Pseudorandom Number Generators

Austin Hill

December 2020

Contents

1 Introduction

The need to be able to generate random numbers is central to many fields, from slot machines in casinos to simulations in physics. True random number generators are typically hardware based and measure physical processes such as thermal noise or radioactive decay. However these methods are relatively slow, making them unsuitable for many applications. They also suffer from a lack of reproducibility: Suppose you ran an experiment using random numbers generated this way - then you would be unable to replicate any findings unless you stored every random number, which would obviously be impractical on a large enough scale. These are just some of the reasons that an alternative in the form of pseudorandom number generators (PRNGs) has been the subject of research ever since the invention of computers.

Possibly the earliest form of random number generation was by using six-sided dice. The first known examples of these date back to around 3000 BC, and were used in games much like we use them today. For research, simply rolling a die or picking a number out of a hat would be impractically slow. One idea was to compile tables of random numbers that could be used for sampling purposes. The first such table was published in 1927 by Tippet. However using the same set of ”random” numbers for every experiment is clearly flawed, since some numbers will inevitably and consistently show up more than others in these tables. With the advent of computers things changed and since around 1950 many hardware based random generators have been used as described above.

The first PRNG was described by von Neumann in 1946. He described a sequence of random numbers by choosing a starting value s_0 with $2n$ digits and obtaining subsequent numbers by squaring the previous value and taking the middle $2n$ digits. This offered the benefits that values could be calculated as fast as a computer could perform arithmetic and that the sequence was reproducible if the starting value was known. However it's output was not sufficiently random for most purposes: For instance a starting value of 1000 would immediately result in all subsequent numbers being zero.

In this essay we begin by introducing the terminology that will be used throughout. We will introduce two of the most commonly used PRNGs and investigate their properties before looking into how we can test the quality of these generators. Our aim is to perform tests checking the suitability of the generators we discuss for Monte Carlo Integration, a numerical method for integrating functions commonly used in higher dimensions.

2 Terminology

PRNGs used for simulations and other non-cryptographically secure applications are typically deterministic algorithms based on the following structure, from L'Ecuyer [1].

Definition 2.1. A *generator* is a structure $G = (S, \mu, f, U, g)$, where S is the finite set of *states*, μ is the probability distribution on S for the *seed* $s_0 \in S$, $f : S \longrightarrow S$ is the *transition function*, U is the finite set of *output* symbols and $g : S \longrightarrow U$ is the *output function.*

A generator operates as follows:

1. Select the seed $s_0 \in S$ according to the distribution μ and let $u_0 = g(s_0)$
2. Define $S = (s_i)$ by $s_i = f(s_{i-1})$ and similarly define $U = (u_i)$ by $u_i = g(s_i)$

Strictly speaking, we have not introduced any restrictions on the randomness of the output of our generator in this definition. We will now introduce the necessary mathematical concepts in order to understand what it means for a generator to be pseudorandom.

Definition 2.2. Given a sequence of n random variables $X_1, \ldots, X_n$ in $I \subset \mathbb{R}$ we say that the sequence is *independent* if

$$F_{X_1,\ldots,X_n}(x_1, \ldots, x_n) = F_{X_1}(x_1) \cdot \ldots \cdot F_{X_n}(x_n) \qquad \forall x_1, \ldots, x_n \in I$$

and we say that the sequence is *identically distributed* (ID) if

$$F_{X_1}(x) = F_{X_k}(x) \qquad \forall k \in \{1, \ldots, n\} \,\text{and}\, \forall x \in I$$

where $F_{X_1,\ldots,X_n}(x_1, \ldots, x_n) = P(X_1 \leq x_1 \wedge \ldots \wedge X_n \leq x_n)$.
If the sequence is *independent and identically distributed*, we denote this (IID).

In this essay we will be looking at deterministic PRNGs, where the sequence of values generated is entirely dependent on the seed. These will typically be ID but clearly not independent. As we will see later on, these can be a very close approximation to IID sequences. The output space U will be typically be either a finite set of integers or a finite set of values between 0 and 1 approximating the standard uniform distribution $U(0, 1)$.

Definition 2.3. The period of a generator $G = (S, \mu, f, U, g)$ is defined as the smallest positive integer ρ such that for some integer $\tau \geq 0$, $s_{\rho+n} = s_\rho$ for all $n \geq \tau$. If this property is satisfied when $\tau = 0$, we say that G is purely periodic.

