Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
Deutsch
Español
Français
한국어
Italiano
Nederlands
日本語
Polski
Русский
ไทย
中文
Edit links

Mersenne Twister

From Wikipedia, the free encyclopedia
(Redirected from Mersenne twister)

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.

Adoption in software systems [edit]

The Mersenne Twister is the default PRNG for the following software systems: R,[3] Python,[4][5] Ruby,[6] PHP,[7] CMU Common Lisp,[8] Steel Bank Common Lisp,[9] Free Pascal,[10] GLib,[11] SageMath,[12] Maple,[13] MATLAB,[14] GAUSS,[15] IDL,[16] Julia,[17] Scilab,[18] GNU Octave,[19] the GNU Scientific Library,[20] the GNU Multiple Precision Arithmetic Library,[21] and Microsoft Visual C++.[22] It is also available in standard C++ (since C++11)[23][24] and Apache.[25] Add-on implementations are provided in many program libraries, including the Boost C++ Libraries[26] and the NAG Numerical Library.[27]

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".[28] The Mersenne Twister is similarly one of the PRNGs in SAS: the other generators are older and deprecated.[29]

Advantages [edit]

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.[30]
2. It is *k*-distributed to 32-bit accuracy for every $1 \le k \le 623$ (see definition below).
3. It passes numerous tests for statistical randomness, including the Diehard tests.

Disadvantages [edit]

The state space is very large and may needlessly stress the CPU cache (a period above 2^{512} is enough for any application[31]). 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.[32]

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

It passes most, but not all, of the stringent TestU01 randomness tests.[33]

Multiple Mersenne Twister instances that differ only in seed value (but not other parameters) are not generally appropriate for Monte-Carlo simulations that require independent random number generators, though there exists a method for choosing multiple sets of parameter values.[34]

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.[35]

k-distribution [edit]

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

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

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 [edit]

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.

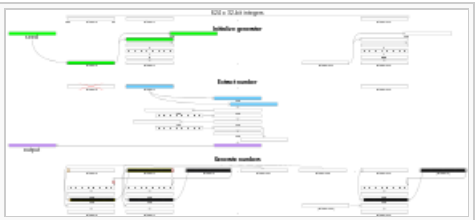
A pair of cryptographic stream ciphers based on output from the Mersenne Twister has been proposed by Matsumoto, Nishimura, and co-authors. The authors claim speeds 1.5 to 2 times faster than Advanced Encryption Standard in counter mode.[36]

An alternative generator, WELL ("Well Equidistributed Long-period Linear"), offers quicker recovery, and equal randomness, and nearly-equal speed.[37] Marsaglia's xorshift generators and variants are the fastest in this class.[38]

Algorithmic detail [edit]

For a *w*-bit word length, the Mersenne Twister generates integers in the range $[0, 2^w-1]$.

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[39] (twisted GFSR, or TGFSR) of rational normal form (TGFSR(R)), with state bit reflection and tempering. The basic idea is to define a series x_i through a simple recurrence relation, and then output numbers of the form x_iT, where T is an invertible F_2 matrix called a tempering matrix.



Visualisation of generation of pseudo-random 32-bit integers using a Mersenne Twister. The 'Extract number' section shows an example where integer 0 has already been output and the index is at integer 1. 'Generate numbers' is run when all integers have been output. (click for detail)

The general algorithm is characterized by the following quantities: (some of these explanations make sense only after reading the rest of the algorithm)

- *w*: word size (in number of bits)
- *n*: degree of recurrence
- *m*: middle word, an offset used in the recurrence relation defining the series ***x***, $1 \le m < n$
- *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*, *d*, *l*: additional Mersenne Twister tempering bit shifts/masks

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.

