Construction of a nonrecursive 64-bit pseudorandom number generator based on beta transformations on [1,2)

Hirotake YAGUCHI*

[1,2) 上のベータ変換に基づく非再帰型６４ビット擬似乱数の構成

谷　口　礼　偉

Abstract

A beta transformation (β-transformation) on [1,2) is a function $M_\beta : [1,2) \to [1,2)$ defined by $M_\beta(t) = \beta t - \lfloor \beta t \rfloor + 1$, $\beta > 1$. In this paper we construct a nonrecursive 64-bit pseudorandom number generator SSI64rand using $M_\beta(t)$ and test the randomness of SSI64rand by applying various statistical tests.

§1. Introduction

In [4] they constructed a nonrecursive pseudorandom number generator SSI32rand using a modified algorithm of SSR32rand whose origin is cancellation error of numerical computations. However a mathematical description of the algorithm of SSI32rand was not investigated there. Recently in [6] they constructed a nonalgebraic hash function called MB$n$hash based on beta transformations $M_\beta(t)$ on [1,2). There, a relation of algorithms between beta transformations $M_\beta(t)$ and SSI32rand was only indicated but not stated. So in this paper, based on $M_\beta(t)$, we construct a nonrecursive 64-bit pseudorandom number generator SSI64rand whose algorithm is the same as SSI32rand except for the bit size, and give a mathematical description of the algorithm of SSI64rand precisely.

*Department of Mathematics, Mie University, Tsu City, 514-8507, Japan. (E-mail: yaguchi@edu.mie-u.ac.jp) This work was supported by JSPS KAKENHI Grant Number 23500086.

In the next section we give algorithm and implementation of SSI64rand, and in §3 test the randomness of SSI64rand by applying NIST's suite of testing randomness [7] and TestU01 by L'Ecuyer [1].

§2. Algorithm and implementation of SSI64rand

2.1. Algorithm of SSI64rand. Let $M_\beta : [1,2) \to [1,2)$ be a beta transformation on [1,2) defined by $M_\beta(t) = \beta t - \lfloor \beta t \rfloor + 1$, $\beta > 1$, where $\lfloor \beta t \rfloor$ is the largest integer not exceeding βt. (We often adopt the notation appeared in [6].) Let $1.e$ and $1.\pi$ be numbers defined by $1.27181\cdots$ and $1.31415\cdots$ respectively. (The notation e and π implies Napier's constant and the mathematical constant pi respectively.) We first use $1.e$. Let the binary representation of $1.e$ be $1.\hat{r}_1\hat{r}_2\cdots\hat{r}_{63}\cdots$. For a given nonnegative 63-bit integer $\nu_1\nu_2\cdots\nu_{63}$ we make a number x in [1,2) from $1.e$ by

$$x = 1.(\hat{r}_1 \oplus \nu_1)(\hat{r}_2 \oplus \nu_2)\cdots(\hat{r}_{63} \oplus \nu_{63})\hat{r}_{64}\hat{r}_{65}\cdots, \tag{1}$$

where $\oplus$ is the logical operation XOR (exclusive or). We write (1) such as $x = (1.e) \oplus (\nu_1\nu_2\cdots\nu_{63})$ for short. Let w_0 be a fixed number in [1,2). Then we compute

$$u \equiv M_{2^5}^{16}{}_x(w_0) = 1.\check{b}_7\check{b}_8\cdots\check{b}_{127}\check{b}_{128}\cdots \quad \text{(in binary)}. \tag{2}$$

If we put $w_0 = x$ and take out $\check{b}_{32}\check{b}_{33}\cdots\check{b}_{95}$ as a 64-bit random number from u, this algorithm is the same as MB32rand in [6] except for the bit size. However in SSI64rand, in order to increase randomness, we further compute

$$y = (1.\pi) \oplus (\tilde{\nu}_1\tilde{\nu}_2\cdots\tilde{\nu}_{63}) \quad \text{and} \quad v \equiv M_{2^5}^{16}{}_y(\tilde{w}_0) = 1.\tilde{b}_7\tilde{b}_8\cdots\tilde{b}_{127}\tilde{b}_{128}\cdots$$

for $\tilde{w}_0$ and $\tilde{\nu}_1\tilde{\nu}_2\cdots\tilde{\nu}_{63}$ which are different from w_0 and $\nu_1\nu_2\cdots\nu_{63}$ respectively. Let $\hat{b}_0\hat{b}_7\hat{b}_8\cdots\hat{b}_{127}$ be the result of binary subtraction $1\check{b}_7\check{b}_8\cdots\check{b}_{127} - 1\tilde{b}_7\tilde{b}_8\cdots\tilde{b}_{127}$. Then as a 64-bit random number ζ we take out $\hat{b}_{32}\hat{b}_{33}\cdots\hat{b}_{95}$. To make a sequence of random numbers $\{\zeta_k\}_{k=0,1,2,\cdots}$, we use

