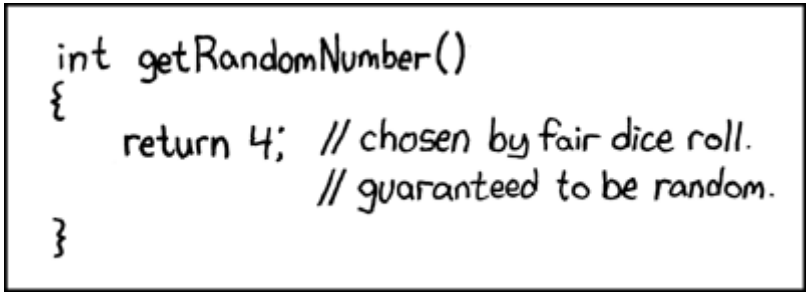
A short tutorial on random number generation

Fabian Bastin
fabian.bastin@umontreal.ca
Université de Montréal – CIRRELT – IVADO

Université de Montréal



https://xkcd.com/221/

U[0,1]-distributed random numbers

- A good uniform random generator on the interval $[0, 1]$ is a major component of any good random generator library.
- Draws from other distributions are usually obtained by adequately transforming a uniformly distributed sample.

Define a transition function $f : \mathcal{S} \to \mathcal{S}$, where $\mathcal{S}$ is the state space. The cardinality of $\mathcal{S}$ is assumed to be finite.

The initial state is denoted by s_0, and we will write

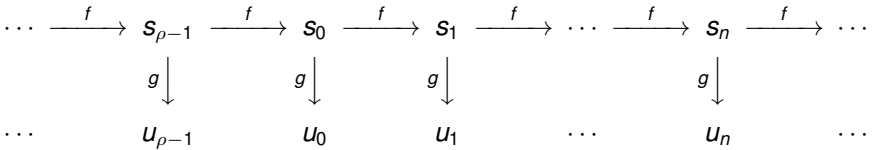
$$s_n = f(s_{n-1}).$$

We will furthermore assume that f is periodic for all n greater of equal to some known τ (often equal to 0), with period ρ:

$$s_{n+\rho} = s_n, \ \forall\, n \geq \tau.$$

U[0,1]-distributed random numbers (cont'd)

- Output space: $\mathcal{U}$.
- We assume here that $\mathcal{U} = (0, 1)$.
- Output function $g : \mathcal{S} \to \mathcal{U}$.
  It transforms the state s_n into an output value u_n.



How to choose f and g?

Goals: large ρ, good uniformity, "random" behavior.

Linear congruential generators (LCGs)

- Introduced by Lehmer (1951).
- State $s = x \in \mathbb{N}_{\geq 0}$.
- Recursive formula:

$$x_i = f(x_{i-1}) = (ax_{i-1} + c) \mod m,$$

where $\mod$ is the modulo operator.
Given two numbers, a (the dividend) and n (the divisor),

$$a \mod n = a - n \left\lfloor \frac{a}{n} \right\rfloor.$$

- If $c = 0$, the generator is often called a multiplicative congruential generator, or "Lehmer RNG"

Linear congruential generators: full period?

Full period: m is $c \neq 0$, $m-1$ otherwise (if $c = 0$, 0 is a fixed point for the recurrence). Consider the case $c \neq 0$.

Theorem (Period)

The LCG has full period iff the following three conditions hold:

1. *the only positive integer that (exactly) divides both m and c is 1;*
2. *if q is a prime number that divides m, then q divides $a-1$;*
3. *if 4 divides m, then 4 divides a-1.*

"Standard minimal" (Park and Miller, 1988):

$$x_{n+1} = 16807 x_n \bmod 2147483647.$$

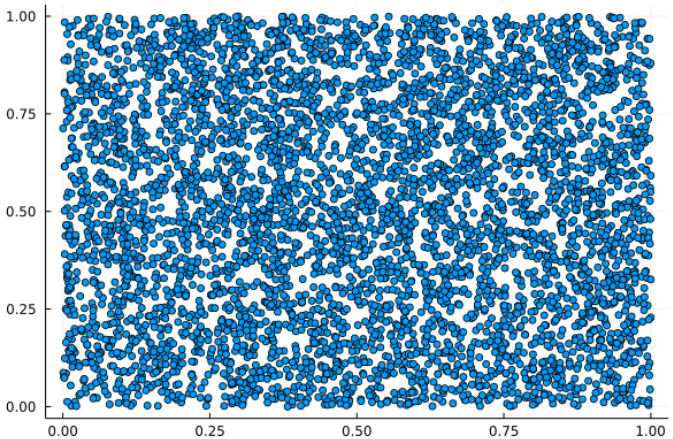
Observe that $2147483647 = 2^{31} - 1$; on 32-bit architectures, the largest representable (signed) integer is $2^{31} - 1$.

