Dynamic Creation of Pseudorandom Number Generators

Makoto Matsumoto[⋆] and Takuji Nishimura[⋆⋆]

Department of Mathematics, Keio University, 3-14-1, Hiyoshi, Kohoku-ku, Yokohama, 223-8522, Japan

Abstract. We propose a new scheme *Dynamic Creation* (DC) of pseudorandom number generators (PRNG) for large scale Monte Carlo simulations in parallel or distributed systems. DC receives user's specification such as word size, period, size of working area, together with a process ID (or a set of IDs). Then DC creates a PRNG (or a set of PRNGs, respectively) satisfying the specification, so that ID number is encoded in the characteristic polynomial of PRNG. Thus, different IDs assure highly independent PRNGs. Each PRNG is a small Mersenne Twister, which we proposed previously.

1 Introduction

Nowadays, large scale Monte Carlo simulations like those in nuclear physics or financial management become popular, and the necessity of random number generation in the parallel machines or distributed systems is increasing. A question is how to generate random numbers in such machines. The usual scheme for PRNG in parallel machines is to use one and the same PRNG for every process, with different initial seeds. However this procedure may yield bad collision, in particular if the generator is based on a linear recurrence, because in this case the sum of two pseudorandom sequences satisfies the same linear recurrence and may appear in the third sequence. The danger becomes non-negligible if the number of parallel streams becomes large compared to the size of the state space.

Here we propose a new scheme named *Dynamic Creation* (DC) of PRNGs. DC receives processor ID or process ID or machine ID, according to the application, and creates a PRNG with the ID encoded in the characteristic polynomial of the PRNG, so that different ID assures relatively prime characteristic polynomial. Thus, different IDs imply highly independent PRNGs. For example, in a parallel machine, each processor has its ID, and we assign different PRNG to each processor by DC with its ID. Similar method can be applied for process ID, too.

DC has another merit, i.e., user specification is possible. We proposed Mersenne Twister (MT) random number generator in [5], and it is now widely used. Many users sent emails with various requirements on MT, like 31-bit

[⋆] email: `matumoto@math.keio.ac.jp`

[⋆⋆] email: `nisimura@comb.math.keio.ac.jp`

version, 8-bit version, smaller working area, etc. DC automatically answers to these requirements, since it receives word size, memory size, and creates a MT with that specification.

2 Mersenne Twister

Here we shall briefly explain about MT. MT is a pseudorandom number generating algorithm with following properties;

- long period
- good k-distribution property
- efficient use of memory
- high speed.

An implementation in C language mt19937.c [1] has the following records.

- period $2^{19937}-1$
- 623-dimensionally equidistributed to 32-bit accuracy
- consumes 624 words of 32 bits.
- about four times faster than `rand()` in usual C.

MT generates the vectors of word size by the recurrence

$$\mathbf{x}_{k+n} := \mathbf{x}_{k+m} + (\mathbf{x}_k^u | \mathbf{x}_{k+1}^l)A, \quad (k = 0, 1, \cdots). \tag{1}$$

Here, $n > m$ are fixed positive integers, $(\mathbf{x}_k)_{k=0,1,\ldots}$ is a sequence of w-dimensional row vectors over the two element field $\mathbb{F}_2 = \{0,1\}$, $(\mathbf{x}_k^u|\mathbf{x}_{k+1}^l)$ is the w-dimensional vector obtained by concatenating the left $w-r$ bits of $\mathbf{x}_k$ and the right r bits of $\mathbf{x}_{k+1}$ (u for upper, l for lower). By multiplying a matrix A (called *twister*) from right, we get $(\mathbf{x}_k^u|\mathbf{x}_{k+1}^l)A$. Every arithmetic operation is modulo 2, i.e., this is a linear recurrence of vectors in the two element field $\mathbb{F}_2$. Since A should be chosen to be quickly computable, we proposed the form called *companion matrix*:

$$A = \begin{pmatrix} & 1 & & & \\ & & 1 & & \\ & & & \ddots & \\ & & & & 1 \\ a_0 & a_1 & \cdots & \cdots & a_{w-1} \end{pmatrix}.$$

For such A, $\mathbf{x}A$ can be calculated by

$$\mathbf{x}A = \begin{cases} \text{if (the least significant bit of } \mathbf{x}) = 0 \text{ then } \mathbf{x} \leftarrow \text{shiftright}(\mathbf{x}), \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \text{else } \mathbf{x} \leftarrow \text{shiftright}(\mathbf{x}) \oplus \mathbf{a}, \end{cases}$$

where $\mathbf{a} = (a_0, a_1, \ldots, a_{w-1})$.