$$(\texttt{0x39f750241c2d5d33}) \times k \mod \texttt{0x7ffffffffffffe7}, \tag{3}$$

$$(\texttt{0x32f50fee9b2a32bb}) \times k \mod \texttt{0x7ffffffffffff5b} \tag{4}$$

as the k-th $\nu_1\nu_2\cdots\nu_{63}$ and $\tilde{\nu}_1\tilde{\nu}_2\cdots\tilde{\nu}_{63}$ respectively. (`0x` means "in the hexadecimal representation".) So x and y in $M_{2^5}^{16}{}_x(w_0)$ and $M_{2^5}^{16}{}_y(\tilde{w}_0)$ are

$$x_k = (1.e) \oplus (\nu_1\nu_2\cdots\nu_{63}) \quad \text{and} \quad y_k = (1.\pi) \oplus (\tilde{\nu}_1\tilde{\nu}_2\cdots\tilde{\nu}_{63})$$

respectively. (Of course all integers in (3) and (4) are prime numbers; and so the period of SSI64rand is approximately $2^{126} \approx 10^{38}$.) Numbers w_0 and $\tilde{w}_0$ are used to change a sequence of SSI64 random numbers. As the initial values we use $w_0 = 1.e$ and $\tilde{w}_0 = 1.\pi$.

2.2. Implementation of SSI64rand. We implement the algorithm of SSI64rand in the following 64-bit number system:

1) a number $1.b_1b_2\cdots b_{63}b_{64}\cdots$ in [1,2) is rounded to 64-bit $1.b_1b_2\cdots b_{63}$,
2) a multiplication of 64-bit numbers $1.t_1t_2\cdots t_{63}$ and $1.x_1x_2\cdots x_{63}$ is performed precisely and yields a 128-bit number $\check{b}_0\check{b}_1.\check{b}_2\check{b}_3\cdots\check{b}_{127}$ with $\check{b}_0\check{b}_1 = 01$ or $\check{b}_0 = 1$ depending on whether the result is in [1,2) or [2,4) respectively,
3) a 64-bit number $1.b_sb_{s+1}\cdots b_{s+62}$ is taken out from a result $b_0b_1.b_2b_3\cdots b_{127}$ of multiplication. (s depends on context and $s = 7$ for $M_{2^5x}(t)$.)

It is convenient for us to identify a 64-bit number $1.b_1b_2\cdots b_{63}$ and a 128-bit number $\check{b}_0\check{b}_1.\check{b}_2\check{b}_3\cdots\check{b}_{127}$ with integers $1b_1b_2\cdots b_{63}$ and $\check{b}_0\check{b}_1\check{b}_2\check{b}_3\cdots\check{b}_{127}$ respectively. Hence below we do it. ($\rightarrow$ Numbers $1.e$ and $1.\pi$ are identified with

`0xa2cb4411ba257552` and `0xa8365eed39e1c070`

respectively.) Since $M_{2^5x}(t)$ is calculated such as

$$\begin{aligned} M_{2^5x}(t) &= (2^5x)t \bmod [1,2) \ = (2^5)(xt) \bmod [1,2) \\ &= 2^5 \times \check{b}_0\check{b}_1 \ . \ \check{b}_2\check{b}_3\check{b}_4\check{b}_5\check{b}_6\check{b}_7\check{b}_8\cdots \bmod [1,2) \\ &= \check{b}_0\check{b}_1\check{b}_2\check{b}_3\check{b}_4\check{b}_5\check{b}_6 \ . \ \check{b}_7\check{b}_8\cdots \bmod [1,2) \\ &= 1 \ . \ \check{b}_7\check{b}_8\cdots, \end{aligned}$$

a computation of $M_{2^5x}(t)$ under our 64-bit integer system is performed such as

$$\begin{matrix} 1x_1x_2\cdots x_{63} \\ 1t_1t_2\cdots t_{63} \end{matrix} \overset{mul}{\longrightarrow} \check{b}_0\check{b}_1\check{b}_2\cdots\check{b}_5\check{b}_6\check{b}_7\cdots\cdots\check{b}_{127} \overset{shift6}{\longrightarrow} \check{b}_6\check{b}_7\cdots\check{b}_{69}\cdots\check{b}_{127}$$

$$\overset{OR}{\longrightarrow} 1\check{b}_7\cdots\check{b}_{69}\cdots\check{b}_{127} \overset{cut}{\longrightarrow} 1\check{b}_7\check{b}_8\cdots\check{b}_{69}.$$

