Mersenne twister

From Wikipedia, the free encyclopedia

The **Mersenne Twister** is a pseudorandom number generator (PRNG). It is by far the most widely used PRNG.[1] Its name derives from the fact that its period length is chosen to be a Mersenne prime.

The Mersenne Twister was developed in 1997 by Makoto Matsumoto (松本 眞) and Takuji Nishimura (西村 拓士).[2] It was designed specifically to rectify most of the flaws found in older PRNGs. It was the first PRNG to provide fast generation of high-quality pseudorandom integers.

The most commonly-used version of the Mersenne Twister algorithm is based on the Mersenne prime $2^{19937}-1$. The standard implementation of that, MT19937, uses a 32-bit word length. There is another implementation that uses a 64-bit word length, MT19937-64; it generates a different sequence.

Contents

Adoption in software systems

The Mersenne Twister is the default PRNG for R,[3] Python,[4][5] Ruby,[6] IDL,[7] Free Pascal,[8] PHP,[9] Maple,[10] MATLAB,[11] GAUSS,[12] Julia,[13] CMU Common Lisp,[14] Microsoft Visual C++,[15] the GNU Multiple Precision Arithmetic Library,[16] and the GNU Scientific Library.[17] It is also available in C++[18][19] since C++11. Add-on implementations are provided by the Boost C++ Libraries,[20] GLib,[21] and the NAG Numerical Library.[22]

The Mersenne Twister is one of two PRNGs in SPSS: the other generator is kept only for compatibility with older programs, and the Mersenne Twister is stated to be "more reliable".[23] The Mersenne Twister is similarly one of the PRNGs in SAS: the other generators are older and deprecated.[24]

Advantages

The commonly-used version of Mersenne Twister, MT19937, which produces a sequence of 32-bit integers, has the following desirable properties:

1. It has a very long period of $2^{19937} - 1$. While a long period is not a guarantee of quality in a random number generator, short periods (such as the 2^{32} common in many older software packages) can be problematic.[25]
2. It is k-distributed to 32-bit accuracy for every $1 \leq k \leq 623$ (see definition below).
3. It passes numerous tests for statistical randomness, including the Diehard tests.

Disadvantages

The state space is very large and may needlessly stress the CPU cache (a period above 2^{512} is enough for any application[26]). In 2011, Saito & Matsumoto proposed a version of the Mersenne Twister to address this issue. The tiny version, TinyMT, uses just 127 bits of state space.[27]

By today's standards, the Mersenne Twister is fairly slow, unless you use the SFMT implementation (see section below).

It passes most, but not all, of the stringent TestU01 randomness tests.[28]

It can take a long time to start generating output that passes randomness tests, if the initial state is highly non-random—particularly if the initial state has many zeros. A consequence of this is that two instances of the generator, started with initial states that are almost the same, will usually output nearly the same sequence for many iterations, before eventually diverging. The 2002 update to the MT algorithm has improved initialization, so that reaching such a state is very unlikely.[29]

k-distribution

A pseudorandom sequence x_i of w-bit integers of period P is said to be k-distributed to v-bit accuracy if the following holds.

Let $trunc_v(x)$ denote the number formed by the leading v bits of x, and consider *P* of the *kv*-bit vectors

$$(\text{trunc}_v(x_i),\ \text{trunc}_v(x_{i+1}),\ \ldots,\ \text{trunc}_v(x_{i+k-1})) \quad (0 \le i < P).$$

Then each of the 2^{kv} possible combinations of bits occurs the same number of times in a period, except for the all-zero combination that occurs once less often.

Alternatives

The algorithm in its native form is not cryptographically secure. The reason is that observing a sufficient number of iterations (624 in the case of MT19937, since this is the size of the state vector from which future iterations are produced) allows one to predict all future iterations.

A pair of cryptographic stream ciphers based on output from the Mersenne Twister has been proposed by Matsumoto, Nishimura, and co-authors. The authors claim speeds 1.5 to 2 times faster than Advanced Encryption Standard in counter mode.[30]

An alternative generator, WELL ("Well Equidistributed Long-period Linear"), offers quicker recovery, and equal randomness, and nearly-equal speed.[31] Marsaglia's xorshift generators and variants are the fastest in this class.[32]

Algorithmic detail

For a *k*-bit word length, the Mersenne Twister generates integers in the range $[0, 2^k-2]$.