It should be clear that a good PRNG should have a large period, in particular much larger than the longest sequence that we intend to produce with the generator, since a truly random sequence of values is not periodic.

Definition 2.4. The *least significant bit* is the final bit of a binary integer, sometimes referred to as the *low-order bit.* The *i least significant bits* are the final i bits of a binary integer.

3 Linear Congruential Generators

Definition 3.1. A Linear Congruential Generator (LCG) is a generator G with it's transition function defined by a linear recurrence of the form

$$s_{n+1} = as_n + c \mod m$$

The integer constants s_0, a, c and m define the sequence of states $S = (s_n)$ where

- s_0, $0 \leq s_0 < m$ is the seed
- a, $0 < a < m$ is the multiplier
- c, $0 \leq c < m$ is the increment
- m, $m > 0$ is the modulus

When $c = 0$ we refer to this as a Multiplicative Linear Congruential Generator (MLCG).

We shall now discuss two common choices of parameters for LCGs.

3.1 m is a power of two, $c \neq 0$

This choice of m is primarily chosen due to the simplicity of calculating the modulus by simply truncating the binary form of the value. We choose $c \neq 0$ because if $c = 0$, the period cannot be larger than $\frac{m}{4}$. In order to ensure that this has maximal period, we require the following theorem, adapted from Hull and Dobell [2].

Theorem 3.2. *A LCG as defined above has maximal period m if and only if*

- *(i) c is an odd number greater than zero*
- *(ii)* $a \equiv 1 \pmod 4$

Proof. Suppose $a = 1$. Then solving $s_n \equiv s_0 + nc \equiv 0 \pmod m$ for the smallest value of n simply gives us $n = m$ because c does not have any factors of two. This of course means that the LCG has maximal period, so we can assume $a \neq 1$. We need to find the smallest value of n satisfying

$$s_n \equiv a^n s_0 + \frac{(a^n - 1)c}{a - 1} \equiv 0 \pmod m$$

Which can be rearranged to

$$\frac{(a^n - 1)(s_0(a - 1) + c)}{a - 1} \equiv 0 \pmod m$$

Note that a and c are odd, so $(s_0(a - 1) + c)$ is odd and we can simplify this to

$$\frac{a^n - 1}{a - 1} \equiv 0 \pmod m \tag{1}$$

Set $a = 1 + 2^\beta k$ for some $\beta \geq 2$ using *(ii)* and the fact that $a \neq 1$. Now let write $m = 2^\alpha$ and let $m = n$. Then

$$\begin{aligned}
\frac{a^n - 1}{a - 1} &= \frac{(1 + 2^\beta k)^n - 1}{2^\beta k} \\
&= \frac{1}{2^\beta k}[1 + 2^\beta k 2^\alpha + 2^{2\beta} k^2 \frac{2^\alpha(2^\alpha - 1)}{2!} + \ldots + 2^{\beta 2^\alpha} k^{(2^\alpha)} - 1] \\
&= 2^\alpha + 2^\beta k \frac{2^\alpha(2^\alpha - 1)}{2!} + \ldots + (2^\beta k)^{2^\alpha - 1}
\end{aligned}$$

To show that this is divisible by 2^α, we see that the first term is divisible and then write the ith term as

$$2^\alpha [\frac{2^{i\beta}(2^\alpha - 1)\ldots(2^\alpha - i + 1)k}{i!}] \qquad \text{for } i \geq 2 \tag{2}$$

and note that the number of factors of two in $i!$ is at most $\frac{i}{2} + \frac{i}{4} + \frac{i}{8} + \ldots = i$. We know that $\beta \geq 2$, so $2^{i\beta}$ has more than i factors of two and hence the value in the square brackets is an integer, so every term is divisible by 2^α and (1) is satisfied. All we need to show now is that this is the smallest value of n giving maximal period. Note that n will satisfy (1) if and only if it is a multiple of the smallest solution ρ (the period). So we simply need to show that $n = 2^{\alpha - 1}$ is not a solution to (1). We can do the binomial expansion again and each term will be the same as in (2), but replacing 2^α with $2^{\alpha-1}$. So we get the ith term as