Suppose $M^{16}_{2^5x_n}(1.e)$ and $M^{16}_{2^5y_n}(1.\pi)$ are identified with $1\check{b}_7\check{b}_8\cdots\check{b}_{127}$ and $1\tilde{b}_7\tilde{b}_8\cdots\tilde{b}_{127}$ respectively. The n-th SSI64 random number ζ_n is the 64 bits $\hat{b}_{32}\hat{b}_{33}\cdots\hat{b}_{95}$ in $\hat{b}_0\hat{b}_7\hat{b}_8\cdots\hat{b}_{127}$ which is the result of binary subtraction $1\check{b}_7\check{b}_8\cdots\check{b}_{127} - 1\tilde{b}_7\tilde{b}_8\cdots\tilde{b}_{127}$. For example the first and the second SSI64 random numbers are $0x8eaafb19f73587f8$ and $0x4bb2533b46fb5cf1$ respectively. In the appendix we give a code of computing an SSI64 random number.

§3. Randomness of SSI64rand

We tested the randomness of random numbers generated by SSI64rand by applying NIST's statistical test suite sts-2.1.1 [7] and L'Ecuyer's BigCrush suite in TestU01 V.1.2.3 [1] as usual. We repeated NIST's test suite 10 times for consecutive 10 files consisting of one gigabits and took average of 10 results of each test. (In the block frequency test we modified the block length to 20000.) The maximum and the minimum of the averaged values are as follows:

[SSI64rand]
P-VALUE AvMax = 0.742517 at NonOverlappingTemplate [$i = 131$]
P-VALUE AvMin = 0.185703 at NonOverlappingTemplate [$i = 45$]
PROPORTION AvMax = 0.992719 at RandomExcursionsVariant [$i = 179$]
PROPORTION AvMin = 0.986145 at RandomExcursions [$i = 159$]

In the result above, p-values which each test of the suite generates are expected to spread uniformly in [0,1]. PROPORTION (FRACTION) is the proportion of sequences of random numbers which pass the test at the level of 0.01. (The i in $[i = x]$ is the serial number of the test in the suite.) Below, for comparison's sake, we cite the result of Mersenne Twister ar pseudorandom number generator [8][6]:

[Mersenne Twister ar]
P-VALUE AvMax = 0.713578 at NonOverlappingTemplate [$i = 105$]
P-VALUE AvMin = 0.188745 at NonOverlappingTemplate [$i = 126$]
PROPORTION AvMax = 0.992879 at RandomExcursionsVariant [$i = 168$]
PROPORTION AvMin = 0.986809 at RandomExcursions [$i = 159$]

Next we applied BigCrush of TestU01. In TestU01 each test requires 32-bit integers or double precision numbers in [0,1]. When a double precision number in [0,1] was requested, we first made a number in [1,2) by putting the first 52 bits of ζ_k in the mantissa and setting its exponent to be $\cdots \times 2^0$, and then subtract one from the number. When a 32-bit number was requested we supplied the first half and then the second half of ζ_k. The tests consisting of BigCrush were executed parallelly, and the result of BigCrush was "All tests were passed". The time required for BigCrush was about 250 hours on (Xeon 3.1 GHz) + (Windows 7 Professional 64-bit). (TestU01 allows only 32-bit gcc, and so we can not use the 64-bit code in the appendix, which is about 7 times faster than a usual 32-bit C code.)

Remark. Distribution of SSI64 random numbers

Essentially a beta transformation $M_\beta(s) = \beta s - \lfloor \beta s \rfloor + 1$ on [1,2) is a special version of linear mod one transformation $T_{\beta,\alpha} : [0,1) \to [0,1)$ defined by

$$T_{\beta,\alpha}(x) \quad = \quad \beta x + \alpha \bmod 1 \quad = \quad \beta x + \alpha - \lfloor \beta x + \alpha \rfloor \quad (0 \le \alpha < 1)$$

[2][3]. In fact M_β and $T_{\beta,\alpha}$ are related such as

$$M_\beta(t) = T_{\beta,\hat{\beta}}(t-1) + 1, \quad t \in [1,2), \tag{5}$$

where $\hat{\beta} = \beta - \lfloor \beta \rfloor (=$ the fractional part of $\beta)$ [6]. So we can make use of results of researches about linear mod one transformation $T_{\beta,\alpha}$. Especially we know that

$$h_{\beta,\alpha}(x) = \sum_{x < T^n_{\beta,\alpha}(1),\ n \ge 0} \frac{1}{\beta^n} - \sum_{x < T^n_{\beta,\alpha}(0),\ n \ge 1} \frac{1}{\beta^n} \tag{6}$$

defines an invariant measure $\nu_{\beta,\alpha}(E) = \int_E h_{\beta,\alpha}(x)d\lambda(x)$ for $T_{\beta,\alpha}$, where $T^n_{\beta,\alpha}(1) = \lim_{x\to 1+0} T^n_{\beta,\alpha}(x)$ and λ is Lebesgue measure on [0,1) [2][3]. From this we have

$$1 - \frac{1}{\beta - 1} \leq h_{\beta,\alpha}(x) \leq 1 + \frac{1}{\beta - 1} \tag{7}$$

