

# 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**



 WILEY

---

# Permutation Tests for Complex Data

Theory, Applications  
and Software



Fortunato Pesarin • Luigi Salmaso

---

WILEY SERIES IN PROBABILITY AND STATISTICS

# 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])
- **TO DO: TAKE A QUOTE FROM MARSAGLIA PAPER, SAYING THAT PRNGs WITH LARGE STATE SPACE ARE GOOD ENOUGH FOR THIS**

# 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

- A **pseudorandom number generator** (PRNG) that produces sequences of bits
- An 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.

# Simple Random Sampling

If there is a problem generating SRSs, it will be even worse for other methods: permutation, bootstrap samples, MCMC, Monte Carlo integration...

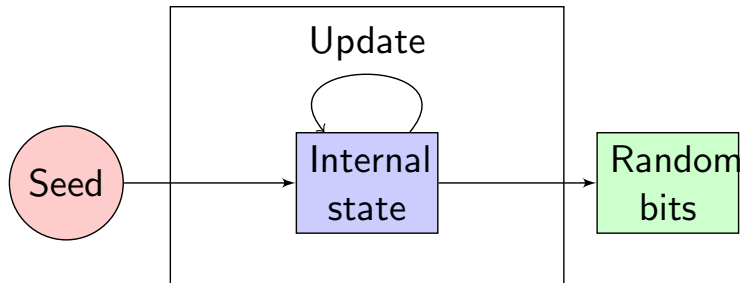
Number of possibilities for  $n = 100$ ,  $k = 50$

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

# 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:

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

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:

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

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
  - New PRNGs

# Contents

1 Introduction

2 Sampling Algorithms

3 Pseudorandomness

4 PRNGs

5 Appendix

# 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

---

**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

---

- 1: **for**  $i = 2, \dots, n$  **do**
  - 2:      $J \leftarrow$  random integer uniformly distributed on  $1, \dots, i$
  - 3:      $(a[J], a[i]) \leftarrow (a[i], a[J])$
  - 4: **end for**
  - 5: Take the first  $k$  to be the sample
- 

