A PRNG specialized in double precision floating point numbers using an affine transition

Mutsuo Saito and Makoto Matsumoto

Abstract We propose a pseudorandom number generator specialized to generate double precision floating point numbers, which generates pseudorandom 52-bit patterns supplemented by a constant most significant 12-bit (sign and exponent), so that the concatenated 64-bit represents a floating point number obeying the IEEE 754 format. To keep the constant part, we adopt an affine transition function instead of usual $\mathbb{F}_2$-linear transition, and extend algorithms computing the period and the dimensions of equi-distribution to the affine case. The resulted generator generates double precision floating point numbers faster than Mersenne Twister generates those of 32-bit precision.

1 Introduction

In [19], we proposed a fast version of the Mersenne twister (MT) of [14], that exploits the single instruction multiple data [20] (SIMD) feature of some recent CPUs, which processes 128-bit at a time. This new pseudorandom number generator (PRNG), named SFMT (which stands for SIMD-oriented fast Mersenne twister), is faster than the original MT and also has better equidistribution. The proposal of [19] also features a block generation procedure, which returns a large array of pseudorandom numbers at each call.

Mutsuo Saito
Department of Mathematics, Graduate School of Science, Hiroshima University, Hiroshima, Japan, e-mail: `saito@math.sci.hiroshima-u.ac.jp`

Makoto Matsumoto
Department of Mathematics, Graduate School of Science, Hiroshima University, Hiroshima, Japan, e-mail: `m-mat@math.sci.hiroshima-u.ac.jp`

In this article, we propose PRNGs specialized in generating floating point numbers, which we call dSFMT (double precision floating point SFMT). It generates a sequence of 64-bit patterns with constant 12 most significant bits (MSBs), so that each of 64-bit patterns represents a double precision floating point numbers in a fixed interval in the standard IEEE754 format. Instead of usual $\mathbb{F}_2$-linear transition function, we adopt $\mathbb{F}_2$-affine transition function to keep the fixed constant in the 64-bit (§4). We extended some of the existing algorithms to compute the period and distribution to the affine case. As a result, we implemented this type of generators whose periods are multiples of 6 Mersenne primes from $2^{521}-1$ to $2^{19937}-1$, respectively. These generators are shown to be faster than MT, SFMT and WELL generators, and have satisfactorily high dimensions of equidistribution (much higher than MT, but lower than WELL which attains the theoretical bounds).

2 Generating floating point numbers

Usually, floating point pseudorandom numbers are obtained by converting integer pseudorandom numbers. One may consider recursion in floating point numbers for PRNG, but it may accumulate approximation errors. Since the rounding-off is not standardized, the generated sequence often depends on CPUs. Consequently, usual PRNGs generate integer random numbers by integer recursion, and converts them to floating point numbers by multiplying a constant. However, this method requires a conversion from an integer to a floating point number, which consumes about 50% of the cpu-time in the generation, according to our experiments using the 64-bit MT [15].

A faster conversion is given by bit operations fitting to a standard floating point format. We recall the most widely-used standard, IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754-2008) [6], which we shall refer as IEEE 754. The standard was defined in 1985 and revised in 2008, and we here treat the 64-bit binary format valid for both. The 64-bit are separated in to the sign bit (the most significant bit, MSB), the exponent 11 bit (the next significant 11 bits representing an integer between 0 to 2047, denoted by e) and the rest 52 fraction bits (representing a real number in $[1, 2)$: 52-bit pattern `xxx`... is interpreted as a binary floating number 1.`xxx`..., denoted by f). When $0 < e < 2047$, the 64-bit represents a floating point number $\pm f \times 2^{e-1023}$ with the sign determined by the sign bit. Thus, if the sign bit is 0 and $e = 1023$ (or equivalently the 12 MSBs being `0x3ff` in hexadecimal form), then the represented number is in $[1, 2)$. If the 52-bit fraction part is uniformly randomly chosen, then the represented number is uniformly randomly distributed over $[1, 2)$ with 52-bit precision. In C-language, this conversion of a 64-bit integer `x` is described as follows:

```
x = (x >> 12) | 0x3FF0000000000000ULL;
y = *((double *)&x);
```

where the first line shifts `x` to the right by 12 bits and set the 12 MSBs to the constant `0x3ff`, and the second line regards the 64-bit pattern as an IEEE 754 format. This method is less portable than the conversion by multiplication, depending on a particular format, but consumes only 5% to 10% of the cpu-time for the conversion, according to our experiments with the 64-bit MT. This method goes back to at least 1997: Agner Fog used this method in his open source library [4], and others seemed to invent it independently, too.