The Standard Minimal Generator

```
function getlcg(seed::Integer, a::Integer,
                c::Integer, m::Integer)
    state = seed
    am_mil = 1.0/m
    return function lcgrand()
        state = mod(a * state + c, m)
        # produce a number in (0,1)
        return state*am_mil
    end
end

stdmin = getlcg(1234, 16807, 0, 2^31-1)
```

Standard minimal generator: illustration



10000 generated points on the unit square.

Multiple Recursive Generator (MRG)

But we want better! Generalize the LCG:

$$x_n = (a_1 x_{n-1} + \cdots + a_k x_{n-k}) \mod m, \quad u_n = x_n/m.$$

In practice, take $u_n = (x_n + 1)/(m + 1)$, or $u_n = x_n/(m + 1)$ if $x_n > 0$ and $u_n = m/(m + 1)$ otherwise, but the structure remains the same, and is easier when studying theoretical properties.

State at step n:

$$s_n = (x_{n-k+1}, \ldots, x_n)^T.$$

State space: $\mathcal{Z}_m^k$, of cardinality m^k.

The maximal period is $\rho = m^k - 1$.

Period of MRGs

It can be shown that for $k > 1$, it is sufficient to have at least two non-zero coefficients, including a_k, in order to get the maximal period.

The cheapest recurrence has therefore the form

$$x_n = (a_r x_{n-r} + a_k x_{n-k}) \mod m.$$

But how to choose a_r and a_k?

It is possible to study theoretical properties of MRG's, and exclude directly some generators that have known strong deficiencies.

Choosing a good MRG

Example: Lagged-Fibonacci

$$x_n = (\pm x_{n-r} \pm x_{n-k}) \mod m.$$

It can be shown the vectors $(u_n, u_{n+k-r}, u_{n+k})$ are all contained in two planes! We therefore know without additional tests that the numbers cannot be considered as random.

In practice, we can impose various conditions on the coefficients, and compute theoretically appealing generators by maximizing some quality measure. This maximization is numerically expensive.

Combined MRG's

Consider two (or more) MRG's working in parallel:

$$\begin{aligned} x_{1,n} &= (a_{1,1}x_{1,n-1} + \cdots + a_{1,k}x_{1,n-k}) \mod m_1, \\ x_{2,n} &= (a_{2,1}x_{2,n-1} + \cdots + a_{2,k}x_{2,n-k}) \mod m_2. \end{aligned}$$

We define the two combinations

$$\begin{aligned} z_n &:= (x_{1,n} - x_{2,n}) \mod m_1; \qquad & u_n &:= z_n/m_1; \\ w_n &:= (x_{1,n}/m_1 - x_{2,n}/m_2) \mod 1. \end{aligned}$$

The sequence $\{w_n,\ n \geq 0\}$ is the output of another MRG, of modulus $m = m_1 m_2$, and $\{u_n,\ n \geq 0\}$ is nearly the same sequence if m_1 and m_2 are close.
We can achieve the period $(m_1^k - 1)(m_2^k - 1)/2$.

MRG32k3a

The following combined MRG was proposed by L'Ecuyer. It has been for a long time amongst the most popular and efficient generators (e.g., used in R), although newer families like xoshiro or CBRNGs are often preferred today for raw performance. It combines 2 MRG's.

$k = 3$,
$m_1 = 2^{32} - 209$, $a_{11} = 0$, $a_{12} = 1403580$, $a_{13} = -810728$,
$m_2 = 2^{32} - 22853$, $a_{21} = 527612$, $a_{22} = 0$, $a_{23} = -1370589$.

Combination: $z_n = (x_{1,n} - x_{2,n}) \bmod m_1$.

Corresponding MRG: $k = 3$,
$m = m_1 m_2 = 18446645023178547541$,
$a_1 = 18169668471252892557$,
$a_2 = 3186860506199273833$,
$a_3 = 8738613264398222622$.

