CPC: 4.13 Statistical methods

PACS: 02.50.-r

Efficient implementation of Bailey and Borwein pseudo-random number generator based on normal numbers

G. Beliakov[a*], D. Creighton[b], M. Johnstone[b], T. Wilkin[a]

[a] *School of Information Technology, Deakin University, Australia*

[b] *Centre for Intelligent Systems Research, Deakin University, Australia.*

[*] *Corresponding author*

Abstract

This paper describes an implementation of a Linear Congruential Generator (LCG) based on the binary representation of the normal number $\alpha_{2,3}$, and of a combined generator based on that LCG. The base LCG with the modulus 3^{33} provides a quality sequence with the period $\approx 3.7 \cdot 10^{15}$, which passes all but two statistical tests from BigCrush test suite. We improved on the original implementation by adapting Barrett's modular reduction method, which resulted in four-fold increase in efficiency. The combined generator has the period of $\approx 10^{23}$ and passes all tests from BigCrush suite.

Keywords: Normal numbers; Random number generation, Linear Congruential Generator

---

PROGRAM SUMMARY

Title of program: BCNRandom

Email address: {gleb,michael.johnstone,dougc, tim.wilkin}@deakin.edu.au (G. Beliakov[a*], D. Creighton[b], M. Johnstone[b], T. Wilkin[a])

Catalogue number: (supplied by Elsevier)

Program obtainable from: CPC Program Library, Queen's University of Belfast, N. Ireland (see application form in this issue)

Licensing provisions: none

Computer for which the program is designed and others on which it has been tested:

Computers: IBM PC; MacBook Pro

Installations:

Operating systems under which the program has been tested: Windows 7, Linux, Unix, OS X

Programming language used: C

Memory required to execute with typical data: 512kb

No. of bits in a word: 64

No. of processors used: 1

Has the code been vectorised?: Yes

Peripherals used:

No. of lines in distributed program, including test data, etc.: 1100

CPC Program Library subprograms used:

Keywords: Random number generation; Linear Congruential Generator; Simulation

Nature of physical problem

This is a fast pseudorandom number generator based on normal numbers, with long period and good statistical properties. It passes all 106 BigCrush statistical tests and is faster than C standard library rand generator. Suitable for parallel execution by multiple threads and/or processes because of skip ahead property.

Method of solution

Linear Congruential Generators with specially chosen parameters are used. A customized fast modular reduction is employed.

Restrictions on the complexity of the problem

Typical running time

Generation rate is 100-200 million numbers per second.

Unusual features of the program

References

LONG WRITE-UP

1. Introduction

Pseudorandom number generators are required for many computational tasks, such as simulating stochastic models, numerical integration and cryptography [1]. Linear Congruential Generators (LCG) and Mersenne Twisters (MT) are two of the most important families of pseudorandom generators used in simulations (cryptographically strong random number generators are treated elsewhere [2, 3]). MTs offer longer periods than LCGs and do not suffer from various correlations between the elements of the sequence as LCGs, especially when an LCG is poorly designed [4]. However MTs require a bigger state space and their implementation is more complex than that of LCGs. Typically, a generator that produces uniform pseudorandom variates from $(0, 1)$ (or random integers from $\{0, 1, \ldots, m\}$) is the base generator used to generate random variates from more complicated distributions [5, 6, 7]. Various statistical tests, such as SmallCrush and BigCrush from the TestU01 library [8] are used to evaluate the quality of the generated sequences.

We focus on a version of LCG based on binary representation of normal numbers, recently proposed in [9]. For an integer $b \geq 2$ we say that a real number K is b-normal if, for all $m > 0$, every m-long string of digits in the base-b expansion of K appears, in the limit, with frequency b^{-m}. That is, with exactly the frequency one would expect if the digits appeared completely "at random". Bailey and Borwein [9] presented a thorough theoretical overview of the topic, and designed a special class of constants $\alpha_{b,c}$, where $b, c > 0$ and co-prime. They proved that such constants are b-normal, and therefore their base-b expansion can serve to generate pseudorandom sequences.