A pseudorandom number r in $[1, 2)$ can be converted into $[0, 1)$ (respectively $(0, 1]$) by taking $r-1$ (respectively $1-r$). In practice, it is often the case that random numbers r in the range $[1, 2)$ can be used without converting into $[0, 1)$: for example, the Box-Muller transformation converts two uniform random numbers s_1, s_2 in $[0, 1)$ into two normally distributed numbers

$$\sqrt{-2\log(1-s_1)}\sin(2\pi s_2), \sqrt{-2\log(1-s_1)}\cos(2\pi s_2).$$

If r_1, r_2 are two uniform random numbers in $[1, 2)$, then the conversion can be done by

$$\sqrt{-2\log(2-r_1)}\sin(2\pi r_2), \sqrt{-2\log(2-r_1)}\cos(2\pi r_2).$$

3 LFSR with lung

Our proposal is to use a linear recursion over $\mathbb{F}_2$ to generate a sequence of 64-bit patterns with 12 MSBs being `0x3ff` as above, by a Linear Feedback Shift Register (LFSR) with additional memory called 'lung.' We identify the set of bits $\{0, 1\}$ with the two element field $\mathbb{F}_2$. This means that every arithmetic operation is done modulo 2. A b-bit register or memory is identified with a horizontal vector in $\mathbb{F}_2^b$, and $+$ denotes the sum as vectors (i.e., bit-wise exclusive or). We consider an array of b-bit integers of size N in computer memory as the vector space $(\mathbb{F}_2^b)^N$.

An LFSR method is to generate a sequence $\mathbf{w_0}, \mathbf{w_1}, \mathbf{w_2}, \ldots$ of elements $\mathbb{F}_2^b$ by a recursion

$$\mathbf{w}_{i+N} = g(\mathbf{w}_i, \ldots, \mathbf{w}_{i+N-1}), \quad (i = 0, 1, 2, \ldots)$$

where g is an $\mathbb{F}_2$-linear map $(\mathbb{F}_2^b)^N \to \mathbb{F}_2^b$. In the implementation, this recursion is computed by using an array `W[0..N-1]` of N words of b-bit size, by the simultaneous substitutions

$$\texttt{W[0]} \leftarrow \texttt{W[1]},\ \texttt{W[1]} \leftarrow \texttt{W[2]}, \ldots, \texttt{W[N-2]} \leftarrow \texttt{W[N-1]},$$
$$\texttt{W[N-1]} \leftarrow g(\texttt{W[0]}, \ldots, \texttt{W[N-1]}).$$

The first $N-1$ substitutions shift the content of the array, hence the name of LFSR. Note that in the implementation we may use an indexing technique to avoid computing these substitutions, see [7, P.28 Algorithm A]. Before starting the generation, we need to set some values to the state array, which is called the initialization. Mersenne Twister [14] (MT) is such an example.

An LFSR with lung is to generate a sequence $\mathbf{w_0}$,$\mathbf{w_1}$, $\mathbf{w_2}$,... of elements $\mathbb{F}_2^b$ by a recursion

$$\mathbf{w}_i = g(\mathbf{w}_{i-N+1}, ..., \mathbf{w}_{i-1}, \mathbf{u}_{i-1}), \quad (1)$$

$$\mathbf{u}_i = h(\mathbf{w}_{i-N+1}, ..., \mathbf{w}_{i-1}, \mathbf{u}_{i-1}). \quad (2)$$

where g and h are $\mathbb{F}_2$-linear maps $(\mathbb{F}_2^b)^N \to \mathbb{F}_2^b$ and $\mathbf{w}_i, \mathbf{u}_i \in \mathbb{F}_2^b$. In the implementation, $\mathbf{w}_i$'s are kept in an array `W[0..N-1]`, and $\mathbf{u}_i$ is (expected to be) kept in a register of CPU, which is called the *lung*. We denote the register by `U`. The first line (1) renews the array `W[0..N-1]`, and the second line (2) renews the register (lung) `U`. The idea of LFSR with lung appeared in the talk of Hiroshi Haramoto in a conference MCM 2005, and used in WELL PRNG [17] in 2006. The lung realizes a short feedback loop, which improves some measures of randomness such as higher dimensional equidistributions and the density of nonzero coefficients in the characteristic polynomial.

4 Affinity introduced by the constant part

Our idea is to design the functions g and h in the recursion (1) (2) for LFSR with lung, so that if the initial values $\mathbf{w}_0, \ldots, \mathbf{w}_{N-1}$ are set to have `0x3ff` at their 12 MSBs, then the following $\mathbf{w}_i$ have the same property, regardlessly of the value of $\mathbf{u}_0$. According to our experiments, this method is 5% to 10% faster than the bit-masking conversion explained in §2.

A new difficulty in this approach is that the state transition is far from being maximal periodic. A linear state transition function is said to be maximal periodic, if every non-zero state lies on the same orbit. The existence of the constant implies that, if the initial state is chosen as above, then 12 MSBs of each member of the array of `W[0..N-1]` are constant in the orbit, and the transition can not be maximal periodic. This makes it difficult to apply standard techniques to compute the period and high-dimensional equidistribution property.

A natural solution to this problem is to redefine the state space by excluding the constant part, and consider the transition function as an affine function. More concretely, let $\mathbf{w}'_i$ denote the lower 52-bit of $\mathbf{w}_i$. Since the upper 12-bit is a constant, the recursion formula (1), (2) can be described by

$$\mathbf{w}'_i = g'(\mathbf{w}'_{i-N+1}, ..., \mathbf{w}'_{i-1}, \mathbf{u}_{i-1}), \quad (3)$$

$$\mathbf{u}_i = h'(\mathbf{w}'_{i-N+1}, ..., \mathbf{w}'_{i-1}, \mathbf{u}_{i-1}). \quad (4)$$

Here, it is easy to see that the linearity of g (resp. h) implies the affinity of g' (resp. h'). (Here affine means linear plus a constant.)