The Mersenne Twister algorithm is based on a matrix linear recurrence over a finite binary field F_2. The algorithm is a twisted generalised feedback shift register[33] (twisted GFSR, or TGFSR) of rational normal form (TGFSR(R)), with state bit reflection and tempering. It is characterized by the following quantities:

- *w*: word size (in number of bits)
- *n*: degree of recurrence
- *m*: middle word, or the number of parallel sequences, $1 \le m \le n$
- *r*: separation point of one word, or the number of bits of the lower bitmask, $0 \le r \le w - 1$
- ***a***: coefficients of the rational normal form twist matrix
- ***b***, ***c***: TGFSR(R) tempering bitmasks
- *s*, *t*: TGFSR(R) tempering bit shifts
- *u*, *l*: additional Mersenne Twister tempering bit shifts

with the restriction that $2^{nw - r} - 1$ is a Mersenne prime. This choice simplifies the primitivity test and *k*-distribution test that are needed in the parameter search.

For a word ***x*** with *w* bit width, it is expressed as the recurrence relation

$$x_{k+n} := x_{k+m} \oplus ({x_k}^u \mid {x_{k+1}}^l)A \qquad\qquad k = 0, 1, \ldots$$

with | as the bitwise or and $\oplus$ as the bitwise exclusive or (XOR), $\boldsymbol{x}^u$, $\boldsymbol{x}^l$ being ***x*** with upper and lower bitmasks applied. The twist transformation *A* is defined in rational normal form

$$A = R = \begin{pmatrix} 0 & I_{w-1} \\ a_{w-1} & (a_{w-2}, \ldots, a_0) \end{pmatrix}$$

with I_{n-1} as the $(n - 1) \times (n - 1)$ identity matrix (and in contrast to normal matrix multiplication, bitwise XOR replaces addition). The rational normal form has the benefit that it can be efficiently expressed as

$$\boldsymbol{x}A = \begin{cases} \boldsymbol{x} \gg 1 & x_0 = 0 \\ (\boldsymbol{x} \gg 1) \oplus \boldsymbol{a} & x_0 = 1 \end{cases}$$

where

$$\boldsymbol{x} := ({x_k}^u \mid {x_{k+1}}^l) \qquad\qquad k = 0, 1, \ldots$$

In order to achieve the $2^{nw - r} - 1$ theoretical upper limit of the period in a TGFSR, $\varphi_B(t)$ must be a primitive polynomial, $\varphi_B(t)$ being the characteristic polynomial of

$$B = \begin{pmatrix} 0 & I_w & \cdots & 0 & 0 \\ \vdots & & & & \\ I_w & \vdots & \ddots & \vdots & \vdots \\ \vdots & & & & \\ 0 & 0 & \cdots & I_w & 0 \\ 0 & 0 & \cdots & 0 & I_{w-r} \\ S & 0 & \cdots & 0 & 0 \end{pmatrix} \begin{matrix} \\ \\ \leftarrow m\text{-th row} \\ \\ \\ \\ \\ \end{matrix}$$

$$S = \begin{pmatrix} 0 & I_r \\ I_{w-r} & 0 \end{pmatrix} A$$

The twist transformation improves the classical GFSR with the following key properties:

- Period reaches the theoretical upper limit $2^{nw - r} - 1$ (except if

initialized with 0)
- Equidistribution in *n* dimensions (e.g. linear congruential generators can at best manage reasonable distribution in 5 dimensions)

As like TGFSR(R), the Mersenne Twister is cascaded with a tempering transform to compensate for the reduced dimensionality of equidistribution (because of the choice of *A* being in the rational normal form), which is equivalent to the transformation $A = R \rightarrow A = T^{-1}RT$, *T* invertible. The tempering is defined in the case of Mersenne Twister as

$$\mathbf{y} := \mathbf{x} \oplus (\mathbf{x} >> u)$$
$$\mathbf{y} := :\mathbf{y} \oplus ((\mathbf{y} << s) \ \& \ \mathbf{b})$$
$$\mathbf{y} := :\mathbf{y} \oplus ((\mathbf{y} << t) \ \& \ \mathbf{c})$$
$$\mathbf{z} := \mathbf{y} \oplus (\mathbf{y} >> l)$$

with <<, >> as the bitwise left and right shifts, and & as the bitwise and. The first and last transforms are added in order to improve lower bit equidistribution. From the property of TGFSR, $s + t \geq \lfloor w/2 \rfloor - 1$ is required to reach the upper bound of equidistribution for the upper bits.

The coefficients for MT19937 are:

- $(w, n, m, r) = (32, 624, 397, 31)$
- $\mathbf{a} = \mathrm{9908B0DF}_{16}$
- $u = 11$
- $(s, \mathbf{b}) = (7, \mathrm{9D2C5680}_{16})$
- $(t, \mathbf{c}) = (15, \mathrm{EFC60000}_{16})$
- $l = 18$

A small lagged Fibonacci generator or linear congruential generator usually is used to seed the Mersenne Twister with random initial values.

