# 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

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 | $n!$ | $100! \approx 10^{158}$ |

# Simple Random Sampling

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

Instead, people tend to draw samples using

- A **pseudorandom number generator** (PRNG)
- 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.

# 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

Does it matter in practice?

# Pigeons and Pigeonholes

Does it matter in practice?

Period of 32-bit linear congruential generators (e.g. RANDU):
$2^{32} \approx 4 \times 10^9$
Samples of size $10$ from $50$: $\binom{50}{10} \approx 10^{10}$
**More than half of samples cannot be generated**

# Pigeons and Pigeonholes

Does it matter in practice?

Period of 32-bit linear congruential generators (e.g. RANDU):
$2^{32} \approx 4 \times 10^9$
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**

# Pigeons and Pigeonholes

But does it *really* matter in practice?

- Social applications may require the PRNG to produce all possible samples – e.g. jury duty summons, gaming machines, lottery tickets (Marsaglia [2003])
- Impossibility bounds show that this introduces a nontrivial amount of bias into statistics estimated from Monte Carlo distributions here

# 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
- Sort: Stata uses a random sort function to break ties! Not reproducible! TO DO: cite http://www.stata-journal.com/sjpdf.html?articlenum=dm0019

# 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 $1, \ldots, i$
    $(a[J], a[i]) \leftarrow (a[i], a[J])$
**end for**
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

# Using random integers

**Algorithm 3** Uniform random integers

$\tilde{n} \leftarrow n$
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 $w$th element in Population indices
    Sample indices $\leftarrow$ Sample indices $\cup \{j\}$
    Population indices $\leftarrow$ Put last remaining index in place $w$
    $\tilde{n} \leftarrow \tilde{n} - 1$
**end for**
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

- Algorithm 2 relies on uniformly distributed integers
- A common way to get integers in the range $\{1, \ldots, m\}$ 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!

**Lemma**

*For $m < 2^w$, the ratio of the largest to smallest selection probability is, to first order, $1 + m2^{-w}$. (See, e.g., Knuth v2 3.4.1.A.)*

TO DO: PROPER CITATION, PUT PROOF IN APPENDIX

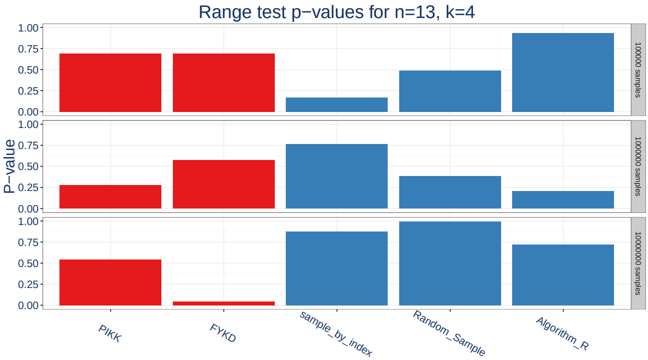
# Generating (non)uniform integers

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

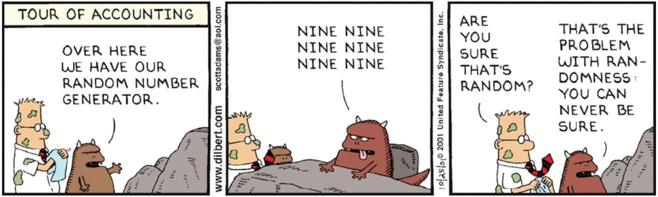
- Let $w = \lfloor \log_2 m \rfloor + 1$
- Generate a $w$-bit integer $J$
- If $J > m$, discard and repeat.

# Sampling Algorithms

- Red: require $n$ PRNs
- Blue: require $k$ PRNs
- No straightforward pattern in how uniform the samples are



# Pseudorandomness

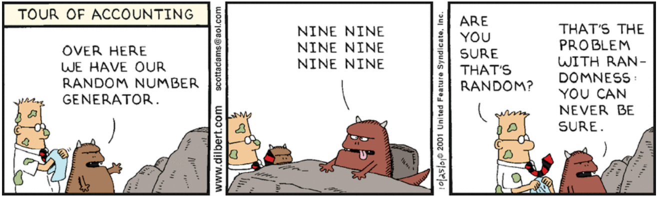



Dilbert

**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

- Fast and memory efficient
- Mimics a random sequence (statistically indistinguishable)
- Unpredictable. this is different from random - if it's deterministic, then it's predictable to some degree
- Jump-ahead feature so we can efficiently skip random numbers, generate multiple streams for parallel applications

# Testing PRNGs

TO DO:

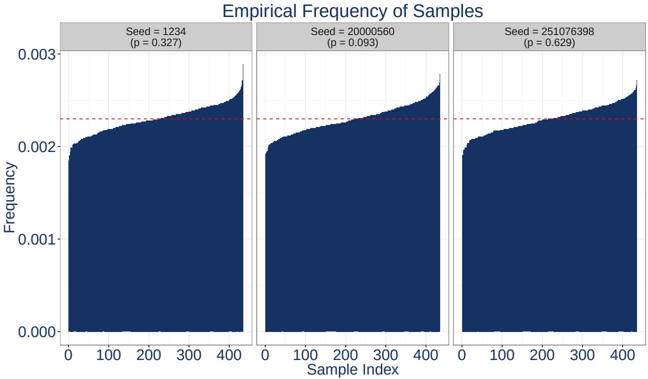
- List and cite test suites
- Describe a few interesting/important tests
- Argue that for statistical use, PRNGs should be able to produce samples uniformly and this ought to be a test
- Describe chi-squared test and range test
- Mention Spearman correlation test of permutations?

# Bad seeds

- **Seed**: an initial value to set the internal state of a PRNG.
- Mersenne Twister has problems with seeds with too many zeros (Saito and Matsumoto [2008])
- LCGs may not achieve their full period with certain seeds (Park and Miller [1988])

# Bad seeds

- **Seed**: an initial value to set the internal state of a PRNG.
- Mersenne Twister has problems with seeds with too many zeros (Saito and Matsumoto [2008])
- LCGs may not achieve their full period with certain seeds (Park and Miller [1988])




$10^5$ samples of size $2$ from a population of $30$ (Mersenne Twister in R)

# 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)**

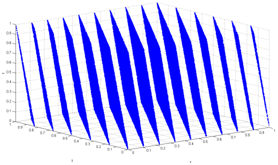
"Random numbers should not be generated with a method chosen at random."

# 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

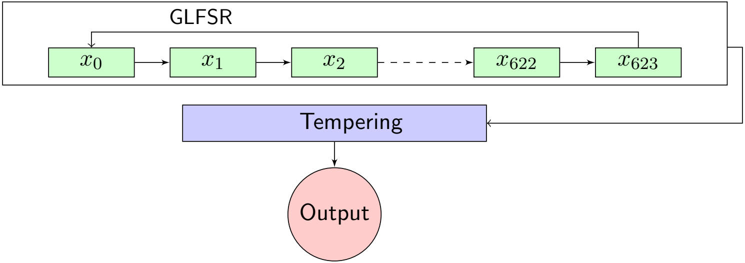
$$x_{n+1} = (65539 x_n) \mod 2^{31}$$

(Wikipedia)

# 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

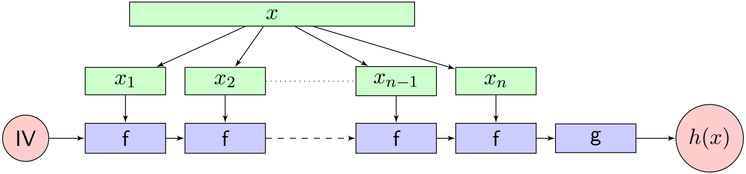
- Fast to compute but has a large state space, not the most memory efficient
- Generally considered “random” enough for Statistics, but fails some TestU01 tests
- Completely predictable after we’ve seen 624 values
- No good jump ahead feature

# 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.
5. Increment the counter
6. Repeat Steps 2-5 as many times as needed

# Hash function PRNGs

- Efficient: based on fast, pre-existing hash function code
- 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 hashing function
- Jump ahead: add the desired number of steps to the counter

## Simulations

TO DO:

- Compare MT/CSPRNG/LCG with different seeds
- For the unique sample frequencies, report
  - Range statistic $p$-value
  - Chi-squared test $p$-value
- Do this for various numbers of samples - 1e6 up to 1e8. How do $p$-values change? How does this relate to equidistribution?

# Next steps

TO DO:

Theoretical:

- Understand how the equidistribution of a PRNG relates to how uniformity of sampling frequencies over different segments of the period
- Study the properties of hash function PRNGs

Practical:

- Find examples where results of a study would change from using a better SA/PRNG
- Add these 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 $\psi$ 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,

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

$\square$

back

# References

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. 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.

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.

Mutsuo Saito and Makoto Matsumoto. SIMD-Oriented Fast Mersenne Twister: a 128-bit Pseudorandom Number Generator. In *Monte Carlo and Quasi-Monte Carlo Methods 2006*, pages 607–622. Springer, Berlin, Heidelberg, 2008.