CHAPTER 5

RANDOM NUMBER GENERATORS

Here we briefly discuss concepts related to the generation of "random" numbers via computer, which we will need to carry out Monte Carlo simulation. The word random is in quotes because calculations performed by computer are not truly random — the computer simply follows instructions coded up by the user. Each time the same set of instructions is executed, the same result ensues. In fact, to emphasize this point many refer to computer generated random numbers as *pseudo-random* numbers. I prefer to drop the cumbersome modifier and simply remember that the numbers generated by random number generators are not really random — they just "look" random in a sense to be discussed below. The subject of random number generators is quite large and lies at the intersection of number theory and computer science. We barely scratch the surface here.

We remark that there are also *hardware random number generators* that rely on inherently random physical phenomena such as quantum mechanical effects to generate random numbers — typically random sequences of zeros and ones. This may be thought of as flipping a coin at the atomic or subatomic level. These are very different from the algorithmic approaches we consider here.

§1. The Basic Idea

We seek a computer-friendly algorithm to generate a very long — infinite for all practical purposes — sequence of numbers $U_1, U_2, \ldots$ between 0 and 1 that statistically behave as if they are independent and uniformly distributed on the interval $(0, 1)$.

The general procedure is to produce an algorithm to construct a sequence of integers $n_1, n_2, \ldots$ that are drawn seemingly randomly (i.e., uniformly and independently) from the numbers $S = \{0, 1, 2, \ldots, m-1\}$, where m is a very large integer. We may then scale them to numbers that are (approximately) uniformly distributed on $(0, 1)$ by taking, for example,

$$U_i \;=\; \frac{n_i + 0.5}{m}, \quad i = 1, 2, \ldots \tag{1}$$

Since the numerator in (1) lies between 0.5 and $m - 0.5$, the value of U_i will be strictly bigger than 0 and less than 1. As a numerical matter, this is useful in some settings. For example, if $U \sim \text{Uniform}\,(0, 1)$, then, as we will see,

$-\log(U) \sim$ Exponential (1), offering a nice way to generate exponential random variables from uniform random variables. If a computer command calls the logarithm function with an argument value 0, a run-time "domain error" warning is issued and program execution is compromised.

One way to generate the sequence n_i is to construct a function $h : S \to S$, as presently illustrated. The process is then *seeded* with a number $s \in S$ and we take $n_1 = h(s)$, $n_2 = h(n_1)$, $n_3 = h(n_2)$, and so forth. In other words,

$$n_{i+1} = h(n_i), \quad \text{for } i \geq 0 \text{ where } n_0 = s. \tag{2}$$

Notice that this process must eventually produce a repetition since S is a finite set. When this happens, the procedure goes into a loop. For example, if $n_j = n_i$ where $j > i$, then $n_{j+1} = h(n_j) = h(n_i) = n_{i+1}$, so necessarily $n_{j+1} = n_{i+1}$. But then, also, $n_{j+2} = n_{i+2}$, etc. The size of this loop, which is at most m, is called the *period*. A good random number generator satisfies the following criteria:

(i) m is large so that (1) produces a discrete distribution that closely approximates a continuous distribution;

(ii) for each seed s, the period should be as long as possible (hopefully m) so that the sequence does not quickly repeat;

(iii) the function $h(\cdot)$ should be efficient to implement on a computer so the algorithm runs fast; and

(iv) the resulting sequence U_i should statistically behave like iid Uniform $(0, 1)$s.

Most common random number generators follow some variation of this paradigm. Presently we briefly discuss the *mechanics* of three classes of generators. The underlying *mathematics* (i.e., *why* they work) is well beyond the scope of this text.

Linear Congruential Generators. For *linear congruential generators* (LCGs), we take

$$h(n) \;=\; (an + c) \bmod m, \tag{3}$$

where each number in (3) is a non-negative integer and a, called the *multiplier*, satisfies $1 < a < m$ and c, called the *increment*, satisfies $0 \leq c < m$. For integers j and m, the notation '$j \bmod m$' denotes the remainder when j is divided by m. For example, $8 \bmod 3 = 2$ and $10 \bmod 5 = 0$. Mod is short for the word *modulo*. Generally, m is taken to be a power of 2 because this works well with computer architecture (computers work in base 2). Often c is taken to be 1. It turns out that if m is a power of 2 and $c = 1$, then the period will be as large as possible (i.e., m) if $a - 1$ is a multiple of 4. (This fact comes under the heading of number theory and is beyond the scope of this text.)

