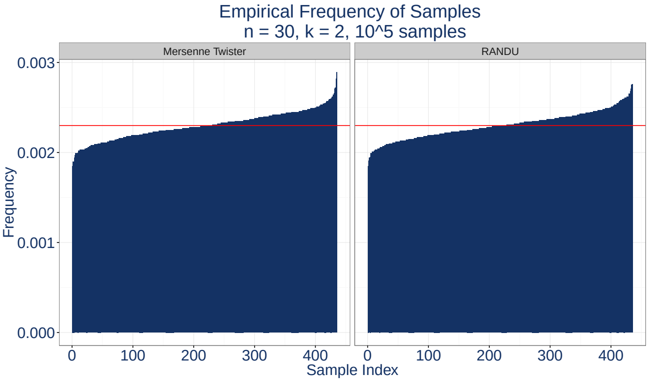
$$H_1 : p = p_1 > s\binom{n}{k}^{-1}$$

- **Peculiar:** $B(n)$ depends on the samples that have been observed before the nth!
  But under the null, p_0 is fixed and doesn't depend on what the samples actually are.

Sequential probability ratio test

Preliminary results:

- Simulations show that the test has the correct level under the null
- Issues of power: how to choose s and p_1? Depends on what we believe the alternative is.



Contents

Linear Congruential Generators

LCGs have the form

$$X_{n+1} = (aX_n + c) \mod m$$

Smart choices of a,c, and m can make the LCG fast to compute and more or less random

Theorem (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.*

The good, the bad, and the ugly

(Knuth, 1997)

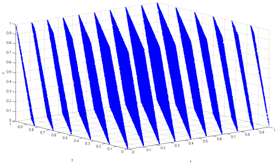
"Random numbers should not be generated with a method chosen at random."

The good, the bad, and the ugly

(Knuth, 1997)

"Random numbers should not be generated with a method chosen at random."

Marsaglia [1968] proved that n-tuples of numbers generated by any LCG will lie on parallel hyperplanes, making them especially non-random.



Triples of RANDU lie on 15 planes in 3D space

$$x_{n+1} = (65539 x_n) \mod 2^{31}$$

(Wikipedia)

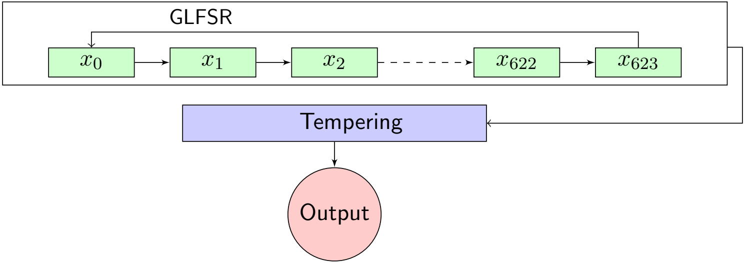
Better LCGs

- Super-Duper: $X_{n+1} = (69069 X_n) \mod 2^{32}$
  - $69069 = 3 \times 7 \times 11 \times 13 \times 23$
  - Considered a good LCG, passes spectral tests in low dimensions
- MINSTD: $X_{n+1} = (16807 X_n) \mod (2^{31} - 1)$
  - $16807 = 7^5$
  - The "minimum standard" against which other PRNGs should be judged (Park and Miller [1988])
- KISS: combines Super-Duper with two other PRNGs
  - Was previously the only PRNG in Stata
  - Period length over 2^{210}
- Wichman-Hill PRNG: a sum of 3 LCGs
  - Was previously the only PRNG in Excel
  - Faulty implementation didn't allow seeding and sometimes produced negative values (McCullough [2008])

Linear Congruential Generators

- Fast to compute and requires little memory
- Some LCGs are more random than others – depends on choosing good constants
- Not unpredictable. We only need 2 values to determine the constants.
- Possible to do jump ahead using mathematical formulas.

Mersenne Twister (Matsumoto and Nishimura [1998])



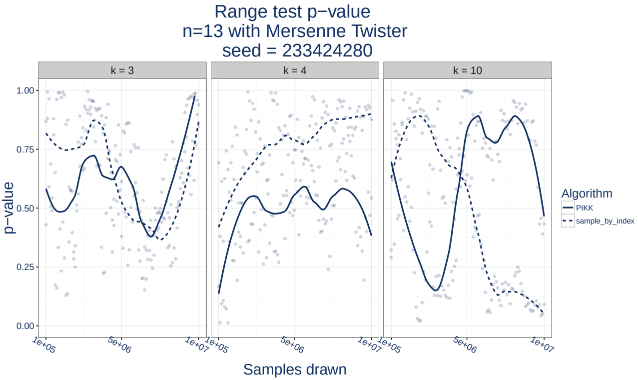
- A "twisted" generalized linear feedback shift register: a complicated sequence of bitwise and linear operations
- Enormous period of $2^{19937} - 1$, a Mersenne prime
- k-distributed to 32-bit accuracy for $1 \leq k \leq 623$, i.e. tuples of up to length 623 occur with equal frequency over the entire period
- Integer seed is used to set the state, a 624×32 binary matrix

