A Pseudo-Random Number Generator Based on Normal Numbers

David H. Bailey

11 Dec 2004

Abstract

In a recent paper, Richard Crandall and the present author established that each of a certain class of explicitly given real constants, uncountably infinite in number, is b-normal, for an integer b that appears in the formula defining the constant. A b-normal constant is one where every string of m digits appears in the base-b expansion of the constant with limiting frequency b^{-m}. This paper shows how this result can be used to fashion an efficient and effective pseudo-random number generator, which generates successive strings of binary digits from one of the constants in this class. The resulting generator, which tests slightly faster than a conventional linear congruential generator, avoids difficulties with large power-of-two data access strides that may occur when using conventional generators. It is also well suited for parallel processing—each processor can quickly and independently compute its starting value, with the collective sequence generated by all processors being the same as that generated by a single processor.

Introduction

In [1], Richard Crandall and the present author established that the class of constants

$$\alpha_{b,c} = \sum_{k=1}^{\infty} \frac{1}{c^k b^{c^k}},$$

where the integer $b > 1$ and c is odd and co-prime to b, is both b-normal and transcendental. The term *b-normal* (also written as *normal base* b) means that each string of m digits in the base-b expansion of $\alpha_{b,c}$ appears with a limiting frequency of b^{-m}, or in other words with exactly the limiting frequency that one would expect if the digits were generated "at random." Actually, the authors established normality for a much larger class (in fact, an uncountably infinite class). But for simplicity this paper will deal with one specific member of this class, namely

$$\begin{aligned}\alpha_{2,3} &= \sum_{k=1}^{\infty} \frac{1}{3^k 2^{3^k}} \\ &= 0.041883680831502985071252898624571682426096758465485 7\ldots_{10} \\ &= \texttt{0.0AB8E38F684BDA12F684BF35BA781948B0FCD6E9E06522C3F35B}\ldots_{16},\end{aligned}$$

which is 2-normal. This particular constant was first proved 2-normal by Stoneham [5]. A fairly simple proof of this result is given in [2, pg. 155].

This result suggests that the binary digits of $\alpha_{2,3}$ could be used to fashion a practical pseudo-random number generator. Indeed, this was suggested in the original paper [1]. This can be done, and the result is a generator that is both efficient on single-processor systems and also well-suited for parallel processing: each processor can quickly and independently calculate the starting seed for its section of the resulting global sequence, which global sequence is the same as the sequence produced on a single-processor system (subject to some reasonable conditions).

The material that follows represents the full solution of a problem originally sketched in [2, pg. 169–170].

2. Mathematical Background

Define x_n to be the binary expansion of $\alpha_{2,3}$ starting with position $n + 1$. Note that $x_n = \{2^n \alpha_{2,3}\}$, where $\{\cdot\}$ means the fractional part of the argument. First consider the case $n = 3^m$ for some integer m. In this case one can write

$$x_{3^m} = \{2^{3^m} \alpha_{2,3}\} = \left\{\sum_{k=1}^{m} \frac{2^{3^m - 3^k}}{3^k}\right\} + \sum_{k=m+1}^{\infty} \frac{2^{3^m - 3^k}}{3^k}$$

Observe that the "tail" term (i.e., the second term) in this expression is exceedingly small once m is even moderately large—for example, when $m = 10$, this term is only about $10^{-35,551}$. This term will hereafter be abbreviated as ϵ_m. By expanding the first term, one obtains

$$x_{3^m} = \frac{(3^{m-1} 2^{3^m - 3} + 3^{m-2} 2^{3^m - 3^2} + \cdots + 3 \cdot 2^{3^m - 3^{m-1}} + 1) \bmod 3^m}{3^m} + \epsilon_m$$

The numerator is taken modulo 3^m, since only the remainder when divided by 3^m is of interest when finding the fractional part. By Euler's totient theorem, the next-to-last term in the numerator, when reduced modulo 3^m, is three. Similarly, it can be seen that every other term in the numerator, when reduced modulo 3^m, is equivalent to itself without the power-of-two part. In other words, the expression above reduces to

$$\begin{aligned} x_{3^m} &= \frac{(3^{m-1} + 3^{m-2} + \cdots + 3 + 1) \bmod 3^m}{3^m} + \epsilon_m \\ &= \frac{3^m - 1}{2 \cdot 3^m} + \epsilon_m = \frac{\lfloor 3^m/2 \rfloor}{3^m} + \epsilon_m \end{aligned}$$

The author is indebted to Helaman Ferguson for this proof. More generally, for n that is not a power of three, one can write

$$x_n = \frac{(2^{n-3^m} \lfloor 3^m/2 \rfloor) \bmod 3^m}{3^m} + \epsilon,$$

where m is chosen so that 3^m is the largest power of three less than or equal to n. In this case, one can be assured that $\epsilon < 10^{-30}$ provided n is not within 100 of any power of three.