MRG32k3a: basic implementation

```
function rand(rng::MRG32k3a)

p1::Int64 = (a12 * rng.Cg[2] + a13 * rng.Cg[1]) % m1
p1 += p1 < 0 ? m1 : 0

rng.Cg[1] = rng.Cg[2]
rng.Cg[2] = rng.Cg[3]
rng.Cg[3] = p1
p2::Int64 = (a21 * rng.Cg[6] + a23 * rng.Cg[4]) % m2
p2 += p2 < 0 ? m2 : 0

rng.Cg[4] = rng.Cg[5]
rng.Cg[5] = rng.Cg[6]
rng.Cg[6] = p2
u::Float64 = p1 > p2 ? (p1 - p2) * norm :
   (p1 + m1 - p2) * norm
end
```

Random numbers generators on $\mathcal{F}_2$

MRGs work on integers. An alternative is to build generators directly on bits, with linear recurrences in $\mathcal{F}_2$.

Let us introduce the Galois field $\mathcal{F}_2$: it is simply the set $\{0, 1\}$ with addition and multiplication computed modulo 2.

Addition ($\oplus$, XOR)

$\oplus$	0	1
0	0	1
1	1	0

Multiplication ($\cdot$, AND)

$\cdot$	0	1
0	0	0
1	0	1

All operations are then bitwise (shifts, xors, masks), which makes these generators numerically very fast.

Generators on $\mathcal{F}_2$: state and output

We construct two sequences of bit vectors, the state and the output, with linear recurrences (k-bit state, w-bit output):

$$\begin{aligned} \boldsymbol{x}_n &= X\boldsymbol{x}_{n-1} && \text{(state vector, } k \text{ bits)}, \\ \boldsymbol{y}_n &= B\boldsymbol{x}_n && \text{(output vector, } w \text{ bits)}, \end{aligned}$$

where X and B are binary matrices and all arithmetic is in $\mathcal{F}_2$.

Reading a binary fraction. The w output bits are stacked after the radix point:

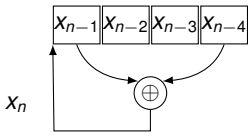
$$u_n = \sum_{j=1}^{w} y_{n,j-1}\, 2^{-j} = .y_{n,0}\; y_{n,1}\; \cdots\; y_{n,w-1}.$$

Example: $y_n = (1, 0, 1, 1)$ gives $2^{-1} + 2^{-3} + 2^{-4} = 0.6875 = .1011_2$.

LFSR: a shift register with linear feedback

A linear feedback shift register (LFSR) is a k-bit register updated in three steps:

1. read the tapped bits — here the positions r and k;
2. xor them: the new bit is $x_n = x_{n-r} \oplus x_{n-k}$ (it is read before the shift);
3. shift the register by one position and insert x_n.



LFSR: a tiny example ($k = 4$, taps $r = 1$ and $r = 4$)

Seed 0001, recurrence $x_n = x_{n-1} \oplus x_{n-4}$:

step	register				new bit
	x_{n-1}	x_{n-2}	x_{n-3}	x_{n-4}	x_n
0	0	0	0	1	–
1	1	0	0	0	1
3	1	1	1	0	1
7	0	1	0	1	0
9	1	1	0	1	1
⋮					⋮

- After 15 steps we are back to the seed 0001: period $2^4 - 1$.
- All $2^4 - 1$ non-zero states have been visited: maximal period.

LFSR: matrix form

The recurrence is a linear recurrence over $\mathcal{F}_2$ on the state vector (we assume $a_k \neq 0$):

$$\boldsymbol{x}_n = X\,\boldsymbol{x}_{n-1}, \qquad X = \begin{pmatrix} 0 & 1 & & \\ & \ddots & \ddots & \\ & & \ddots & 1 \\ a_k & a_{k-1} & \dots & a_1 \end{pmatrix}.$$

- The last row encodes the taps (the feedback coefficients), the rest of the matrix is a pure shift.
- X^ν shifts the register by ν bits in one step: this is the spacing ν used when assembling output words.
- For Tausworthe generators, $B = I$ (no output transformation).

Tausworthe generators: assembling words

Tausworthe (1965): read the bit stream w bits at a time, spaced ν apart, and interpret them as a binary fraction:

$$u_n = \sum_{\ell=1}^{w} x_{n\nu+\ell-1}\, 2^{-\ell} = .x_{n\nu}\; x_{n\nu+1}\; \ldots\; x_{n\nu+w-1}.$$

