Random Numbers

Dieter W. Heermann

Monte Carlo Methods

2018

Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.
John Von Neumann, 1951

Intuitively, we can list a number of criteria that a sequence of numbers must fulfill to pass as a random number sequence:

- unpredictability,
- independence,
- without pattern.

These criteria appear to be the minimum request for an algorithm to produce random numbers. More precisely we can formulate:

- uniform distribution,
- uncorrelated,
- passes every test of randomness,
- large period before the sequence repeats (see later),
- sequence repeatable and possibility to vary starting values,
- fast algorithm.

The most common generators use very basic operations and apply them repeatedly on the numbers generated in previous steps. We formulate this as a recursion relation

$$x_{i+1} = G(x_i), \qquad x_0 = \text{initial value} \quad , \tag{1}$$

where we have made explicit only the dependence on the immediate predecessor. The most important representatives of this class of generators are the

- linear congruential,
- lagged Fibonacci,
- shift-register or a
- combination of linear congruential.

Linear Congruential Generators

A very simple generator is constructed using the modulo function.

$$G(x) = (xa + b) \bmod M \tag{2}$$

This function produces a dilatation, translation and a folding back into the interval. Random number generators based on this function are called *linear congruential generators* or LCG(a,b,M) for short. If we assume integers as the set on which the modulo function is defined, then for example, the range of integer numbers for a 32-bit architecture is at most $M = 2^{31} - 1$. Here we assume that one bit is taken for the sign of the number. Then the numbers range at most from 0 to $M - 1$. Of course, we can map these onto the real interval between 0 and 1, recognizing that this is an approximation to the real-valued random numbers.

Inverse Congruential Generators

A very simple generator is constructed using the modulo function.

$$x_{n+1} = (x_n^{-1} a + b) \bmod M \quad , \tag{3}$$

where x_n^{-1} is the multiplicative inverse of x_n in the integers mod m with 0^{-1} defined as 0.

- The choice of the parameters a, b and M determine the statistical properties and how many different numbers we can expect before the sequence repeats itself.
- The period can be shown to be maximal, if M is chosen to be a prime number. Then the whole range of numbers occurs.
- Here we only consider modulo generators with $b = 0$.
- Such generators are called *multiplicative* and the short form MLCG(a,M) is used for such generators.
- These are the most commonly used, since one can show that *additive* generators, i.e. generators with b in general non zero have undesirable statistical properties.
- The choices for the parameter a are manifold. For example $a = 16807$, 630360016 or 397204094 are possible choices with $M = 2^{31} - 1$.

```
/*------------------------------------------------------------------------ */
/*                        Modulo Generator                                 */
/*------------------------------------------------------------------------ */
int ModGenerator(modul,multi,inc,seed,max_sweeps,x)
    int             modul;
    int             multi;
    int             inc;
    int             seed;
    int             max_sweeps;
    float           *x;
{
    /*---------------------------------------------------------- */
    /*                    Declarations                          */
    /*---------------------------------------------------------- */
    int i;
    double r;
    double factor, increment, modulus;
    /*---------------------------------------------------------- */
    /*               End of declares                            */
    /*---------------------------------------------------------- */

    r         = (double) seed;
    factor    = (double) multi;
    increment = (double) inc;
    modulus   = (double) modul;

    for(i=0; i< max_sweeps; i++) {
      r=fmod(r*factor + increment,modulus);
      x[i] = (float) r / modulus;
    }
    return 0;
}
```

In C/C++:

```
#include <stdlib.h>
int iseed, randInt;
float randFloat;

srand(iseed);
randInt = rand();
randFloat = (float) randInt / (float) RAND MAX;
```

Add-with-Carry/Subtract-with-Carry Generators

- Add-with-carry and subtract-with-carry generators rely on two numbers, the carry c and the modular base M.
- Add-with-carry generator;

$$x_{n+1} = (x_{n-s} + x_{n-r} + c) \bmod M \tag{4}$$

- Subtract-with-carry

$$x_{n+1} = (x_{n-s} - x_{n-r} - c) \bmod M \tag{5}$$

- Problems:
  - Require an initial seed of a sufficiently long sequence.
  - Pairs (or triplets) of terms fall on planes (see modulo generator).