$$2^{\alpha-1}[\frac{2^{i\beta}(2^{\alpha-1} - 1)\ldots(2^{\alpha-1} - i + 1)k}{i!}] \qquad \text{for } i \geq 2$$

$$= 2^{\alpha}[\frac{2^{i\beta-1}(2^{\alpha-1}-1)\dots(2^{\alpha-1}-i+1)k}{i!}] \qquad \text{for } i \geq 2$$

similarly, we see that $i\beta - 1 > i$ since $\beta \geq 2$ so $2^{i\beta-1}$ has at least i factors of two and so all terms $i \geq 2$ are divisible by 2^{α}. However, the first term $2^{\alpha-1}$ is not, so (1) is not satisfied and the proof is complete. □

This type of generator has some flaws, one of which we shall now explore.

Lemma 3.3. *The period of the i least significant bits of the LCG is at most* 2^i

Proof. Consider an LCG defined by the recurrence $s_{n+1} = (as_n + c) \pmod{m}$, where $m = 2^k$. Let $m' = 2^i$ for $1 \leq i \leq k$. Then the sequence defined by $x_n = s_n \pmod{m'}$ is equivalent to taking the i least significant bits of the LCG for each term in the sequence. Then

$$\begin{aligned} s_{n+1} &= a(x_n + pm') + c - qm && \text{for p, q} \in \mathbb{Z} \\ \implies s_{n+1} \mod m' &= ax_n + c + apm' - qm \mod m' && \\ \implies x_{n+1} &= ax_n + c \mod m' && \text{since } m' \mid m \end{aligned}$$

Clearly the sequence (x_n) of i least significant bits can take at most m' values and each term is completely determined by the previous term, so the period is at most $m' = 2^i$. □

Some implementations of LCGs where m is a power of two will combat this lack of randomness in the lower order bits by simply discarding a number of the lowest order bits. In this essay, we are only interested in how well distributed the numbers are, hence the lack of randomness in the low order bits is relatively unimportant.

3.2 m is prime, $c = 0$

This form of LCG, originally suggested by Lehmer in 1951, does not suffer from the same short period for the lower bits as in the case above. These generators are slower due to the need to perform division with remainder by a prime number m. Most generators of this form choose m to be a Mersenne prime, i.e. $m = 2^k - 1$ such that m is prime for some integer k; there are algorithms to compute the prime modulus more efficiently in this case. We will not discuss this form of generator due to it's comparative lack of speed, since the randomness of the low-order bits is relatively unimportant for our purposes.

3.3 Notable flaws of LCGs

A problem which all LCGs face is illustrated by the following theorem, from Marsaglia [3]. The proof is beyond the scope of this essay.

Theorem 3.4. *(Marsaglia's Theorem) Define an LCG $G = (S, \mu, f, U, g)$ by $s_{i+1} = (as_i + c) \mod m$, and define the output space $U = (u_i)$ by $u_i = \frac{s_i}{m}$. Define the sequence of points $P = (\pi_i)$ by $\pi_i = (u_i, \ldots, u_{i+n})$ such that the points of P lie in a unit $n-$cube. Then all the points of p will lie in at most $\sqrt[n]{n!m}$ hyperplanes.*

An example of an LCG which is poorly distributed when used to generate points in three dimensions in this way is the RANDU generator, created by IBM in the 1960s. The values chosen for this generator were $a = 65539$, $c = 0$ and $m = 2^{31}$, typically used with the output funcion $g(s) = \frac{s}{m}$. It turns out that when consecutive outputs are used to generate 3-tuples, all of these points lie within 15 planes in $\mathbb{R}^3$.