[1] This code is available via `http://www.math.keio.ac.jp/matumoto/emt.html`

DC embeds the given ID into a part of this vector **a**. DC searches for **a** so that the period attains the maximal value $2^{nw-r}-1$, where this number should be a prime (i.e. so-called Mersenne prime). Then, search for a matrix T (called tempering[4]) such that the sequence $\mathbf{x}_n T$ has good higher dimensional distribution for most significant bits.

3 Dynamic Creation

3.1 What is Dynamic Creator?

DC is a program which

1. receives **users' specification**, i.e., word size, period, etc.
2. receives **ID** (process ID, machine ID, etc)
3. creates a (set of parameters for) MT such that
   (a) satisfying users' specification,
   (b) ID is encoded in a parameter of MT so that different ID assures essentially different[2] PRNG (i.e., the characteristic polynomial of the recurring sequence is irreducible and distinct to each other).

3.2 A Hypothesis on Independence of PRNGs

We used a hypothesis that a set of PRNGs based on linear recurrences is mutually "independent" if the characteristic polynomials are relatively prime to each other.

There is no mathematically rigorous proof on this hypothesis, but many researchers on PRNG would agree with this. For example if $(x_i)_{i\in\mathbb{N}}$ has characteristic polynomial $f(t)$, $(y_i)_{i\in\mathbb{N}}$ has $g(t)$, and if f and g are coprime, then (x_i+y_i) has characteristic polynomial $f(t)g(t)$. Thus an immediate correlation like $x_i+y_i=x_{i+1}$ can never happen.

An ideal method is to select a different theory of random number generation for each process, (e.g. LCG for one, MT for another, etc), but it is difficult since we have only a few theories.

Second best strategy would be PRNGs with relatively prime periods, but is still difficult by the same reason. So we compromise by listing relatively prime characteristic polynomials. At least, this strategy seems to be safer than the merely changing initial values, or merely changing c in the LCG $x_n = ax_{n-1}+c \bmod M$ for each generator, since if $y_n = ay_n + c' \bmod M$ then $(x_n+y_n) = a(x_n+y_n)+(c+c') \bmod M$ may coincide with a third generator.

[2] under the hypothesis in §3.2

3.3 Encoding ID in the Parameter

Recall that MT has a vector parameter $\mathbf{a}$. One can prove that the characteristic polynomial $\varphi_{\mathbf{a}}(t)$ of the linear recurrence of MT is distinct if $\mathbf{a}$ is distinct [5, §A.1] (i.e. $\mathbf{a} \mapsto \varphi_{\mathbf{a}}(t)$ is injective, for fixed w, n, m).

DC embeds ID into $\mathbf{a}$. A problem is that not all $\mathbf{a}$ satisfy the maximal period $2^{nw-r}-1$. Those $\mathbf{a}$ with irreducible $\varphi_{\mathbf{a}}(t)$ attain the maximal period. So we use half of the word $\mathbf{a}$ as ID, and search for a maximal periodic parameter by changing the rest part of $\mathbf{a}$. DC receives $w/2$-bit integer as an ID, then searches for $\mathbf{a}$ giving irreducible $\varphi_{\mathbf{a}}$ with lower half word fixed to that ID. We change the higher half word of $\mathbf{a}$ deterministically (i.e. pseudorandomly) and search for an irreducible $\varphi_{\mathbf{a}}(t)$ until we find one. Note that this will not work for small w.

