WIKIPEDIA
The Free Encyclopedia

Mersenne Twister

The **Mersenne Twister** is a general-purpose pseudorandom number generator (PRNG) developed in 1997 by Makoto Matsumoto (松本 眞) and Takuji Nishimura (西村 拓士).[1][2] Its name derives from the fact that its period length is chosen to be a Mersenne prime.

The Mersenne Twister was designed specifically to rectify most of the flaws found in older PRNGs.

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 (with five variants[3]) that uses a 64-bit word length, MT19937-64; it generates a different sequence.

Application

Software

The Mersenne Twister is used as default PRNG by the following software:

- Programming languages: Dyalog APL,[4] IDL,[5] R,[6] Ruby,[7] Free Pascal,[8] PHP,[9] Python (also available in NumPy, however the default was changed to PCG64 instead as of version 1.17[10]),[11][12][13] CMU Common Lisp,[14] Embeddable Common Lisp,[15] Steel Bank Common Lisp,[16] Julia (up to Julia 1.6 LTS, still available in later, but a better/faster RNG used by default as of 1.7)[17]
- Linux libraries and software: GLib,[18] GNU Multiple Precision Arithmetic Library,[19] GNU Octave,[20] GNU Scientific Library[21]
- Other: Microsoft Excel,[22] GAUSS,[23] gretl,[24] Stata,[25] SageMath,[26] Scilab,[27] Maple,[28] MATLAB[29]

It is also available in Apache Commons,[30] in the standard C++ library (since C++11),[31][32] and in Mathematica.[33] Add-on implementations are provided in many program libraries, including the Boost C++ Libraries,[34] the CUDA Library,[35] and the NAG Numerical Library.[36]

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".[37] The Mersenne Twister is similarly one of the PRNGs in SAS: the other generators are older and deprecated.[38] The Mersenne Twister is the default PRNG in Stata, the other one is KISS, for compatibility with older versions of Stata.[39]

Advantages

- Permissively-licensed and patent-free for all variants except CryptMT.
- Passes numerous tests for statistical randomness, including the Diehard tests and most, but not all of the TestU01 tests.[40]
- A very long period of $2^{19937} - 1$. Note that 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.[41]

- *k*-distributed to 32-bit accuracy for every $1 \leq k \leq 623$ (for a definition of *k*-distributed, see below)
- Implementations generally create random numbers faster than hardware-implemented methods. A study found that the Mersenne Twister creates 64-bit floating point random numbers approximately twenty times faster than the hardware-implemented, processor-based RDRAND instruction set.[42]

Disadvantages

- Relatively large state buffer, of 2.5 KiB, unless the TinyMT variant (discussed below) is used.
- Mediocre throughput by modern standards, unless the SFMT variant (discussed below) is used.[43]
- Exhibits two clear failures (linear complexity) in both Crush and BigCrush in the TestU01 suite. The test, like Mersenne Twister, is based on an $\mathbf{F}_2$-algebra.[40]
- Multiple 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.[44][45]
- Poor diffusion: 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 beginning with such a state is very unlikely.[46] The GPU version (MTGP) is said to be even better.[47]
- Contains subsequences with more 0's than 1's. This adds to the poor diffusion property to make recovery from many-zero states difficult.
- Is not cryptographically secure, unless the CryptMT variant (discussed below) is used. 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.

Alternatives

An alternative generator, WELL ("Well Equidistributed Long-period Linear"), offers quicker recovery, and equal randomness, and nearly equal speed.[48]

Marsaglia's xorshift generators and variants are the fastest in the class of LFSRs.[49]

64-bit MELGs ("64-bit Maximally Equidistributed $\mathbf{F}_2$-Linear Generators with Mersenne Prime Period") are completely optimized in terms of the *k*-distribution properties.[50]

The ACORN family (published 1989) is another *k*-distributed PRNG, which shows similar computational speed to MT, and better statistical properties as it satisfies all the current (2019) TestU01 criteria; when used with appropriate choices of parameters, ACORN can have arbitrarily long period and precision.