A useful feature of the numbers $\alpha_{b,c}$ is that their $n-$th base$-b$ digit can be calculated directly, without needing to compute any of the first $n-1$ digits. It offers the valuable skip ahead feature of a pseudorandom number generator, suitable in particular for parallel processing, as consecutive subsequences of random numbers can be generated concurrently in different threads (or processors), and the entire sequence is identical to the one generated in a single thread. This feature, together with the small state of the LCG are very useful for computations on Graphics Processing units (GPU) (see recent works [10, 11]), which have recently become a powerful alternative to traditional high performance computing. Skip ahead is much harder to achieve in other types of pseudorandom generators.

Bailey and Borwein [9] present a pseudorandom number generator based on the binary representation of $\alpha_{2,3}$, and outline its implementation as a version of LCG, with the period $P = 2 \cdot 3^{32} \approx 3.7 \cdot 10^{15}$. The seed of this generator is the index of the starting element, which is calculated directly through a simple formula. They suggest its use in parallel computations, by splitting the sequence of variates into m subsequences (m is the number of threads), each subsequence i starting from the $1 + \frac{(i-1)n}{m}$-th element, where n is the desired number of random variates.

In this work we describe a particularly efficient algorithm implementing Bailey and Borwein's generator, referred to as BCN, based on modular arithmetic. Our implementation is four times faster than the original implementation in [9, 12]. We have studied various approaches to 128-bit integer multiplication on CPU and GPU, which is one of the steps of Bailey and Borwein's generator [9], and found that the efficiency depends on both hardware and the compiler. We also implemented differently the seeding step of Bailey and Borwein's generator, using modular exponentiation.

The resulting generator BCN passes all but two of the 106 statistical tests from the BigCrush suite [8]. If fails Birthday Spacings and Closed Pairs tests, which is expected for any LCG (Birthday Spacings), and any generator with the period less than $2^{60} \approx 10^{18}$ (Closed Pairs). An approach based on combining several generators advocated in [13, 14] was adopted. By combining the BCN generator with another simple LCG, we obtained a generator with the period of approximately 10^{23}, which passes all 106 statistical tests from BigCrush suite, and is only marginally slower than BCN. Its state is 128 bits and it also offers the skip ahead feature.

The paper is structured as follows: Section 2 reviews the LCG proposed by Bailey and Borwein [9]. Section 3 describes improvements made to the original implementation using modular arithmetic. Numerical experiments are described and results presented in Section 4. The combined generator is presented in Section 5. Finally we draw conclusions in Section 6.

2. Random number generator based on normal numbers

Bailey and Borwein [9] investigated the following class of constants, which was introduced previously in [15],

$$\alpha_{b,c}(r) = \sum_{k=1}^{\infty} \frac{1}{c^k b^{c^k + r_k}}$$

where integers $b, c > 1$ and co-prime and $r \in [0, 1]$, and where r_k denotes the k-th binary digit of r. Bailey and Crandall [15] proved the following:

Theorem 1. *Every number $\alpha_{b,c}(r)$ is b-normal.*

Then Bailey and Borwein [9] focused on the number

$$\alpha = \alpha_{2,3}(0) = \sum_{k=1}^{\infty} \frac{1}{3^k 2^{3^k}}.$$

This number is 2-normal, but is not 6-normal, as the authors show [9]. Further they show that all constants $\alpha_{b,c}$ are not $bc-$normal for co-prime integers $b, c > 1$.

The normality of α suggests that its binary expansion can be used to generate a sequence of pseudorandom numbers. Let $x_n = \{2^n \alpha\}$ be the binary expression of α starting from position $n + 1$, where $\{\cdot\}$ denotes the fractional part of the argument. Bailey and Borwein [9] show that when n is not the power of three

$$x_n = \frac{(2^{n-3^m} \lfloor 3^m/2 \rfloor) \mod 3^m}{3^m} + \varepsilon, \tag{1}$$

and the tail of the series $\varepsilon < 10^{-30}$ if n is not within 100 of any power of three.