For the LCGs, (2) and (3) may be combined as

$$n_{i+1} \;=\; (an_i + c) \bmod m, \quad i \geq 0, \tag{4}$$

where $n_0 = s$, the seed.

Example 1. For a concrete example, take $m = 2^4 = 16$, $c = 1$, and $a = 5$. With a seed of $n_0 = s = 6$, beginning with n_0 (4) produces the sequence

$$6, 15, 12, 13, 2, 11, 8, 9, 14, 7, 4, 5, 10, 3, 0, 1, 6, 15, 12, 13, 2, \dots$$

The period here is 16 (maximal) as $n_{16} = n_0 = 6$, with no earlier repetition. In this situation $a - 1 = 4$, which is a multiple of 4, and number theory tells us that the period must be maximal.

Of course, in this silly example our choice of m is ridiculously small. A common version of the LCG takes $m = 2^{32} = 4{,}294{,}967{,}296$, $c = 1$, and $a = 22{,}695{,}477$ ($a - 1$ is a multiple of 4 so the period, which is m, is maximal). The number 2^{32} is used for m because this facilitates the mod m computation in (4). With many compiler/hardware combinations, in C or C++ the largest unsigned integer, and also unsigned long integer, is the 32 bit (a bit is a 0 or 1) integer

$$1111, 1111, 1111, 1111, 1111, 1111, 1111, 1111 \text{ (base 2)},$$

which is $2^{32} - 1 = 4{,}294{,}967{,}295$ (base 10). I have grouped the base 2 digits in blocks of 4 to make it easy to see that there are 32 bits.

The code N = `22695477 * N + 1` implements (4) for this choice of a and c. This computation corresponds to (4), except that the 'mod 4294967296' is missing. That is because it is unnecessary. The number `22695477 * N + 1` will typically be huge, requiring in excess of 32 binary digits to hold. In doing the computation, the computer retains only the rightmost 32 binary digits of the number — the rest simply disappear. (This is called *overflow*.) But the value of these 32 digits is precisely the remainder when $22695477N + 1$ is divided by 2^{32}. We observe the same phenomenon in base 10. For example, when 53,467 is divided by $1000 = 10^3$, the remainder is 467 — the 3 rightmost digits.

Donald Knuth's 64 Bit LCG. Donald Knuth (the "K" is not silent — incidentally he created the TeX mathematics word processor that typeset these words) has proposed an LCG for compiler/hardware combinations where an unsigned long integer contains 64 bits rather than 32. The largest integer in this regime is $2^{64} - 1 = 18{,}446{,}744{,}073{,}709{,}551{,}615$. For this LCG, in (4) we have $m = 2^{64}$, $a = 6{,}364{,}136{,}223{,}846{,}793{,}005$, and $c = 1{,}442{,}695{,}040{,}888{,}963{,}407$. The period of this 64 bit LCG is a whopping 1.8×10^{19}. This LCG does well when subjected to various statistical tests for uniformity and independence. However, my compiler (and probably yours) does not take advantage of 64 bit hardware and unsigned long integers are only 32 bits. Consequently my implementation of this LCG (found in the accompanying code) is relatively inefficient.

§2. A More General Framework

Many RNGs have a slightly more general structure. Instead of cycling through a set of numbers $\{0, 1, 2, \dots, m-1\}$, they cycle through an often different collection of objects which we will still call S. As before, there is a function $h : S \to S$ and a seed $s \in S$. Then, as before, $s_0 = s$ and, for $i > 0$, $s_i = h(s_{i-1})$. Additionally however, there is a function $n : S \to N$, where $N = \{0, 1, 2, \ldots, m-1\}$. For each i, it is the number $n_i = n(s_i)$ that is inserted in (1) to produce U_i. Generally, though not in our first example below, S will be much larger than the 2^{32} numbers cycled through by our 32 bit LCG. This will allow for much longer periods depending on how large S is. In Figure 1 each '$\to$' is an application of $h(\cdot)$ and each '$\downarrow$' is an application of $n(\cdot)$.