The series ***x*** is defined as a series of *w*-bit quantities with the recurrence relation:

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

where $\mid$ denotes the bitwise or, $\oplus$ the bitwise exclusive or (XOR), ${x_k}^u$ means the upper $w-r$ bits of x_k, and x_{k+1}^l means the lower r bits of x_{k+1}. The twist transformation *A* is defined in rational normal form as:

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

with I_{n-1} as the $(n-1) \times (n-1)$ identity matrix. The rational normal form has the benefit that multiplication by *A* can be efficiently expressed as: (remember that here matrix multiplication is being done in F_2, and therefore bitwise XOR takes the place of addition)

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

where x_0 is the lowest order bit of *x*.

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). Note that this is equivalent to using the matrix *A′* where $A' = T^{-1}AT$ for *T* an invertible matrix, and therefore the analysis of characteristic polynomial mentioned below still holds.

As with *A*, we choose a tempering transform to be easily computable, and so do not actually construct *T* itself. The tempering is defined in the case of Mersenne Twister as

y := ***x*** ⊕ ((***x*** >> *u*) & ***d***)
y := ***y*** ⊕ ((***y*** << *s*) & ***b***)
y := ***y*** ⊕ ((***y*** << *t*) & ***c***)
z := ***y*** ⊕ (***y*** >> *l*)

where ***x*** is the next value from the series, ***y*** a temporary intermediate value, ***z*** the value returned from the algorithm, 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)
- *a* = 9908B0DF$_{16}$
- (*u*, *d*) = (11, FFFFFFFF$_{16}$)
- (*s*, *b*) = (7, 9D2C5680$_{16}$)
- (*t*, *c*) = (15, EFC60000$_{16}$)
- *l* = 18

Note that 32-bit implementations of the Mersenne Twister generally have *d* = FFFFFFFF$_{16}$. As a result, the *d* is occasionally omitted from the algorithm description, since the bitwise and with *d* in that case has no effect.

The coefficients for MT19937-64 are:[40]

- (*w*, *n*, *m*, *r*) = (64, 312, 156, 31)
- *a* = B5026F5AA96619E9$_{16}$
- (*u*, *d*) = (29, 5555555555555555$_{16}$)
- (*s*, *b*) = (17, 71D67FFFEDA60000$_{16}$)
- (*t*, *c*) = (37, FFF7EEE000000000$_{16}$)
- *l* = 43

Initialization [edit]

As should be apparent from the above description, the state needed for a Mersenne Twister implementation is an array of *n* values of *w* bits each. To initialize the array, a *w*-bit seed value is use to supply x_0 through x_{n-1} by setting x_0 to the seed value and thereafter setting

$x_i = f \times (x_{i-1} \oplus (x_{i-1} >> (w\text{-}2))) + i$

for i from 1 to n-1. The first value the algorithm then generates is based on x_n. The constant f forms another parameter to the generator, though not part of the algorithm proper. The value for f for MT19937 is 1812433253 and for MT19937-64 is 6364136223846793005.[41]

Comparison with classical GFSR [edit]

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

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

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

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)

Pseudocode [edit]

The following piece of pseudocode implements the general Mersenne Twister algorithm. The constants **w**, **n**, **m**, **r**, **a**, **u**, **d**, **s**, **b**, **t**, **c**, **l**, and **f** are as in the algorithm description above. It is assumed that **int** represents a type sufficient to hold values with **w** bits:

```
// Create a length n array to store the state of the generator
int[0..n-1] MT
int index := n+1
const int lower_mask = (1 << r) - 1 // That is, the binary number of r 1's
const int upper_mask = lowest w bits of (not lower_mask)

// Initialize the generator from a seed
function seed_mt(int seed) {
    index := n
    MT[0] := seed
    for i from 1 to (n - 1) { // loop over each element
        MT[i] := lowest w bits of (f * (MT[i-1] xor (MT[i-1] >> (w-2))) + i)
    }
}

// Extract a tempered value based on MT[index]
// calling twist() every n numbers
function extract_number() {
    if index >= n {
        if index > n {
          error "Generator was never seeded"
          // Alternatively, seed with constant value; 5489 is used in reference C
code[42]
        }
        twist()
    }

    int y := MT[index]
    y := y xor ((y >> u) and d)
    y := y xor ((y << s) and b)
    y := y xor ((y << t) and c)
    y := y xor (y >> l)

    index := index + 1
    return lowest w bits of (y)
```

```
}

// Generate the next n values from the series x_i
function twist() {
    for i from 0 to (n-1) {
        int x := (MT[i] and upper_mask)
                  + (MT[(i+1) mod n] and lower_mask)
        int xA := x >> 1
        if (x mod 2) != 0 { // lowest bit of x is 1
            xA := xA xor a
        }
        MT[i] := MT[(i + m) mod n] xor xA
    }
    index := 0
}
```