Then Bailey and Borwein [9] construct the following algorithm to calculate pseudorandom 64-bit real numbers in $(0, 1)$, which would contain in their mantissas 53-bit segments of the binary expansion of α

Algorithm BB

1. Select starting index a in the range $3^{33} + 100$ to 2^{53}, referred to as the seed of the generator.

2. Calculate
$$z_0 = 2^{a-3^{33}} \left\lfloor \frac{3^{33}}{2} \right\rfloor \mod 3^{33} \tag{2}$$

3. Generate iterates
$$z_k = 2^{53} z_{k-1} \mod 3^{33}, \quad k = 1, 2, \ldots \tag{3}$$
and return $z_k 3^{-33}$, which are 64-bit floating point random variates in $(0, 1)$.

Bailey and Borwein [9] note that several operations need to be done with accuracy 106 mantissa bits, i.e., in 106-bit floating point arithmetic, and describe and implement their algorithm in double-double arithmetic [9, 12]. Note that double-double arithmetic allows exactly the representations of integers up to 2^{106}.

The algorithm BB is a version of the LCG, and therefore its period can be established from the theory of LCGs and results in $P = 2 \cdot 3^{32} \approx 3.7 \cdot 10^{15}$. The algorithm can be checked by calculating z_k-th iterate recursively, or directly by using formula (2). The binary digits of α within a range of indices spanned by successive powers of three are given by an LCG with a modulus that is a large power of three.

This approach has an advantage over LCGs that use a power of two value as the modulus, where arrays of pseudorandom data, of size matching a power of two, are accessed by row and column and therefore have the potential for a reduced period [9]. A modulus of the sort proposed by Bailey and Borwein [9] removes this issue.

The direct formula (2) produces the valuable skip ahead property: the ability to calculate the k-th iterate without iterating through the previous steps. This property is particularly useful for parallel random number generators [10, 11], as every parallel thread can calculate its starting iterate directly from (2). We presented such a parallel implementation for Graphics processing Units (GPUs) in [16].

3. Implementation based on modular arithmetic

Using Bailey and Borwein's [9] implementation as the starting point, we translated it from FORTRAN to C to investigated possible improvements to the LCG. Our main focus was the generation step Eq. (3).

3.1. Generation step

Generation step (3) is a modular reduction, simply the computation of the remainder of an integer division. However, division is a much more expensive operation than multiplication, thus there are a number of special methods for performing modular reduction using multiplication, summation and subtraction, and single precision division that could be employed to improve the efficiency of the calculation. We have investigated and compared various modular reduction

```
ALGORITHM 1: Barrett's reduction
  Input: $x = (x_{2k-1} \dots x_1 x_0)_2, m = (m_{k-1} \dots, m_1 m_0)_2$, $\mu = \left\lfloor \frac{2^{2k}}{m} \right\rfloor$,
         $x, m > 0$, $m_{k-1} \neq 0$
  Output: $r = x \mod m$
    1. $q_1 = \lfloor \frac{x}{2^{k-1}} \rfloor$, $q_2 = \mu \cdot q_1$, $q_3 = \lfloor \frac{q_2}{2^{k+1}} \rfloor$
    2. $r_1 = x \mod 2^{k+1}$, $r_2 = q_3 \cdot m \mod 2^{k+1}$, $r = r_1 - r_2$
    3. if $r < 0$ then $r = r + 2^{k+1}$
    4. while $r \geq m$ do $r = r - m$
    5. return $r$.
```

methods in [16] and found that by modifying and tailoring Barrett's reduction algorithm to our specific set of numbers we could achieve four-fold improvement to the speed of generation.