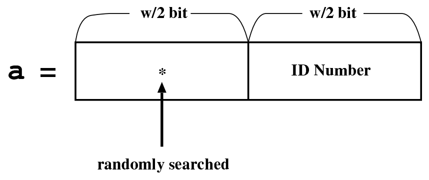

Fig. 1. Encoding ID in the parameter

After finding $\mathbf{a}$, DC searches for a tempering matrix T. For the tempering matrix $\mathbf{x} \mapsto \mathbf{z} = \mathbf{x}T$, we choose the following successive transformations [4]

$$\mathbf{y} := \mathbf{x} \oplus (\mathbf{x} >> u) \tag{2}$$
$$\mathbf{y} := \mathbf{y} \oplus ((\mathbf{y} << s) \text{ AND } \mathbf{b}) \tag{3}$$
$$\mathbf{y} := \mathbf{y} \oplus ((\mathbf{y} << t) \text{ AND } \mathbf{c}) \tag{4}$$
$$\mathbf{z} := \mathbf{y} \oplus (\mathbf{y} >> l), \tag{5}$$

where l, s, t, and u are integers, $\mathbf{b}$ and $\mathbf{c}$ are suitable bitmasks of word size, and $(\mathbf{x} >> u)$ denotes the u-bit shiftright ($(\mathbf{x} << u)$ the u-bit shiftleft).

Usually u, s, t, l are fixed, and $\mathbf{b}$, $\mathbf{c}$ are searched so that they optimize the k-distribution property. Here, k-distribution is one of the strongest measures of the quality of PRNGs defined as follows.

Definition 1. A periodic sequence $x_0, x_1, x_2, \ldots \in [0,1]$ of period P is said to be *k-distributed to v-bit accuracy* if the points $(x_i, x_{i+1}, \ldots, x_{i+k-1}) \in [0,1]^k$ $(i = 0, 1, \ldots, P-1)$ are uniformly dense in $[0,1]^k$ up to v-bit accuracy.

By $k(v)$, we denote the maximum such k for a given v. Note the trivial upper bound $k(v) \leq \lfloor \log_2(P)/v \rfloor$. Tempering parameters $\mathbf{b}$, $\mathbf{c}$ are searched so that $k(v)$ comes near to this bound for every v.

3.4 Experiments on Dynamic Creation

We implemented the idea of DC in a C-code, and measured the speed using Sun Ultra 1 (Solaris 2.5, gcc-2.7.2).

Distribution of Time for Finding One MT Fix $p = 521, n = 17, m = 9, r = 23, w = 32$. We have $65536 = 2^{16}$ different IDs, and DC finds a corresponding $\mathbf{a}$ with maximal periodicity for each ID. We measured the time to find a parameter for each ID, and made a distribution table. (We didn't temper here.) Figure 2 shows that the distribution seems to be exponential,

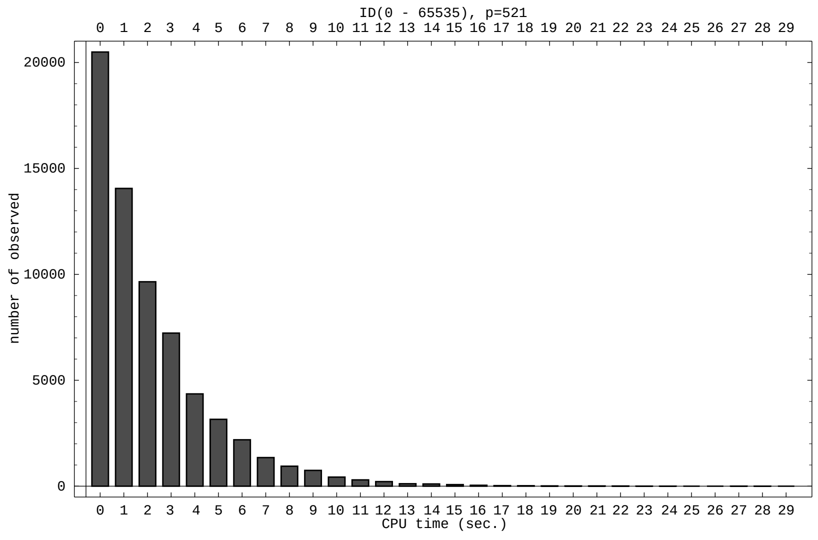

Fig. 2. Distribution of CPU time for finding one MT (p=521)

and it would be difficult to expect exactly how much time will be necessary to create one MT in advance. Same data are listed in Table 1. At worst, it takes 28 seconds to find one MT. Average time is 2.56 seconds. For tempering, it takes about 5 seconds independently of $\mathbf{a}$.