Pseudocode

The following piece of pseudocode generates uniformly distributed 32-bit integers in the range $[0, 2^{32} - 1]$ with the MT19937 algorithm:

```
// Create a length 624 array to store the state of the generator
int[0..623] MT
int index = 0

// Initialize the generator from a seed
function initialize_generator(int seed) {
    index := 0
    MT[0] := seed
    for i from 1 to 623 { // loop over each element
        MT[i] := lowest 32 bits of(1812433253 * (MT[i-1] xor (right shift by 30 bits(MT[i-1]))))
    }
}

// Extract a tempered pseudorandom number based on the index-th value,
// calling generate_numbers() every 624 numbers
function extract_number() {
    if index == 0 {
        generate_numbers()
    }

    int y := MT[index]
    y := y xor (right shift by 11 bits(y))
    y := y xor (left shift by 7 bits(y) and (2636928640)) // 0x9d2c5680
    y := y xor (left shift by 15 bits(y) and (4022730752)) // 0xefc60000
    y := y xor (right shift by 18 bits(y))

    index := (index + 1) mod 624
    return y
}

// Generate an array of 624 untempered numbers
function generate_numbers() {
    for i from 0 to 623 {
        int y := (MT[i] and 0x80000000)                       // bit 31 (32nd bit) of MT[i]
                       + (MT[(i+1) mod 624] and 0x7fffffff)   // bits 0-30 (first 31 bits) of M
        MT[i] := MT[(i + 397) mod 624] xor (right shift by 1 bit(y))
        if (y mod 2) != 0 { // y is odd
            MT[i] := MT[i] xor (2567483615) // 0x9908b0df
        }
    }
}
```

SFMT

SFMT, the Single instruction, multiple data-oriented Fast Mersenne Twister, is a variant of Mersenne Twister, introduced in 2006,[34] designed to be fast when it runs on 128-bit SIMD.

- It is roughly twice as fast as Mersenne Twister.[35]
- It has a better equidistribution property of v-bit accuracy than MT but worse than WELL ("Well Equidistributed Long-period Linear").
- It has quicker recovery from zero-excess initial state than MT, but slower than WELL.
- It supports various periods from 2^{607}−1 to 2^{216091}−1.

Intel SSE2 and PowerPC AltiVec are supported by SFMT. It is also used for games with the Cell BE in the PlayStation 3.[36]

MTGP

MTGP is a variant of Mersenne Twister optimised for graphics processing units published by Mutsuo Saito and Makoto Matsumoto.[37] The basic linear recurrence operations are extended from MT and parameters are chosen to allow many threads to compute the recursion in parallel, while sharing their state space to reduce memory load. The paper claims improved equidistribution over MT and performance on a high specification GPU (Nvidia GTX260 with 192 cores) of 4.7ms for 5x10^7 random 32-bit integers.

References

1. ^ E.g. Marsland S. (2011) *Machine Learning* (CRC Press), §4.1.1. Also see the section "Adoption in software systems".
2. ^ Matsumoto, M.; Nishimura, T. (1998). "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator". *ACM Transactions on Modeling and Computer Simulation* **8** (1): 3–30. doi:10.1145/272991.272995 (http://dx.doi.org/10.1145%2F272991.272995).
3. ^ "Random Number Generators" (http://cran.r-project.org/web/views/Distributions.html). *CRAN Task View: Probability Distributions*. Retrieved 2012-05-29.
4. ^ "9.6 random — Generate pseudo-random numbers" (https://docs.python.org/release/2.6.8/library/random.html). *Python v2.6.8 documentation*. Retrieved 2012-05-29.
5. ^ "8.6 random — Generate pseudo-random numbers" (https://docs.python.org/release/3.2/library/random.html). *Python v3.2 documentation*. Retrieved 2012-05-29.
6. ^ " "Random" class documentation" (http://www.ruby-doc.org/core-1.9.3/Random.html). *Ruby 1.9.3 documentation*. Retrieved 2012-05-29.
7. ^ "RANDOMU (IDL Reference)" (http://www.exelisvis.com/docs/RANDOMU.html). *Exelis VIS Docs Center*. Retrieved 2013-08-23.
8. ^ "random" (http://www.freepascal.org/docs-html/rtl/system/random.html). *free pascal documentation*. Retrieved 2013-11-28.
9. ^ "mt_srand" (http://php.net/manual/en/function.mt-srand.php). *php documentation*. Retrieved 2012-05-29.
10. ^ "random number generator" (http://www.maplesoft.com/support/help/Maple/view.aspx?path=rand). *Maple Online Help*. Retrieved 2013-11-21.
11. ^ Random number generator algorithms (http://www.mathworks.co.uk/help/matlab/ref/randstream.list.html) — Documentation Center, MathWorks
12. ^ GAUSS 14 Language Reference (http://www.aptech.com/wp-content/uploads/2014/01/GAUSS14_LR.pdf)
13. ^ Julia Language Documentation — The Standard Library (http://julia.readthedocs.org/en/latest/stdlib/base/#random-numbers)
14. ^ "Design choices and extensions" (http://common-lisp.net/project/cmucl/doc/cmu-user/extensions.html). *CMUCL User's Manual*.