TO DO: PUT PROOF THAT SHUFFLE WORKS IN APPENDIX

- 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

---

```
1:  $\tilde{n} \leftarrow n$ 
2: Population indices  $\leftarrow \{1, \dots, n\}$ 
3: for  $i = 1, \dots, k$  do
4:    $w \leftarrow$  A random integer on  $\{1, \dots, \tilde{n}\}$ 
5:    $j \leftarrow$  The  $w$ th element in Population indices
6:   Sample indices  $\leftarrow$  Sample indices  $\cup \{j\}$ 
7:   Population indices  $\leftarrow$  Put last remaining index in place  $w$ 
8:    $\tilde{n} \leftarrow \tilde{n} - 1$ 
9: end for
10: 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 get integers in the range  $\{1, \dots, m\}$  is  $X = \lfloor mU \rfloor + 1$ ,  $U \sim U[0, 1)$ .
- 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

# Generating (non)uniform integers

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

1 Introduction

2 Sampling Algorithms

3 Pseudorandomness

4 PRNGs

5 Appendix





# 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

- “Independent” and “uniform” are broad criteria – many ways to define and check for these properties
  - Uniformity at varying levels of granularity
  - Correlations 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

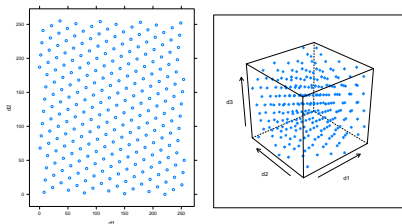
- 1 Gap test: count length of sequence between values in a given range
- 2 Permutation test: look at ordering of values in subsequences of length  $t$
- 3 Serial correlation test: correlation between consecutive pairs of values
- 4 Many more possibilities!

# 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  $(x_n, \dots, x_{n+d-1})$ .

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

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



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

# Testing 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  $(U_1, \dots, U_k)$  where  $U_j$  is uniform on  $\{1, \dots, n - j + 1\}$
  - The sequences  $(U_1, \dots, U_k)$  should themselves be equiprobable

# Testing 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  $(U_1, \dots, U_k)$  where  $U_j$  is uniform on  $\{1, \dots, n - j + 1\}$
  - The sequences  $(U_1, \dots, U_k)$  should themselves be equiprobable
- 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  $1/\binom{n}{k}$

# Testing PRNGs

Test proposals:

- Chi-squared or range test
- Sequential probability ratio test



TO DO:

# Contents

- 1 Introduction
- 2 Sampling Algorithms
- 3 Pseudorandomness
- 4 PRNGs
- 5 Appendix

# Linear Congruential Generators

LCGs have the form

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

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)

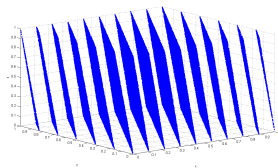
“Random numbers should not be generated with a method chosen at random.”

# 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

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

(Wikipedia)

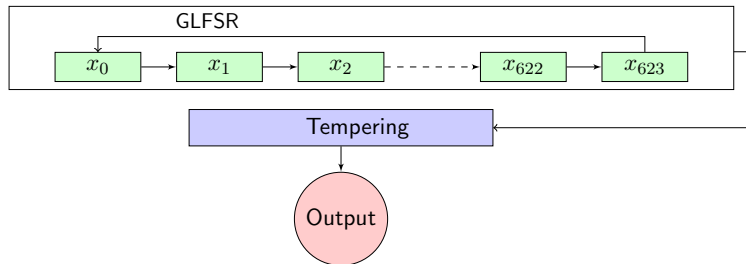
# Better LCGs

- Super-Duper:  $X_{n+1} = (69069X_n) \bmod 2^{32}$ 
  - $69069 = 3 \times 7 \times 11 \times 13 \times 23$
  - Considered a good LCG, passes spectral tests in low dimensions
- MINSTD:  $X_{n+1} = (16807X_n) \bmod (2^{31} - 1)$ 
  - $16807 = 7^5$
  - 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  $624 \times 32$  binary matrix

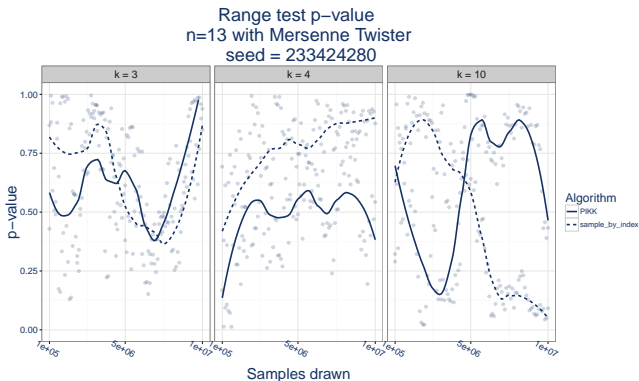


# Mersenne Twister

- 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

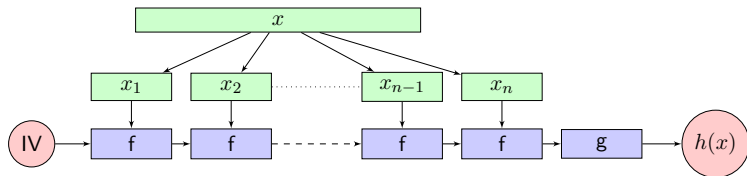


## A better alternative

**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

Procedure for using a cryptographic hash function as PRNG:

- 1 Set a seed, a large random integer
- 2 Set the counter to 0
- 3 Set the state to be “seed,counter”
- 4 Hash the state value. This is your random number (expressed in hexadecimal).
- 5 Increment the counter
- 6 Repeat Steps 2-5 as many times as needed

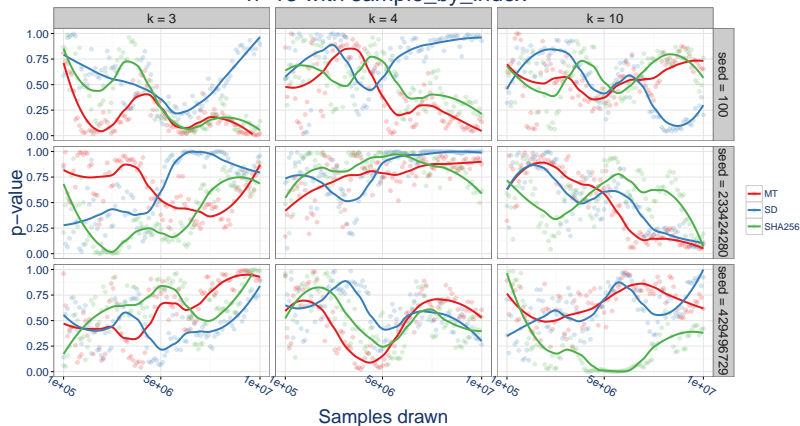
PRNG based on the SHA256 hash function passes the tests described. [see appendix](#)

# 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

Range test p-value  
n=13 with sample\_by\_index



# Sampling tests

## TO DO: SUMMARIZE SIM RESULTS

- Results are similarly indeterminate for the chi-squared test
- Even though sampling  $k$  from  $n$  is complementary to sampling  $n - k$  from  $n$ , the resulting samples are not equally uniform
- None of the 3 PRNGs is best overall – results vary by seed value
- $p$ -value curves tend to dip around  $5 \times 10^6$  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  $\psi : \Omega \rightarrow \mathbb{R}$  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  $\psi$  such that*

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

## 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,

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

□

## Proof.

Define  $Y = \lfloor mX \rfloor + 1$  and  $\tilde{X}$  to be a uniform random integer on  $\{0, 1, \dots, 2^w - 1\}$  (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  $i = 0, \dots, m-1$  and  $k^+(m) \equiv 2^w$ . The maximum ratio of selection probabilities is

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

□

TO DO: INSERT TABLE OF P-VALUES FOR SEVERAL TESTS +  
SEVERAL SEEDS [back](#)

# References

- 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.
- 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>.
- 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>.
- 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>.
- Juan Soto. Statistical testing of random number generators. In *Proceedings of the 22nd National Information Systems Security Conference*, volume 10, page 12. NIST Gaithersburg, MD, 1999.
- D. H. Young. Two Alternatives to the Standard 2-Test of the Hypothesis of Equal Cell Frequencies. *Biometrika*, 49 (1/2):107–116, 1962. ISSN 0006-3444. doi: 10.2307/2333471. URL <http://www.jstor.org/stable/2333471>.