Average Time for Different Periods Figure 3 shows the average CPU time to find a MT of period $2^p - 1$, for $p = 127, 521, \ldots, 4423$, i.e. all the Mersenne exponents between 100 and 5000. (We didn't temper here.) Same data are in Table 2.

time(sec.)	0	1	2	3	4	5	6	7	8	9
observed	20495	14055	9649	7227	4356	3156	2190	1350	943	745
time(sec.)	10	11	12	13	14	15	16	17	18	19
observed	429	297	216	115	105	74	43	26	20	12
time(sec.)	20	21	22	23	24	25	26	27	28	
observed	10	10	7	2	1	0	0	2	1	

Table 1. Distribution of CPU time for finding one MT (p=521)

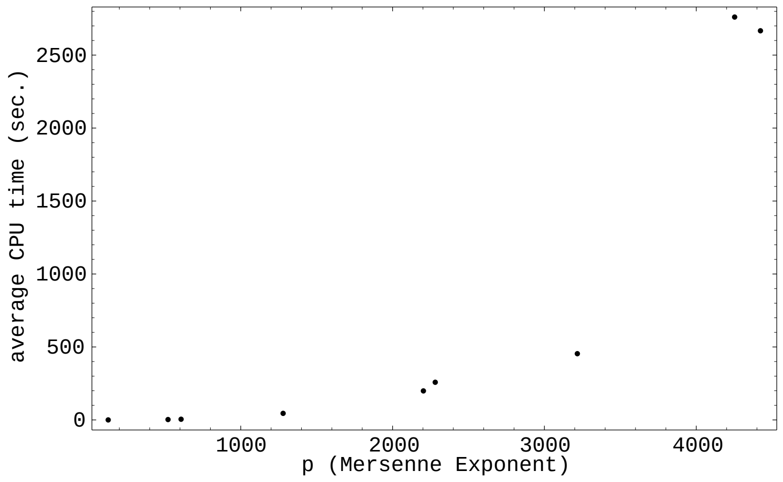

Fig. 3. Average CPU time to find one MT(various p)

p	127	521	607	1279	2203	2281	3217	4253	4423
average(sec.)	0.045	2.56	4.37	44.96	198.86	258.09	453.91	2760.72	2666.74
number of searched	100	100	100	100	50	50	10	10	10

Table 2. Average CPU time to find one MT(various p)

Gap between the Realized k-Distribution and the Theoretical Upper Bound We chose 1000 different **a** for $p = 521$ randomly. After tempering[3], we measure its k-distribution $k(v)$ and show the gap from the theoretical upper bound. For each v, we show the worst (i.e. smallest $k(v)$) among 1000 MT. A tendency is that for $v = 4, 8, 11, 15$, $k(v)$ is not close to the bound. Same data are in Table 3.

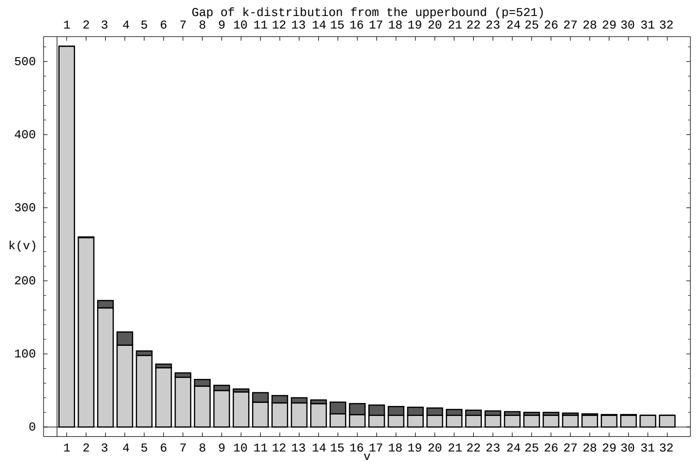

Fig. 4. Maximum gap of $k(v)$ from the bound ($p = 521$)

v	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
defect	0	1	10	18	6	5	6	9	7	4	13	10	7	5	16	15
v	17	18	19	20	21	22	23	24	26	25	27	28	29	30	31	32
defect	14	12	11	10	8	7	6	5	4	4	3	2	1	1	0	0