Let b_w denote the number of variable bits in each `W[i]` (52 in the above case), and b_u denote the number of bits in the lung `U`. This LFSR with lung (not linear but affine) is considered as an automaton, with the state space $S = \mathbb{F}_2^{b_u + b_w \times (N-1)}$. The state transition function $F : S \to S$ is given by

$$\begin{aligned}&(\mathbf{w}'_0, \ldots, \mathbf{w}'_{N-2}, \mathbf{u}_0)\\ &\qquad \mapsto (\mathbf{w}'_1, \ldots, \mathbf{w}'_{N-2}, g'(\mathbf{w}'_0, \ldots, \mathbf{w}'_{N-2}, \mathbf{u}_0), h'(\mathbf{w}'_0, \ldots, \mathbf{w}'_{N-2}, \mathbf{u}_0)).\end{aligned}$$

As a b_w-bit vector generator (i.e., removing the constant bits), the output function is

$$o : S \to \mathbb{F}_2{}^{b_w}; \quad (\mathbf{w}'_0, \ldots, \mathbf{w}'_{N-2}, \mathbf{u}_0) \mapsto \mathbf{w}'_0.$$

Now, both F and o are not linear but affine. Namely, they have the form $x \mapsto Ax + c$ where x is a vector, A is an $\mathbb{F}_2$ matrix, and c is a constant vector. (If $c = 0$, it is linear.)

5 Reduction from affine to linear: fixed points

Let f denote the linear part of F, namely, put $c := F(0)$ and

$$F(x) = f(x) + c \tag{5}$$

with linear $f : S \to S$. If F has a fixed point $F(z) = z$, then $F(x - z) = f(x - z) + c = f(x) - z$, and consequently $F^n(x - z) = f^n(x) - z$. Thus, for the state transition $x_0, x_1, x_2, \ldots$ by F, its translation $x_0 + z, x_1 + z, \ldots$ by the constant z is obtained by the linear state transition f, hence can be analyzed by the existing methods. Since the period and the distribution property of the sequence is unchanged by a parallel translation, computation of those for the affine F is reduced to those for the linear f. If f has the maximal period, then the equidistribution property can be computed as usual.

The equation $F(z) = z$ is equivalent to $(f - \mathrm{Id})(z) = c$, where Id denotes the identity transformation on S. Thus, a fixed point exists if the characteristic polynomial χ_f of f does not have 1 as a root, in particular if irreducible with degree ≥ 2.

6 Reducible transition function in affine case

Usually, to assure the period, we need to check the primitivity of χ_f. This is often computationally difficult, since we need the integer factorization of $2^{\deg(\chi(t))} - 1$, which is hard if the degree is high (say, > 10000). There are two methods to avoid this: (1) to tune the size of the state space to be a Mersenne exponent (i.e. a prime number p such that $2^p - 1$ is also prime) where $2^{\deg(\chi(t))} - 1$ is a prime, and (2) to use f such that χ_f has an irreducible factor of a Mersenne prime degree denoted by p. We here adopt the latter method, named reducible transition method (RTM) in [19]. This is advantageous over the former in the generation speed, because of no need of discarding a part of the state array (as was required in MT [14] and WELL [17]). Note that this idea appeared in somewhat different purposes previously in [5][1][2].

We here recall RTM very briefly. Let $f : S \to S$ be an $\mathbb{F}_2$-linear transition function, $o : S \to O$ be an $\mathbb{F}_2$-linear output function. Assume that a linear transition function $f : S \to S$ has a decomposition $S = V_p \oplus V_r$, $f = f_p \oplus f_r$ with $f_p : V_p \to V_p$, $f_r : V_r \to V_r$. In other words, f is the combined generator obtained from the two generators (f_p, V_p, o_p) and (f_r, V_r, o_r), in the sense of §2.3 of [10]. A linear output function $o : S \to O$ is then the sum of there restrictions $o_p : V_p \to O$ and $o_r : V_r \to O$. The output of the combined generator is obtained by taking xor of the outputs of each generator. The period of the combined generator (f, S, o) is the least common multiple of the two generators. Thus, once we know that (f_p, V_p, o_p) has a large period, then the combined generator has at least that period.

Our strategy is to fix a Mersenne prime p, to determine the size N of the state array so that $p \leq \dim S$, and then search for parameters with a factorization $\chi_f = \phi_p \phi_r$, where ϕ_p is irreducible of degree p and ϕ_r has degree r with $r < p$. Then, it is automatic to have a decomposition $S = V_p \oplus V_r$ into p-dimensional and r-dimensional subspaces, so that the restriction f_p (respectively f_r) of f to V_p (respectively to V_r) has the characteristic polynomial ϕ_p (respectively ϕ_r). Once we have such decomposition, then the component $f_p : V_p \to V_p$ has the Mersenne exponent dimension p, and hence an existing method searches for the parameters that assure the period of $2^p - 1$. Then we can assure $2^p - 1$ as the lower bound of the period of the combined generator, provided that the initial state $s = s_p \oplus s_r \in S = V_p \oplus V_r$ has non zero component $s_p \neq 0$.

