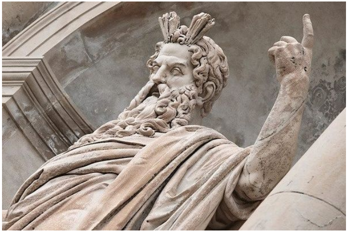
Random Number Generation

Corina Petrescu

Ladislaus von Bortkiewicz Chair of Statistics
Humboldt–Universität zu Berlin

Outline

Motivation



Blame the RNG Gods

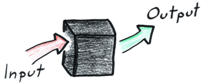
What is Random Number Generation?

Types

> *"One thing that traditional computer systems aren't good at is coin flipping"*

-Steve Ward, Professor of Computer Science and Engineering at MIT

Computers are deterministic. They are fed input which determines an action. How can they produce a random number?



Types

True RNGs

- can be generated **only** by relying on external input from a process assumed to be random like physical phenomenon
- think white noise, background radiation

Figure 1: Noise Figure Meter

Types

Pseudo RNGs

- a deterministic algorithm starts with a **seed** and follows a pattern
- the pattern must be sufficiently complex that it cannot be identified
- **seed** - the beginning of the pattern, the truly random element, a single number which determines the entire sequence

Seed

We're all guilty!

Advantages and Disadvantages

True RNGs

- offer actual random numbers
- slow
- not reproducible
- require extra hardware
- unknown distribution

Pseudo RNGs

- "look" random
- fast
- reproducible if you know the seed
- known distribution

PRNGs

Most useful PRNGs are based on Multiple Recursion:

$$x_i = f(x_{i-1}, x_{i-2}, .., x_{i-k})$$

The amount of previous numbers k is called the **order** of the generator. At some point, the sequence will repeat because of the finite number of states in a computer. The length of the sequence prior to beginning to repeat is called the **period** or **cycle length**.

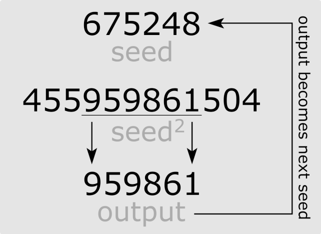
PRNGs

- **Blum Blum Shub**
- Counter-based random number generator (CBRNG)
- ISAAC (cipher)
- KISS (algorithm)
- Lagged Fibonacci generator
- Naor Reingold
- Park Miller
- RC4 PRGA
- **Linear congruential generator** - of historical importance
- **Mersenne Twister**
- **Middle-square method**
- MIXMAX generator
- Wichmann Hill
- Xorshift
- Yarrow

What should I use?

Middle Square Algorithm

Also known as the "paper and pen" method.



How it works: Take a 6 digit seed, square it and extract the middle 6 digits.

Middle Square Algorithm

Issues

- if you can't extract the middle 6 digits, you supply the number with leading 0's
- if the middle n digits are all zeroes, the generator then outputs zeroes forever
- if the first half of a number in the sequence is zeroes, the subsequent numbers will be decreasing to zero
- other seed values form very short repeating cycles:

$$0540 \xrightarrow{2916} \xrightarrow{5030} \xrightarrow{3009} 0540$$

Linear Congruential Algorithm

Besides the seed (X_0), this algorithm requires 3 other inputs from the user: **multiplier (a), increment (c)** and **modulus (m)**

How it works:

$$X_{n+1} = (a * X_n + c) \mod m$$

- $m > 0$
- $0 < a < m$
- $0 \leq c < m$
- $0 \leq X_0 < m$
- $(X_0, m) = 1$

Multiplicative Congruential Algorithm

Period: length of the sequence prior to repeat

- depends on the smallest k for which

$$a^k \equiv 1 \mod m$$

- thus, period can't be greater than k
- $(X_0, m) = 1$

Output is usually expressed in one of three types:

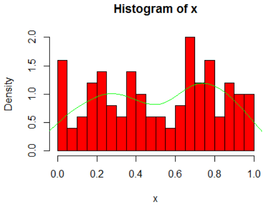
1. $g : \mathbb{N} \to [0, 1), \quad g(x) = \frac{x}{m}$
2. $g : \mathbb{N} \to (0, 1], \quad g(x) = \frac{x}{m-1}$
3. $g : \mathbb{N} \to (0, 1), \quad g(x) = \frac{x+\frac{1}{2}}{m}$

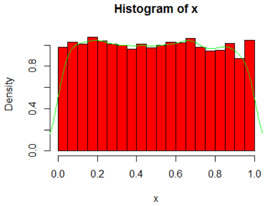
Multiplicative Congruential Algorithm

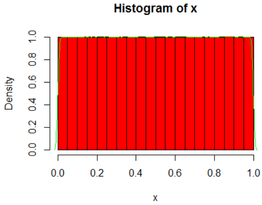
The following values of a and m were chosen by Lewis, Goodman and Miller (1969). ("minimal congruential generator")

```
lcg.rand <- function(n=100000)
rng <- vector(length = n)
m <- 2^31 - 1
a <- 16807
c <- 0
Xn <- as.numeric(Sys.time()) * 1000
for (i in 1:n)
Xn <- (a * Xn + c)
rng[i] <- Xn / m
return(rng)
```

Multiplicative Congruential Algorithm







Matrix Congruential Algorithm

A generalization of the linear congruential algorithm for vectors.

- $x_i \equiv (Ax_{i-1} + c) \mod m$
- x_i, x_{i-1} and c are vectors of length d
- A is a dxd matrix
- elements of vectors and matrices are integers between 1 and m-1
- vector elements then scaled into (0,1)

Matrix Congruential Algorithm

```
lcg.rand <- function(n=121)
d <- 11
rng <- matrix(nrow = d,ncol = d)
m <- 10**5
A <- matrix(c(sample(1:m-1,n)), nrow = d,ncol = d,byrow = T)
c <- rep(0,d)
Xn <- as.numeric(Sys.time()) * 1000
for (i in 1:n)
Xn <- (A * Xn + c)
rng[i] <- Xn / m
return(rng)
```

Matrix Congruential Algorithm

1	0.68904	0.67432	0.54056	0.54248	0.73384	0.47272	0.44776	0.96008	0.35464	0.67912	0.01896
2	0.20168	0.76744	0.82152	0.87816	0.52328	0.82024	0.08392	0.69736	0.83688	0.40904	0.43432
3	0.32456	0.01448	0.10984	0.28072	0.31176	0.07208	0.85064	0.44712	0.56296	0.44168	0.68744
4	0.94952	0.30216	0.11528	0.15624	0.57192	0.00136	0.46888	0.06504	0.48232	0.40456	0.65448
5	0.88584	0.28872	0.77576	0.98408	0.74664	0.41512	0.70696	0.79368	0.10344	0.30952	0.18216
6	0.50728	0.89224	0.25992	0.90536	0.90088	0.12104	0.73032	0.78856	0.92648	0.00584	0.24872
7	0.83976	0.69608	0.04264	0.58312	0.45096	0.94568	0.91944	0.63752	0.20616	0.54728	0.21224
8	0.14792	0.40936	0.13288	0.57704	0.17832	0.77256	0.99848	0.18184	0.45672	0.51976	0.13608
9	0.73864	0.95112	0.79496	0.89768	0.13544	0.16552	0.83016	0.73928	0.34824	0.70792	0.88936
10	0.16488	0.43304	0.82632	0.35656	0.87048	0.75784	0.86472	0.18376	0.64808	0.25864	0.11112
11	0.73896	0.64968	0.75144	0.89352	0.05416	0.73128	0.88424	0.79592	0.99336	0.00488	0.15304

Linear Congruential Algorithms

The bad news: Marsaglia[1968] shows that even with the best chosen constants, using an LCG has a defect which makes it unsuitable for Monte Carlo problems: using an LCG to choose points in an n-dimensional space will generate points that will lie on, at most, $(n!m)^{1/n}$ hyperplanes.

The good news: In a 2014 paper on the desirable properties of PRNGs we see evidence that despite its flaws, the LCG has endured with good reason. They have speed, easy implementation and are fairly space efficient.

