Véletlenszám-generátorok

1. rész

– egy MATLAB® alapú megközelítés –

Baja Zsolt, Vas Orsolya

Matematika és Informatika Intézet, Babeş–Bolyai Tudományegyetem, Kolozsvár, Románia

(bajazsolt98@gmail.com, vas.orsolya@yahoo.com)

1. labor / 2023. október 3–6.

- Tekintsük a természetes számok halmazán az $m \geq 1$ osztót, az $a \in \{0, 1, \ldots, m-1\}$ szorzótényezőt, a $c \in \{0, 1, \ldots, m-1\}$ növekedményt, valamint az $X_1 \in \{0, 1, \ldots, m-1\}$ kezdőértékkel ellátott

$$X_{i+1} = (aX_i + c) \pmod m, \; i \geq 2 \tag{1}$$

  rekurzív multiplikatív lineáris kongruenciát!

- Vegyük észre, hogy az (1)-es rekurzió az m, a, c és X_1 paramétereknek bármely beállítására nem generál „véletlenszerű'' számokat! Rossz paraméterezés esetén a generált sorozat túlságosan hamar periodikussá válhat, nyomon követhető mintát eredményezve. Például az $m = 10$, $X_1 = 2$ és az $a = c = 3$ feltételek esetén a

$$\mathbf{2}, 9, 0, 3, \mathbf{2}, 9, 0, 3, \mathbf{2}, 9, 0, 3, \mathbf{2}, \ldots$$

  periodikus számsorozatot kapjuk eredményül.

- Nyilván az (1)-es rekurzió bármely változata periodikus számsorozatot generál, célunk viszont az, hogy a periodikusan ismétlődő minta hosszát minél nagyobbra nyújtsuk.

- Az m számmal osztva összesen m darab különböző maradékot kaphatunk, de – ahogy a fenti példa is mutatja – nem megfelelő beállítás esetén a létrehozott sorozat nem tartalmazza az összes lehetséges maradékot.

- Ha $c \neq 0$, akkor az alábbi tétel [Hull, Dobell, 1962] segítségével elérhetjük, hogy az (1)-es rekurzió által felépített egész számsorozat periódusa maximális – azaz m-mel megegyező – legyen.

Tétel (Maximális periódus biztosítása, ha $c > 0$)

Az (1)-es rekurzió által generált egész számsorozat periódusa akkor és csakis akkor egyezik meg az m osztó értékével, ha teljesül az alábbi három feltétel:

1. *a $c \neq 0$ növekmény és az m osztó relatív prím;*
2. $a \equiv 1 \pmod p$, *ha p az m szám prímosztója;*
3. $a \equiv 1 \pmod 4$, *ha az m osztó négynek többszöröse.*

- A fenti tétel értelmében, ha az m osztó 2-nek hatványa (ami természetes egy bináris számrendszerre épülő architektúra esetén), akkor a maximális periódus biztosításához elégséges, ha a c növekmény páratlan és $a \equiv 1 (\bmod 4)$.

Értelmezés (Primitív gyök modulo m)

Az $m = p_1^{\alpha_1} p_2^{\alpha_2} \ldots p_r^{\alpha_r} > 1$ prímtényezős felbontású természetes szám esetén az olyan g számot nevezzük „g primitív gyök modulo m"-nek, amelyre a

$$g, g^2, \ldots, g^{\varphi(m)}$$

hatványok különböző maradékot adnak m-mel osztva, azaz a g rendje modulo m pontosan $\varphi(m)$, ahol

$$\varphi(m) = m \prod_{i=1}^{r} \left(1 - \frac{1}{p_i}\right)$$

az Euler-féle φ-függvényt jelöli.

- A $c = 0$ esetben a [Carmichael, 1910] cikkben közölt és alább kijelentett tételt alkalmazhatjuk a maximális periódus biztosítására.

Tétel (Maximális periódus biztosítása, ha $c = 0$)

Ha az (1)-es képletbeli c növekmény nulla, az X_1 és m számok relatív prímek, valamint a primitív gyök modulo m, akkor a rekurzió által generált egész számsorozat periódusa maximális és ez a maximum megegyezik a

$$\lambda(m) = \begin{cases} 1, & m = 2, \\ 2, & m = 2^2 = 4, \\ 2^{e-2}, & m = 2^e,\ e \geq 3, \\ p^{e-1}(p-1), & m = p^e,\ p > 2, \\ \text{lkkt}\left\{\lambda\left(p_1^{\alpha_1}\right), \lambda\left(p_2^{\alpha_2}\right), \ldots, \lambda\left(p_r^{\alpha_r}\right)\right\}, & m = p_1^{\alpha_1} p_2^{\alpha_2} \ldots p_r^{\alpha_r} \end{cases}$$

számmal.