Example (previous LFSR, $w = 3$, $\nu = 1$): the output bit stream is

$$\underbrace{1\,1\,1}_{\frac{7}{8}}\;\underbrace{1\,0\,1}_{\frac{5}{8}}\;\underbrace{0\,1\,1}_{\frac{3}{8}}\;\underbrace{0\,0\,1}_{\frac{1}{8}}\;\underbrace{0\,0\,0}_{0}$$

so the uniforms are $u = .111,\ .101,\ .011,\ .001,\ .000,\ \ldots$

Tausworthe generators: maximal period

Maximum period. The output period reaches its maximal value

$$\rho = 2^k - 1$$

iff the characteristic polynomial is primitive and the spacing is coprime to the period:

$$Q(z) = z^k - a_1 z^{k-1} - \cdots - a_{k-1} z - a_k \text{ primitive},$$

$$\gcd(\nu, 2^k - 1) = 1.$$

In most applications, only two coefficients are non-zero:

$$Q(z) = z^k - a_r z^{k-r} - a_k,$$

which, in $\mathcal{F}_2$, brings us back to the xor recurrence $x_n = x_{n-r} \oplus x_{n-k}$ of the LFSR.
Caveat. Linearity in $\mathcal{F}_2$ is a strong structural flaw: plain LFSRs/Tausworthe are not good on their own, and are combined, or replaced by MT19937 and xoshiro (next slide).

Implementations

More generally, we construct a fast implementation by using shifts, xor's, masks,.... We can also combine them.

Most popular:

- Mersenne Twister MT19937 (Matsumoto and Nishimura); period of $2^{19937} - 1$
- xoshiro: `https://prng.di.unimi.it/`; by default, Julia uses `xoshiro256++`

But even these recent generators are not without flaws! See e.g.

- It Is High Time We Let Go Of The Mersenne Twister
- Unveiling patterns in xorshift128+ pseudorandom number generators

PCGs

- Melissa O'Neil introduced the PCG family (`https://www.pcg-random.org/`), which applies permutation functions to the output of an LCG.
- Vigna (author of xorshift/xoshiro) strongly criticizes PCG (`https://pcg.di.unimi.it/pcg.php`). He argues that PCG's mechanism for generating independent streams (varying the LCG increment) lacks the rigorous guarantees of MRGs or CBRNGs.
- Vigna also claims PCG is "not competitive" in speed compared to xoshiro.
- However, this performance claim is debated: practical benchmarks (e.g., Julia's `RandomNumbers.jl`, or its adoption in NumPy) often show PCG to be extremely fast and competitive with xoshiro.

Counter-based generators (CBRNG)

Recall the general RNG framework introduced earlier:

- Transition function: $s_n = f(s_{n-1})$
- Output function: $u_n = g(s_n)$

In previously covered RNGs (LCG, MRG, LFSR, etc.):

- The transition function f does most of the transformation (mixing the state).
- The output function g is very simple (e.g., extracting the state as a fraction $u_n = s_n/m$).

Counter-based generators (CBRNGs) take the exact opposite approach:

- The transition f is trivial: $s_n = s_{n-1} + 1$ (the state is just a counter).
- The output function g is a complex function (often based on cryptographic block ciphers) that ensures the output looks random.

CBRNGs properties

- Because the state is just a counter, it is trivially easy to jump ahead. To compute the n-th number, we just evaluate the complex function on the n-th counter value, without generating the preceding values!
- This makes CBRNGs extremely suitable for parallel computing (GPUs, massive multithreading).
- They pass demanding test suites such as TestU01's BigCrush, with sufficiently many rounds, and are efficient on both CPUs and GPUs.
- The most popular implementation, Philox, detailed next, is the default generator in TensorFlow and PyTorch (JAX uses Threefry, another CBRNG).

Counter-based generators: a stateless map

A CBRNG is a stateless map onto a block of random bits:

$$R = E_K(C),$$

where

- E_K is a bijective function parameterized by K. For Philox, it acts as a non-cryptographic block cipher (a substitution-permutation network) using wide integer multiplications and XORs;
- $C = (c_1, \ldots, c_N)$ is a counter: the position in the sequence;
- $K = (k_1, \ldots, k_M)$ is a key: it identifies the stream; for Philox, $M = N/2$.

The sequence follows by incrementing the counter:

$$C,\ C+1,\ C+2,\ \ldots$$

Consequence: identical (C, K) always yields identical bits on any architecture — ideal for reproducible parallel Monte Carlo.

Counter-based generators: parallel streams

- **Streams via keys:** The key identifies the stream, the counter the position. Jumping ahead is a simple integer addition.
- **Stateless & Reproducible:** No internal state to update sequentially. Identical (C, K) pairs yield identical bits everywhere (ideal for GPUs).
- **Stochastic optimization:** One independent stream per scenario.

Widespread adoption (Philox / Threefry):

- **Default in ML:** PyTorch (GPU), TensorFlow, JAX.
- **Available in:** NumPy (`Generator`), NVIDIA cuRAND.