Barrett [17] introduced the idea of estimating the quotient $\lfloor \frac{x}{m} \rfloor$ with operations that are either less expensive in time than a multi-precision division by m or can be done as a pre-calculation for a given m. Barrett [17] uses the approximation

$$\left\lfloor \frac{x}{m} \right\rfloor - 2 \leq \left\lfloor \frac{\lfloor \frac{x}{2^{k-1}} \rfloor \cdot \left\lfloor \frac{2^{k-1} 2^{k+1}}{m} \right\rfloor}{2^{k+1}} \right\rfloor = \left\lfloor \frac{\lfloor \frac{x}{2^{k-1}} \rfloor \cdot \left\lfloor \frac{2^{2k}}{m} \right\rfloor}{2^{k+1}} \right\rfloor \leq \left\lfloor \frac{x}{m} \right\rfloor$$

and the equation $x \mod m = x - m \lfloor \frac{x}{m} \rfloor$. The term $\lfloor \frac{2^{2k}}{m} \rfloor$ depends only on m and can be pre-computed. The other divisions are efficient binary shift operations. The value of k here is the number of binary digits of m. The algorithm is shown in Algorithm 1.

In our case $k = 53$, and μ is pre-computed in extended arithmetic but is a

64-bit integer itself. Integer q_2 needs extended precision, but q_1 and q_3 need only 64 bits using binary shift operations with high and low parts of q_2. Integer $q_3 \cdot m$ requires extended arithmetic. Finally r_1, r_2 and r are 64 bit integers. These observations allow one to reduce CPU time. At most two subtractions at step 4 are needed.

Next we present a modified approach which allows us to short-circuit several arithmetical operations.

We note that with our choice of $\mu = \left\lfloor \frac{2^{2k}}{m} \right\rfloor$ at step 1 of Barrett's algorithm, we have $q_1 = \left\lfloor \frac{x}{2^{53-1}} \right\rfloor = \left\lfloor \frac{2^{53}z}{2^{52}} \right\rfloor = 2z$, and consequently $q_2 = 2\mu z$. Similarly at Step 2 we have $r_1 = 2^{53}z \mod 2^{52} = z \mod 2$. We now eliminate these two operations altogether.

We rewrite Barrett's formula as follows

$$\left\lfloor \frac{\left\lfloor \frac{x}{2^k} \right\rfloor \cdot \left\lfloor \frac{2^k 2^k}{m} \right\rfloor}{2^k} \right\rfloor = \left\lfloor \frac{\left\lfloor \frac{x}{2^k} \right\rfloor \cdot \left\lfloor \frac{2^{2k}}{m} \right\rfloor}{2^k} \right\rfloor \leq \left\lfloor \frac{x}{m} \right\rfloor.$$

Then we can keep the same $\mu = \left\lfloor \frac{2^{2k}}{m} \right\rfloor$, but the two mentioned operations simplify as follows:

1) at Step 1 we calculate $q_1 = \left\lfloor \frac{x}{2^{53}} \right\rfloor = \left\lfloor \frac{2^{53}z}{2^{53}} \right\rfloor = z$;

2) at Step 2 we have $r_1 = 2^{53}z \mod 2^{53} = 0$.

Therefore these two instructions become redundant, and hence we save on CPU time. Furthermore, since $r = 0 - r_2 < 0$, the `if` statement in Step 3 is redundant. Finally, because in our case $3^{33} = m < 2^k = 2^{53} < 2m$, and because $r < 2^{53}$, we have $r < 2m$ and therefore `while` in Step 4 can be replaced with a cheaper `if`. Our modified Barrett's algorithm is shown in Algorithm 2.

3.2. Seeding step

Next, we modified the seeding step (2) of the Algorithm BB, using modular exponentiation. At this step we need to compute $z_0 = 2^b \left\lfloor \frac{3^{33}}{2} \right\rfloor \mod 3^{33}$ with $100 < b < 2^{53} - 3^{33}$. Let number b have binary representation $b = (b_{k-1} \dots, b_1, b_0)_2$. We express z_0 as

$$\begin{aligned} z_0 &= (2^{b_{k-1}} 2^{b_{k-2}} \cdots 2^{b_1} 2^{b_0} \mod 3^{33}) \left\lfloor \frac{3^{33}}{2} \right\rfloor \mod 3^{33} \\ &= [2^{b_{k-1}} (2^{b_{k-2}} (\cdots 2^{b_0} \mod 3^{33}) \mod 3^{33}) \mod 3^{33}] \left\lfloor \frac{3^{33}}{2} \right\rfloor \mod 3^{33}. \end{aligned}$$