- Vegyük észre, hogy a fenti tétel értelmében, ha m prím, akkor $m-1$ periódusú számsorozatot generálhatunk az (1)-es rekurzióval!

Példák

A továbbiakban az (1)-es rekurzió néhány olyan paraméterezését ismertetjük, amelyek által szült egyenletes eloszlású számgenerátorok „véletlenszerűséget" ellenőrző, szigorú statisztikai teszteken is átmentek:

- URNG1 generátor: $m = 2^{31} - 1$ (Mersenne-féle prím), $a = 7^5$, $c = 0$, míg $x_1 \geq 1$ tetszőlegesen rögzített természetes szám;
- URNG2 generátor: $m = 2^{31} - 1$, $a = 2^{16} + 3$, $c = 0$, míg $x_1 = 2k + 1$ alakú tetszőlegesen rögzített természetes szám.

1. Kódrészlet. Multiplikatív lineáris kongruencia módszere

```
%----------------
% Description
%----------------
% The function implements the linear congruential[1] generator X_{i+1} = (aX_i + c) mod m, i ≥ 2.
%
%--------
% Input
%--------
% m                   - the modulus[2] m > 0
% a                   - the multiplier[3] a ∈ {0, 1, ..., m − 1}
% c                   - the increment[4] c ∈ {0, 1, ..., m − 1}
% initial_value       - an integer that represents the first element of the
%                       generated sequence (0 < intial_value < m)
% n                   - the size of the output sequence
%
%---------
% Output
%---------
% X = [X_i]_{i=1}^n   - an array of uniformly distributed integer random numbers[5]
% new_initial_value - an integer that can be used as an initial value in case
%                       of consecutive random sequence generations
function [X, new_initial_value] = LinearCongruentialGenerator(m, a, c, initial_value, n)

X = zeros(1, n);
X(1) = initial_value;

for i = 2:n
    X(i) = mod(a * X(i-1) + c, m);
end

```

```
new_initial_value = mod(a * X(n) + c, m);
```

[1]**congruence** *kongruencia*; **(combined) multiplicative linear** ∼ *(összetett) multiplikatív lineáris kongruencia*
[2]**modulus** *osztó*
[3]**multiplier** *szorzó*
[4]**increment** *növekedmény*
[5]**number** *szám*; **(pseudo)random** ∼ *(ál) véletlen szám*; **random** ∼ **generator** *véletlenszám-generátor*

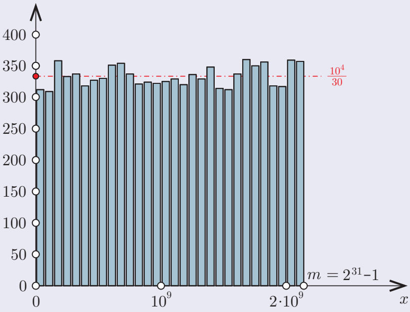
2. Kódrészlet. Egyenletes eloszlású egész véletlenszám-generátor

```
% ------------
% Description
% ------------
% By means of parameters m = 2^31 - 1, a = 7^5 and c = 0, the function implements a specialized
% version of the linear congruential generator X_{i+1} = (aX_i + c) mod m, i >= 2.
% ------
% Input
% ------
% initial_value     - an integer that must be >= 1 and < 2^31 - 1 and which represents
%                     the first element of the output sequence
% n                 - the size of the output sequence
% ------
% Output
% ------
% X = [X_i]_{i=1}^n - a sequence of uniformly distributed integer random numbers
% new_initial_value - an integer that can be used as an initial value in
%                     case of consecutive random sequence generations
function [X, new_initial_value] = URNG1(initial_value, n)

m = 2^31-1;
a = 7^5;
c = 0;
[X, new_initial_value] = LinearCongruentialGenerator(m, a, c, initial_value, n);
```

1. feladat

Az URNG1 egyenletes eloszlású véletlenszám-generátor 2. kódrészéletbeli implementálásához hasonlóan írjatok egy függvényt az URNG2 generátorra is! A hist parancsot használva készítsetek abszolút gyakoriságdiagramot (másképpen hisztogramot) az új függvény által visszatérített sorozatokról is!