Counter-based vs conventional generators

	Conventional (MRG, PCG, xorshift)	**Counter-based** (Philox, Threefry)
Stored state	full generator state	counter + key
Workload	transition function	output function (rounds)
Jump ahead	dedicated algorithms	addition on the counter
Independent streams	dedicated machinery	distinct keys
Draw order	sequential	position-indexed
GPUs	serial dependency	stateless threads

Philox: an encrypt-increment CBRNG

Introduced by Salmon, Moraes, Dror & Shaw (SC 2011), together with Threefry and AES-based designs: the *Random123* library, `https://github.com/DEShawResearch/random123`.

- Default variant (PyTorch, TensorFlow, cuRAND):

$$\text{Philox4} \times 32{-}10, \qquad N = 4,\ W = 32 \text{ bits},\ r = 10 \text{ rounds},$$

  i.e. a block of $4 \times 32 = 128$ bits per call.
- Not cryptographic: the few rounds are meant to pass statistical tests, not to resist cryptanalysis (e.g. the key space is finite: 2^{64} keys).
- Fewer rounds are faster but display measurable correlations.

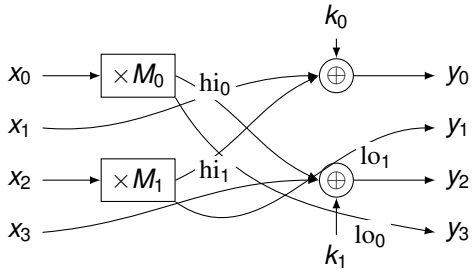
Philox: one round — four elementary operations

Each round is a bijection on (x_0, x_1, x_2, x_3), given two subkeys (k_0, k_1). It alternates arithmetic (multiplication) and bitwise (xor) mixing.

1. Multiply. Build the 64-bit products $P_0 = M_0 x_0$ and $P_1 = M_1 x_2$, with the odd multipliers $M_0 = \texttt{D2511F53}_{16}$, $M_1 = \texttt{CD9E8D57}_{16}$.
2. Split. Cut each product into a high and a low 32-bit half, $P_j = \mathrm{hi}_j\, 2^{32} + \mathrm{lo}_j$: bits travel from the low counter words to all 64 positions.
3. Fold. Mix a high half, a passing lane and a subkey with xor: $y_0 = \mathrm{hi}_1 \oplus x_1 \oplus k_0$ and $y_2 = \mathrm{hi}_0 \oplus x_3 \oplus k_1$.
4. Swap the lows across the lanes: $y_1 = \mathrm{lo}_1$ and $y_3 = \mathrm{lo}_0$.

Why a bijection? M_0, M_1 odd $\Rightarrow$ invertible modulo 2^{32}: any y has a unique predecessor. Bijectivity avoids collisions and gives a clean period.

Philox: one round — wiring



$$(y_0,\ y_1,\ y_2,\ y_3) = (\mathrm{hi}_1 \oplus x_1 \oplus k_0,\ \mathrm{lo}_1,\ \mathrm{hi}_0 \oplus x_3 \oplus k_1,\ \mathrm{lo}_0),$$

where $(\mathrm{hi}_j, \mathrm{lo}_j)$ are the high/low 32-bit halves of $M_j x_j$.

Philox: the key schedule (Weyl sequence)

Each round must use different subkeys, otherwise round symmetry would leak structure. Between two rounds, the subkeys advance by a fixed Weyl step:

$$(k_0,\, k_1) \mathrel{+}= (W_0,\, W_1)$$

with the constants

$$W_0 = \texttt{9E3779B9}_{16}, \qquad W_1 = \texttt{BB67AE85}_{16},$$

taken as odd parts of well-chosen irrationals (W_0 from the golden ratio φ, W_1 from $\sqrt{3} - 1$).

- A single 64-bit integer addition per round: cheap.
- The schedule is invertible, so each round stays a bijection.

Philox: from blocks to a number sequence

After $r = 10$ rounds, the counter/key pair has been fully scrambled into a block $R = E_K(C)$. We map a 32-bit word to $[0, 1)$ with a plain division:

$$u = \frac{R_j}{2^{32}} \in [0, 1),$$

so each call provides up to 4 uniforms (or a 128-bit block).

Summary of the streaming model:

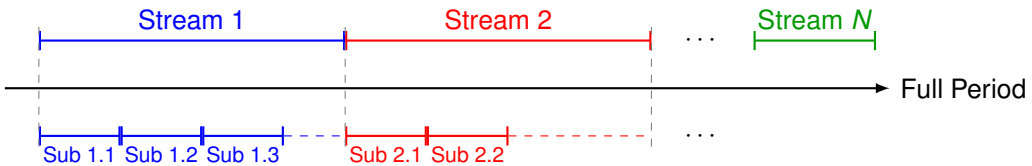
- draw next block: increment the counter, $C \leftarrow C + 1$;
- jump m blocks ahead: $C \leftarrow C + m$ (an integer addition);
- switch stream: change the key K;
- no state to store or synchronize: natural for GPUs and parallel workers.

