Uniform Variate Generation

Refs: Chapter 7 in Law,
Pierre Lecuyer Tutorial, Winter Simulation Conference 2015

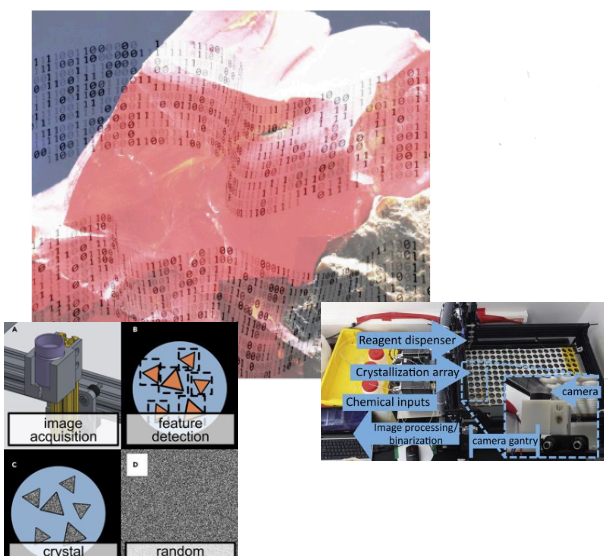
Peter J. Haas

CS 590M: Simulation
Spring Semester 2020

Mad Scientists Are Using Crystals to Generate Random Numbers

It's all to make your computer safer. But it sounds fun, too.

By Courtney Linder Feb 19, 2020 Popular Mechanics



Pseudo-Random Numbers

A deterministic sequence that "looks random"

- Deterministic recurrence relation: sequence of integer seeds
- Each seed converted into a uniform "random number"
- A good generator has desirable theoretical properties, passes statistical tests

Repeatability is a good thing

- Facilitates debugging and verification
- Allows "common random numbers" (efficiency improvement)

Versus "natural" sources of randomness

- Ex: Silicon graphics / Cloudflare "lava lamp" generator
- Ex: HotBits site uses radioactive decay
- Ex: random.org uses atmospheric noise (radio static)
- Non-reproducible and slow
  (OK for lotteries, games, generating encryption keys)

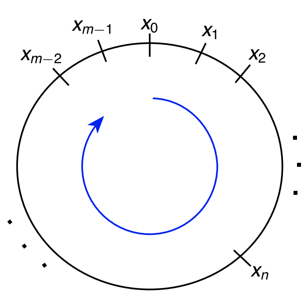
Linear Congruential Generators (LCGs)

Fundamental recurrence:

$x_{n+1} = (ax_n + c) \bmod m$

- a = multiplier, c = increment, m = modulus
- $k \bmod m$ is remainder after dividing k by m (e.g., $14 \bmod 5 = 4$ and $2 \bmod 10 = 2$)
- x_n's take values in $\{0, 1, 2, \ldots, m-1\}$
- Return $U_n = x_n/m$
- In C: `rand()` returns seed between 1 and `RAND_MAX`
- Historically the earliest (effective) rng

Period of an LCG



An LCG is periodic

- Period $\leq m$
- Want full period (every number in $[0..m-1]$ appears once)
  - Maximizes number of available random variates in a simulation
  - Otherwise, gaps may cause statistical anomalies

Multiplicative Congruential Generators (MCGs)

Fundamental recurrence:

$x_{n+1} = ax_n \bmod m$

- Special case of LCG with increment c equal to 0
- Full period: all values in $\{1, 2, \ldots, m-1\}$ visited in cycle

Theorem:

An MCG has full period if m is prime and a is a primitive element modulo m

- I.e., $r(a) \triangleq \min\{k > 0 : a^k \bmod m = 1\} = m - 1$
- Ex: $x_{n+1} = 3x_n \bmod 4$
- Ex: $x_{n+1} = 3x_n \bmod 5$

k	1	2	3	4
$3^k \bmod 5$	3	4	2	1

Classical MCGs

Modulus choices

- $m = 2^b$ for convenience on binary computer
  - mod 2^b is simple: retain b lowest-order bits
  - Ex: IBM RANDU generator with $m = 2^{31}$ and $a = 2^{16} + 3$
  - For $b > 3$, period is at most $m/4$
- $m = 2^{31} - 1$ is, fortuitously, a (Mersenne) prime number
  - Because $2^{31} - 1$ is "almost" 2^{31}, can compute mod quickly [Bratley et al., pp. 212–213]