Megjegyzés



1. ábra. $X_1 = 2907$ kezdőértékkel inicializált, URNG1 generátorral előállított 10000 elemű, egyenletes eloszlású véletlen természetes számokat tartalmazó tömb hisztograma.

- Vegyük észre, hogy a

```
seed = 2907;
count = 10000;
[sequence, seed] = URNG1(seed, count);

bin_count = 30;
[frequency, bins] = hist(sequence, bin_count);

bar(bins, frequency);
```

  parancsokkal létrehozott 1. ábrán az egyes részintervallumok abszolút gyakorisága (azaz az oszlopok magassága) közel megegyező!
- Vizuálisan ez jelzi azt, hogy a generált számok egyenletes eloszlásúak (az egyes cellák körülbelül ugyanannyiszor fordulnak elő, ha elég nagy mintavételezést végzünk).

- Egy $[\alpha, \beta] \subset \mathbb{R}$ intervallumba eső egyenletes eloszlású véletlen valós számokból álló sorozatot egy alkalmas lineáris transzformációval állíthatunk elő, ahogy azt a 3. kódrészlet is mutatja.

3. Kódrészlet. Adott intervallumon egyenletes eloszlású valós véletlen számok generálása

```
%------------
% Description
%------------
% The function generates uniformly distributed real random numbers in the range [$\alpha$, $\beta$],
% where $\alpha < \beta$.
%
%------
% Input
%------
% intial_value      - an integer which is required by the uniform integer
%                     random number generator (0 < initial_value < m)
% generator_type    - specifies the type of the used uniform integer radom
%                     number generator, that can be set as 'URNG1'/'URNG2' or 1/2
% [alpha, beta]     - all elements of the output sequence Y will fall in this range
% n                 - the size of the output sequence
%
%-------
% Output
%-------
% $\mathbf{Y} = [Y_i]_{i=1}^{n}$       - $\mathbf{Y} \sim \mathcal{U}([\alpha, \beta])$
% new_initial_value - an integer that can be used as an initial value in
%                     case of consecutive random sequence generations
%
function [Y, new_initial_value] = URealRNG(
        initial_value, generator_type, alpha, beta, n)

switch (generator_type)
    case {'URNG1', 1}
```

```
        m = 2^31 - 1;
        [Y, new_initial_value] = URNG1(initial_value, n);
        Y = Y ./ (m - 1);

    % handle another integer uniform random number generator
    ...
end

Y = alpha + Y .* (beta - alpha);
```

2. feladat

Fejezzétek be és teszteljétek is a 3. kódrészletbeli URealRNG függvényt!

- Az (1)-es rekurzió különböző beállításaiból származó multiplikatív lineáris kongruenciákat ötvözhetjük is egyenletes eloszlású számgenerátorok kialakítására.
- Az idevágó elméleti hátteret mellőzzük, viszont megemlítjük, hogy Pierre L'Ecuyer az

$$\left\{ \begin{array}{rcl} X_{1,i} & = & (40692 \cdot X_{1,i-1}) \pmod{2147483399}, \\ X_{2,i} & = & (40014 \cdot X_{2,i-1}) \pmod{2147483563}, \\ X_i & = & (X_{1,i} + X_{2,i} - 2) \pmod{2147483562}, \\ U_i & = & \dfrac{X_i + 1}{2147483563} \end{array} \right. \quad (2)$$

  összetett multiplikatív lineáris kongruenciákra épülő módszert ajánlotta a $(0, 1)$ intervallumon egyenletes eloszlású valós számok generálására [L'Ecuyer, 1986], ahol $i \geq 2$, illetve $X_{1,1}$ és $X_{2,1}$ alkalmasan megválasztott kezdőértékek.
- A (2)-es képletekbeli lineáris kongruenciák periódusa rendre $p_1 = 2147483398$, illetve $p_2 = 2147483562$. Az összetett számgenerátor p periódusát a p_1 és p_2 számok legkisebb közös többszöröseként kaphatjuk meg, azaz

$$p = \frac{p_1 \cdot p_2}{2} = 2,305842648436451838 \cdot 10^{18},$$

  ami tökéletesen megfelel a céljainknak.