Figure 1.

$$\begin{tikzcd}
s_0 \arrow[r] & s_1 \arrow[r] \arrow[d] & s_2 \arrow[r] \arrow[d] & s_3 \arrow[r] \arrow[d] & \cdots & \in & S \\
 & n_1 & n_2 & n_3 & \cdots & \in & N
\end{tikzcd}$$

Borland's Variation. A variation of the 32 bit LCG is implemented by Borland's C/C++ compiler in the `rand()` function. There, $S = \{0, 1, \ldots, 2^{32} - 1\}$ and $h(s) = (22695477s + 1) \bmod 2^{32}$. This is exactly the 32 bit LCG discussed above. However, the `rand()` function reports as n_i the portion of s_i obtained by restricting attention to the bits in locations 17 to 31, reading from right to left. For example, if the base 2 representation of the number s_i were 1$\underline{\text{010,1100,0010,1000}}$,1010,0110,0011,0101, the value reported by `rand()` would be $n_i = n(s_i) =$ 10,1100,0010,1000 (base 2), which is 11,304 (base 10). Apparently, these digits exhibit superior qualities of "randomness" — better satisfying criterion (iv) above. The function `rand()` does not scale to the interval $(0, 1)$ as in (1), but returns an integer ranging in value from 0 (in either base) to 111,1111,1111,1111 (base 2) which is $2^{15} - 1 =$ 32,767 (base 10). This RNG's period is still 4,294,967,296 — even though it only reports 32,768 distinct values — because S has 4,294,967,296 elements (the numbers 0 through 4,294,967,295).

Multiply-With-Carry RNGs. The 32 bit LCG generator discussed above has a period of over 4 billion. This may seem large, but as we shall see it is not large enough. Multiply-With-Carry (MWC) are a class of generators introduced by Marsaglia and Zaman (1991) that have a substantially longer period. For the LCG generators of (4) the value of c is held constant throughout the process. The simplest MWC generators additionally update the number c with each iteration:

$$n_{i+1} = (an_i + c_i) \bmod m, \text{ and } c_{i+1} = \left\lfloor \frac{an_i + c_i}{m} \right\rfloor, \quad i \geq 0, \tag{5}$$

where $n_0 = s$ and c_0 is also assigned some initial value, often 1. Here c_{i+1} is the (integer) quotient and n_{i+1} is the remainder when $an_i + c_i$ is divided by m.

Example 2. To illustrate, we again take $m = 2^4 = 16$, but here with $a = 15$ and and initial values $n_0 = s = 5$ and $c_0 = 1$ producing the sequences:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	$\cdots$
n_i	12	8	3	5	14	6	7	15	7	7	0	7	9	13	11	$\cdots$
c_i	4	11	8	3	4	13	6	6	14	7	7	0	6	8	12	$\cdots$

We carry out the first computation: $i = 1$. Here $an_0 + c_0 = 15 \cdot 5 + 1 = 76$. When 76 is divided by 16 (the value of m), the remainder is $n_1 = 12$ — the first number in the n row. Also, $\frac{76}{16} = 4.75$, whose integer part is $c_1 = 4$ — the first number in the c row. When 76 is divided by 16, the quotient is 4 and the remainder is 12, i.e., $76 = 4 \cdot 16 + 12$. One more, with $i = 2$. Here $an_1 + c_1 = 15 \cdot 12 + 4 = 184$. When 184 is divided by 16, the remainder is $n_2 = 8$, and $\frac{184}{16} = 11.5$ whose integer part is $c_2 = 11$, i.e., $184 = 11 \cdot 16 + 8$.

A couple of observations follow. First, if $c_0 < a$ (as it is if $c_0 = 1$), then we will have $c_i < a$ for all $i \geq 0$. To see this, suppose $c_i < a$. Then, since each $n_i \leq m - 1$, we get that

$$c_{i+1} = \left\lfloor \frac{an_i + c_i}{m} \right\rfloor \leq \frac{an_i + c_i}{m} < \frac{a(m-1) + a}{m} = a.$$

Notice that 7 in the n_i row appears multiple times without the sequence repeating. This is because they occur with different values of c_i. For example, $n_7 = n_9 = 7$. However, $c_7 = 6$ but $c_9 = 14$, so the computations for $i = 8$ and $i = 10$ are different. This algorithm cycles through a subset of the pairs

$$S = \{(n, c) : 0 \leq n < m,\ 0 \leq c < a\}.$$

Here $m = 16$ and $a = 15$, so there are 240 such pairs and that is the maximum possible period. The actual period is 119 because the algorithm does not cycle through all of S. Next, notice that some numbers (7, for example) are over-represented in the sequence while others are then necessarily under-represented. This particular choice of m and a does not produce a sequence n_i that is uniformly distributed on the values $\{0, 1, \ldots, 15\}$.