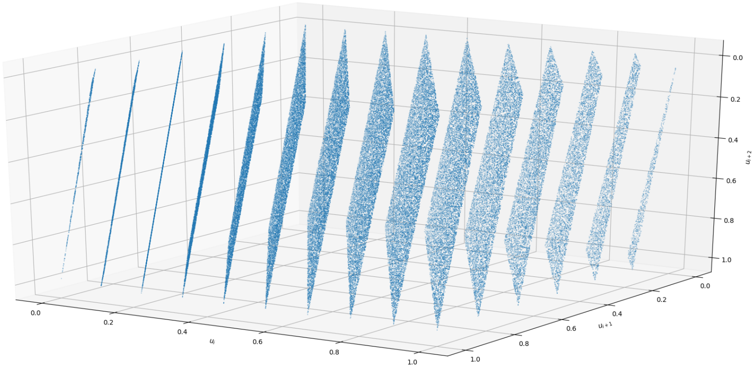

Figure 1: Distribution of 3D points generated using RANDU

For tests in section 5, we will be using a 64 bit LCG with parameters $a = 2862933555777941757, m = 2^{64}, c = 1$ chosen from L'Ecuyer [4]

4 Linear-Feedback Shift Registers

We begin with the following definition, from Brent [5]

Definition 4.1. A *linear-feedback shift register* (LFSR) is a sequence of scalars or vectors (s_i) satisfying the linear recurrence

$$\sum_{k=0}^{d} \alpha_k s_{i-k} = 0 \qquad \text{for } i \geq d$$

Where $\alpha_0, \alpha_1, \ldots, \alpha_d \in \mathbb{F}_2$ and $\alpha_0 \coloneqq 1$.

Note that in this definition, to determine the sequence (s_i) we must know the value of $s_0, \ldots, s_{d-1}$, implying that the seed of the generator is actually a set of initial states. We will however only be looking at cases where only one seed s_0 is required.

Remark 4.2. Define a sequence of binary vectors (s_i) by $s_{i+1} = s_i A$ for some matrix $A \in M_n(\mathbb{F}_2)$ so that $s_i = s_0 A^i$. Then we can write the minimal polynomial of A as

$$\mu_A(A) = \sum_{k=0}^{d} \alpha_k A^{d-k} = 0$$

chosen so that $\alpha_0 = 1$. Multiplying on the left by $s_0 A^{i-d}$ for $i \geq d$, we get

$$\begin{aligned} &\sum_{k=0}^{d} \alpha_k s_0 A^{i-k} = 0 \qquad \text{for } i \geq d \\ \implies &\sum_{k=0}^{d} \alpha_k s_{i-k} = 0 \qquad \text{for } i \geq d \end{aligned}$$

Hence the vectors (s_i) satisfy the requirements of being an LFSR.

We will now discuss one of the fastest and most commonly used LSFRs, first introduced by Marsaglia [6].

4.1 Xorshift PRNGs

Definition 4.3. An Xorshift PRNG is a generator G where each subsequent state is generated by repeatedly taking the exclusive-or (XOR) of the previous state with the bit-shifted version of itself. We refer to this process as an xorshift operation on a binary integer or vector of binary bits.

The seed s_0 of G is given by a binary vector chosen at random from the uniform initial distribution μ of binary vectors $(b_1, \ldots, b_n)$ over the field $\{0, 1\}$ so that vector addition is equivalent to the XOR function. n is typically a multiple of 32 or 64 in order to fit modern computer architectures. We choose our transition function to be the invertible linear map T represented by the $n \times n$ nonsingular binary matrix A. Now we must look at how to ensure this type of generator achieves sufficient period. The following proof follows Marsaglia [6].

Theorem 4.4. *Let A be a nonsingular $n \times n$ binary matrix. Then the sequence $S = s_0, s_0 A, s_0 A^2 \ldots$ has period $2^n - 1$ for every non null initial state s_0 in μ if and only if the order of A in the group of nonsingular $n \times n$ binary matrices is $2^n - 1$.*

Proof. Suppose the the period of S is $\rho = 2^n - 1$. Then $s_0 A^\rho = s_0$ and hence $s_0(A^\rho - I_n) = 0$ for all (non null) binary vectors s_0. So the nullspace of $A^\rho - I_n$ has dimension n, giving $A^\rho = I_n$. Notice that if $A^k = 0$ for some $k < \rho$, the the period of S would be less than ρ and we have finished the first implication.