The PCG family is a more modern long-period generator, with better cache locality, and less detectable bias using modern analysis methods.[51]

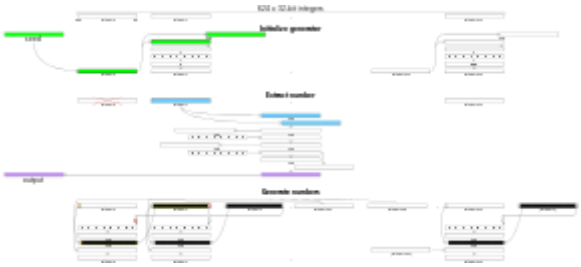
k-distribution

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 $\text{trunc}_v(x)$ denote the number formed by the leading v bits of x, and consider P of the k v-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.

Algorithmic detail

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



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.

The Mersenne Twister algorithm is based on a matrix linear recurrence over a finite binary field $\mathbf{F}_2$. The algorithm is a twisted generalised feedback shift register[52] (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_i^T, where T is an invertible $\mathbf{F}_2$-matrix called a tempering matrix.

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:

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

where $\mid$ denotes concatenation of bit vectors (with upper bits on the left), $\oplus$ 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_{w-1} as the $(w-1)(w-1)$ 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 $\mathbf{F}_2$, and therefore bitwise XOR takes the place of addition)

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

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

$$\begin{aligned} y &\equiv x \oplus ((x \gg u) \ \& \ d) \\ y &\equiv y \oplus ((y \ll s) \ \& \ b) \\ y &\equiv y \oplus ((y \ll t) \ \& \ c) \\ z &\equiv y \oplus (y \gg l) \end{aligned}$$

where x is the next value from the series, y is a temporary intermediate value, and z is the value returned from the algorithm, with $\ll$and $\gg$ 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, $s + t \geq \lfloor \frac{w}{2} \rfloor - 1$ is required to reach the upper bound of equidistribution for the upper bits.

The coefficients for MT19937 are:

$$\begin{aligned} (w, n, m, r) &= (32, 624, 397, 31) \\ a &= \mathrm{9908B0DF}_{32} \\ (u, d) &= (11, \mathrm{FFFFFFFF}_{32}) \\ (s, b) &= (7, \mathrm{9D2C5680}_{32}) \\ (t, c) &= (15, \mathrm{EFC60000}_{32}) \\ l &= 18 \end{aligned}$$

Note that 32-bit implementations of the Mersenne Twister generally have d = $\mathrm{FFFFFFFF}_{16}$. 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:[53]

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

Initialization

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 used to supply x_0 through x_{n-1} by setting x_0 to the seed value and thereafter setting

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

for i from 1 to $n-1$.

- The first value the algorithm then generates is based on x_n, not on x_0.
- The constant *f* forms another parameter to the generator, though not part of the algorithm proper.
- The value for *f* for MT19937 is 1812433253.
- The value for *f* for MT19937-64 is 6364136223846793005.[53]

Comparison with classical GFSR

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

- The 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 five dimensions)