In the case of affine transition $F(x) = f(x) + c$, we assume that its linear part f satisfies the above factorizing condition $\chi_f = \phi_p \phi_r$. Let us decompose $c = c_p \oplus c_r$ and $x = x_p \oplus x_r$ along $V_p \oplus V_r$, then

$$F(x) = f(x) + c = (f_p(x_p) + c_p) \oplus (f_r(x_r) + c_r) =: F_p(x_p) \oplus F_r(x_r). \quad (6)$$

This implies that the affine generator (F, S, o) is obtained by combining two affine generators (F_p, V_p, o_p) and (F_r, V_r, o_r). Now f_p is irreducible, and the fixed point argument in §5 reduce the computation of the periods and the high-dimensional equidistribution property for F_p to those for f_p.

7 Period certification

We explain how to choose parameters realizing the period $2^p - 1$, for a given Mersenne exponent p. For the linear transition function, the method is described in [19], which we briefly recall. Let N be the smallest length of the array such that the dimension of the state space $S = \mathbb{F}_2^{b_u + b_w \times (N-1)}$ is greater than or equal to p. Thus, $r := \dim S - p < b_w$ holds.

We randomly choose parameters for the recursion (3) and (4). Let $F : S \to S$ be the corresponding affine transition function, and $f : S \to S$ be its linear part. We compute the characteristic polynomial $\chi_f(t)$ by using Berlekamp-Massey algorithm, and check whether it decomposes to

$$\chi_f = \phi_p \phi_r$$

where ϕ_p is a primitive polynomial of degree p and ϕ_r is a polynomial of degree $r := \dim S - p < b_w$. We assume $r < p$, which is natural in our context where p is large, and also $b_w \leq b_u$, since b_w is the number of the non-constant part in a b_u-bit word. We continue the random search of parameters, until we obtain a primitive ϕ_p.

Once we found such a set of parameter, then we have $S = V_p \oplus V_r$ and the projector $P_p : S \to V_p$. To assure the period of a multiple of $2^p - 1$ for the initial state $s \in S$, it suffices to assure $s_p := P_p(s) \neq 0$. In the implementation, to compute $P_p(s)$ is a time-consuming procedure in the initialization. Instead, we propose the following method, named Period Certification Vector (PCV) method, by which the period is certified by looking at one word in the state.

Let V_U denote the b_u-dimensional vector space corresponding to the lung `U` in (4). To certify the period for the initial state $s \in S$, it suffices to show that $s \notin V_r$. Let $\pi : S \to V_U$ be the projection obtained by extracting the lung from the state space S. Since we assumed $b_u = \dim(V_U) > r$, the image $\pi(V_r)$ is a proper subspace of V_U. Hence, there is a nonzero vector q in V_U which is orthogonal to every vector in $\pi(V_r)$. We call such a vector *the period certification vector* (PCV). For a given initial state s, if the inner product $\pi(s) \cdot q$ is nonzero, then $\pi(s) \notin \pi(V_r)$ and hence $s \notin V_r$, and the period is certified. If the inner product is zero, then we make the inner product nonzero by reversing appropriate one bit in $\pi(s)$.

The period certification for affine case easily reduces to the linear case. Let $z_p \in V_p$ be a fixed point of F_p. For the initial state $s \in S$, it suffices to show that $s - z_p \notin V_r$ to assure the period. This can be done by precomputing $\pi(z_p)$, and check that $(\pi(s) - \pi(z_p)) \cdot q \neq 0$. In this method, only two constant b_u-bit words $\pi(z_p)$ and q need to be precomputed and stored, and at the initialization stage, only the last inner-product is necessary to compute.

8 Computation of the dimension of equidistribution

We briefly recall the definition of dimension of equidistribution (cf. [3][8][19]).

Definition 1. Let $F : S \to S$ be an affine transition function over $\mathbb{F}_2$. Let v be an integer, and $o : S \to \mathbb{F}_2^v$ be a v-bit affine output function. The generator (S, F, o) is said to be k-dimensionally equidistributed, if the map

$$S \to (\mathbb{F}_2^v)^k, \quad s \mapsto (o(s), o(F(s)), o(F^2(s)), \ldots, o(F^{k-1}(s)))$$

is surjective. The largest value of such k is called the dimension of equidistribution (DE).

For a b-bit integer generator, its *dimension of equidistribution at v-bit accuracy* $k(v)$ is defined as the DE of the v-bit sequence, obtained by extracting the v MSBs from each of the b-bit integers.