Lewis-Learmonth Generator ("Minimal Standard Generator")

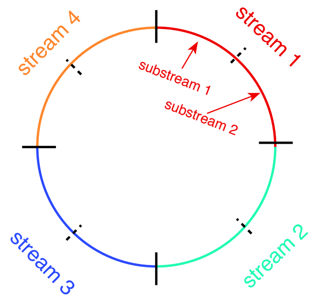
$x_{n+1} = 7^5 x_n \bmod (2^{31} - 1) = 6807 x_n \bmod 2{,}147{,}483{,}647$

Used for many years, but fails modern statistical tests, cycle is too short

Streams and Substreams in an MCG

Jumping ahead

- Goal: quickly compute x_k for k large
- $x_k = (a^k x_0) \bmod m = \big((a^k \bmod m)\, x_0\big) \bmod m$
- Precompute numbers $\alpha_k = a^k \bmod m$ for multiple values of k
- Allows partitioning of cycle into streams and substreams
  - Better than, e.g., setting $y_n = x_{2n}$ and $z_n = x_{2n+1}$
  - Caution: For an MCG, non-overlap is not sufficient (see demo)

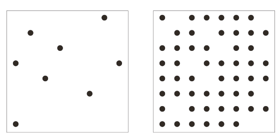



```
unigen.ResetNextSubstream()
unigen.ResetStartStream()
unigen.ResetStartSubstream()
```


Pitfalls of MCGs (and Other Generators)

Short cycles

- MCG numbers fall on a lattice
- Only want to use $O(\sqrt{\text{period}})$ numbers



Early in cycle   Late in cycle

Low-order bits

Claim:

If $x_{n+1} = a x_n \bmod 2^b$ and $r_{n+1} = x_{n+1} \bmod 2^k$, where $3 < k < b$, then $(r_n : n \geq 1)$ has period at most 2^{k-2}

- r_n's are k low-order bits
- Ex: $x_{n+1} = 13 x_n \bmod 2^{31}$ with $k = 4$ [period $= (2^{31}/4) - 1 \approx 537 \times 10^6$]
- So avoid algorithm that sets $X = \lfloor nU \rfloor$ and $V = nU - X$ where $U \overset{D}{\sim} \text{Uniform}(0,1)$

n	x_n	r_n
1	16049371	11
2	208641823	15
3	564860051	3
4	900729719	7
5	972068107	11
6	1899467151	15
7	1070752835	3
8	1034884967	7

Other Pitfalls (Demo)

Starting seeds for Lewis-Learmonth generator

- X: use starting seed $s = 1$
- Y: use starting seed $s' = 2$
- s and s' are over 1.3 billion steps apart in cycle
- Plot (X, Y) pairs
- Plot histogram of $X + Y$

Box-Muller and MCG

1. Generate U, V i.i.d. $U[01,]$
2. Set $X = \sqrt{-2 \log u} \cos(2\pi V)$
3. Set $Y = \sqrt{-2 \log u} \sin(2\pi V)$
4. Return X and Y independent $N(0,1)$

Combined Generators

Example: rngStream (used in Arena and other packages)

$$x_n = (1403580x_{n-2} - 810728x_{n-3}) \bmod 4294967087$$
$$y_n = (527612y_{n-1} - 1370589y_{n-3}) \bmod 4294944443$$
$$z_n = (x_n - y_n) \bmod 4294967087$$
$$u_n = z_n/4294967087$$

- Seed = vector of six 32-bit integers
- Cycle length $\approx 2^{191} \approx 10^{57}$ (1 octodecillion)
- # streams $= 2^{64} \approx 10^{19}$ (10 quintillion)
- Stream length $= 2^{127} \approx 10^{38}$ (100 undecillion)
- # substreams $= 2^{51} = 10^{15}$ (1 trillion)
- Substream length $= 2^{76} \approx 10^{22}$ (10 sextillion)
- Well-behaved up to at least 45 dimensions

Fun Fact
Time to mostly use up a generator with period of 2^{191} with 1 trillion computers generating one seed per nanosecond: $> 10^{38}$ years