Fibonacci Generators

The *lagged Fibonacci generator*, symbolically denoted by LF(p,q,$\otimes$) with $p > q$, is based on a Fibonacci sequence of numbers with respect to an operation which we have given the generic symbol $\otimes$.
Let $\mathcal{S}$ be the model set for the operation $\otimes$, for example the positive real numbers, the positive integers, or the set $\mathcal{S} = \{0, 1\}$. The binary operation $\otimes$ computes a new number from previously generated numbers with a lag p

$$x_n = x_{n-p} \otimes x_{n-q} \quad , \quad p > q \qquad . \tag{6}$$

To start the generator we need p numbers. These can be generated using for example a modulo generator. The advantage of the lagged Fibonacci generator, apart from removing some of the deficiencies that are build into the modulo type generators, is that one can operate on the level of numbers or on the level of bits.

```
for(i=0; i< max_sweeps; i++) {
    mf[p] = mf[p] + mf[q];
    if ( mf[p] > 1 ) mf[p] -= 1;
    x[i] = mf[p];
    if ( ++p == lagP-1 )  p = 0;
    if ( ++q == lagP-1 )  q = 0;
}
```

In the following I have listed some lagged Fibonacci generators:

Recursion Relation	Period
$x_i = x_{i-17} - x_{i-5} \bmod (2^n)$	$(2^{17} - 1)2^{n-1}$
$x_i = x_{i-17} + x_{i-5} \bmod 2^n$	$(2^{17} - 1)2^{n-1}$
$x_i = x_{i-31} - x_{i-13} \bmod 2^n$	$(2^{31} - 1)2^{n-1}$
$x_i = x_{i-55} - x_{i-24}$	$2^{24}(2^{97} - 1)$ with 24 Bit Mantissa

Example: The shift bit register generator R250

R250.C Page 1 of 2
Printed For: Heermann

```
# include <math.h>
# define RAND_MAX 2147483647

/*==============================================================*/
/*                                                              */
/*       Random Number Generator:  R 2 5 0                      */
/*                                                              */
/*       program version 1.0 for C                              */
/*       Dieter W. Heermann                                     */
/*       may 1990                                               */
/*==============================================================*/

int init_r250(seed, m_f_ptr)
   int seed;
   int *m_f_ptr;
   {

    int  i,tmp, dummy, one ;
    int  *start;

    start = m_f_ptr;
    srand(seed);
    one    = 1;

    /* warm up the usual random number generator */
    for (i=0; i< 100; i++)
       {dummy = rand();
        }

    /* now draw the 250 (251)initial bit sequences */
    for (i=0 ; i<251; i++)
       {*m_f_ptr++ = rand();
       }

    /* now orthogonalize as best as we can  */
    m_f_ptr = start;
    for (i=0; i < 30; i++)
       { tmp      = *m_f_ptr;
  *m_f_ptr = tmp | one;
  one      = one << 1;
  m_f_ptr++;
       }

    return 0;
  }

int r250 (n, x_ptr, m_f_ptr, save)
   int    n;
   float  *x_ptr;
   int    *m_f_ptr;
   int    save;
   {

    int ind ;
    int j, min,k,ll;
    float *ran_ptr;

    ind     = save;
    ll      = n + 250;
    ran_ptr = x_ptr;
    j       = 1;
```

For example, we can construct a *generalized shift-register generator* GFSR(p,q,$\otimes$), where the operation is interpreted as the *exclusive or*, which acts on every of the 32 bits in a computer word. This generator is also known under the name of *R*250. (Follow this link to access the code for the R250.c.)

Non Uniform Distributions

- Let us turn now to the generation of non-uniform distributions. First we look at the normal or Gaussian distribution.
- Typically algorithms generating non-uniform variates do so by converting uniform variates.
- In its most straightforward form a normal deviate x with mean $< x >$ and standard deviation σ is produced as follows:
- Let n be an integer, determined by the needed accuracy. Then
  - sum n uniform random numbers r_i from the interval $(-1, 1)$:

$$s_n = \sum_{i=1}^{n} r_i$$

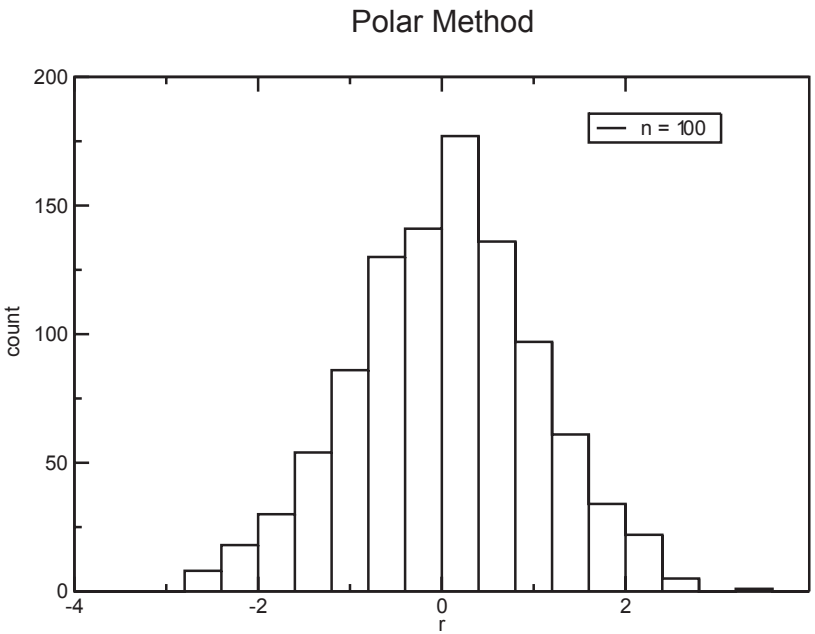
  - and let $x =< x > +\sigma s_n \sqrt{3.0/n}$.

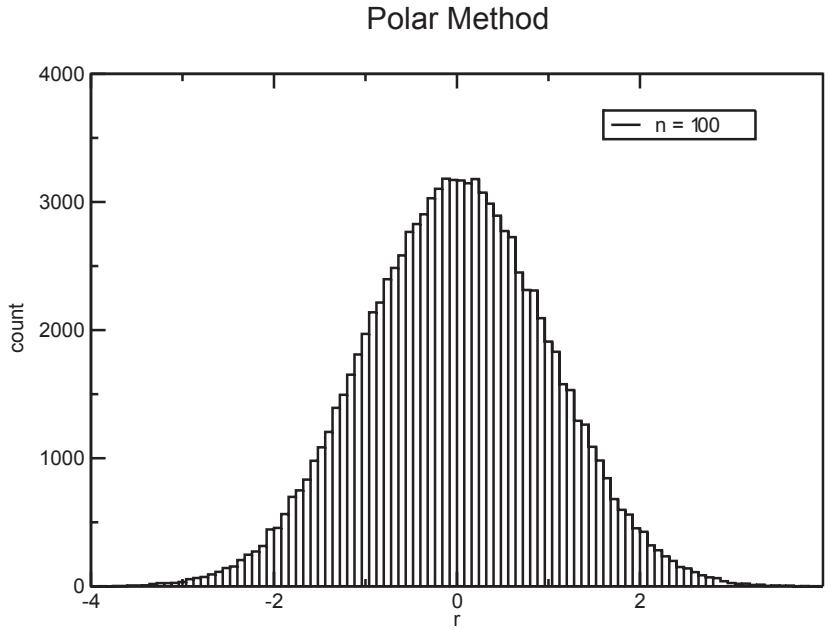
- Let $G(x)$ be a function on the interval $[a, b]$ with $0 < G(x) < 1$ and $f(x)$ the probability distribution $f(x) = a \exp[-G(x)]$, where a is a constant.

1: Generate r from a uniform distribution on $(0, 1)$
2: Set $x = a + (b - a)r$
3: Calculate $t = G(x)$
4: Generate $r_1, r_2, ..., r_k$ from a uniform distribution on $(0, 1)$ (k is determined from the condition $t > r_1 > r_2 > ... > r_{k-1} < r_k$)
5: **if** $t < r_1$ **then**
6: $k = 1$
7: **end if**
8: **if** k is even **then**
9: reject x and go to 1
10: **else**
11: x is a sample
12: **end if**

- An interesting method for generating normal variates is the *polar method*. It has the advantage that two independent, normally distributed variates are produced with practically no additional cost in computer time.