Let $P = 2^p - 1$ be the period of the generated sequence. Then, there is an upper bound $k(v) \le \lfloor p/v \rfloor$, and their gap $d(v)$ is called the dimension defect at v of the sequence, and their sum Δ over $v = 1, \ldots, b$ is called the total dimension defect, namely:

$$d(v) := \lfloor p/v \rfloor - k(v) \text{ and } \Delta := \sum_{v=1}^{b} d(v). \tag{7}$$

We adopt RTM as in §6, and the dimensions of the equidistribution of the larger component (F_p, V_p, o_p) gives the lower bound of these dimensions [9] [19]. Accordingly, we define $k(v)$ and $d(v)$ of RTM to be those for this larger component. Let f_p be the linear part of F_p. Since χ_{f_p} is irreducible, there is a fixed point of F_p as explained in §5. Thus, computation of $k(v)$ for F_p is reduced to that for the linear part f_p, which was done in [19].

9 Implementation of dSFMT

As a result of the preceding discussion, we propose a generator using SIMD features, an affine transition function to keep the MSBs constant, and reducible characteristic polynomial. The generator is named dSFMT (double precision floating point SIMD-oriented Fast Mersenne Twister).

Remark 1. In the homepage [18], we released "dSFMT" in 2007, but no corresponding article exists. The generator proposed here is its improved version, by adopting the lung and a more efficient recursion, and is referred to as dSFMT ver. 2 in the homepage. In this manuscript, we call the former as dSFMT-old, and the latter as dSFMT simply.

The dSFMT generator is an LFSR with lung, whose recursion formulas are (1) and (2) with

$$h(\mathbf{w}_0, \ldots, \mathbf{w}_{N-2}, \mathbf{u}_0) = \mathbf{w}_0 A + \mathbf{w}_M + \mathbf{u}_0 B, \tag{8}$$
$$g(\mathbf{w}_0, \ldots, \mathbf{w}_{N-2}, \mathbf{u}_0) = \mathbf{w}_0 + h(\mathbf{w}_0, \ldots, \mathbf{w}_{N-2}, \mathbf{u}_0)C, \tag{9}$$

where $\mathbf{w}_i$'s and $\mathbf{u}$ are 128-bit integers regarded as horizontal vectors in $\mathbb{F}_2^{128}$, and A, B, C are linear transformations described below, computable by a few SIMD operations. The number b_w of variable bits is $128 - 12 \times 2 = 104$, while $b_u = 128$. It generates two 52-bit precision floating point numbers at each step.

- $\mathbf{w}A := \mathbf{w} \overset{64}{<<} \mathrm{SL1}$
  This notation means that $\mathbf{w}$ is regarded as two 64-bit memories, and $\mathbf{w}A$ is the result of the left-shift of each 64-bit by SL1 bits. There is such a SIMD operation in Pentium SSE2, and can be emulated in PowerPC AltiVec SIMD. SL1 is a parameter with $12 \le \mathrm{SL1} < 64$.
- $\mathbf{u}B := \mathbf{u}\,\mathrm{perm}(4, 3, 2, 1)$
  This notation means that $\mathbf{u}$ is regarded as four 32-bit memories and $\mathbf{u}\,\mathrm{perm}(4, 3, 2, 1)$ is the result of reverse the order of 32-bit block in 128-bit. The permutation can be done by one SIMD operation.
- $\mathbf{u}C := (\mathbf{u} \overset{64}{>>} 12) + (\mathbf{u}\,\&\,\mathrm{MASK})$
  The notation $\mathbf{u} \overset{64}{>>} 12$ means that $\mathbf{u}$ is regarded as two 64-bit memories and each right-shifted by 12 bit. The notation & means 128-bit bitwise logical '**AND**' with a 128-bit constant vector **MASK**. MASK is a concatenation of two 64-bit vectors with 0s in the 12 MSBs for both.

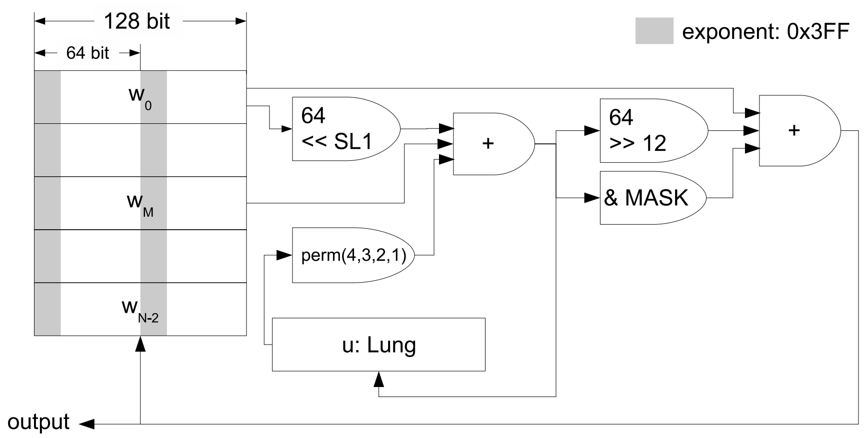

Fig. 1 Diagram of dSFMT

Fig. 1 shows the recursion in a circuit-like diagram. Note that the recursion (8) and (9) is linear, and constant `0x3ff` (in hexadecimal) of IEEE 754 exponent part does not appear. The recursion is carefully selected so that once the initial values $w_0, \ldots, w_{N-2}$ have `0x3ff` in their 12 MSBs, then these constant parts are preserved through the recursion. This trick contributes to the generation speed, by avoiding constant-setting.