Retrieved 2014-02-03.

15. ^ <random> (http://msdn.microsoft.com/en-us/library/bb982398.aspx) —Microsoft Developer Network
16. ^ "Randum Number Algorithms" (http://gmplib.org/manual/Random-Number-Algorithms.html). *GNU MP*. Retrieved 2013-11-21.
17. ^ "Random number environment variables" (http://www.gnu.org/software/gsl/manual/html_node/Random-number-environment-variables.html). *GNU Scientific Library*. Retrieved 2013-11-24.
18. ^ Random Number Generation in C++11 (https://isocpp.org/files/papers/n3551.pdf) —Standard C++ Foundation
19. ^ "std::mersenne_twister_engine" (http://en.cppreference.com/w/cpp/numeric/random/mersenne_twister_engine). *Pseudo Random Number Generation*. Retrieved 2012-09-25.
20. ^ "boost/random/mersenne_twister.hpp" (http://www.boost.org/doc/libs/1_49_0/boost/random/mersenne_twister.hpp). *Boost C++ Libraries*. Retrieved 2012-05-29.
21. ^ "Changes to GLib" (http://developer.gnome.org/glib/stable/glib-changes.html). *GLib Reference Manual*. Retrieved 2012-05-29.
22. ^ "G05 – Random Number Generators" (http://www.nag.co.uk/numeric/fl/nagdoc_fl23/xhtml/G05/g05intro.xml). *NAG Library Chapter Introduction*. Retrieved 2012-05-29.
23. ^ "Random Number Generators" (http://pic.dhe.ibm.com/infocenter/spssstat/v20r0m0/index.jsp?topic=%2Fcom.ibm.spss.statistics.help%2Fidh_seed.htm). *IBM SPSS Statistics*. Retrieved 2013-11-21.
24. ^ "Using Random-Number Functions" (http://support.sas.com/documentation/cdl/en/lrdict/64316/HTML/default/viewer.htm#a001281561.htm). *SAS Language Reference*. Retrieved 2013-11-21.
25. ^ Note: 2^{19937} is approximately 4.3×10^{6001}; this is many orders of magnitude larger than the estimated number of particles in the observable universe, which is 10^{87}.
26. ^ *Numerical Recipes*, §7.1.
27. ^ http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/TINYMT/index.html
28. ^ P. L'Ecuyer and R. Simard, "TestU01: "A C library for empirical testing of random number generators (http://www.iro.umontreal.ca/~lecuyer/myftp/papers/testu01.pdf)", *ACM Transactions on Mathematical Software*, 33, 4, Article 22 (August 2007).
29. ^ http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html
30. ^ Matsumoto, Makoto; Nishimura, Takuji; Hagita, Mariko; Saito, Mutsuo (2005). "Cryptographic Mersenne Twister and Fubuki Stream/Block Cipher" (http://eprint.iacr.org/2005/165.pdf).
31. ^ P. L'Ecuyer, "Uniform Random Number Generators", *International Encyclopedia of Statistical Science*, Lovric, Miodrag (Ed.), Springer-Verlag, 2010.
32. ^ "xorshift*/xorshift+ generators and the PRNG shootout"

(http://prng.di.unimi.it).

33. ^ Matsumoto, M.; Kurita, Y. (1992). "Twisted GFSR generators". *ACM Transactions on Modeling and Computer Simulation* **2** (3): 179–194. doi:10.1145/146382.146383 (http://dx.doi.org/10.1145%2F146382.146383).
34. ^ SIMD-oriented Fast Mersenne Twister (SFMT) (http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/index.html)
35. ^ SFMT:Comparison of speed (http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/speed.html)
36. ^ PLAYSTATION 3 License (http://www.scei.co.jp/ps3-license/index.html)
37. ^ Mutsuo Saito; Makoto Matsumoto (2010). "Variants of Mersenne Twister Suitable for Graphic Processors". arXiv:1005.4973v3 (http://arxiv.org/abs/1005.4973v3) [cs.MS (http://arxiv.org/archive/cs.MS)].

External links

- The academic paper for MT, and related articles by Makoto Matsumoto (http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/ARTICLES/earticles.html)
- Mersenne Twister home page, with codes in C, Fortran, Java, Lisp and some other languages (http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html)
- SFMT in Action (http://www.codeproject.com/KB/DLL/SFMT_dll.aspx?msg=3130186) –The Code Project