Mersenne Twister

- Fast to compute but has a large state space, not the most memory efficient
- Fails some TestU01 tests but has been generally considered "random" enough for Statistics... (but stay tuned)
- Completely predictable after we've seen 624 values
- No good jump ahead feature

Sample uniformity

- Generate samples using MT with a single seed, increasing number of samples
- PIKK requires n PRNs; sample_by_index requires only k PRNs
- No straightforward pattern in how uniform the samples are

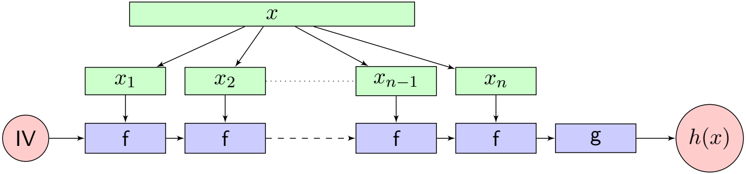


A better alternative

One solution: Find a class of PRNGs with infinite state space

Hash function PRNGs

Hash functions take in a message x of arbitrary length and return a value $h(x)$ of fixed size (e.g. 256 bits)



Cryptographic hash functions:

- computationally infeasible to invert
- difficult to find two inputs that map to the same output
- small input changes produce large, unpredictable changes to output
- resulting bits are uniformly distributed

Hash function PRNGs

Hash function PRNGs are a subset of a wide range of **cryptographically secure PRNGs**:

- NIST gives guidelines on using hash functions and stream ciphers for cryptographically secure PRNGs (Barker and Kelsey [2015])
- The OpenBSD OS uses the ChaCha20 stream cipher to generate PRNs (OpenBSD [2014], Bernstein [2008])
- Hash function PRNGs have been recommended for random selection of committees and election auditing (Laboratories [2004], Rivest [2011])

These PRNGs are usually written in low level languages, not in widely used statistical software.

Hash function PRNGs

Procedure for using a cryptographic hash function as PRNG:

Algorithm 5 Hash function PRNG

1: seed $\leftarrow$ a large random integer
2: counter $\leftarrow 0$
3: **for** the number of PRNs desired **do**
4: internal state $\leftarrow$ "seed,counter"
5: Hash the internal state value. This is your random number (expressed in hexadecimal).
6: counter $\leftarrow$ counter$+1$

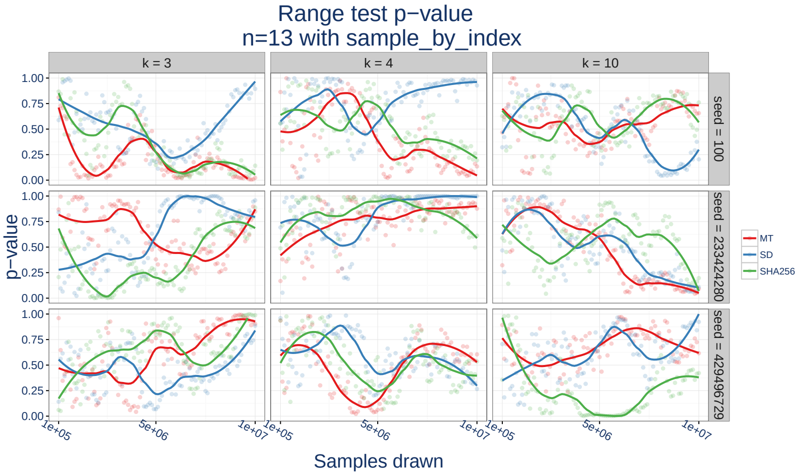
We use the SHA256 hash function:

- tested against a reference implementation (Rivest [2011])
- passes the tests described earlier. see appendix

Hash function PRNGs

- Efficient: based on fast, pre-existing hash function code. Some cryptographic hash functions are even built into hardware (e.g. AES and Intel)
- Memory efficient: only need to store the seed and counter
- Unpredictable: small changes to input produce large unpredictable changes to output. The only way to figure out the sequence is to know the seed.
- Jump ahead: add the desired number of steps to the counter

Sampling tests



Sampling tests

- Results are similarly indeterminate for the chi-squared test
- None of the 3 PRNGs is best overall – results vary by seed value
  - SHA256 PRNG should behave like "random oracle" and results should not vary by seed
  - What is the right way to aggregate results across seeds?
- Even though sampling k from n is complementary to sampling $n - k$ from n, the resulting samples are not equally uniform
- p-value curves tend to dip around 5×10^6 samples – trade-off between statistical power and equidistribution of PRNG?