Table 1 lists the parameters for dSFMT with various sizes. Table 2 lists the corresponding fixed points and period certification vectors(PCV) as discussed in §7.

Table 1 Parameter sets. MEXP denotes the Mersenne exponents. The column MASK(HIGH) shows the higher 64-bit of the constant mask, and the column MASK(LOW) shows the lower 64-bit in hexadecimal.

MEXP	N	M	SL1	MASK(LOW)	MASK(HIGH)
521	5	3	25	`0x000fbfefff77efff`	`0x000ffeebfbdfbfdf`
1279	13	9	19	`0x000efff7ffddffee`	`0x000fbffffff77fff`
2203	21	7	19	`0x000fdffff5edbfff`	`0x000f77fffffffbfe`
4253	41	19	19	`0x0007b7fffef5feff`	`0x000ffdffeffefbfc`
11213	108	37	19	`0x000ffffffdf7fffd`	`0x000dfffffff6bfff`
19937	192	117	19	`0x000ffafffffffb3f`	`0x000ffdfffc90fffd`

Table 2 Fixed points and period certification vectors (PCVs). Two 64-bit integers (in hexadecimal) piled in one place represent one 128-bit integer with higher (respectively lower) 64-bit being the upper (respectively lower) piled integer. For example, the PCV in the first row is `0xccaa5880000000000000000000000001`.

MEXP	Fixed Point	PCV
521	`0xcfb393d661638469` `0xc166867883ae2adb`	`0xccaa588000000000` `0x0000000000000001`
1279	`0xb66627623d1a31be` `0x04b6c51147b6109b`	`0x7049f2da382a6aeb` `0xde4ca84a40000001`
2203	`0xb14e907a39338485` `0xf98f0735c637ef90`	`0x8000000000000000` `0x0000000000000001`
4253	`0x80901b5fd7a11c65` `0x5a63ff0e7cb0ba74`	`0x1ad277be12000000` `0x0000000000000001`
11213	`0xd0ef7b7c75b06793` `0x9c50ff4caae0a641`	`0x8234c51207c80000` `0x0000000000000001`
19937	`0x90014964b32f4329` `0x3b8d12ac548a7c7a`	`0x3d84e1ac0dc82880` `0x0000000000000001`

10 Comparison of speed

We compared generators MT19937, 64-bit MT19937, SFMT19937, dSFMT-old 19937 and dSFMT 19937, each with implementations using and without using SIMD instructions. For MT and SFMT, 'mask' means the conversion by bit operation described in §2 from 64 bit integer, and '$\times$ const' means the conversion by multiplying 2^{-64}. Note that original MT and SFMT do not use 'mask' conversion.

We measured the speeds for five different CPUs: Pentium M 1.4GHz, Pentium IV 3GHz, core 2 duo 1.83GHz (32-bit mode, use 1 core), AMD Athlon 64 3800+ (64-bit mode), and PowerPC G4 1.33GHz. In returning the random values, we used two different methods. One is sequential generation, where one double floating point random number is returned per call. The other is block generation, where an array of random double floating point numbers is generated per call. We used Intel C Compiler for intel CPUs (Pentium M, Pentium IV, core 2 duo) and GNU C Compiler for others (AMD Athlon, Power PC G4).

We measured the consumed cpu-time in second, for 10^8 generations of floating point numbers in the range $[0, 1)$ to compare with other generators. In case of the block generation, we generate 10^5 floating point numbers per call, and this is iterated for 10^3 times. For sequential generation, the same 10^8 floating point numbers are generated, one per call. We used the inline declaration `inline` to avoid the function call. Implementations without SIMD are written in INTERNATIONAL STANDARD ISO/IEC 9899 : 1999(E) Programming Language-C, Second Edition (which we shall refer to as C99 in the rest of this article), whereas those with SIMD use some standard SIMD extension of C99 supported by the Intel C compiler and GNU C Compiler.

Table 3 summarises the speed comparison using SIMD and Table 4 shows the speed comparison without using SIMD. The 64-bit MT is not listed in Table 3, because we do not have SIMD version. The first two lines list the CPU time (in seconds) needed to generate 10^8 floating point numbers, for a Pentium-M CPU. The first line lists the timings for the block-generation scheme, and the second line lists those for the sequential generation scheme. The result is that dSFMT is the fastest for all CPUs, all returning methods, using SIMD and without using SIMD. Table 5 shows the speed of other generators. Although dSFMT has 52-bit precision while the others have only 32-bit precision, dSFMT's sequential generation using standard C (i.e. the slowest case) is faster than the other generators, except xorshift128 [13], whose quality is reported to be questionable in [16].