1: Generate two independent random variables, U_1, U_2 from the interval $(0, 1)$.
2: Set $V_1 = 2U_1 - 1$, $V_2 = 2U_2 - 1$
3: Compute— $S = V_1^2 + V_2^2$
4: **if** $S \geq 1$ **then**
5: return to step 1
6: **else**
7: $x_1 = V_1 \sqrt{-2 \ln S / S}$
8: $x_2 = V_2 \sqrt{-2 \ln S / S}$
9: **end if**


Polar Method
n = 100
count
r
200
150
100
50
0
-4
-2
0
2


Polar Method
n = 100
count
r
4000
3000
2000
1000
0
-4
-2
0
2

The Accept/Reject Method

- Another idea of converting one distribution into another is to accept or reject drawn number for an initial distribution such that the accepted numbers have the desired distribution.
- Assume that we are given a uniform random number generator $U \sim (0,1)$ and $X \sim g$.
- We want to generate $Y \sim f$.
- Assume that there exists a constant c such that $f(x) < cg(x)$ for all x.

1: Generate $X \sim g$
2: Generate $U \sim (0,1)$
3: **if** $U \leq f(X)/cg(X)$ **then**
4: $\quad Y = X$
5: **else**
6: $\quad$ Goto 1
7: **end if**

- To proof that this is correct we show that

$$P(X < y | U \leq f(X)/cg(X)) = P(Y \leq y) \quad .$$

Note that

$$\frac{P(X < y | U \leq f(X)/cg(X)) = P(Y \leq y)}{P(U \leq f(X)/cg(X))} = \frac{\int_{-\infty}^{y} \int_{0}^{f(x)/cg(x)} g(x) du dx}{\int_{-\infty}^{\infty} \int_{0}^{f(x)/cg(x)} g(x) du dx}$$

which simplifies to

$$\int_{-\infty}^{y} f(x) dx \quad .$$

Testing Random Numbers

- A number of statistical tests have been devised to check for the properties of random number generators. To name just a few prominent tests
  - χ^2,
  - Kolmogorov-Smirnov,
  - correlation,
  - run and
  - visual test.
- The statistical tests are tests how well the empirical distribution, i.e., the generated sequence, fits a test distribution. For example, the simple frequency test χ^2 is a test that virtually all random number generators will pass.

- The *run test* tests whether an empirical distribution has monotone decreasing or increasing subsequences and confronts these with the expectation for their occurrence.
- Let us assume that we want to test for monotone increasing subsequences. Such a test is called a *run test up*, otherwise *run test down*.
- A run of length r of a sequence $x = (x_1, ..., x_n)$ is a maximal strictly monotonically increasing (decreasing) subsequence $(x_i, ..., x_{i+r-1})$, i.e.,

$$x_{i-1} > x_i < ... < x_{i+r-1} > x_{i+r} \tag{7}$$

with x_0 positive infinite and x_{n+1} negative infinite.

- The expectation for the distribution of runs is derived from a simple permutation argument and will not be reproduced here.
- For large n, one can show, that the probability to get a run of length r is given by $r/(r+1)!$, hence

$$1/2, 1/3, 1/8, 1/30, 1/144, \ldots \quad . \tag{8}$$

- The way the test is derived shows that this tests for correlation in the generated sequence.
- The expected probabilities for the sequences reflect the independence from correlation.

- A very easy test is the *lattice test*.
- Suppose we have to visit the sites of a simple cubic lattice L^3 at random.
- The three coordinates are obtained from three successively generated random numbers $r_1, r_2, r_3 \in (0, 1)$, as

  1: $ix = rz * L + 1$
  2: $iy = rz * L + 1$
  3: $iz = rz * L + 1$

  where ix, iy, iz are integer variables, implying a conversion of the real right-hand sides to integers, i.e., removal of the fractional part.
- If there are no correlations between successively generated random numbers all sites will eventually be visited.
- However, only certain hyperplanes are visited if correlations exist.

Literature

- F. James, A review of pseudorandom number generators,Computer Physics Communications 60 (1990) 329-344
- G. Marsaglia et al., A random number generator for PC's, Computer Physics Communications 60 (1990) 345-349
- G. Marsaglia, Random numbers fall mainly in the planes, Proc Natl Acad Sci USA 1968; 61(1): 25ˇ201328.
- G. Marsaglia et al., Toward a universal random number generator, Stat. Prob. Lett. 9 (1990) 35
- CERN program library CERNLIB: RIWIAD, RADMUL, DIVONNE etc. VEGAS: G. P. Lepage, Journal of Computational Physics 27 (1978) 192- 203
- RAMBO: R. Kleiss et al., Computer Physics Communications 40 (1986) 359-373