3. The Algorithm

With this explicit expression in mind, an algorithm can be given for generating pseudorandom deviates, in the form of a sequence of IEEE 64-bit floating-point numbers in $(0, 1)$. These deviates contain, in their mantissas, successive 53-bit segments of the binary expansion of $\alpha_{2,3}$, beginning at some given starting position.

Initialization. First select a starting index a in the range $3^{33} + 100 = 5559060566555623 \leq a \leq 2^{53} = 9007199254740992$. The value of a can be thought of as the "seed" of the generator. Then calculate

$$z_0 = (2^{a-3^{33}} \cdot \lfloor 3^{33}/2 \rfloor) \bmod 3^{33}.$$

Generate iterates. Successive iterates of the generator can then be recursively computed by iterating

$$z_k = (2^{53} \cdot z_{k-1}) \bmod 3^{33}$$

and then returning the values $z_k 3^{-33}$, which are 64-bit IEEE floating-point results in the unit interval.

Double-double arithmetic. Several of the operations used in this scheme must be done with an accuracy of 106 mantissa bits. "Double-double" precision arithmetic of this sort can be implemented simply and efficiently. Here a double-double datum is represented by a pair of IEEE double-precision floating-point numbers—the first word is the closest 64-bit IEEE value to the double-double value, and the second word is the difference. For convenience, three key algorithms used in double-double arithmetic are given below, from which a complete set of double-double arithmetic functions can be constructed [3] or [2, pg. 218-220]. Here $\oplus, \ominus$ and $\otimes$ denote the result of IEEE 64-bit floating-point operations.

A: Double + double. This computes the high- and low-order words of the sum of two IEEE 64-bit values a and b.

```
1. $s := a \oplus b$;
2. $v := s \ominus a$;
3. $e := (a \ominus (s \ominus v)) \oplus (b \ominus v)$;
4. Return $(s, e)$.
```

B: Split. This splits an IEEE 64-bit value a into $a_{\rm hi}$ and $a_{\rm lo}$, each with 26 bits of significance and one hidden bit, such that $a = a_{\rm hi} + a_{\rm lo}$.

```
1. $t := (2^{27} + 1) \otimes a$;
2. $a_{\rm hi} := t \ominus (t \ominus a)$;
3. $a_{\rm lo} := a \ominus a_{\rm hi}$;
4. Return $(a_{\rm hi}, a_{\rm lo})$.
```

C: Double × double. This computes the high- and low-order words of the product of two IEEE 64-bit values a and b.

```
1. $p := a \otimes b$;
2. $(a_{\rm hi}, a_{\rm lo}) := \text{SPLIT}(a)$;
3. $(b_{\rm hi}, b_{\rm lo}) := \text{SPLIT}(b)$;
4. $e := ((a_{\rm hi} \otimes b_{\rm hi} \ominus p) \oplus a_{\rm hi} \otimes b_{\rm lo} \oplus a_{\rm lo} \otimes b_{\rm hi}) \oplus a_{\rm lo} \otimes b_{\rm lo}$;
5. Return $(p, e)$.
```

With regards to C, note that some processors, notably IBM PowerPC and RS6000 processors and Intel IA-64 processors, have a "fused multiply-add" instruction that greatly simplifies double × double operations. In this case, one can simply write $p := a \otimes b$ and $e := a \otimes b - p$, and the "Split" operation is not needed. Note, however, that it is often necessary to specify a special compiler option (such as -qstrict on IBM systems) to insure that this code is performed as written.

Complete C++ and Fortran-90 double-double computation software packages, including both basic-level arithmetic functions as well as common algebraic and transcendental functions, are available from `http://crd.lbl.gov/~dhbailey/mpdist`.