Table 3. Table of maximum gap of $k(v)$ from the bound (p = 521)

[3] In [4][5], our tempering strategy is back-tracking to obtain nearly optimal $k(v)$, but here in DC, we adopted more a greedy algorithm, i.e. we obtain tempering parameter from $v = 1, 2, ..., i$, then $v = i + 1$ tempering parameter is searched so that $k(i + 1)$ attains the maximal for the possible tempering parameters, with fixed parameter for $v = 1, ..., i$. Thus, in DC, tempering is faster but more coarse than that of [5].

4 Why we could do this? ...Due to Mathematics on $\mathbb{F}_2$

There is a slogan in number theory :

$\mathbb{Z}$ and $\mathbb{F}_2[t]$ are alike, but the latter is much easier.

4.1 LCG-type and $\mathbb{F}_2$-type Generators

There are two classes in linear recurrence for PRNG. Linear Congruential Generator (LCG-type) generates an integer sequence $x_1, x_2, \ldots,$ by a recurrence

$$x_n \equiv a \cdot x_{n-1} \bmod M.$$

For a fixed M, we search for a large order a in the multiplicative group modulo M. LCG-type generators including SWB use *integer arithmetic.*

$\mathbb{F}_2$-type Generator (e.g. MT, GFSR, Tausworthe) generates a sequence of polynomials $x_1(t), x_2(t), \ldots, \in \mathbb{F}_2[t]$ by a recurrence

$$x_n(t) \equiv t \cdot x_{n-1}(t) \bmod M(t)$$

with high degree ($=: p$) $M(t)$. For fixed p , we search for $M(t)$ making the order of t large. $\mathbb{F}_2$-type Generators use *polynomial arithmetic.*

4.2 Transforming the Lattice Structure

Both LCG-type and $\mathbb{F}_2$-type generators have “lattice structure” in higher dimension. Strongest tests on PRNGs seem to be those which measure defects in the lattice structure in k-dimension (for $k = 1, 2, 3, \cdots$). For LCG-type, it is the spectral test[2]. For $\mathbb{F}_2$-type, it is the k-distribution test [8].

To improve the lattice structure, we may transform the output by a function f, i.e, if $x_1, x_2, \cdots$ are the original outputs, then use $f(x_1), f(x_2), \cdots$ instead. But to apply these tests, f must preserve the algebraic structure. Thus, for LCG-type, the test requires f to be

$$f \in \text{ Aut } (\mathbb{Z}/M\mathbb{Z}) \cong (\mathbb{Z}/M\mathbb{Z})^\times,$$

but in this case f does not change the lattice structure. So, this transformation is meaningless. Thus, even if we found an a with large order, we must discard it if the lattice structure is not good. For $\mathbb{F}_2$-type, the test requires

$$f \in \text{ Aut }_{\mathbb{F}_2}(\mathbb{F}_2[t]/(M(t))) \cong GL_p(\mathbb{F}_2),$$

where p is degree of $M(t)$, and f may and do change the lattice structure.

For each $M(t)$, we can select an f which nearly optimizes the lattice structure, i.e., tempering[4]. Thus, we can obtain MT with good k-distribution for every irreducible $M(t)$, by selecting f. Note that tempering does not change the characteristic polynomial of the recurrence. This is why we can encode

ID in $M(t)$. (If we try to encode ID in a in LCG, the probability to find a with large order and good lattice structure would be too small.)

Also, k-distribution test is faster than spectral test. k-distribution test for MT is fast because of Couture-Tezuka-L'Ecuyer's lattice method [1] [6] [7] using Lenstra's algorithm [3] (c.f. spectral tests in ≥ 500 dimension is nearly impossible).

5 Remark

A referee commented the following questions about DC.

In [9], it is claimed that, for shift register-type pseudorandom number generators, the decimated sequence $(x_{id})_{i\in\mathbb{N}}$ with d fixed should also have good k-distribution property for up to some large d. In this respect, there arise the following questions about DC:

- How is the k-distribution of the decimated sequence obtained from DC?
- How to guarantee that decimated sequences do not follow trinomial recurrence relation for, say, $d < 1000$.
- Denote two different sequences produced from DC by $(x_i)_{i\in\mathbb{N}}$ and $(y_i)_{i\in\mathbb{N}}$. How to avoid the danger that the decimated sequence (x_{di}) has the same linear recurrence as (y_i)?