Jump ahead

- A very useful possibility proposed by some implementation is the possibility to make a jump of m positions in the random number sequences, with m very large.
- This allows one to easily define independent random variables.
- Useful in simulation when relying on common random numbers.

Historically, MRG32k3a was one of the few generators offering built-in functions for independent streams and substreams. Today, the library `RandomDataStreams.jl` extends this capability to a wide variety of generators in a unified framework.

Streams and Substreams



- Streams: The global sequence is partitioned into huge, non-overlapping segments (e.g., 2^{127} values).
- Substreams: Each stream is further subdivided (e.g., 2^{76} values).
- **Use case:** Allocate one stream per scenario, and advance to the next substream for each simulation run (replication) to maintain alignment (Common Random Numbers).

RandomDataStreams.jl

A Julia package providing a unified framework for independent streams and substreams.

Installation: `]  add RandomDataStreams`

Code example: One stream per scenario, one substream per replication.

```
using RandomDataStreams

gen = MRG32k3aGen()      # Generator instance
rng = next_stream!(gen) # Independent stream (Scenario 1)

# Replication 1
u1 = rand(rng)

# Replication 2
next_substream!(rng)     # Fast jump to the next substream
u2 = rand(rng)
```

Non-uniform random variables generation

A good reference: Luc Devroye, *Non-Uniform Random Variate Generation*, `http://luc.devroye.org/rnbookindex.html`.

Assume that we have a good uniform random variates generator, but we want to generate random variables following various probability laws: Normal, Weibull, Poisson, binomial,...

The desired properties are:

- correct method (or good approximation);
- as simple as possible, but as fast as possible;
- low memory consumption;
- robust;
- compatible with variance reduction technique (as quasi-Monte Carlo).

Inversion Method: Principle

Consider a random variable X with cumulative distribution function F. Let $U \sim U(0,1)$ and define:

$$X = F^{-1}(U) = \min\{x : F(x) \geq U\} \quad \text{(Generalized inverse)}.$$

Then,

$$P[X \leq x] = P[F^{-1}(U) \leq x] = P[U \leq F(x)] = F(x),$$

i.e., X precisely follows the desired distribution.

This foundational principle applies universally:

- **Continuous case:** $F(X) \sim U[0,1]$;
- **Discrete case:** ensures $P[X = x_i] = p(x_i)$ assuming ordered x_i;
- **Mixed distributions:** works seamlessly.

Inversion: The Gold Standard

- **Monotonic mapping:** Inversion guarantees a monotonic relationship between the uniform draw U and the generated variable X.
- **Variance Reduction:** This monotonicity is strictly required by variance reduction techniques like *Common Random Numbers* (CRN) or *Antithetic Variates*, which break down with other generation methods (like acceptance-rejection).
- **Copulae:** It is the preferred method when building dependent joint distributions using copulae.

Inversion (2)

- Advantage: monotone, only one U for all X.
- Weakness: for some laws, F is very difficult to invert. But we can often approximate F^{-1}.

Example: normal law.
If $Z \sim N(0,1)$, then $X = \sigma Z + \mu : N(\mu, \sigma^2)$.
It is therefore sufficient to be able to generate a $N(0,1)$, of density
$f(x) = (2\pi)^{-1/2} e^{-x^2/2}$.
We do not have any formula for $F(x)$ or $F^{-1}(x)$. Efficient codes however exist to approximate $F^{-1}(x)$.

Chi-square, gamma, beta, etc.: it is much more complicated since the form of F^{-1} depends of the distribution parameters.

Inversion for discrete distributions

Recall that

$$p(x_i) = P[X = x_i]; \quad F(x) = \sum_{x_i \le x} p(x_i).$$

We have to generate $U \sim U(0,1)$, search

$$I = \min\{i \mid F(x_i) \ge U\}$$

and return $X = x_I$.

Initialization: store the values x_i and the cumulative probabilities $F(x_i)$ in arrays, for $i = 1, \ldots, n$.

Various algorithms can perform this search. Their efficiency heavily depends on the distribution and the number of outcomes n.

Discrete Inversion: Search Algorithms

1. **Linear search** (time in $\mathcal{O}(n)$):
   `generate` $U \sim U(0,1)$;
   $i \leftarrow 1$;
   `while` $F(x_i) < U$ `do` $i \leftarrow i + 1$;
   `return` x_i.