Implementation details. The operation $(2^{53} \cdot z_{k-1}) \bmod 3^{33}$ can be performed efficiently as follows: (1) multiply 2^{53} by z_{k-1} by using Algorithm C above; (2) multiply the high-order word of the result of Step 1 by 3^{-33}, using ordinary double precision arithmetic, and take the greatest integer; (3) multiply the result of Step 2 by 3^{33}, again by using Algorithm C above; and (4) subtract the result of Step 3 (which is of type double-double) from the result of step 1, using a double-double subtraction routine. It is possible the final result may be either negative or exceed the modulus 3^{33}, due to the fact that the result of Step 2 might be one unit too high, or one too low. This difficulty can be easily remedied by adding 3^{33}, if the final result is negative, or subtracting 3^{33}, if the final result exceeds 3^{33}.

Exponentiation. The exponentiation required in the initialization may be done efficiently using the binary algorithm for exponentiation. This is merely the formal name for the observation that exponentiation can be economically performed by means of a factorization based on the binary expansion of the exponent. For example, one can write $3^{17} = ((((3^2)^2)^2)^2) \cdot 3$, thus producing the result in only five multiplications, instead of the usual 16. According to Knuth, this technique dates back at least to 200 BCE [4, pg. 461]. In this application, the exponentiation result is required modulo a positive integer k. This can be done very efficiently by reducing modulo k the intermediate multiplication result at each step of the exponentiation algorithm. A formal statement of this scheme is as follows:

To compute $r = b^n \bmod k$, where r, b, n and k are positive integers: First set t to be the largest power of two such that $t \le n$, and set $r = 1$. Then

```
    A: if $n \ge t$ then $r \leftarrow br \bmod k$;   $n \leftarrow n - t$;   endif
    $t \leftarrow t/2$
    if $t \ge 1$ then $r \leftarrow r^2 \bmod k$;   go to A;   endif
```

Note that the above algorithm is performed entirely with positive integers that do not exceed k^2 in size.

A full implementation of the above pseudo-random scheme, which runs on any computer system with IEEE 64-bit arithmetic and a Fortran-90 compiler, can be obtained from the author's website: `http://crd.lbl.gov/~dhbailey/mpdist`. The code is straightforward and can easily be converted to other languages, such as C or Java.

4. Analysis

It can be seen from the above that the recursive sequence generating iterates, which contain successive 53-long segments of binary digits from the expansion of $\alpha_{2,3}$, is nothing more than a special type of linear congruential pseudo-random number generator, a class that has been studied extensively by computer scientists and others [4, pg. 10–26]. In other words, the binary digits of $\alpha_{2,3}$ are "locally" (within a range of indices spanned by successive powers of three) given by a linear congruential generator, with a modulus that is a large power of three.

This observation makes it an easy matter to determine the period P of the resulting generator [4, pg. 17]: as specified above, $P = 2 \cdot 3^{32} \approx 3.706 \cdot 10^{15}$. Note, however, that the binary digits of the resulting sequence will match that of $\alpha_{2,3}$ only if $[a, a + 53n]$, where a is the starting index and n is the number of floating-point results generated, does not include a power of three or come within 100 of a power of three.

This scheme has one significant advantage over conventional linear congruential generators that use a power-of-two modulus: it cleanly avoids anomalies that sometimes arise in large scientific codes, when arrays with dimensions that are large powers of two are filled with pseudo-random data and then accessed both by row and by column (or plane), or which otherwise are accessed by large power-of-two data strides (as in a power-of-two FFT). This is because the pseudo-random data sequence accessed in this manner has a reduced period as thus may be not as "random" as desired. The usage of a modulus that is a large power of three is immune to these problems. The author is not aware of any scientific calculation that involves data access strides that are large powers of three.

The algorithm given in Section 3 is designed for straightforward implementation using IEEE 64-bit arithmetic, which features 53 mantissa bits, by means of "double-double" arithmetic functions. But by using a different implementation scheme, for example, by using 128-bit integer operations, an even stronger generator can be obtained. For example, in the scheme as described above, one double-precision floating-point result, if it is by chance rather small, will include as its tailing bits a few of the leading bits of the next result. While the author is not aware of any application for which this feature would have any material impact, it can be virtually eliminated by generating, say, 64 bits of $\alpha_{2,3}$ for each result instead of only 53 bits. Also, a 128-bit integer implementation would permit an even larger modulus, say 3^{40}, which yields a period that is 2,187 times larger than the scheme defined in Section 3.