Python Implementation [edit]

This python implementation hard-codes the constants for MT19937:

```
def _int32(x):
    # Get the 32 least significant bits.
    return int(0xFFFFFFFF & x)

class MT19937:

    def __init__(self, seed):
        # Initialize the index to 0
        self.index = 624
        self.mt = [0] * 624
        self.mt[0] = seed  # Initialize the initial state to the seed
        for i in range(1, 624):
            self.mt[i] = _int32(
                1812433253 * (self.mt[i - 1] ^ self.mt[i - 1] >> 30) + i)

    def extract_number(self):
        if self.index >= 624:
            self.twist()

        y = self.mt[self.index]

        # Right shift by 11 bits
        y = y ^ y >> 11
        # Shift y left by 7 and take the bitwise and of 2636928640
        y = y ^ y << 7 & 2636928640
        # Shift y left by 15 and take the bitwise and of y and 4022730752
        y = y ^ y << 15 & 4022730752
        # Right shift by 18 bits
        y = y ^ y >> 18

        self.index = self.index + 1

        return _int32(y)

    def twist(self):
        for i in range(0, 624):
            # Get the most significant bit and add it to the less significant
            # bits of the next number
            y = _int32((self.mt[i] & 0x80000000) +
                       (self.mt[(i + 1) % 624] & 0x7fffffff))
            self.mt[i] = self.mt[(i + 397) % 624] ^ y >> 1

            if y % 2 != 0:
                self.mt[i] = self.mt[i] ^ 0x9908b0df
        self.index = 0
```

SFMT [edit]

This section requires expansion.
(June 2007)

SFMT, the Single instruction, multiple data-oriented Fast Mersenne Twister, is a variant of Mersenne Twister, introduced in 2006,[43] designed to be fast when it runs on 128-bit SIMD.

- It is roughly twice as fast as Mersenne Twister.[44]
- 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.[45]

MTGP [edit]

MTGP is a variant of Mersenne Twister optimised for graphics processing units published by Mutsuo Saito and Makoto Matsumoto.[46] 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 [edit]