Mersenne Twister

The "golden standard" used in most commercial packages, it's a standard generator in Matlab, Octave, R, S+ etc..

Why?

- amazing period length $2^{19937} - 1$ (6002 character number)
- outputs identically distributed random numbers with close to 0 correlation
- based on bit operations
- passes Diehard test But! not cryptographically secure

Mersenne Twister

- instead of 1 seed, we have an array of 624 seeds
- take first bit from S1 and last 31 bits from S2 and merge them together
- we shift everything right one bit

The twist

- if 1, we A=0x9908B0DF in an XOR manner (addition modulo 2)
- take S397 and combine it XOR
- Congratulations! You've reached a new random number. **BUT WAIT, IT'S NOT OVER...**
- the new random number goes through tempering

Mersenne Twister

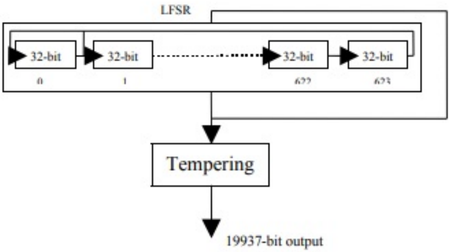
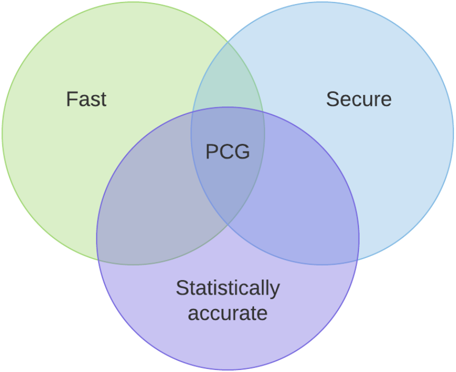


Fig 1. Block diagram of MT

Latest PRNGs

PCG Family

M. E. O'Neill (2014), "Pcg: A family of simple fast space-efficient statistically good algorithms for random number generation"



Why is it better?

PCG=Permuted Congruential Generator

$$LCG + \text{ permutation based output function } = WIN$$

- excellent statistical quality
- harder to predict than MT
- arbitrary period
- faster than MT
- little space usage

What does it look like?

LCG PART:
state = state * multiplier + increment;

OUTPUT PART:
uint32_t output = state » (29 - (state » 61));

Take-Away

Remember

- Don't use the system's generator (runif. rnorm etc)
- Make use of CongruRand in R
- Try to understand what's going on behind your generation, it is the base for everything
- MT not magic
- LCG actually quite useful
- if you want to impress, PCG

Thank you for your attention!

Bibliography

David Jones (2010), Good Practice in (Pseudo) Random Number Generation for Bioinformatics Applications, UCL Bioinformatics Group, London

M. E. O'Neill (2014), "Peg: A family of simple fast space-efficient statistically good algorithms for random number generation", ACM Trans. Math. Softw., http://www.pcg-random.org/pdf/toms-oneill-pcg-family-v1.02.pdf.

Code adapted from MULTIPLICATIVE CONGRUENTIAL GENERATOR IN R by Aaron Schlegel

What about cryptography?

Cryptography

- earliest known usage in 1900 BC
- PRNGs used to create session keys and stream ciphers
- CSPRNGs= Cryptographically Secure PRNGs
  - pass all statistical randomness tests (Next Bit Test)
  - resistance under attacks
- Mersenne Twister is **NOT** cryptographically secure

Notes

Properties of RN

- the number of different states that a computer can be in is bounded by its memory (a computer with 1âKiByte of memory and 4 8-bit registers can only be in 8*21024 + 4*28 = 10300 distinct states
- And since the program itself is part of the state and computers are deterministic, this means that it will do the exact same thing it did the last time it was in this state, thus it will end up in the same followup-state it ended up the last time, which means that it will now do the same thing it did â and so on.
- Any programs which runs long enough will at some point end up in a state it was in before and from that point on run in exactly the same way it did the last time.