2. **Binary search** (time in $\mathcal{O}(\log(n))$):
   `generate` $U \sim U(0,1)$;
   $L \leftarrow 0$; $R \leftarrow n$;
   `while` $L < R - 1$ `do`
     $m \leftarrow \lfloor (L + R)/2 \rfloor$;
     `if` $F(x_m) < U$ `then` $L \leftarrow m$ `else` $R \leftarrow m$;
   `return` x_R.

Other approaches: composition

Assume that F is a convex combination of several cumulative distribution functions:

$$F(x) = \sum_{j=0}^{\infty} p_j F_j(x),$$

and that it is easier to invert F_j, $j = 0, \ldots, \infty$ than F.

Generate $J = j$ with the probability p_j, than generate X following F_J.

The method therefore requires two uniforms for each random variable, and exploit the decomposition

$$P[X \leq x] = \sum_{j=1}^{\infty} P[X \leq x | J = j] P[J = j] = \sum_{j=1}^{\infty} F_j(x) p_j.$$

Convolution

Convolution. Assume that

$$X = Y_1 + Y_2 + \ldots + Y_n,$$

where the Y_i are independent, of given laws. We generate the Y_i, $i = 1, \ldots, n$, and we sum.

Examples: Erlang (sum of exponentials with same mean), binomial.

Acceptance/rejection: the most important technique after inversion.

We consider the case where X is continuous (the discrete case is analogous). Let $f(x)$ be the density of X, and let t be a "hat" function that majors f, i.e. $f(x) \leq t(x)\ \forall\, x$.

Acceptance/rejection

We can normalize t in a density r:

$$r(x) = t(x)/a, \text{ where } a = \int_{-\infty}^{\infty} t(s)ds.$$

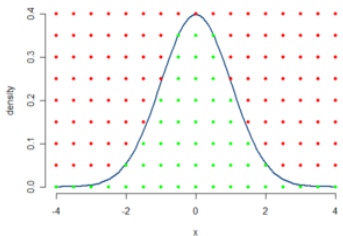
We choose t so that

1. it is easy to generate random variables of density r;
2. a is small (close 1), or in other terms, $t(x)$ is close to $f(x)$.

The choice of t may be automatized.

Algorithm: Repeat

1. generate Y of density $r(x)$;
2. generate $U : U(0,1)$ independently of Y;
3. until $U \leq f(Y)/t(Y)$;
4. return Y.



Particular cases

Sometimes, we can benefit of mathematical transformations.
The main weakness is that they are seldom compatible with variance reduction techniques.

Example: Box-Muller method for the normal law.

Idea: it is easier to generate a point (X, Y) from the bivariate normal law, of density on $\mathbb{R}^2$

$$f(x, y) = \frac{1}{2\pi} e^{-(x^2+y^2)/2}.$$

We change the cartesian coordinates (X, Y) by the polar coordinates (R, Θ):

$$R^2 = X^2 + Y^2; Y = R \sin \Theta.$$

It gives an elegant approach, but incompatible with variance reduction techniques and can be slower than inversion.

Box-Muller Algorithm

1. Independently draw U_1, U_2 from a $U(0,1)$ distribution.
2. Set

$$R = \sqrt{-2\log(U_1)}$$
$$\theta = 2\pi U_2$$

3. Set

$$X = R\cos(\theta)$$
$$Y = R\sin(\theta)$$

Multivariate distributions

What if we want to draw from $\boldsymbol{X} : \Omega \to \mathcal{X} \subseteq \mathbb{R}^d$?
General approach:

- Generate the marginals
- Combine the marginals as desired.

Multivariate distributions: simple cases

- Ideally, we can explicitly transform the marginals.
- Example: multivariate normal $\boldsymbol{X} \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma)$, with

$$\boldsymbol{\mu} = \begin{pmatrix} \mu_1 \\ \mu_2 \\ \vdots \\ \mu_d \end{pmatrix} \qquad \Sigma = \begin{pmatrix} \sigma_1^2 & \sigma_{1,2} & \dots & \sigma_{1,d} \\ \sigma_{1,2} & \sigma_2^2 & \dots & \sigma_{2,d} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{1,d} & \sigma_{2,d} & \dots & \sigma_d^2 \end{pmatrix}$$

Then,

$$\boldsymbol{X} = \boldsymbol{\mu} + L\boldsymbol{Z}, \quad \text{where } \boldsymbol{Z} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$$

with $\Sigma = LL^T$, and L is the Cholesky factor.
Drawing from $\mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$: generate d $\mathcal{N}(0, 1)$ independently.