Next steps

Theoretical:

- Understand how the equidistribution of a PRNG relates to how uniformity of sampling frequencies over different segments of the period
- Implement a sequential hypothesis test for uniformity of samples
- Account for multiple testing over many seeds

Practical:

- Find examples where results of a study would change from using a better sampling algorithm/PRNG
- Add the proposed statistical tests to a more thorough test battery for PRNGs for Statistics
- Create plug-in hash function PRNGs for R and Python

Impossibility bounds

Let F be the uniform distribution on all samples of size k from a population of n. For some subset of samples S, define $\mathcal{G} = \{G : G(S) = 0, S \in \mathcal{S}\}$ and $\nu = |S|$.

Lemma

For any $G \in \mathcal{G}$, $\|F - G\|_1 \geq \frac{2\nu}{\binom{n}{k}}$

For any bounded function $\psi : \Omega \to \mathbb{R}$ and for any $G \in \mathcal{G}$,

$$\left| \int \psi dG - \int \psi dF \right| \leq \|F - G\|_1 \|\psi\|_\infty$$

Corollary

There exists a statistic ψ such that

$$|\mathbb{E}_F(\psi) - \mathbb{E}_G(\psi)| \geq \frac{2\nu \|\psi\|_\infty}{\binom{n}{k}}$$

back

Proof of Lemma.

Fix S and choose $G \in \mathcal{G}$ such that $G(S) = 0$, $G(\omega) > 0$ for $\omega \in S^c$.

$$\begin{aligned}
\|F - G\|_1 &= \sum_{\omega \in \Omega} |F(\omega) - G(\omega)| \\
&= \sum_{\omega \in S} |F(\omega) - G(\omega)| + \sum_{\omega \in S^c} |F(\omega) - G(\omega)| \\
&= \sum_{\omega \in S} |F(\omega)| + \sum_{\omega \in S^c} |F(\omega) - G(\omega)| \\
&= \frac{|S|}{\binom{n}{k}} + \sum_{\omega \in S^c} |F(\omega) - (F(\omega) + \varepsilon_\omega)|
\end{aligned}$$

where $\varepsilon_\omega \in [-\binom{n}{k}^{-1}, 1 - \binom{n}{k}^{-1}]$ and $\sum_{\omega \in S^c} \varepsilon_\omega = \sum_{\omega \in S} F(\omega) = \frac{|S|}{\binom{n}{k}}$. NB this must be the case to ensure that $\sum_\omega G(\omega) = 1$, since

$$\sum_\omega G(\omega) = \sum_{\omega \in S^c} G(\omega) = \sum_{\omega \in S^c} F(\omega) + \varepsilon_\omega = \sum_{\omega \in S^c} F(\omega) + \sum_{\omega \in S} F(\omega) = 1.$$

Therefore,

$$\begin{aligned}
\|F - G\|_1 &= \frac{|S|}{\binom{n}{k}} + \sum_{\omega \in S^c} |\varepsilon_\omega| \\
&= \frac{|S|}{\binom{n}{k}} + \sum_{\omega \in S} |F(\omega) \\
&= \frac{2|S|}{\binom{n}{k}}
\end{aligned}$$

$\square$

back

Fisher-Yates-Knuth-Durstenfeld Shuffle.

We prove by induction that the FYKD the algorithm gives all possible permutations of $\{1, \ldots, n\}$ with equal probability, and thus all possible orderings of the first k have equal probability too. When $n = 2$, this is trivial. We sample $J = 1$ with probability $\frac{1}{2}$ to get the ordered pair $(2, 1)$ or sample $J = 2$ with probability $\frac{1}{2}$ to get the ordered pair $(1, 2)$.

Suppose the algorithm works for $n = 1, \ldots, j$ and we're at the $j + 1$st step. There are two possibilities:

1. $J = j + 1$ with probability $\frac{1}{j+1}$. Then we don't swap anything and we simply append $j + 1$ to the other permutations. This enumerates $j!$ permutations.
2. $J = i < j + 1$ with probability $\frac{1}{j+1}$. Then we swap i with $j + 1$. There are $j!$ equally likely ways that the first j items may be arranged, and j possible choices for J. This enumerates $j(j!)$ permutations.