We precompute the quantities $Q_i = 2^{b_i} \mod 3^{33}$, $i = 5, \dots, 53$. Starting with $q = 2^{(b_4 b_3 b_2 b_1 b_0)} \mod 3^{33} = 2^{(b_4 b_3 b_2 b_1 b_0)}$ we recursively compute

$$q \leftarrow Q_i q \mod 3^{33}, i = 5, 6, \dots, 53, \text{ if } b_i \neq 0.$$

At the last step we compute

$$z_0 = q \left\lfloor \frac{3^{33}}{2} \right\rfloor \mod 3^{33}.$$

Each mod operation is evaluated using Barrett's Algorithm 1. Every multiplier is at most 53 bits long and therefore the product fits 106 (and hence 128-bit) data type. There are at most 48 calls to Algorithm 1.

An improvement to this approach can be achieved by precomputing more quantities to be used in a lookup table. Let us precompute the arrays

$$\begin{aligned} Q_1 &= ((1 \cdot 2^{32}) \mod 3^{33}, (2 \cdot 2^{32}) \mod 3^{33}, (3 \cdot 2^{32}) \mod 3^{33}, \dots, (31 \cdot 2^{32}) \mod 3^{33}), \\ Q_2 &= ((1 \cdot 2^{2^{10}}) \mod 3^{33}, (2 \cdot 2^{2^{10}}) \mod 3^{33}, (3 \cdot 2^{2^{10}}) \mod 3^{33}, \dots, (31 \cdot 2^{2^{10}}) \mod 3^{33}), \\ &\dots \\ Q_{10} &= ((1 \cdot 2^{2^{50}}) \mod 3^{33}, (2 \cdot 2^{2^{50}}) \mod 3^{33}, (3 \cdot 2^{2^{50}}) \mod 3^{33}, \dots, (31 \cdot 2^{2^{50}}) \mod 3^{33}). \end{aligned}$$

```
ALGORITHM 2: Modified Barrett
  Input: $k = 53$, $z = (z_{k-1} \ldots z_1 z_0)_2$,$m = (m_{k-1} \ldots, m_1 m_0)_2$, $\mu = \left\lfloor \frac{2^{2k}}{m} \right\rfloor$,
         $x, m > 0$, $m_{k-1} \neq 0$
  Output: $r = 2^k z \mod m$
    1. $q_2 = \mu \cdot z$, $q_3 = \left\lfloor \frac{q_2}{2^k} \right\rfloor$
    2. $r_2 = q_3 \cdot m \mod 2^k$
    3. $r = 2^k - r_2$
    4. if $r \geq m$ then $r = r - m$
    5. return $r$.
```

Then Execution of Step (2) of the BB algorithm is achieved by the following sequence of calls, starting with $q = 2^{(b_4 b_3 b_2 b_1 b_0)}$:

$$b \leftarrow b/32;\ c \leftarrow b\&31;\ \ (\&\text{ denotes the binary AND operation })$$

$$\text{if } c \neq 0 \text{ then } q \leftarrow Q_i(c)q \mod 3^{33}, i = 1, 2, \ldots, 10.$$

As previously, at the last step we compute

$$z_0 = q \left\lfloor \frac{3^{33}}{2} \right\rfloor \mod 3^{33}.$$

Here we need to precompute at most 320 values, and execute Algorithm 1 at most 10 times.

4. Implementations and numerical experiments

We experimented with several alternative implementations of the algorithm, which were tailored to specific C compiler and hardware. Note that only the number q_2 needs to be a 128-bit integer at the generation step in Algorithm 2 (the rest are standard 64-bit unsigned integers we do not worry about).

The Gnu-C compiler on Linux provides native support for a 128 bit result of signed and unsigned 64 bit multiplications (when compiled with `-m64` and executed on x86_64 systems). This operation translates into just three assembler instructions on Intel 64-bit processors, and is considered efficient.

The Microsoft Visual C++ compiler offers the intrinsic function `_umulh` which performs multiplication of two 64-bit unsigned integers and returns the high 64 bits of the 128-bit result. The low 64 bits of the result can be obtained from the native arithmetic multiplication operator.

