A short tutorial of random numbers generation

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

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 transform an 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 the period denoted by ρ. In other terms, we have $s_{n+\rho} = s_n$ for all $n \geq \tau$.

U[0,1]-distributed random numbers (2)

- 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.

$$
\begin{array}{ccccccccccccc}
\cdots & \xrightarrow{f} & s_{\rho-1} & \xrightarrow{f} & s_0 & \xrightarrow{f} & s_1 & \xrightarrow{f} & \cdots & \xrightarrow{f} & s_n & \xrightarrow{f} & \cdots \\
 & & g\big\downarrow & & g\big\downarrow & & g\big\downarrow & & & & g\big\downarrow & & \\
\cdots & & u_{\rho-1} & & u_0 & & u_1 & & \cdots & & u_n & & \cdots
\end{array}
$$

How to choose f and g?

Goals: large ρ, good uniformity, “random” behavior.

Linear congruential generators

Linear congruential generators (LCGs) have been introduced by Lehmer in 1951. We use the recursive formula

$$Z_i = f(Z_{i-1}) = (aZ_{i-1} + c) \mod m.$$

Given two numbers, a (the dividend) and n (the divisor),

$$a \text{ modulo } n$$

(abbreviated as a mod n) is the remainder, on division of a by n. For instance, the expression ”7 mod 3” gives 1, while “9 mod 3” leads to 0. In other terms,

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

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).à Considere the case $c \neq 0$.

Theorem (Period)

The LCG has full period if and only if 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.*

A popular LCG is the "standard minimal", as known from the terminology introduced by Park and Miller in 1988:

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

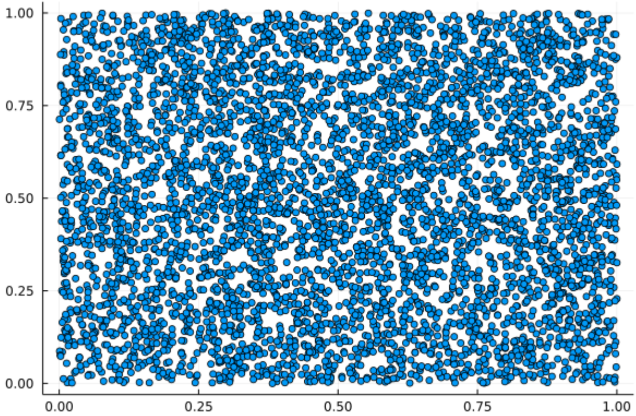
Observe that $2147483647 = 2^{31} - 1$; on 32-bit architectures, the largest representable (signed) integer is 2^{31}.

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)
        return state*am_mil  # produce a number in (0,1)
    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 linear congruential generator:

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

In practice, we will 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. This kind of generators is very popular.

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 if $\rho = m^k - 1$.

Period of MRG's

It can be shown that for $k > 1$, it is sufficient to have at least two non-zero coefficient, 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 excluding directly some generators that have known strong deficiencies.

Choosing a good MRG's

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 plans! 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 module $m = m_1 m_2$, 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, and is amongst the most popular and efficient known generators. 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: $k = 3$,
$m = m_1 m_2 = 18446645023178547541$,
$a_1 = 18169668471252892557$,
$a_2 = 3186860506199273833$,
$a_3 = 8738613264398222622$.

Période $\rho = (m_1^3 - 1)(m_2^3 - 1)/2 \approx 2^{191}$.