(see [6]). In our SSI64rand the β is in $[2^5, 2^6)$, and so $\frac{1}{63} < \frac{1}{\beta-1} \leq \frac{1}{31}$. Also concerning the normalized measure $\mu_{\beta,\alpha}$ of $\nu_{\beta,\alpha}$ we have

$$\lim_{N\to\infty} \frac{1}{N} \sum_{n=0}^{N-1} f(T^n_{\beta,\alpha}(x)) = \int_{[0,1)} f(y)d\mu_{\beta,\alpha}(y) \quad \text{a.a. } x. \tag{8}$$

From these circumstances we think that we can prove that if k is large, then the distribution of $M^k_{2^5 x}(x)$ under the assumption that x spreads uniformly in [1,2) is approximated by the normalized measure on [1,2) derived from

$$H(\cdot) = \int_1^2 h_{2^5 w, \widehat{(2^5 w)}}(\cdot - 1)dw$$

[4][6].

References

[1] P. L'Ecuyer and R. Simard, TestU01: A C Library for empirical testing of random number generators, *ACM Transactions on Mathematical Software*, vol. 33, no. 4 (2007), Article 22. (http://www.iro.umontreal.ca/~simardr/testu01/tu01.html)

[2] W. Parry, Representations for real numbers, *Acta Math. Acad. Sci. Hungar.*, vol. 15 (1964), pp. 95-105.

[3] K. M. Wilkinson, Ergodic properties of certain linear mod one transformations, *Advances in Math.*, vol. 14 (1974), pp. 64-72.

[4] H. Yaguchi and I. Kubo, A new nonrecursive pseudorandom number generator based on chaos mappings, *Monte Carlo Methods Appl.*, vol. 14 (2008), pp. 85-98. DOI 10.1515/MCMA.2008.005

[5] H. Yaguchi and S. Ueda, Construction, randomness and security of new hash functions derived from chaos mappings, *Interdisciplinary Information Sciences*, vol. 18 no. 1 (2012), pp. 1-11. DOI 10.4036/iis.2012.1

[6] H. Yaguchi, Construction and security of nonalgebraic hash functions based on β-transformations on [1,2).

[7] http://csrc.nist.gov/groups/ST/toolkit/rng/index.html

[8] http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/mt19937ar.html

Appendix. Here we give a code of generating an SSI64 random number. The code was tested by Intel's icl (64-bit) and gcc (64-bit). The whole SSI64rand is at http://math1.edu.mie-u.ac.jp/yaguchi/SSI64rand.html .

```
//*******************************
#define MB_W64_X64_16(out, in1, in2) \
     __asm__("mov %3, %%rcx;    \
              mov %2, %%rax;    \
              mov  $0x8000000000000000, %%r10; \
              or  %%r10, %%rax; \
              mul %%rcx;        \
              shl $6, %%rdx;    \
              shr $58, %%rax;   \
              or  %%rdx, %%rax; \

              repeat from "or" to "or" above 14 times

              or  %%r10, %%rax; \
              mul %%rcx;        \
              mov %%rdx, %0;    \
              mov %%rax, %1;    \
             "                  \
             : "=r"(out.hi64), "=r"(out.lo64) \
             : "r"(in1), "r"(in2) \
             : "%rax", "%rcx", "%rdx", "%r10" \
             );
//*******************************
#define SUB_128_128_SHL32(out, in1, in2) \
     __asm__("mov %2, %%rax;    \
              mov %3, %%rdx;    \
              mov %5, %%rcx;    \
              sub %%rcx, %%rdx; \
              mov %4, %%rcx;    \
              sbb %%rcx, %%rax; \
              mov %%rdx, %%rcx; \
              shl $32, %%rax;   \
              shr $32, %%rcx;   \
              or  %%rcx, %%rax; \
              shl $32, %%rdx;   \
              mov %%rax, %0;    \
              mov %%rdx, %1;    \
             "                  \
             : "=r"(out.hi64), "=r"(out.lo64) \
             : "r"(in1.hi64), "r"(in1.lo64),"r"(in2.hi64), "r"(in2.lo64) \
             : "%rax", "%rcx", "%rdx" \
             );
//*******************************
typedef struct
    {
```

```
    unsigned long long int lo64;
    unsigned long long int hi64;
    } my_i128;
//*********************************//
unsigned long long int genSSI64rand(unsigned long long int x,
                                    unsigned long long int t,
                                    unsigned long long int y,
                                    unsigned long long int s)
    {
    my_i128 u, v, umv;

    MB_W64_X64_16(u, t, x);
    MB_W64_X64_16(v, s, y);
    SUB_128_128_SHL32(umv, u, v);

    return(umv.hi64);
    }
//*******************************
```