We present C code of our implementation of the original and modified Barrett's reduction, as well as of the initialization and generation steps in Figure 1 and Figure 2 .

We expect that the efficiency of the generation on CPU and GPU will depend on the hardware and compiler being used, and also the compilation parameters. In Table 1 we present generation rates on different machines under different compilers. Cross-comparison should consider that the target chipsets were different for the three machines and thus direct comparison of numbers is not advised. The important part is the difference in generation rates of the alternative methods on the same architecture and compiler.

We benchmark the modified Barrett's reduction against the original implementation in [9, 12] called BCN. The standard library `rand()` function was also evaluated on the same task as a baseline for comparison. Efforts were made to reduce timing complications due to cache flushes and process interruptions due to CPU scheduling.

What is most noticeable is the performance of the modified Barrett method, which greatly exceeds the performance of the other algorithms across all compilers. In particular, the combination of this algorithm and a modern chipset (Intel i7-2760QM processor) provides exceptionally good performance compared with the other algorithms across the various architectures.

A different implementation of the seeding step in our algorithm allowed us to detect an error in the original BCN algorithm implementation from [9, 12]. The original algorithm calculated the starting value z_0 by formula (2) incorrectly in some rare cases, e.g. for $a = 17196091$, $a = 34392182$, $a = 34392183$, and a few larger seeds. We traced this error to their implementation of the modulus operation in double-double arithmetics, and found a fix for it by changing one `if` statement to `while`, which is executed at most twice. The proposed implementation based on modular exponentiation does not have this issue. In addition it is much shorter than the double-double implementation.

As far as the quality of the generated sequence is concerned, the testing suite BigCrush [8] was applied to the generator algorithm BB. Of the 106 statistical tests, all tests passed except for the Birthday Spacings tests #13-21 and the Close Pairs tests #22-24.

The Birthday Spacings tests check certain correlations of the generated numbers. For any LCG, if the generated values are used as coordinates of points in an n-dimensional space, then they fall onto a small number of equally spaced

Table 1: The generation rate for various methods on CPU using different compilers and systems (Gnum/sec). We measured the rates on the following processors and systems: Intel i7-860 @ 2.8 GHz (Fedora 16), Intel Core i7-2720QM @ 2.2 GHz (Windows 7) and MacBook Pro Intel Core i7-2760QM @ 2.40 GHz (Mac OSX 10.7.3) respectively.

Compiler and	Method			
architecture	original BCN	modified Barrett	rand()	Combined
gcc 4.4 on Linux (Fedora 12) 32 bit implementation	0.023	0.031	0.062	0.024
gcc 4.4 on Linux (Fedora 12) 64 bit implementation	0.026	0.121	0.10	0.091
MSVC 10 on Windows 7 64 bit implementation	0.032	0.179	0.065	0.141
gcc 4.5 on Mac OSX 64 bit implementation	0.035	0.24	0.14	0.19

parallel planes (Marsaglia's Theorem). Hence failure of Birthday Spacings tests is expected for an LCG.

The Close Paris tests, as studied in [18], would expect an LCG to fail with period less than 2^{60}. Here we have a period of $P = 2 \cdot 3^{32}$, and a failed test. The rest of the tests (94 in total) were passed, and we conclude that the generated sequence is of a sufficiently good statistical quality. However, improvements can be made by considering combinations of several generators, which is what we did in the sequel.

5. Combined generator

Combining outputs of several generators was considered in the literature many times. In [13] the author proposed combining several LCGs using the following formula

$$z_k = \left(\sum_{j=1}^{l} (-1)^{j-1} s_{j,k} \right) \mod (m_1 - 1),$$

where l is the number of generators combined, and $s_{j,k}$ is the output of the j-th generator at iteration k.

The maximum period of the combined generator is half the product of the periods of the generators being combined, and is achieved when the values $(m_i - 1)/2$ and $(m_j - 1)/2$ are relatively prime for $i \neq j$ [13], which is the case with our choices of the moduli.