11 Dimension of equidistribution

We calculated $d(v)$s for our generators, by using method described in §8. Table 6 lists the dimension defects $d(v)$ of dSFMT, for Mersenne exponent (mexp) $= 521, 1279, 2203, 4253, 11213, 19937$ and $v = 1, 2, \ldots, 52$. The $d(v)$ for $1 \leq v \leq 22$ are very small. The larger mexp seems to lead to the larger $d(v)$ for $v > 22$. Still, the case mexp=19937 has total dimension defect $\Delta = 2608$,

Table 3 The CPU time (sec.) for 10^8 generations using SIMD.

		dSFMT (new)	dSFMT-old (old)	MT mask	SFMT mask	SFMT $\times$ const
Pentium M	blk	0.626	0.867	1.526	0.928	2.636
1.4 Ghz	seq	1.422	1.761	3.181	2.342	3.671
Pentium 4	blk	0.254	0.640	0.987	0.615	3.537
3 Ghz	seq	0.692	1.148	3.339	3.040	3.746
core 2 duo	blk	0.199	0.381	0.705	0.336	0.532
1.83GHz	seq	0.380	0.457	1.817	1.317	2.161
Athlon 64	blk	0.362	0.637	1.117	0.623	1.278
2.4GHz	seq	0.680	0.816	1.637	0.763	1.623
PowerPC G4	blk	0.887	1.151	2.175	1.657	8.897
1.33GHz	seq	1.212	1.401	5.624	2.994	7.712

Table 4 The CPU time (sec.) for 10^8 generations (without using SIMD).

		dSFMT (new)	dSFMTold (old)	MT 64 mask	MT mask	SFMT mask	SFMT $\times$ const
Pentium M	blk	1.345	2.023	2.031	3.002	2.026	3.355
1.4 Ghz	seq	2.004	2.386	2.579	3.308	2.835	3.910
Pentium 4	blk	1.079	1.128	1.432	2.515	1.929	3.762
3 Ghz	seq	1.431	1.673	3.137	3.534	3.485	4.331
core 2 duo	blk	0.899	1.382	1.359	2.404	1.883	1.418
1.83GHz	seq	0.777	1.368	1.794	1.997	1.925	2.716
Athlon 64	blk	0.334	0.765	0.820	1.896	1.157	1.677
2.4GHz	seq	0.567	0.970	1.046	2.134	1.129	2.023
PowerPC G4	blk	1.834	3.567	2.297	4.326	4.521	12.685
1.33GHz	seq	1.960	2.865	4.090	5.489	5.464	9.110

Table 5 The CPU time (sec.) for 10^8 generations for other generators, where conversion to floating point numbers uses constant multiplication.

	WELL1024	WELL19937	MT19937	XORSHIFT128
Pentium M	2.076	2.876	2.028	1.233
Pentium 4	1.626	2.031	1.232	1.023
core 2 duo	1.165	1.913	1.032	0.653
Athlon 64	0.804	1.191	0.971	0.975
Power PC G4	2.947	7.524	3.082	2.267

which is smaller than 4188 of SFMT19937's 32-bit Δ and 6750 of MT19937's 32-bit Δ. Note that it is natural to guess that Δ increases at least proportionally to the word size b, by its definition (7).

Remark 2. The number of non-zero terms in $\chi_f(t)$ is an index measuring the amount of bit-mixing. The column "weight" in Table 7 shows these numbers: dSFMT19937 has the ratio $9756/19992 = 0.488$ which is higher than those of MT ($135/19937{=}0.00677$), WELL19937a ($8585/19937 = 0.431$) and WELL19937b ($9679/19937 = 0.485$).

Table 6 $d(v)$ $(1 \le v \le 52)$ of 52-bit fraction part of dSFMT.

	521	1279	2203	4253	11213	19937		521	1279	2203	4253	11213	19937
d(1)	0	1	0	0	4	0	d(27)	0	0	1	1	33	4
d(2)	0	1	1	0	0	1	d(28)	0	6	7	28	33	10
d(3)	0	2	1	0	0	1	d(29)	1	5	7	23	28	67
d(4)	0	0	0	0	1	1	d(30)	3	3	15	18	80	126
d(5)	0	0	0	0	0	0	d(31)	2	6	13	15	68	107
d(6)	0	1	1	0	1	0	d(32)	4	4	10	10	58	88
d(7)	0	0	0	0	0	1	d(33)	6	12	25	43	120	220
d(8)	0	0	0	0	0	1	d(34)	6	12	23	44	114	202
d(9)	0	1	0	0	0	0	d(35)	5	11	21	40	105	185
d(10)	1	0	0	0	0	0	d(36)	5	10	20	37	96	169
d(11)	0	0	0	0	0	0	d(37)	5	9	18	33	88	155
d(12)	0	0	0	0	0	0	d(38)	4	8	16	30	80	141
d(13)	0	0	0	0	0	0	d(39)	4	7	15	28	72	128
d(14)	0	0	0	0	0	1	d(40)	4	6	14	25	65	115
d(15)	0	0	0	0	0	1	d(41)	3	6	12	22	58	103
d(16)	0	0	0	0	0	1	d(42)	3	5	11	20	51	91
d(17)	0	0	0	0	0	0	d(43)	3	4	10	17	45	80
d(18)	0	0	0	0	0	0	d(44)	2	4	9	15	39	70
d(19)	0	0	0	0	0	0	d(45)	2	3	7	13	34	60
d(20)	1	0	0	0	0	0	d(46)	2	2	6	11	28	50
d(21)	0	0	0	0	7	0	d(47)	2	2	5	9	23	41
d(22)	0	0	0	0	0	134	d(48)	1	1	4	7	18	32
d(23)	0	0	7	16	22	94	d(49)	1	1	3	5	13	23
d(24)	0	1	3	9	19	58	d(50)	1	0	3	4	9	15
d(25)	0	1	0	6	7	25	d(51)	1	0	2	2	4	7
d(26)	0	0	0	0	0	0	d(52)	1	0	1	0	0	0
total dimension defect Δ								73	135	291	531	1423	2608