Multivariate distributions: copulae

Intuition: Copulae allow us to completely decouple the *marginal distributions* (what each variable looks like individually) from their *dependence structure* (how they move together).

A d-variate copula $C : [0,1]^d \to [0,1]$is the cdf of a random vector $(U_1, \ldots, U_d)$ with $U(0,1)$ margins:

$$C(u) = P[U_1 \leq u_1, \ldots, U_d \leq u_d]$$

where

$$P[U_j \leq u_j] = u_j$$

for $j = 1, \ldots, d$, and $0 \leq u_j \leq 1$.

Sklar's theorem

Theorem (Sklar's theorem (1959))

- *Let H be a multivariate distribution function with margins $F_1, \ldots, F_d$. There exists a copula C such that*

$$H(x_1, \ldots, x_d) = C\left(F_1(x_1), \ldots, F_d(x_d)\right), \quad x_1, \ldots, x_d \in \overline{\mathbb{R}}.$$

  If F_i, $i = 1, \ldots, d$, are continuous then C is unique. Otherwise C is uniquely determined on the Cartesian product of the range of the marginals $F_1\left(\overline{\mathbb{R}}\right) \times \ldots \times F_d\left(\overline{\mathbb{R}}\right)$.

- *Conversely, if C is a copula and $F_1, \ldots, F_d$ are univariate distribution functions, then the function H defined above is a multivariate distribution function with margins $F_1, \ldots, F_d$.*

Copula

If we know the joint CDF F and the marginals $F_1, \ldots, F_d$, we can find the copula via

$$C(u_1, \ldots, u_d) = F\left(F_1^{-1}(u_1), \ldots, F_d^{-1}(u_d)\right),$$

where $F_j^{-1}(\cdot)$ is the generalized inverse for margin j:

$$F_j^{-1}(u) = \min\{x \,:\, F_j(x) \geq u\}.$$

Fréchet–Hoeffding bounds

Any bivariate copula C verifies

$$\max(F_X(x) + F_Y(y) - 1, 0) \leq C(x, y) \leq \min(F_X(x), F_Y(y))$$

Upper bound proof

$$\begin{aligned} C(x, y) &= P[X \leq x \cap Y \leq y] \\ &\leq \min\left(P[X \leq x], P[Y \leq y]\right) = \min(F_X(x), F_Y(y)) \end{aligned}$$

since $\{X \leq x\} \cap \{Y \leq y\} \subseteq \{X \leq x\}$ and $\subseteq \{Y \leq y\}$.

Fréchet–Hoeffding bounds

Lower bound proof

$$\begin{aligned} 1 - C(x, y) &= P[X > x \cup Y > y] \\ &\leq P[X > x] + P[Y > y] \\ &= 1 - F_X(x) + 1 - F_Y(y) \end{aligned}$$

Thus,

$$C(x, y) \geq F_X(x) + F_Y(y) - 1.$$

Note: this can be viewed as a particular of the Bonferroni inequality

$$P\left[\cap_{i=1}^{d} A_i\right] \geq 1 - d + \sum_{i=1}^{d} P[A_i].$$

Gaussian copula

$$C\left(F_1(x_1), \ldots, F_d(x_d)\right) = \Phi_\Sigma\left(\Phi^{-1}\left(F_1(x_1)\right), \ldots, \Phi^{-1}\left(F_d(x_d)\right)\right),$$

where

- Φ is the cdf of a $\mathcal{N}(0,1)$,
- Φ_Σ is the cdf of a d-dimensional $\mathcal{N}(\mathbf{0}, \Sigma)$.

Copulae estimation

- Parametric estimation;
- Non-parametric estimation;
- Machine-learning models (see Jutras-Dubé et al., *Transportation Research Part C*, 2024).

Further reading

- **General RNG principles:** L'Ecuyer, P. (2012). *Random Number Generation*. https://www.iro.umontreal.ca/~lecuyer/myftp/papers/wsc12rng.pdf
- **Philox & CBRNGs:** Salmon et al. (2011). *Parallel Random Numbers: As Easy as 1, 2, 3*. https://doi.org/10.1145/2063384.2063405
- **PCG family:** O'Neill, M. E. (2014). https://www.pcg-random.org/
- **xoshiro / xorshift:** Blackman, D. & Vigna, S. (2021). *Scrambled Linear Pseudorandom Number Generators*. https://prng.di.unimi.it/
- **Non-uniform generation:** Devroye, L. (1986). *Non-Uniform Random Variate Generation*. http://luc.devroye.org/rnbookindex.html