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.
3. ^ "Random Number Generators". *CRAN Task View: Probability Distributions*. Retrieved 2012-05-29.
4. ^ "9.6 random — Generate pseudo-random numbers". *Python v2.6.8 documentation*. Retrieved 2012-05-29.
5. ^ "8.6 random — Generate pseudo-random numbers". *Python v3.2 documentation*. Retrieved 2012-05-29.
6. ^ ""Random" class documentation". *Ruby 1.9.3 documentation*. Retrieved 2012-05-29.
7. ^ "mt_srand". *php documentation*. Retrieved 2012-05-29.
8. ^ "Design choices and extensions". *CMUCL User's Manual*. Retrieved 2014-02-03.
9. ^ "Random Number Generation". *SBCL User's Manual*.
10. ^ "random". *free pascal documentation*. Retrieved 2013-11-28.
11. ^ Random Numbers —GLib Reference Manual
12. ^ Probability Distributions —Sage Reference Manual
13. ^ "random number generator". *Maple Online Help*. Retrieved 2013-11-21.
14. ^ Random number generator algorithms —Documentation Center, MathWorks
15. ^ GAUSS 14 Language Reference
16. ^ "RANDOMU (IDL Reference)". *Exelis VIS Docs Center*. Retrieved 2013-08-23.
17. ^ Julia Language Documentation — The Standard Library
18. ^ Random numbers —Scilab Help
19. ^ GNU Octave: §16.3 —Built-in Function: rand
20. ^ "Random number environment variables". *GNU Scientific Library*. Retrieved 2013-11-24.
21. ^ "Randum Number Algorithms". *GNU MP*. Retrieved 2013-11-21.
22. ^ <random> —Microsoft Developer Network
23. ^ Random Number Generation in C++11 —Standard C++ Foundation
24. ^ "std::mersenne_twister_engine". *Pseudo Random Number Generation*. Retrieved 2012-09-25.
25. ^ Data Generation —Apache Commons Math User Guide
26. ^ "boost/random/mersenne_twister.hpp". *Boost C++ Libraries*. Retrieved 2012-05-29.
27. ^ "G05 – Random Number Generators". *NAG Library Chapter Introduction*. Retrieved 2012-05-29.
28. ^ "Random Number Generators". *IBM SPSS Statistics*. Retrieved 2013-11-21.
29. ^ "Using Random-Number Functions". *SAS Language Reference*. Retrieved 2013-11-21.
30. ^ 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}.
31. ^ *Numerical Recipes*, §7.1.
32. ^ http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/TINYMT/index.html
33. ^ P. L'Ecuyer and R. Simard, "TestU01: "A C library for empirical testing of random number generators", *ACM Transactions on Mathematical Software*, 33, 4, Article 22 (August 2007).
34. ^ Makoto Matsumoto; Takuji Nishimura. "Dynamic Creation of Pseudorandom Number Generators" (PDF). Retrieved 19 July 2015.
35. ^ http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html
36. ^ Matsumoto, Makoto; Nishimura, Takuji; Hagita, Mariko; Saito, Mutsuo (2005). "Cryptographic Mersenne Twister

36. ^ Matsumoto, Makoto; Nishimura, Takuji; Hagita, Mariko; Saito, Mutsuo (2005). "Cryptographic Mersenne Twister and Fubuki Stream/Block Cipher" (PDF).
37. ^ P. L'Ecuyer, "Uniform Random Number Generators", *International Encyclopedia of Statistical Science*, Lovric, Miodrag (Ed.), Springer-Verlag, 2010.
38. ^ "xorshift*/xorshift+ generators and the PRNG shootout".
39. ^ Matsumoto, M.; Kurita, Y. (1992). "Twisted GFSR generators". *ACM Transactions on Modeling and Computer Simulation* **2** (3): 179–194. doi:10.1145/146382.146383.
40. ^ "std::mersenne_twister_engine". *Pseudo Random Number Generation*. Retrieved 2015-07-20.
41. ^ "std::mersenne_twister_engine". *Pseudo Random Number Generation*. Retrieved 2015-07-20.
42. ^ Takuji Nishimura; Makoto Matsumoto. "A C-program for MT19937, with initialization improved 2002/1/26.". Retrieved 20 July 2015.
43. ^ SIMD-oriented Fast Mersenne Twister (SFMT)
44. ^ SFMT:Comparison of speed
45. ^ PLAYSTATION 3 License
46. ^ Mutsuo Saito; Makoto Matsumoto (2010). "Variants of Mersenne Twister Suitable for Graphic Processors". arXiv:1005.4973v3 [cs.MS].

External links [edit]

- The academic paper for MT, and related articles by Makoto Matsumoto
- Mersenne Twister home page, with codes in C, Fortran, Java, Lisp and some other languages
- SFMT in Action —The Code Project

v · t · e	**Marin Mersenne**	[hide]
Mersenne conjectures · Mersenne's laws · Mersenne prime (Double Mersenne number · Great Internet Mersenne Prime Search) · **Mersenne Twister**		

Categories: Pseudorandom number generators