Pseudocode

The following 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[54]
        }
        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
}
```

Variants

CryptMT

CryptMT is a stream cipher and cryptographically secure pseudorandom number generator which uses Mersenne Twister internally.[55][56] It was developed by Matsumoto and Nishimura alongside Mariko Hagita and Mutsuo Saito. It has been submitted to the eSTREAM project of the eCRYPT

network.[55] Unlike Mersenne Twister or its other derivatives, CryptMT is patented.

MTGP

MTGP is a variant of Mersenne Twister optimised for graphics processing units published by Mutsuo Saito and Makoto Matsumoto.[57] 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 very old GPU (Nvidia GTX260 with 192 cores) of 4.7 ms for 5×10^7 random 32-bit integers.

SFMT

The SFMT (SIMD-oriented Fast Mersenne Twister) is a variant of Mersenne Twister, introduced in 2006,[58] designed to be fast when it runs on 128-bit SIMD.

- It is roughly twice as fast as Mersenne Twister.[59]
- 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.[60]

TinyMT

TinyMT is a variant of Mersenne Twister, proposed by Saito and Matsumoto in 2011.[61] TinyMT uses just 127 bits of state space, a significant decrease compared to the original's 2.5 KiB of state. However, it has a period of $\mathbf{2^{127}-1}$, far shorter than the original, so it is only recommended by the authors in cases where memory is at a premium.

References

1. Matsumoto, M.; Nishimura, T. (1998). "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator" (http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/ARTICLES/mt.pdf) (PDF). *ACM Transactions on Modeling and Computer Simulation*. **8** (1): 3–30. CiteSeerX 10.1.1.215.1141 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.215.1141). doi:10.1145/272991.272995 (https://doi.org/10.1145%2F272991.272995). S2CID 3332028 (https://api.semanticscholar.org/CorpusID:3332028).
2. E.g. Marsland S. (2011) *Machine Learning* (CRC Press), §4.1.1. Also see the section "Adoption in software systems".
3. John Savard. "The Mersenne Twister" (http://www.quadibloc.com/crypto/co4814.htm). "A subsequent paper, published in the year 2000, gave five additional forms of the Mersenne Twister with period 2^19937-1. All five were designed to be implemented with 64-bit arithmetic instead of 32-bit arithmetic."
4. "Random link" (http://help.dyalog.com/latest/#Language/System%20Functions/rl.htm). *Dyalog Language Reference Guide*. Retrieved 2020-06-04.
5. "RANDOMU (IDL Reference)" (http://www.exelisvis.com/docs/RANDOMU.html). *Exelis VIS Docs Center*. Retrieved 2013-08-23.

6. "Random Number Generators" (https://cran.r-project.org/web/views/Distributions.html). *CRAN Task View: Probability Distributions*. Retrieved 2012-05-29.
7. " "Random" class documentation" (http://www.ruby-doc.org/core-1.9.3/Random.html). *Ruby 1.9.3 documentation*. Retrieved 2012-05-29.
8. "random" (http://www.freepascal.org/docs-html/rtl/system/random.html). *free pascal documentation*. Retrieved 2013-11-28.
9. "mt_rand — Generate a better random value" (http://php.net/manual/en/function.mt-rand.php). *PHP Manual*. Retrieved 2016-03-02.
10. "NumPy 1.17.0 Release Notes — NumPy v1.21 Manual" (https://numpy.org/doc/stable/release/1.17.0-notes.html?highlight=random). *numpy.org*. Retrieved 2021-06-29.
11. "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.
12. "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.
13. "random — Generate pseudo-random numbers — Python 3.8.3 documentation" (https://docs.python.org/3/library/random.html). *Python 3.8.3 documentation*. Retrieved 2020-06-23.
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 states" (https://common-lisp.net/project/ecl/manual/ch12s02.html). *The ECL manual*. Retrieved 2015-09-20.
16. "Random Number Generation" (http://www.sbcl.org/manual/#Random-Number-Generation). *SBCL User's Manual*.
17. "Random Numbers · The Julia Language" (https://docs.julialang.org/en/v1/stdlib/Random/). *docs.julialang.org*. Retrieved 2022-06-21.