Table 7 The number of non-zero terms in $\chi_f(t)$

mexp	521	1279	2203	4253	11213	19937
degree of $\chi_f(t)$	544	1376	2208	4288	11256	19992
weight	273	673	1076	2233	5684	9756
ratio	0.50	0.49	0.49	0.52	0.50	0.49

The dSFMT generators passed the DIEHARD statistical tests [12]. They also passed TestU01 [11] consisting of 144 different tests, except for `LinearComp` (fail unconditionally) and MatrixRANK tests (fail if the size of dSFMT is smaller then the matrix size). These tests measure $\mathbb{F}_2$-linear dependency of the outputs, and reject $\mathbb{F}_2$-linear generators, such as MT, SFMT and WELL.

We shall keep the latest version of the codes in the web page [18].

Acknowledgements This study is partially supported by JSPS/MEXT Grant-in-Aid for Scientific Research No.19204002, No.18654021, and JSPS Core-to-Core Program No.18005. The second author is partially supported as a visiting professor of The Institute of Statistical Mathematics. The authors are thankful to the anonymous referees and the editor for valuable comments.

References

1. R.P. Brent and P. Zimmermann. Random number generators with period divisible by a Mersenne prime. In *Computational Science and its Applications - ICCSA 2003*, volume 2667, pages 1–10, 2003.
2. R.P. Brent and P. Zimmermann. Algorithms for finding almost irreducible and almost primitive trinomials. *Fields Inst. Commun.*, 41:91–102, 2004.
3. R. Couture, P. L'Ecuyer, and S. Tezuka. On the distribution of k-dimensional vectors for simple and combined Tausworthe sequences. *Math. Comp.*, 60(202):749–761, 1993.
4. Agner Fog. Pseudo random number generators. `http://www.agner.org/random/`.
5. M. Fushimi. Random number generation with the recursion $x_t = x_{t-3p} \oplus x_{t-3q}$. *Journal of Computational and Applied Mathematics*, 31:105–118, 1990.
6. IEEE standard for binary floating-point arithmetic 754, 2008. `http://ieeexplore.ieee.org/servlet/opac?punumber=4610933`.
7. D. E. Knuth. *The Art of Computer Programming. Vol.2. Seminumerical Algorithms.* Addison-Wesley, Reading, Mass., 3rd edition, 1997.
8. P. L'Ecuyer. Maximally equidistributed combined tausworthe generators. *Math. Comp.*, 65(213):203–213, 1996.
9. P. L'Ecuyer and J. Granger-Piché. Combined generators with components from different families. *Mathematics and Computers in Simulation*, 62:395–404, 2003.
10. P. L'Ecuyer and F. Panneton. $\mathbb{F}_2$-linear random number generators. In *Advancing the Frontiers of Simulation: A Festschrift in Honor of George Samuel Fishman*, pages 175–200, 2009. C. Alexopoulos, D. Goldsman, and J. R. Wilson Eds.
11. P. L'Ecuyer and R. Simard. TestU01: A C library for empirical testing of random number generators. *ACM Transactions on Mathematical Software*, 15(4):346–361, 2006.
12. G. Marsaglia. The Marsaglia Random Number CDROM, with the DIEHARD Battery of Tests of Randomness. Department of Statistics, Florida State University, (1996) `http://www.stat.fsu.edu/pub/diehard/`.
13. G. Marsaglia. Xorshift RNGs. *Journal of Statistical Software*, 8(14):1–6, 2003.
14. M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Trans. on Modeling and Computer Simulation*, 8(1):3–30, January 1998. `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html`.
15. T. Nishimura. Tables of 64-bit mersenne twisters. *ACM Trans. on Modeling and Computer Simulation*, 10(4):348–357, October 2000.
16. F. Panneton and P. L'Ecuyer. On the Xorshift random number generators. *ACM Transactions on Modeling and Computer Simulation*, 15(4):346–361, 2005.
17. F. Panneton, P. L'Ecuyer, and M. Matsumoto. Improved long-period generators based on linear reccurences modulo 2. *ACM Transactions on Mathematical Software*, 32(1):1–16, 2006.
18. M. Saito and M. Matsumoto. SFMT Homepage. `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/index.html`.
19. M. Saito and M. Matsumoto. SIMD-oriented fast Mersenne twister : a 128-bit pseudorandom number generator. In *Monte Carlo and Quasi-Monte Carlo Methods 2006*, LNCS, pages 607–622. Springer, 2008.
20. SIMD From Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/SIMD`.