4. Kódrészlet. L'Ecuyer egyenletes eloszlású számgenerátora

```
%------------
% Description
%------------
% The function generates continuous uniform pseudorandom numbers on the interval (0,1).
% Its implementation is based on the article [L'Ecuyer, 1986].
%
%-----------------
% Usage and syntax
%-----------------
%   scalar     = ULEcuyerRNG
%   row_matrix = ULEcuyerRNG(column_count)
%   matrix     = ULEcuyerRNG(row_count, column_count)
%
function result = ULEcuyerRNG(varargin)

% checking the possible input parameters
optional_input_argument_count = size(varargin,2);

if (optional_input_argument_count == 0)
    row_count = 1;
    column_count = 1;
else
    if (optional_input_argument_count == 1)
        row_count = 1;

        if (isscalar(varargin{1}))
            column_count = round(varargin{1});
        else
            error('Wrong_column_number!');
        end
    else
        if (optional_input_argument_count == 2)
```

```
            if (isscalar(varargin{1}))
                row_count = round(varargin{1});
            else
                error('Wrong row number!');
            end

            if (isscalar(varargin{2}))
                column_count = round(varargin{2});
            else
                error('Wrong column number!');
            end
        else
            error('Too many input arguments!');
        end
    end
end

% Implementation of the combined multiplicative linear congruential generator [L'Ecuyer, 1986]
% for U((0,1)) random sampling:
persistent seed1 seed2

if (isempty(seed1))
    seed1 = 55555;
end

if (isempty(seed2))
    seed2 = 99999;
end

persistent factor

if (isempty(factor))
    factor = 1.0/2147483563.0;
end

result = zeros(row_count, column_count);
```

```
for i = 1:row_count
    for j = 1:column_count

        k = seed1 / 53668;
        seed1 = 40014 * mod(seed1, 53668) - k * 12211;

        if (seed1 < 0)
            seed1 = seed1 + 2147483563;
        end

        k = seed2 / 52774;

        seed2 = 40692 * mod(seed2, 52774) - k * 3791;

        if (seed2 < 0)
            seed2 = seed2 + 2147483399;
        end

        z = (seed1 - 2147483563) + seed2;

        if (z < 1)
            z = z + 2147483562;
        end

        result(i, j) = z * factor;
    end
end
```

Megjegyzés

- Vegyük észre a persistent kulcsszó használatát! A persistent típusú változók a globálisakhoz hasonlóan viselkednek (azaz értékeik permanens memóriaterületen tárolódnak el), azzal a különbséggel, hogy csak a változókat deklaráló függvény számára elérhetőek. Ezért értéküket nem változtathatjuk meg más függvényekben, vagy a MATLAB® parancssorából. Ezzel az apró kis trükkel megszabadulhatunk az ULEcuyerRNG egyenletes eloszlású számgenerátor által várt kezdőértékek folytonos megadásától, hiszen sorozatos függvényhívások esetén a már módosult kezdőértékek lépnek életbe.
- A varargin, isscalar kulcsszavak alkalmazásával az ULEcuyerRNG függvényt egyaránt használhatjuk $(0, 1)$ intervallumon egyenletes eloszlású skalárok, sormátrixok, illetve tetszőleges méretű mátrixok generálására.

- Ha mégsem lennénk megelégedve a 4. kódrészletben ismertetett L'Ecuyer-féle egyenletes eloszlású számgenerátorral, akkor helyette – a [Matsumoto, Nishimura, 1998] cikkre alapozva – a Mersenne-twister számgenerátort ajánlhatjuk, amely szintén 32-bites architektúrát feltételez, és számos szigorú, véletlenszerűséget ellenőrző statisztikai (pl. Diehard-típusú [Marsaglia, 1985]) tesztet kiállt. A módszer a nevét a

$$2^{19937} - 1 = 4,315424797388162648055235516337\,9 \cdot 10^{6001}$$

kolosszális nagyságú periódusát meghatározó Mersenne-féle prímszámról kapta.

- Mejegyezzük, hogy a lent kódolt módszer a TestU01-típusú [L'Ecuyer, Simard, 2007] véletlenszerűséget tesztelő statisztikák zöme alapján is kiváló egyenletes eloszlású számgenerátornak bizonyult.
- Ha az 5. kódrészletbeli implementáció nem is teljesen optimális, tanítópédának mindenképpen alkalmas, mert hűen követi az ▸ algoritmus pszeudokódját. Az algoritmus helyességének és komplexitásának tanulmányozása nem tartozik a jelen kézirat célkitűzései közé.
- Az 5. kódrészletbeli UMersenneTwisterRNG függényt egyaránt használhatjuk $(0, 1)$ intervallumon egyenletes eloszlású skalárok, sormátrixok, illetve tetszőleges méretű mátrixok generálására.