Moving on to the reverse implication, suppose that the order of A is $\rho = 2^n - 1$. Define the set of distinct powers of A by

$$P = \{I_n, A, A^2, \ldots, A^{\rho-1}\}$$

Let $\mu_A(x)$ be the minimal polynomial of A with degree $t \leq n$. Let Q be the set of binary polynomials in T of degree less than or equal to t. It is clear that since we are dealing only in binary coefficients, $|Q| = 2^t$. We can now write elements of P as elements of Q as follows:

$$T^s = q(T)\mu_A(T) + r(T) \qquad\qquad \text{where } deg(r) < t$$

Hence P is a subset of $Q \setminus \{0\}$ giving

$$2^n - 1 = |P| \leq \big|Q \setminus \{0\}\big| = 2^t - 1$$

with the condition $t \leq n$ giving us $t = n$ and hence $P = Q \setminus \{0\}$. We know that the elements of P are nonsingular, so the elements of $Q\setminus\{0\}$ are also nonsingular. Note that $A^k s_0 = s_0$ implies $A^k s_0 + s_0 = (A^k + I_n)s_0 = s_0 + s_0 = 0$ which requires $A^k + I_n$ to be singular, so cannot be in P if $k < \rho$. Hence the sequence $S = s_0, s_0A, s_0A^2 \ldots$ has period ρ for every non null s_0. This completes the second implication. □

In practice, we look at a special case where A represents a sequence of xorshift operations, since these are very fast to compute. Let L be a binary matrix of zeros except for ones on the principle subdiagonal. It is clear that computing xL gives us x, bit-shifted one position to the left. Hence we can reperesent a bit shift of a places by computing xL^a. Finally, we compute x XOR xL^a by the equivalent operation $x(I_n + L^a)$. We can define a similar operation with a right bit-shift using the matrix $R = L^T$ by $x(I_n + R^a)$. Using the theorem above, it is simply a matter of testing all possible combinations of these matrices to find a matrix A with order $2^n - 1$. It turns out that when n = 32 or n = 64 as is often the case, there are no combinations of two matrices that provide a good quality output (Brent [5]). For this reason, most generators in this family use a combination of xorshifts represented by the matrix

$$A = (I_n + L^a)(I_n + R^b)(I_n + L^c)$$

For a suitable choice of a, b and c. For tests of this generator in section 5, we chose $a = 18, b = 31, c = 11$ based on the test results from Vigna [7].

4.2 Scrambling the output

Although the xorshift generators described above are very fast and ended up passing all of the tests described in section 5, they certainly do not pass the tests described by L'Ecuyer and Simard [8]. An alternative suggested by Vigna [7] is to use an output function $g : S \longrightarrow S$ that is nonlinear in $\mathbb{Z}/2\mathbb{Z}$. We would like this function to produce a permutation of the output in order to guarantee full period, so g must be invertible. A function satisfying this would be $g(s) = ks \mod m$ for any element k (the multiplier) which generates S. For suitable choices of xorshift parameters and multiplier, this generator fares a lot better in the tests described in [8]. We chose the parameters $a = 12, b = 25, c = 27$ with multiplier $k = 2685821657736338717$ based on the results from Vigna [7]. We will refer to this generator as xorshift 64*.

5 Quality Testing

There is a limit to what we can mathematically prove about the properties of a generator. Typically, we should always endeavour to show that a generator has sufficient period for every possible seed and that the output is uniform and identically distributed. However, we must then implement generators and perform hypothesis tests to see if the sequences generated have the necessary statistical properties for our application. Here we will look at one particular application as described below.

Definition 5.1. *Monte Carlo integration* is a way of calculating definite integrals, where values are drawn at random from a simple domain D that contains the domain of integration D_I. We can count the number of points lying in D_I and calculate the integral I by the following formula:

$$I = \frac{mV_D}{n}$$

Where m is the number of points lying in D_I, n is the total number of points sampled and V_D is the area or n-dimensional volume of D.

This method is particularly useful for integrals in higher dimensions, where traditional methods of integration are very slow. However care must be taken to ensure that a PRNG is suitable for this purpose. For instance, using the RANDU generator as described above to calculate the volume of a sphere would clearly be completely unsuitable. Hence we must test that the values generated by our PRNG are uniformly distributed over the output space which we will assume to be the unit hypercube (other output spaces could be achieved by using a suitable output function g). The following test is from L'Ecuyer and Simard [8].