In actual practice, with $m = 2^{32}$, the choice $a =$ 4,294,967,118 works well. For this choice of m and a, S has 1.8×10^{19} elements and the period is approximately 10^{19} — a huge number. The algorithm is also quite efficient — on my notebook computer I can generate roughly 21 million random numbers per second using this algorithm. Yet, at that rate, it would take about 15 thousand years to run through a single period of the sequence.

The Mersenne Twister. The *Mersenne Twister* (MT), an algorithm developed by M. Matsumoto and T. Nishimura in 1998, is more complicated. We vaguely describe it here — feel free to skip this paragraph. For the LCG and MWC generators, the sequence of 32 bit unsigned integers generated are thought of as numbers, held by the computer in base two. For the MT, these unsigned integers are best thought of as vectors with 32 components each of which is a 0 or 1. Each unsigned integer thus has a dual interpretation — as both a number and a vector. Two such vectors may be added componentwise to produce a third such vector — provided the addition is modulo two, so $1 + 1 = 0$. If x and y are two such vectors, $x \oplus y$ will denote this sum (in C it is `x ^ y`). When the MT is seeded, numbers $x_1, x_2, \ldots, x_{624}$ are generated by an LCG (or any other generator). For $k > 624$, the vector x_k is generated as follows. First, the leftmost component of x_{k-624} is concatenated with the rightmost 31 components of x_{k-623} producing a vector y. Then, a function A is applied to y producing another vector $A(y)$. The function A is actually an invertible linear transformation that takes advantage of computer architecture so as to be easily (quickly) computed. We then put $x_k = A(y) \oplus x_{k-227}$. The number x_k is not the value reported by the MT, rather another invertible linear *tempering* function $n(\cdot)$, similarly designed to exploit computer architecture, is applied to x_k producing $n_k = n(x_k)$. This produces the next fresh uniform $U_k = (n_k + 0.5)/4{,}294{,}967{,}296$. In the calculation of U_k, of course, n_k is viewed as a number rather than a vector of zeros and ones.

A *Mersenne prime* is a prime M of the form $M = 2^p - 1$, where the exponent p is also prime (in fact, p must be prime for M to be prime). The period of the Mersenne Twister is the Mersenne prime $M^* = 2^{19937} - 1 \approx 4.3 \times 10^{6001}$, which is a truly super-astronomical number. The MT cycles through a large subset of the set

$$S = \{(n_1, n_2, \ldots, n_{624}) : 0 \leq n_i < 2^{32} \text{ for each } i\},$$

which has $\left(2^{32}\right)^{624} = 2^{19968}$ elements.) My computer generates the Mersenne Twister sequence at a blistering rate of about 63 million numbers per second. At that rate, over the estimated 13.7 billion year life (to date) of the universe, it would have generated a mere 2.7×10^{25} numbers — a miniscule fraction of the period!

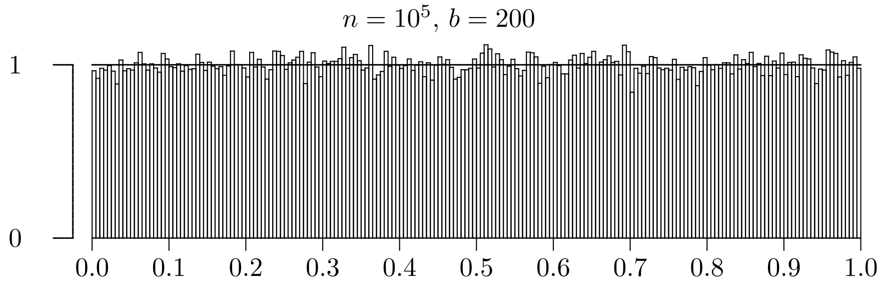
§3. A Few Statistical Tests

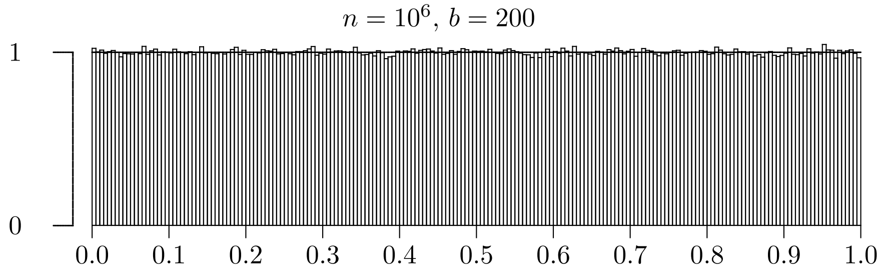
There are many statistical tests that can be applied to random number generators to measure uniformity and independence. We illustrate a few of them here.