5. Kódrészlet. 32-bites Mersenne-twister egyenletes eloszlású számgenerátor

```
%————————
% Description
%————————
% The function generates continuous uniform pseudorandom numbers on the interval (0,1).
% Its implementation is based on the article [Matsumoto, Nishimura, 1998].
%
%————————————
% Usage and syntax
%————————————
%   scalar     = UMersenneTwisterRNG
%   row_matrix = UMersenneTwisterRNG(column_count)
%   matrix     = UMersenneTwisterRNG(row_count, column_count)
%
function result = UMersenneTwisterRNG(varargin)

    persistent MT
    persistent index

    % initialize the generator from a seed
    function initialize_generator(seed)

        index = 0;

        MT = zeros(1, 624);
        MT(1) = uint64(seed);

        for k = 1:623
            MT(k+1) = bitand(...
                uint64(1812433253 * bitxor(MT(k), bitshift(MT(k), -30)) + k), ...
                4294967295);
        end

```

Implementálás – II

```
end

% generate an array of 624 untempered numbers
function generate_numbers()

    for k = 0:623
        y = bitand(MT(k + 1), 2147483648) + ...
            bitand(2147483647, MT( mod(k + 1, 624) + 1));

        MT(k + 1) = bitxor(MT(mod(k + 397, 624) + 1), bitshift(uint32(y), -1));

        if (mod(y, 2) ~= 0)
            MT(k + 1) = bitxor(MT(k + 1), 2567483615);
        end
    end

end

%  extract a tempered pseudorandom number based on the index-th value,
%  calling generate_numbers() every 624 numbers
function y = extract_number()

    if (index == 0)
        generate_numbers();
    end

    y = MT(index + 1);
    y = bitxor(y, bitshift(y, -11));
    y = bitxor(y, bitand(bitshift(y, 7), 2636928640));
    y = bitxor(y, bitand(bitshift(y, 15), 4022730752));
    y = bitxor(y, bitshift(y, -18));

    index = mod(index + 1, 624);

end
```

Implementálás – III

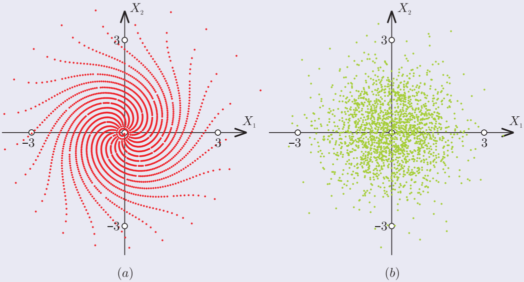
Mersenne-twister $\mathcal{U}((0,1))$-eloszlású számgenerátor

```
% checking the possible input parameters
optional_input_argument_count = size(varargin,2);

if (optional_input_argument_count == 0)
    row_count = 1;
    column_count = 1;
else
    if (optional_input_argument_count == 1)
        row_count = 1;
        if (isscalar(varargin{1}))
            column_count = round(varargin{1});
        else
            error('Wrong column number!');
        end
    else
        if (optional_input_argument_count == 2)
            if (isscalar(varargin{1}))
                row_count = round(varargin{1});
            else
                error('Wrong row number!');
            end

            if (isscalar(varargin{2}))
                column_count = round(varargin{2});
            else
                error('Wrong column number!');
            end
        else
            error('Too many input arguments!');
        end
    end
end

% we use a default initial value that is equal to the prime number 6199
if (isempty(MT) && isempty(index))
    initialize_generator(6199);
```

```
    end

    % allocate memory for the output sequence of uniform
    result = zeros(row_count, column_count);

    % generate random uniform integers and normalize them
    for i = 1:row_count
        for j = 1:column_count
            result(i, j) = extract_number() / 4294967295;
        end
    end

end
```

Megjegyzés

- Nemlineáris transzformációkra épülő és nemegyenletes eloszlású valószínűségi változókat mintavételező algoritmusok erősen függnek az általuk alkalmazott egyenletes eloszlású számgenerátortól is. Olyan egyenletes eloszlású valószínűségi változókat kell generálnunk, amelyek transzformálása során kapott valószínűségi változók is kellőképpen lefedik az eseményteret úgy, hogy semmilyen szabályosságot, ismétlődést, vagy mintát nem észlelünk az így nyert mintavételezésben.