Presently, we have not checked k-distribution of the decimated sequence, and actually we don't have an efficient scheme to check it. There are possible collisions among decimated sequences and trinomials, but we consider the probability is negligibly low when we use irreducible polynomials of degree, say, 521.

6 Conclusion

We proposed Dynamic Creator (DC) of PRNGs, for

1. admitting user's specification (word-size, period, etc...)
2. multi-processor or distributed systems.

DC receives ID-number, and encodes it in PRNG, so that different-ID PRNGs are essentially different. Creation of PRNGs is deterministic and reproducible. Simplicity of the number theory in $\mathbb{F}_2[t]$ makes this method possible.

A Implementation of DC and its Usage

This section explains an implementation of DC in a C-code and its usage.

A.1 Implementation of DC

The struct `mt_struct` (Table 4) has all the information to reconstruct a MT, including the state vector. The parameters for an MT are stored in the struct. Since the state vector is an array of variable size, `mt_struct` stores a pointer to an array, namely `mt_struct->state`.

```
typedef unsigned int uint32;
typedef struct {
        uint32 aaa;
        int mm,nn,rr,ww;
        uint32 wmask,umask,lmask;
        int shift0, shift1, shiftB, shiftC;
        uint32 maskB, maskC;
        int i;
        uint32 *state;
}mt_struct;
```

Table 4. Definition of `mt_struct`

Main ingredients in this package are the following five functions.

- `mt_struct *get_mt_parameter(int w, int p)`
- `mt_struct *get_mt_parameter_id(int w, int p, int id)`
- `mt_struct **get_mt_parameters(int w, int p, int max_id)`
- `void sgenrand_mt(uint32 seed, mt_struct *mts)`
- `uint32 genrand_mt(mt_struct *mts)`

`get_mt_parameter(w, p)` gets two arguments; `w` is the word size of PRNG's output, and `p` is a Mersenne exponent, i.e, a prime number such that $2^p - 1$ is a prime. The created PRNG has period $2^p - 1$. This function deterministically randomly searches for a set of parameters for each call, and if it finds one, allocates the working area for the created MT, makes a `mt_struct`, and returns the pointer to the created struct. This pointer is used to generate a random number by `genrand_mt(mts)`, where `mts` is a pointer to a `mt_struct`, and `genrand_mt(mts)` returns a pseudorandom integer in $[0, 2^w - 1]$, where w is specified word size in `mts`. Thus, for example,

```
mt_struct *mts;
long int x,y;
mts = get_mt_parameter(31,521);
x = genrand_mt(mts);
y = genrand_mt(mts);
```

stores a set of data of one MT in `mts`, then assigns two different random integers of 31-bit in `x` and `y`. To be precise, we need to call `sgenrand_dc(seed)` once, which initializes a pseudorandom number generator used internally in the dynamic creator, before calling `get_mt_parameter()`. Also, we need to call `sgenrand_mt(seed, mts)` once before calling `genrand_mt(mts)` to initialize the MT associated with `mts`. The seed is unsigned 32-bit integer, which is not zero. In an application one may want to have more variety for the initial seed. In this case, one may use the whole array `mts->state[0..n-1]` as the (non zero) initial value, expect that the least significant r bits of `mts->state[0]` is neglected.

The crux of DC is its deterministic nature, i.e, if we use the same seeds, the result of computation is independent of the machine, parallelism, etc. That is, if `mts1` and `mts2` are two different parameters, then `genrand_mt(mts1)` and `genrand_mt(mts2)` can run in parallel. In particular,

```
    x = genrand_mt(mts1);
    y = genrand_mt(mts2);
```

will give the same result with

```
    y = genrand_mt(mts2);
    x = genrand_mt(mts1);   .
```

Thus DC sustains the reproducibility of PRNG even in parallel machines.

`get_mt_parameter_id(w, p, id)` gets one more argument `id`, which is a 16-bit integer, so that different id assures different irreducible characteristic polynomial.