5.1 Serial tests for uniformity

A simple way to do this is to partition the set $[0, 1]^d$ of all d-tuples in the unit hypercube into $k = l^d$ smaller hypercubes of side length l and volume $v = \frac{1}{k}$. The test consists of generating n points $\boldsymbol{u}_i = (u_{di}, u_{di+1}, \ldots, u_{di+d-1})$, and counting the number x_j lying in the jth smaller hypercube (we can order them arbitrarily). Clearly, if the points follow a uniform distribution then the expectation of x_j is $E = \frac{n}{k}$. Let the null hypothesis H_0 be that the points $\boldsymbol{u}_i$ follow a uniform distribution over the unit hypercube for $0 \leq i \leq n-1$. We can compute the chi-squared test statistic

$$\chi^2 = \sum_{j=0}^{k-1} \frac{(x_j - E)^2}{E}$$

which has a chi-squared distribution under H_0 with $k-1$ degrees of freedom if $n \longrightarrow \infty$ for a fixed k. Since we often need k to be large in order for the test to be effective, it is preferable to have n as small as possible in order to minimise the runtime of the test. We only need approximate p-values, so will be using the common rule $\frac{n}{k} \geq 5$.
We can use this to test some of the generators described earlier. The following table shows the χ^2 p-values given by the serial tests with $n = 2^{30}$.

Parameters	LCG 64	Xorshift 64	Xorshift 64*	RANDU
$d = 1, k = 2^{27}$	0.95356	0.93023	0.80430	0.00000
$d = 2, k = 2^{26}$	0.74279	0.28127	0.36475	0.00000
$d = 3, k = 2^{27}$	0.37732	0.34886	0.10554	1.00000

From this table, we can see that all three of our chosen "good" generators give acceptable p-values and hence we conclude that they are well distributed in up to three dimensions up to the precision offered by our choice of partition when used to generate 2^{30} values. However, the RANDU generator failed all of the tests; in up to two dimensions, the p-value of 0.000 indicates that the distribution was too close to a perfect uniform distribution to approximate a random sequence. This is because the period of this generator is $\frac{2^{31}}{4} = 2^{29}$, which means it completed two full periods during the test. In three dimensions, the p-value of 1.000 shows the result we discussed earlier, with the points generated not being at all well distributed in three dimensions.

This approach is very effective for testing uniformity, however it is very computationally intensive in higher dimensions and hence is of limited use. For instance, if we wanted to divide each side of a four dimensional hypercube into 1000 we would have a total of 1000^4 cells; if we counted collisions in each of these cells and stored them as an 8 bit integer, we would require a terabyte of computer memory! Another test from L'Ecuyer and Simard [8] is a way of getting around this problem.

5.2 Single cell tests

A way of getting around this is to only count collisions X with a single cell chosen at random from the output space. Clearly, if a generator offers a good approximation of a uniform random distribution then the expected value number of collisions for n samples would be $\lambda = \frac{nv}{V}$, where v is the volume of the cell and V is the volume of the output space. For a true uniform random generator, the number of collisions X has an approximately Poisson distribution $Po(\lambda)$ for large n. We can then use this test with the null hypothesis as before to determine whether any cells are visited too frequently or rarely, assuming that λ is large enough. The below table illustrates the results of this test with $n = 2^{30}$, where d is the number of dimensions and $\delta = \frac{v}{V}$.

Parameters	LCG 64	Xorshift 64	Xorshift 64*	RANDU
$d = 1, \delta = 2^{-24}$	0.384548	0.483376	0.629766	0.533179
$d = 2, \delta = 2^{-24}$	0.337056	0.881036	0.249352	0.533179
$d = 3, \delta = 2^{-24}$	0.210230	0.210230	0.337056	0.000000
$d = 4, \delta = 2^{-24}$	0.433572	0.793843	0.903062	0.000000
$d = 5, \delta = 2^{-25}$	0.790100	0.274123	0.338006	0.000000

We can see that this test is not as strong as the serial test because the RANDU generator passed the test in one and two dimensions. However, it easily shows the lack of uniformity of this generator in 3 or more dimensions, which is what we are most interested in measuring. We can see that the other generators chosen passed this test for the given δ.

5.3 Probabilistic tests

Testing just a single cell in this way makes for a much faster and more practical test, however it must be repeated a number of times to be confident in the result (think about the RANDU generator in three dimensions; the test cell could easily overlap the right amount with a hyperplane to give an acceptable number of ”hits”). A more powerful test is to count collisions with a fixed sample of cells from the output space and perform a Chi-Squared test exactly as described in the serial test, but for only a sample of cells. The table below illustrates the result of this test using a sample of 20 cells, with the same parameters as in the single cell tests.