Mersenne Twister

General Form (Bit-wise Generator):

$$X_n = (A_1X_{n-1} + \cdots + A_kX_{n-k}) \bmod 2$$
$$Y_n = BX_n \bmod 2 = (y_{n,1}, \ldots, y_{n,W}) \text{ where } W = 32 \text{ or } 64$$
$$u_n = \sum_{j=1}^{W} y_{n,j}2^{-j} \text{ or } u_n = 0.y_{n,1}y_{n,2}\cdots y_{n,W}$$

- $X_n = (x_{n,1}, \ldots, x_{n,L})^\top$ with each $x_{n,i} = 0$ or 1
- Binary matrices $A_1, \ldots, A_k$ ($L \times L$) and B ($W \times L$)
- Fast: XOR operations and bit shifting

Mersenne Twister

- Default generator in Python and many other systems
- Seed = vector of 623 integers (32 bit)
- Period $= 2^{19937} - 1 \approx 10^{6002}$ and well-behaved up to $d = 623$
- Drawbacks: large, slow initialization (demo), some stat. issues
- WELL generator (Lecuyer et al.) scambles bits better

More Generators

Cryptographic generators

- "Impossible" to guess the next number in the sequence
- Ex: RC4 (open source: Arc4Random), Threefish, ChaCha20
- Good for security, slow and statistically shaky for simulation

Counter-based generators

- Trivial seeds: $s_n = n$ for $n \geq 1$ (great for substreams!)
- $u_n = f(n)$ where f is a "weak" but fast encryption function
- Perform well, equidistribution properties not well understood

Permuted congruential generator (PCG)

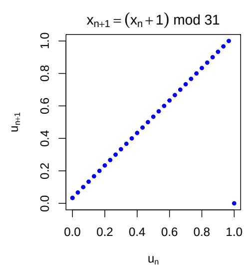
- Use "improving" transformation of fast but shaky LCG
- Under evaluation

Testing Uniform Random Number Generators

A simple generator:
$x_{n+1} = (x_n + 1) \bmod m$

Properties

- Full period
- Values uniformly spread out over $[0, 1]$
- Yet: this is a *terrible* generator



Two ways of showing poor quality

- Compare to expected statistical behavior of uniform sequence $U_1 = \frac{1}{m-1}, U_2 = \frac{2}{m-1}, U_3 = \frac{3}{m-1}, \ldots$ (not very random)
- Look at possible values in higher dimensions (see plot)

Structural (Theoretical) Tests

Possible values in d dimensions

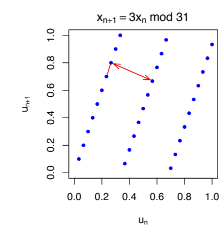
- Group the cycle into d-vectors: $V_i = (U_i, \ldots, U_{i+d-1})$
- Want equidistribution over $[0, 1]^d$

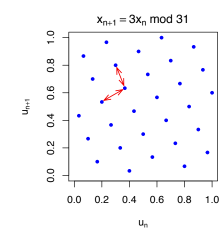
Structural tests for MCGs

- Points of MCG lie on lattice: how even? (spectral test)
- For modulus m, points lie on at most $(d!m)^{1/d}$ hyperplanes
- RANDU (demo)

For other generators:

- Not a lattice; 32/64-bit U values can appear multiple times in cycle
- Want same # of points in each "grid cell"





d	Upper bound
1	2^{31}
2	2^{16}
3	2344
4	476
5	192
6	108

Statistical (Empirical) Tests

How does generator jump from point to point?

- Do U_i numbers look i.i.d. uniform to a statistician?

Many kinds of statistical tests

- General tests for goodness of fit (e.g., χ^2 test)
  1. Divide $[0, 1]$ into k (> 100) equal intervals
  2. Generate $U_1, \ldots, U_n$ (where $n \approx 10k$)
  3. Count number $f_1, \ldots, f_k$ that fall into each interval
  4. Compute likelihood under i.i.d. uniform hypothesis

$$\chi^2 = \sum_{j=1}^{k} \frac{(f_j - \frac{n}{k})^2}{n/k}$$

- Serial test: essentially d-dimensional version of χ^2 test
- Runs-up test (see homework)

Test suites (the PRNG arms race)

- Gold standard: TestU01 suite (incl. "SmallCrush", "Crush") [Lecuyer et al., simul.iro.umontreal.ca/testu01/tu01.html]