`get_mt_parameters(w, p, max_id)` gets another argument `max_id` and creates a bunch of MTs, i.e, an array of `mt_struct` of size `max_id`. It returns a pointer to the array of `mt_struct`. So, if we execute

```
    mt_struct **mtss;
    mtss = get_mt_parameters(31,521,255);
```

then each of `mtss[0]`, ..., `mtss[255]` becomes a pointer to an `mt_struct`, and `genrand_mt(mtss[0])`, ..., `genrand_mt(mtss[255])` give 256 independent pseudorandom number streams.

Note The functions `get_mt_parameter()`, `get_mt_parameter_id()` and `get_mt_parameters()` will return the value `NULL` if it could not find an appropriate set of parameters.

A.2 Simple Example for Usage

An Example Code for `get_mt_parameter()` Table 5 shows a C-code using `get_mt_parameter()`. It creates an MT of word size 31 and period $2^{521}-1$, and then prints the first 100 outputs. The header file `dc.h` included there contains the definition of struct and functions for the dynamic creation package.

```
#include <stdio.h>
#include "dc.h"

int main()
{
        int i;
        mt_struct *mts;

        sgenrand_dc(1111);

        mts = get_mt_parameter(31,521);
        if (mts == NULL) exit(1);

        sgenrand_mt(32437, mts);
        for (i=0; i<100; i++) {
                printf ("%8x\n", genrand_mt(mts));
        }

        return 0;
}
```

Table 5. A simple example for usage of `get_mt_parameter()`

An Example Code for `get_mt_parameters()` Table 6 shows a C-code which finds 256 different MTs of word size 31 and period $2^{521}-1$. `mtss[i]` has distinct parameters and state vector for $i = 0, \cdots, 255$. Then `genrand_mt(mtss[i])` generates a pseudorandom integers for each call, and prints first 100 outputs for each generator. `genrand_mt(mtss[i])` for distinct `i` can run completely in parallel.

Remark This package will be delivered in the Mersenne Twister home page, when it is ready.

Acknowledgments

The authors wish to thank the anonymous referees for the valuable comments.

```
#include <stdio.h>
#include "dc.h"

int main()
{
        int i,j;
        unsigned long seed;
        mt_struct **mtss;

        sgenrand_dc(1111);
        mtss = get_mt_parameters(31,521,0xff);
        if (mtss == NULL) exit(1);

        for (i=0; i<=0xff; i++) {
                do { seed = genrand_dc();} while(seed == 0);
                sgenrand_mt((uint32)seed,mtss[i]);
                for (j=0; j<100; j++) {
                        printf ("%8x ", genrand_mt(mtss[i]));
                        if (j%5 == 4) printf("\n");
                }
        }

        return 0;
}
```

Table 6. A simple example for usage of `get_mt_parameters()`

References

1. Couture, R., L'Ecuyer P., Tezuka, S.: On the distribution of k-dimensional vectors for simple and combined Tausworthe sequences, Math. Comput., **60** (1993) 749–761
2. Knuth, D.E.: Seminumerical Algorithms. 3rd ed., Vol 2, The Art of Computer Programming, Addison Wesley, Reading, MA., (1997)
3. Lenstra, A.K.: Factoring multivariate polynomials over finite fields, J. Comput. Syst. Sci., **30** (1985) 235–248
4. Matsumoto, M., Kurita, Y.: Twisted GFSR generators II, ACM Trans. Model. Comput. Simul., **4** (1994) 254–266
5. Matsumoto, M., Nishimura, T.: Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator, ACM Trans. Model. Comput. Simul., **8** (1998) 3–30
6. Tezuka, S.: Lattice structure of pseudorandom sequence from shift register generators, Proceedings of the 1990 Winter Simulation Conference, IEEE (1990) 266–269
7. Tezuka, S.: The k-dimensional distribution of combined GFSR sequences, Math. Comput., **62** (1994) 809–817
8. Tezuka, S.: Uniform Random Numbers: Theory and Practice, Kluwer, (1995)
9. Tootill, J.P.R., Robinson, W.D., Adams, A.G.: The Runs Up-and-Down Performance of Tausworthe Pseudo-Random Number Generators, J. ACM, **18** (1971), 381–399