Parameters	LCG 64	Xorshift 64	Xorshift 64*	RANDU
$d = 1, \delta = 2^{-24}$	0.672618	0.623083	0.865398	0.000000
$d = 2, \delta = 2^{-24}$	0.797172	0.851435	0.961652	0.000000
$d = 3, \delta = 2^{-24}$	0.626749	0.019519	0.502466	1.000000
$d = 4, \delta = 2^{-24}$	0.775481	0.724214	0.734008	1.000000
$d = 5, \delta = 2^{-25}$	0.487975	0.721921	0.572842	1.000000

5.4 Example simulation

Let us now illustrate the results of these tests by using each generator to estimate the the unit hypersphere in a range of dimensions d. We will draw n points from $U(-1, 1)$ in each dimension, and count the number m of points satisfying $\sqrt{\sum_{i=1}^{d} x_i^2} \leq 1$. The volume can be calculated as above, $V = \frac{2^d m}{n}$.

d	LCG 64	Xorshift 64	Xorshift 64*	RANDU
2	-3.5932×10^{-6}	7.9105×10^{-5}	-3.9471×10^{-6}	1.9173×10^{-7}
3	-1.6332×10^{-4}	-1.0876×10^{-4}	-1.3007 $\times 10^{-4}$	-2.8792×10^{-3}
4	8.0738×10^{-5}	-6.4793×10^{-4}	3.8420×10^{-4}	-1.7994×10^{-3}
5	4.4276×10^{-4}	4.1889×10^{-4}	-1.2188×10^{-4}	-4.7955×10^{-3}

From this we can see that all of the generators provided a good estimate of the result, except for the RANDU generator for $d \geq 3$. This is in agreement with the single cell and probabilistic tests above.

6 Conclusion

The most important thing to take from this essay is the need to test the suitability of a PRNG for a given application. In some cases, a faster yet poorer quality generator would be sufficient. For instance the RANDU generator may be suitable for generating a smaller sample of random numbers in one dimension - perhaps to colour textures in a video game - but should certainly not be used for a large scale Monte Carlo integration in three or more dimensions. Based on my results, the other three generators discussed would be suitable for Monte Carlo integration in up to five dimensions. Out of these, the Xorshift 64 generator was the fastest when implemented in Microsoft Visual C++.

References

[1] Pierre L'Ecuyer, *Random Numbers for Simulation*, Communications of the ACM, Volume 33, Issue 10 (1990); https://doi.org/10.1145/84537.84555

[2] T. E. Hull, A. R. Dobell, *Random Number Generators*, SIAM review, Volume 4 No. 3 (1962); https://doi.org/10.1137/1004061

[3] George Marsaglia, *Random Numbers Fall Mainly in the Planes*, PNAS, Volume 61, Issue 1, pp.25-28 (1968); https://doi.org/10.1073%2Fpnas.61.1.25

[4] Pierre L'Ecuyer, *Tables of Linear Congruential Generators of Different Sizes and Good Lattice Structure*, Mathematics of Computation, Volume 68, No. 225, pp.249–260 (1999); https://doi.org/10.1090/S0025-5718-99-00996-5

[5] Richard P. Brent, *Note on Marsaglia's Xorshift Random Number Generators*, Journal of Statistical Software, Volume 11 Issue 5 (2004); http://dx.doi.org/10.18637/jss.v011.i05

[6] George Marsaglia, *Random Number Generators*, Journal of Modern Applied Statistical Methods, Volume 2 Issue 1 (2003); https://doi.org/10.22237/jmasm/1051747320

[7] Sebastiano Vigna, *An Experimental Exploration of Marsaglia's xorshift Generators, Scrambled*, ACM Transactions on Mathematical Software, Article No. 30 (2016); https://doi.org/10.1145/2845077

[8] L'ecuyer, P. and Simard, R. 2007. *TestU01: A C library for empirical testing of random number generators.* ACM Trans. Math. Softw. 33, 4, Article 22 (August 2007), 40 pages; http://doi.acm.org/10.1145/ 1268776.1268777