18. "Random Numbers: GLib Reference Manual" (https://developer.gnome.org/glib/stable/glib-Random-Numbers.html).
19. "Random Number Algorithms" (http://gmplib.org/manual/Random-Number-Algorithms.html). *GNU MP*. Retrieved 2013-11-21.
20. "16.3 Special Utility Matrices" (https://www.gnu.org/software/octave/doc/interpreter/Special-Utility-Matrices.html). *GNU Octave*. "Built-in Function: rand"
21. "Random number environment variables" (https://www.gnu.org/software/gsl/manual/html_node/Random-number-environment-variables.html). *GNU Scientific Library*. Retrieved 2013-11-24.
22. Mélard, G. (2014), "On the accuracy of statistical procedures in Microsoft Excel 2010", *Computational Statistics*, **29** (5): 1095–1128, CiteSeerX 10.1.1.455.5508 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.455.5508), doi:10.1007/s00180-014-0482-5 (https://doi.org/10.1007%2Fs00180-014-0482-5), S2CID 54032450 (https://api.semanticscholar.org/CorpusID:54032450).
23. "GAUSS 14 Language Reference" (http://www.aptech.com/wp-content/uploads/2014/01/GAUSS14_LR.pdf) (PDF).
24. "uniform (http://gretl.sourceforge.net/gretl-help/funcref.html#uniform)". *Gretl Function Reference*.
25. "New random-number generator—64-bit Mersenne Twister" (https://www.stata.com/new-in-stata/random-number-generators/).
26. "Probability Distributions — Sage Reference Manual v7.2: Probablity" (http://doc.sagemath.org/html/en/reference/probability/sage/gsl/probability_distribution.html).
27. "grand - Random numbers" (https://help.scilab.org/docs/5.5.2/en_US/grand.html). *Scilab Help*.
28. "random number generator" (http://www.maplesoft.com/support/help/Maple/view.aspx?path=rand). *Maple Online Help*. Retrieved 2013-11-21.
29. "Random number generator algorithms" (http://www.mathworks.co.uk/help/matlab/ref/randstream.list.html). *Documentation Center, MathWorks*.

30. "Data Generation" (http://commons.apache.org/proper/commons-math/userguide/random.html). *Apache Commons Math User Guide*.
31. "Random Number Generation in C++11" (https://isocpp.org/files/papers/n3551.pdf) (PDF). *Standard C++ Foundation*.
32. "std::mersenne_twister_engine" (http://en.cppreference.com/w/cpp/numeric/random/mersenne_twister_engine). *Pseudo Random Number Generation*. Retrieved 2012-09-25.
33. [1] (http://reference.wolfram.com/language/tutorial/RandomNumberGeneration.html#569959585) Mathematica Documentation
34. "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.
35. "Host API Overview" (http://docs.nvidia.com/cuda/curand/host-api-overview.html#generator-types). *CUDA Toolkit Documentation*. Retrieved 2016-08-02.
36. "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.
37. "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.
38. "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.
39. Stata help: set rng -- Set which random-number generator (RNG) to use (https://www.stata.com/help.cgi?set%20rng)
40. 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).
41. 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}.
42. Route, Matthew (August 10, 2017). "Radio-flaring Ultracool Dwarf Population Synthesis". *The Astrophysical Journal*. **845** (1): 66. arXiv:1707.02212 (https://arxiv.org/abs/1707.02212). Bibcode:2017ApJ...845...66R (https://ui.adsabs.harvard.edu/abs/2017ApJ...845...66R). doi:10.3847/1538-4357/aa7ede (https://doi.org/10.3847%2F1538-4357%2Faa7ede). S2CID 118895524 (https://api.semanticscholar.org/CorpusID:118895524).
43. "SIMD-oriented Fast Mersenne Twister (SFMT): twice faster than Mersenne Twister" (http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/). *Japan Society for the Promotion of Science*. Retrieved 27 March 2017.
44. Makoto Matsumoto; Takuji Nishimura. "Dynamic Creation of Pseudorandom Number Generators" (http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/DC/dgene.pdf) (PDF). Retrieved 19 July 2015.
45. Hiroshi Haramoto; Makoto Matsumoto; Takuji Nishimura; François Panneton; Pierre L'Ecuyer. "Efficient Jump Ahead for F2-Linear Random Number Generators" (http://www.iro.umontreal.ca/~lecuyer/myftp/papers/jumpf2.pdf) (PDF). Retrieved 12 Nov 2015.