We chose the second supplementary LCG with the modulus $m_2 = 2^{31} + 1 = 3 \cdot 715827883$ and $a = 39373$, and the combined generator is

$$z_k = (LCG_{supp}(x_{k-1}) - BCN(y_{k-1})) \mod (m_2 - 1), \tag{4}$$

where $\{x_i\}$ and $\{y_j\}$ are the sequences of random numbers produced by the LCG_{supp} and BCN generators respectively. The reason for our choice of m_2 is that the modulus in (4) is 2^{31} and hence this operation is implemented efficiently using bit masking. On the other hand, the value of m_2 is rather small (32 bits), and taking the modulus in the supplementary LCG is efficiently implemented using integer division (operation % in C language). The period of the supplementary generator was confirmed to be 119304648.

By using a good quality sequence generated by BCN, and a very fast supplementary generator, we hoped to inherit and improve on the statistical properties of BCN at a very little computational cost. This approach was successful: the resulting pseudo-random sequence of the combined generator has successfully passed all 106 statistical tests of the BigCrush suite, and added only a small overhead to the generation rate (about 25% slower), that can be appreciated in Table 1, last column. It is still faster than the `rand()` function on the newest CPUs. The period of the new generator exceeds 10^{23}, which is more than sufficient for practical purposes (with the current fastest generation rate, it would take one CPU over 10^{14} sec $\approx$ 3 million years to complete one cycle).

As far as the skip ahead feature is concerned, we implemented it for the supplementary generator using the same approach as the one discussed in Section 3.2.

```
#define ULL(x) APPEND(x, ull)
static const uint64_t BCN_m    = ULL(5559060566555523);    /* m = 3^33  */
static const uint64_t BCN_mulo = ULL(0x33D9481681D79D);
static const uint64_t BCN_t  = ULL(2779530283277761);    /* floor(m/2) */

uint64_t BarrettStep(uint64_t z, uint64_t q_t53)
{
  uint64_t r1lo, r2lo;

#if (NATIVE_128BIT_INT_TYPES)
  uint128_t x, q;
  x = (uint128_t)z * q_t53;
  q = x * BCN_mulo;
  r1lo = (x >> 64) & ULL(0x3FFFFFFFFFFFFF);
  r2lo = (q >> 64) * BCN_m;
#else
  uint64_t xlo, xhi, qlo, qhi;
  xlo = umul64(z, q_t53, &xhi);
  qlo = (xhi << 12) | (xlo >> 52);
  qlo = umul64(qlo, BCN_mulo, &qhi);
  qlo = (qhi << 10) | (qlo >> 54);
  r1lo = xlo & ULL(0x3FFFFFFFFFFFFF);
  r2lo = qlo * BCN_m;
#endif
  r2lo &= ULL(0x3FFFFFFFFFFFFF);
  z = r1lo - r2lo;
  if (r1lo < r2lo) z += ULL(0x40000000000000);
  while (z >= BCN_m) z -= BCN_m;
  return z;
}


uint64_t BarrettInit(uint64_t k)
{
  uint32_t i = 0;
  uint64_t q = ULL(1) << (k & 0x1F);
  k = k >> 5;
  for (i = 5; i < 54; i++) {
     if (k & 0x01)  q = BarrettStep(q,BCN_Q[i]);
     k = k >> 1;
  }
  return BarrettStep(q, BCN_t);
}
```

Figure 1: Implementation of the Barrett Algorithm 1 and of the initialization (seeding) step (2) of BB algorithm via modular exponentiation. Constants in the array BCN_Q are $2^{2^i} \bmod 3^{33}, i = 0, \ldots, 53$. Function `umul64` refers to multiplication of two unsigned 64-bit integers.

```
static const double BCN_minv = 1.7988650924514300510763861722128e-16;
/* 1.0 /BCN_m; */
static uint64_t  SeedBCN_z_k = 0 ;

uint64_t randlcgc_increment( uint64_t *z )
{
  register uint64_t qt;
#if !defined (NATIVE_128BIT_INT_TYPES)
  uint64_t qlo, qhi;
  qlo = umul64(*z,BCN_mulo, &qhi);
  qt = ( ( (qhi << 11 | qlo >> 53) * BCN_m ) & ULL(0x1FFFFFFFFFFFFF) );
#else
  qt = ( (umul128(*z,BCN_mulo) >> 53) * BCN_m ) & ULL(0x1FFFFFFFFFFFFF);
#endif
  *z = ULL(0x20000000000000) - qt;
  while (*z >= BCN_m)
     *z -= BCN_m;
  return (*z);
}

/* Generation step */
double randbcn()
{ return BCN_minv * randlcgc_increment(&SeedBCN_z_k); }
```