MRG32k3a: 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
```

RDST library

`https://github.com/JLChartrand/RDST.jl`

To update!

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

Alternatives to MRG's: random numbers generators based on linear recurrence in $\mathcal{F}_2$.

Galois field $\mathcal{F}_2$: set $\{0, 1\}$ on which we define addition and multiplication operation modulo 2.

We construct two sequence of bits vectors $\boldsymbol{x}_n$ and $\boldsymbol{y}_n$ with the linear recurrences

$$\boldsymbol{x}_n = X\boldsymbol{x}_{n-1} \quad \text{(state vector, } k \text{ bits)},$$

and

$$\begin{aligned} \boldsymbol{y}_n &= B\boldsymbol{x}_n \quad \text{(output vector, } w \text{ bits)}, \\ u_n &= \sum_{j=1}^{w} y_{n,j-1} 2^{-j} = .y_{n,0}\ y_{n,1}\ y_{n,2}\ \cdots \quad \text{(sortie)}. \end{aligned}$$

Random numbers generators on $\mathcal{F}_2$ (2)

The implementation of $\mathcal{F}_2$-generators is often quite complex, but it is possible to operate bitwise, so that they are numerically very fast.

The LFSR (linear feedback shift register), while known to have important deficiencies, gives an illustration of such generators.

We use the relations (with $a_k \neq 0$)

$$
\begin{aligned}
u_n &= \sum_{l=1}^{w} x_{n\nu+j-1} 2^{-l} = .x_{n\nu} x_{n\nu+1} x_{n\nu+2} \ldots x_{n\nu+l-1} \\
X &= \begin{pmatrix} & 1 & & \\ & & \ddots & \\ & & & 1 \\ a_k & a_{k-1} & \ldots & a_1 \end{pmatrix}^{\nu} \text{ et } B = I.
\end{aligned}
$$

Tausworthe Generator (2)

Maximum period. $\rho = 2^k - 1$ iff
$Q(z) = z^k - a_1 z^{k-1} - \cdots - a_{k-1} z - a_k$ is primitive and
$\text{pgcd}(\nu, 2^k - 1) = 1$.

In most applications, only two coefficients are nonzero in order to simplify implementation, so we obtain
$Q(z) = z^k - a_r z^{k-r} - a_k$.

Since we are working in $\mathcal{F}_2$, the recurrence on x_n becomes

$$x_n = (x_{n-r} + x_{n-k}) \mod 2.$$

The execution of the addition modulo 2 is equivalent to the instruction exclusive-or (xor) ont the bits:

$$x_n = \begin{cases} 0 & \text{si } x_{n-r} = x_{n-k}, \\ 1 & \text{si } x_{n-r} \neq x_{n-k}. \end{cases}$$

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/`

The generators are slightly less efficient on a statistical point of view, but are faster.

Jump ahead

- A very useful possibly 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 to easily define independant random variables.
- Useful in simulation when relying on common random random numbers.

The MRG32k3a implementation proposes functions to generate independent streams and substreams.

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

This is the prefered method, if it can be applied. The reason is that it is compatible with variance reduction.

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

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

Then

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

i.e., X has the desired distribution. Indeed,

- in the continuous case, $F(X)\ U[0,1]$;
- in the discrete case, it is easy to prove that $P[X = x_i] = p(x_i)$, for all i, and we assume $x_1 < x_2 < \ldots < x_n$;
- The principle still works for mixte distributions.

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, search $I = \min\{i | F(x_i) \ge U\}$ and return x_I.

Various algorithms perform this search. Their efficiency depends of the distribution.

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

1. Linear search (time in $O(n)$): $U \leftarrow U(0,1); \quad i \leftarrow 1;$
   while $F(x_i) < U$ do $i \leftarrow i + 1$; return x_i.
2. Binary search (time in $O(\log(n))$):
   $U \leftarrow U(0,1); \quad L \leftarrow 0; \quad R \leftarrow n;$
   while $L < R - 1$
       $m \leftarrow \lfloor (L+R)/2 \rfloor;$
       if $F(x_m) < U$ then $L \leftarrow m$ otherwise $R \leftarrow m$;
   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.

Exemples: 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 cas is analoge). 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:

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

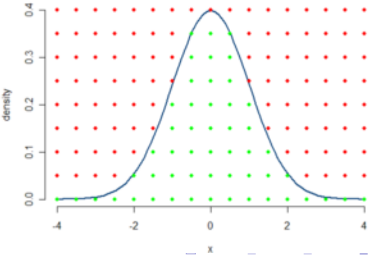
We choose t so that

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 $U : U(0, 1)$ independantly 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

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

3. Set

$$
\begin{aligned}
X &= R\cos(\theta) \\
Y &= R\sin(\theta)
\end{aligned}
$$