46. "mt19937ar: Mersenne Twister with improved initialization" (http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html). *hiroshima-u.ac.jp*. Retrieved 4 October 2015.
47. Fog, Agner (1 May 2015). "Pseudo-Random Number Generators for Vector Processors and Multicore Processors" (https://doi.org/10.22237%2Fjmasm%2F1430454120). *Journal of Modern Applied Statistical Methods*. **14** (1): 308–334. doi:10.22237/jmasm/1430454120 (https://doi.org/10.22237%2Fjmasm%2F1430454120).
48. P. L'Ecuyer, "Uniform Random Number Generators", *International Encyclopedia of Statistical Science*, Lovric, Miodrag (Ed.), Springer-Verlag, 2010.
49. "xorshift*/xorshift+ generators and the PRNG shootout" (http://prng.di.unimi.it).

50. Harase, S.; Kimoto, T. (2018). "Implementing 64-bit Maximally Equidistributed F_2-Linear Generators with Mersenne Prime Period" (https://github.com/sharase/melg-64). *ACM Transactions on Mathematical Software*. **44** (3): 30:1–30:11. arXiv:1505.06582 (https://arxiv.org/abs/1505.06582). doi:10.1145/3159444 (https://doi.org/10.1145%2F3159444). S2CID 14923086 (https://api.semanticscholar.org/CorpusID:14923086).
51. "The PCG Paper" (https://www.pcg-random.org/paper.html). 27 July 2017.
52. Matsumoto, M.; Kurita, Y. (1992). "Twisted GFSR generators" (http://ir.lib.hiroshima-u.ac.jp/00015037). *ACM Transactions on Modeling and Computer Simulation*. **2** (3): 179–194. doi:10.1145/146382.146383 (https://doi.org/10.1145%2F146382.146383). S2CID 15246234 (https://api.semanticscholar.org/CorpusID:15246234).
53. "std::mersenne_twister_engine" (http://en.cppreference.com/w/cpp/numeric/random/mersenne_twister_engine). *Pseudo Random Number Generation*. Retrieved 2015-07-20.
54. Takuji Nishimura; Makoto Matsumoto. "A C-program for MT19937, with initialization improved 2002/1/26" (http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/CODES/mt19937ar.c). Retrieved 20 July 2015.
55. "CryptMt and Fubuki" (http://www.ecrypt.eu.org/stream/cryptmtfubuki.html). eCRYPT. Retrieved 2017-11-12.
56. Matsumoto, Makoto; Nishimura, Takuji; Hagita, Mariko; Saito, Mutsuo (2005). "Cryptographic Mersenne Twister and Fubuki Stream/Block Cipher" (http://eprint.iacr.org/2005/165.pdf) (PDF).
57. Mutsuo Saito; Makoto Matsumoto (2010). "Variants of Mersenne Twister Suitable for Graphic Processors". arXiv:1005.4973v3 (https://arxiv.org/abs/1005.4973v3) [cs.MS (https://arxiv.org/archive/cs.MS)].
58. "SIMD-oriented Fast Mersenne Twister (SFMT)" (http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/index.html). *hiroshima-u.ac.jp*. Retrieved 4 October 2015.
59. "SFMT:Comparison of speed" (http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/speed.html). *hiroshima-u.ac.jp*. Retrieved 4 October 2015.
60. "PlayStation3 License" (http://www.scei.co.jp/ps3-license/index.html). *scei.co.jp*. Retrieved 4 October 2015.
61. "Tiny Mersenne Twister (TinyMT)" (http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/TINYMT/index.html). *hiroshima-u.ac.jp*. Retrieved 4 October 2015.

Further reading

- Harase, S. (2014), "On the $\mathbb{F}_2$-linear relations of Mersenne Twister pseudorandom number generators", *Mathematics and Computers in Simulation*, **100**: 103–113, arXiv:1301.5435 (https://arxiv.org/abs/1301.5435), doi:10.1016/j.matcom.2014.02.002 (https://doi.org/10.1016%2Fj.matcom.2014.02.002), S2CID 6984431 (https://api.semanticscholar.org/CorpusID:6984431).
- Harase, S. (2019), "Conversion of Mersenne Twister to double-precision floating-point numbers", *Mathematics and Computers in Simulation*, **161**: 76–83, arXiv:1708.06018 (https://arxiv.org/abs/1708.06018), doi:10.1016/j.matcom.2018.08.006 (https://doi.org/10.1016%2Fj.matcom.2018.08.006), S2CID 19777310 (https://api.semanticscholar.org/CorpusID:19777310).

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)
- Mersenne Twister examples (https://github.com/bmurray7/mersenne-twister-examples) — a collection of Mersenne Twister implementations, in several programming languages - at GitHub

- SFMT in Action: Part I – Generating a DLL Including SSE2 Support (http://www.codeproject.com/KB/DLL/SFMT_dll.aspx?msg=3130186) – at Code Project

---