Figure 2: Implementation of the modified Barrett Algorithm 2 and of the generation step.

6. Conclusion

We presented a pseudorandom number generation algorithm based on Bailey and Borwein's work on normal numbers and their original implementation as a version of LCG. We confirmed that the generated sequence is statistically suitable for simulation purposes by running a comprehensive BigCrush suite of tests. We improved Bailey and Borwein's implementation by implementing a special modular reduction algorithm, which gave speedup of a factor of 4, in some cases 7, on a CPU. We also improved the initialization step of BB Algorithm using modular exponentiation. It has a valuable skip ahead property, which makes it suitable for parallel random number generation in multiple threads. Our generator is twice as fast as the standard C function `rand()`.

We also constructed a combined generator using a fast supplementary LCG, which passes all BigCrush statistical tests, has a large period, small state, valuable skip ahead property, and is nearly as efficient as the Bailey and Borwein's generator. Our implementation is available from
`http://www.deakin.edu.au/~gleb/bcn_random.html`.

References

[1] L'Ecuyer, P., Annals of Operations Research **53** (1994) 77.

[2] Blum, M., How to generate cryptographically strong sequences of pseudo random bits, in *23rd Annual Symposium on Foundations of Computer Science*, 1982, p. 112.

[3] Menezes, A., van Oorschot, P., and Vanstone, S. A., editors, *Handbook of Applied Cryptography*, CRC Press, Boca Raton, 1996.

[4] L'Ecuyer, P., Simard, R., Chen, J., and Kelton, D., Operations Research **50** (2002) 1073.

[5] Atkinson, A. C. and Pearce, M. C., Journal of the Royal Statistical Society. Series A (General) **139** (1976) 431.

[6] Hörmann, W., Leydold, J., and Derflinger, G., *Automatic Nonuniform Random Variate Generation*, Springer, Berlin; New York, 2004.

[7] Beliakov, G., Computer Physics Communications **170** (2005) 93.

[8] L'Ecuyer, P. and Simard, R., ACM Transactions on Mathematical Software **33** (2007) article 22.

[9] Bailey, D. and Borwein, J., Normal numbers and pseudorandom generators, in *Proceedings of the Workshop on Computational and Analytical Mathematics in Honour of Jonathan Borwein's 60th Birthday, Springer, 2011, available online at: http://www.davidhbailey.com/dhbpapers/normal-pseudo.pdf.*

[10] Gao, S. and Peterson, G., Computer Physics Communications **184** (2013) 1241.

[11] Wu, P.-C. and Huang, K.-C., Computer Physics Communications **175** (2006) 25.

[12] Bailey, D., High-precision software directory http://www.davidhbailey.com/mpdist/, accessed 18 March 2013.

[13] L'Ecuyer, P., Commun. ACM **31** (1988) 742.

[14] Deng, L., Guo, R., Lin, D., and Bai, F., Computer Physics Communications **178** (2008) 401.

[15] Bailey, D. and Crandall, R., Experimental Mathematics **11** (2000) 527.

[16] Beliakov, G., Johnstone, M., Creghton, D., and Wilkin, T., Computing (2013) DOI: 10.1007/s00607-012-0234-8.

[17] Barrett, P., Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor, in *Proceedings on Advances in cryptology—CRYPTO '86*, London, UK, 1987, Springer-Verlag, p. 311.

[18] L'Ecuyer, P., Cordeau, J., and Simard, R., Oper. Res. **48** (2000) 308.