2. ábra. Mindkét ábrán a Box–Muller-transzformáció alkalmazása során nyert 2000 darab, komponensekben független és standard normális eloszlású valószínűségi vektort láthatunk. Az (a) esetben a $v_i = 97v_{i-1} \left(\bmod 2^{17}\right)$, $v_1 = 1$ ($i \geq 2$) multiplikatív lineáris kongruencia által szült természetes számok normalizált értékeire, a (b) esetben pedig a 4. kódrészletben implementált L'Ecuyer-féle összetett multiplikatív lineáris kongruenciákra épülő számgenerátor által eredményezett $(0, 1)$ intervallumbeli, egyenletes eloszlású, véletlen számokra alkalmaztuk a transzformációt. Látható, hogy az (a) esethez tartozó egyenletes eloszlású számgenerátor nem alkalmas véletlenszerű folyamatok modellezésére.

Elfogadási feltételek

- Az alábbi feladatokat egyrészt elméleti úton, másrészt valamilyen programozási nyelvben, vagy matematikai szoftvercsomagban írt – a teljes eseményteret meghatározó elemek nagyszámú mintavételezésén alapuló – szimulációval is kötelező megoldani!
- Nyilván a szimuláción alapuló megoldások csak közelíteni fogják az elméleti úton meghatározott valószínűségi értékeket.
- Ahol csak lehetséges, ábrázoljátok grafikusan a vizsgált eseményhez tartozó teljes eseményteret, illetve a kedvező elemi események halmazát! Ugyanakkor animáljátok is a szimuláció egyes lépéseihez tartozó jelenségeket, valamint a közelített számértékeket! Ha egy feladat több paramétertől is függ, akkor ezeket soroljátok fel a szimulációt meghívó függvények paraméterlistájában!
- A feladatokat személyesen kell bemutatni a megfelelő gyakorlati órán! Ha az ellenőrzés során kiderül, hogy egy adott feladat nincs megoldva, vagy annak megoldása hibás, vagy esetlegesen másolt, akkor az adott feladat továbbra is házi feladat marad, továbbá egy újabb feladatot is meg kell oldani a mulasztás törlesztéséért.

3. feladat

Egy dobozban összesen húsz játékautó van, amelyek 25%-a kamion, 40%-a versenyautó, a maradék pedig tűzoltóautó. Visszatevéssel kiválasztva a dobozból egy tucat autót, mekkora a valószínűsége annak, hogy a kiválasztottak között:

a) pontosan három kamion és öt versenyautó lesz?

b) legalább egy versenyautó és pontosan három tűzoltóautó lesz?

Mekkora valószínűséggel következnek be az előző események ha az autókat nem visszatevéssel, hanem visszatevés nélkül választjuk ki a dobozból?

4. feladat

Rajzoljuk fel az $A(-3,6)$, $B(4,6)$, $C(8,0)$, $D(4,-4)$, $E(-4,-2)$ és $F(-8,3)$ csúcsokkal rendelkező $ABCDEF$ hatszöget, majd vegyük fel a BC és CD oldalak G és H felezőpontját, végül a hatszög belsejében rajzoljuk fel az $I(0,2)$ középpontú és négy egységnyi sugarú kört. Mekkora annak a valószínűsége, hogy az $ABCDEF$ hatszög belsejében véletlenszerűen felvett egyenletes eloszlású pont:

a) a $DFGH$ négyszög belsejében helyezkedik el?

b) a BE szakasz felett és a körön kívül helyezkedik el?

R.D. Carmichael, **1910**.
Note on a new number theory function,
Bulletin of the American Mathematical Society, **16**(5):232–238.

T.E. Hull, A.R. Dobell, **1962**.
Random number generators,
SIAM Review, **4**(3):230–254.

P. L'Ecuyer, **1986**.
Efficient and portable 32-bit random variate generators.
In Proceedings of the 1986 Winter Simulation Conference (J. Wilson, J. Henriksen, S. Roberts, eds.), 275–277.

P. L'Ecuyer, R. Simard, **2007**.
TestU01: A C library for empirical testing of random number generators,
ACM Transactions on Mathematical Software, **33**(4):Article 22.

G. Marsaglia, **1985**.
A current view of random numbers.
In Computer Science and Statistics, Proceedings of the Sixteenth Symposium on The Interface (L. Billard, ed.), North-Holland, Amsterdam, 3–10.

M. Matsumoto and T. Nishimura, **1998**.
Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator,
ACM Transactions on Modeling and Computer Simulation, **8**(1):3–30.