Therefore, at the $j + 1$st step there are $(j + 1)(j!) = (j + 1)!$ equally likely permutations we could construct. □

back

Non-uniform random sampling probabilities.

Define $Y = \lfloor mX \rfloor + 1$ and $\tilde{X}$ to be a uniform random integer on $\{0, 1, \ldots, 2^w - 1\}$ (while X has the same distribution scaled by 2^{-w}). The selection probability for a particular integer value is

$$\begin{aligned}
\mathbb{P}(Y = y) &= \mathbb{P}(1 + \lfloor mX \rfloor = y) \\
&= \mathbb{P}(y - 1 \leq mX < y) \\
&= \mathbb{P}\left(\tilde{X} < \frac{y2^w}{m}\right) - \mathbb{P}\left(\tilde{X} < \frac{(y-1)2^w}{m}\right) \\
&= \mathbb{P}\left(\tilde{X} < \left\lceil \frac{y2^w}{m} \right\rceil\right) - \mathbb{P}\left(\tilde{X} \leq \left\lceil \frac{(y-1)2^w}{m} \right\rceil\right) \\
&= 2^{-w}\left(k^-(y) - k^-(y-1) + 1\right) = 2^{-w}\left(k^+(y-1) - k^-(y-1)\right)
\end{aligned}$$

where, for fixed m, we define $k^-(i) \equiv \min\{k : k2^{-w} \geq i/m\}$ for all i,
$k^+(i) \equiv \max\{k : k2^{-w} < i/m\} = k^-(i+1) - 1$ for $i = 0, \ldots, m-1$ and $k^+(m) \equiv 2^w$. The maximum ratio of selection probabilities is

$$\begin{aligned}
\max_{i,j \in \{0,\ldots,m-1\}} \frac{k^+(i) - k^-(i)}{k^+(j) - k^-(j)} &= \frac{\max_{i=0}^{m-1}(k^+(i) - k^-(i))}{\min_{i=0}^{m-1}(k^+(i) - k^-(i))} \\
&= \frac{\max_{i=0}^{m-1}(k^+(i) - k^+(i+1) + 1)}{\min_{i=0}^{m-1}(k^-(i+1) - k^-(i) - 1)} \\
&= \frac{\lceil 2^w/m \rceil + 1}{\lfloor 2^w/m \rfloor - 1} \\
&= 1 + 2^{-w}m + \ldots
\end{aligned}$$

□

back

TO DO: Insert table of p-values for several tests + several seeds back

References

Elaine Barker and John Kelsey. Recommendation for random number generation using deterministic random bit generators. Released Special Publication (SP) 800-90A Revision 1, NIST, June 2015.

Daniel J. Bernstein. Chacha, a variant of salsa20. *Workshop Record of SASC*, 8, 2008.

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.

Motorola Laboratories. Publicly verifiable nominations committee (nomcom) random selection. https://tools.ietf.org/html/rfc3797, June 2004.

P L'Ecuyer and R. Simard. TestU01: A C Library for Empirical Testing of Random Number Generators, 2007.

George Marsaglia. Random numbers fall mainly in the planes. *Proceedings of the National Academy of Sciences of the United States of America*, 61(1):25–28, September 1968. ISSN 0027-8424. URL `http://www.ncbi.nlm.nih.gov/pmc/articles/PMC285899/`.

George Marsaglia. The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness, 1995.

George Marsaglia. Seeds for Random Number Generators. *Commun. ACM*, 46(5):90–93, May 2003. ISSN 0001-0782. doi: 10.1145/769800.769827. URL `http://doi.acm.org/10.1145/769800.769827`.

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`.

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`.

OpenBSD. Arc4random library functions manual. http://man.openbsd.org/arc4random.3, November 2014.

S. K. Park and K. W. Miller. Random Number Generators: Good Ones Are Hard to Find. *Commun. ACM*, 31(10): 1192–1201, October 1988. ISSN 0001-0782. doi: 10.1145/63039.63042. URL `http://doi.acm.org/10.1145/63039.63042`.

Ronald L. Rivest. Reference implementation code for pseudo-random sampler for election audits or other purposes. https://people.csail.mit.edu/rivest/sampler.py, November 2011.

Andrew Rukhin, Juan Soto, James Nechvatal, Elaine Barker, Stefan Leigh, Mark Levenson, David Banks, Alan Heckert, James Dray, San Vo, et al. Statistical test suite for random and pseudorandom number generators for cryptographic applications, NIST special publication, 2010.

L. Philip Schumm. Stata tip 28: Precise control of dataset sort order. *The Stata Journal*, 6(1):144–146, 2006. URL `http://www.stata-journal.com/sjpdf.html?articlenum=dm0019`.