5. Performance

As mentioned above, a Fortran-90 implementation of the scheme described in Section 3 is available on the author's website. For comparison purposes, the author implemented the conventional linear congruential generator

$$z_n \quad = \quad (5^{21} \cdot z_{n-1}) \bmod 2^{53},$$

using the same software and programming style. These two codes were then tested on an 2 GHz Apple G5 workstation, using the IBM XLF compiler. This Apple system has a fused multiply-add instruction, so that the critical double $\times$ double equals double-double operation can be done with just two instructions, as mentioned in Section 3. This hardware feature was utilized in both implementations.

The results of these tests are as follows. The program implementing the scheme in Section 3 required 7.96 seconds to generate an array of 100 million double-precision deviates. By comparison, the conventional linear congruential system required 8.20 seconds. In other words, the normal-number based program is actually slightly faster, although the difference in run time is minor.

6. Parallel Implementation

The scheme described in Section 3 is very well suited for parallel processing, a trait not shared by a number of other commonly used pseudo-random schemes. Consider for example an implementation of the above pseudo-random scheme on a distributed memory system. Suppose that k is the processor number and p is the total number of processors used. Assume that a total of n pseudo-random deviates are to be generated, and assume that n is evenly divisible by p. Then each processor generates n/p results, with processor p using as a starting value $a+nk/p$. Note that each processor can quickly and independently generate its own value of z_0.

In this way, the collective sequence generated by all processors coincides precisely with the sequence that is generated on a single processor system. This feature is crucially important in parallel processing, permitting one can verify that a parallel program produces the same answers (to within reasonable numerical round-off error) as the single-processor version. It is also important, for the same reason, to permit one to compare results on, say, on 64 CPUs of a given system with one run on 128 CPUs.

This scheme has already been exploited in a fast Fourier transform (FFT) benchmark that is being prepared as part of the benchmark suite for the High Productivity Computing Systems (HPCS) program, funded by the U.S. Defense Advanced Research Projects Agency (DARPA) and the U.S. Department of Energy. In this case, it is used to generate the initial data for a large three-dimensional FFT, performed on a highly parallel computer system.

7. Conclusion

The new class of normal numbers defined above can indeed be used to fashion an efficient pseudo-random number generator. While the generator above is designed to replicate the binary digits of the constant

$$\alpha_{2,3} = \sum_{k=1}^{\infty} \frac{1}{3^k 2^{3^k}},$$

there is no reason that other constants from this class could not also be used in a similar way. For example, a very similar generator could be constructed by replacing "3" in this formula with "5." The resulting generator generates successive strings of binary digits from the constant $\alpha_{2,5}$. One could also construct pseudo-random generators based on constants that are 3-normal or 5-normal, although one would lose the property that successive digits are precisely retained in separate computer words (which are based on binary arithmetic). The specific choice of multiplier and modulus can be made based on application requirements and the type of high-precision arithmetic that is available (e.g., double-double or 128-bit integer).

This construction has theoretical as well as practical utility, because it provides some insight into why the digits of the normal numbers defined in the introduction behave in a "random" manner—their expansions consist of successive segments of exponentially growing length, within which the digits are given by a specific type of linear congruential generator, with a correspondingly growing period. Such insights may ultimately lead to other results in the field of normal numbers, such as long-sought proofs that common mathematical constants π or $\log 2$ are normal.

References

[1] David H. Bailey and Richard E. Crandall, "Random Generators and Normal Numbers," *Experimental Mathematics*, vol. 11, no. 4 (2004), pg. 527–546.

[2] Jonathan M. Borwein and David H. Bailey, *Mathematics by Experiment: Plausible Reasoning in the 21st Century*, AK Peters, Natick, MA, 2004.

[3] Yozo Hida, Xiaoye S. Li and David H. Bailey, "Algorithms for Quad-Double Precision Floating Point Arithmetic," *15th IEEE Symposium on Computer Arithmetic*, IEEE Computer Society, 2001, pg. 155–162.

[4] Donald E. Knuth, *The Art of Computer Programming*, vol. 2, Addison-Wesley, Boston, 1998.

[5] R. Stoneham, "On Absolute (j, ϵ)-Normality in the Rational Fractions with Applications to Normal Numbers," *Acta Arithmetica*, vol. 22 (1973), 277-286.