Uniformity. The obvious first test is to determine, at least visually, whether the sequence $U_1, U_2, \ldots$ generated is distributed uniformly on the interval $(0, 1)$. To study this, we have partitioned the unit interval into $b = 200$ subintervals each of length 0.005. We then generate $U_1, U_2, \ldots, U_n$ for various (large) values of n and count how many of the U_i fall into each subinterval. If the U_i are uniformly distributed, we would expect roughly the same number (namely $\frac{n}{b}$) to land in each interval — naturally with some degree of randomness. Specifically, let interval $I_j = [(j-1)\delta, j\delta)$ for $1 \le j \le b$ and $\delta = \frac{1}{b}$ (which is 0.005 when $b = 200$). Let N_j denote the number of the U_i that fall into I_j (so $\sum_{j=1}^{b} N_j = n$, the number of simulations). In Figure 2, we plot *histograms* showing the value of N_j that corresponds to each interval (scaled by the factor b/n so that the anticipated value for each interval is 1). A horizontal line is plotted at $y = 1$ for comparison. This is shown for the Mersenne Twister for $n = 10^5$ and $n = 10^6$. The corresponding pictures for the two LCGs and the MWC look very similar.

Visually, at least, the pictures seem reasonable. The number of outcomes in each subinterval is approximately as expected and the randomness about that anticipated level decreases with the number of simulations n.

Figure 2. Histograms for the U_i produced by the Mersenne Twister.



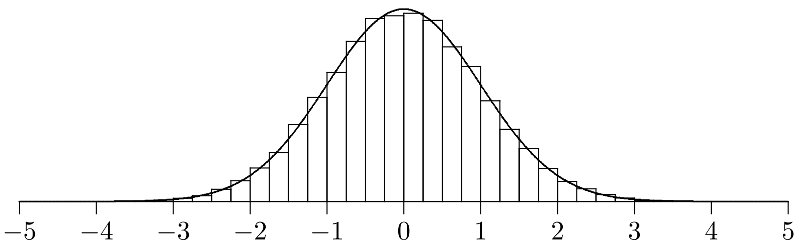


A More Refined View. Here we test whether the "bumpiness" in pictures like Figure 2 is as expected. As above, we divide the interval $(0, 1)$ into b subintervals of equal length and let N_j denote the number of the n simulations that lie in interval j. If the sequence $U_1, U_2, \ldots$ were truly iid Uniform $(0, 1)$, then each newly generated U_i would independently lie in a fixed subinterval I_j with probability $p = \frac{1}{b}$, so N_j would be distributed like Binomial (n, p). Subtracting off the mean and dividing by the standard deviation produces

$$X_j = \frac{N_j - np}{\sqrt{np(1-p)}}, \tag{6}$$

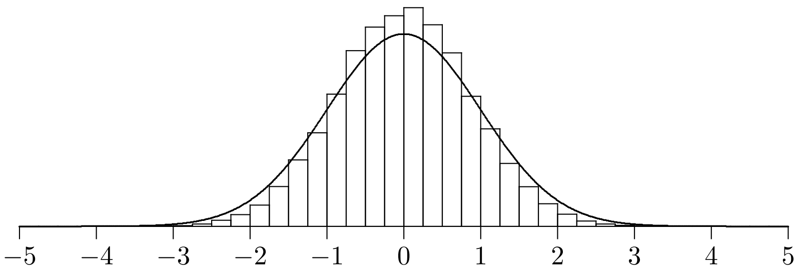
which, by construction, has mean 0 and variance 1. If n is large relative to b, the Central Limit Theorem ensures that each X_j would be approximately distributed like a standard normal. Furthermore, if the number of subintervals b is large, the values of $N_1, N_2, \ldots, N_b$ would be almost independent (one can show that $\text{Corr}(N_i, N_j) = -\frac{1}{b-1}$ for $i \neq j$). The values of $X_1, X_2, \ldots, X_b$ would therefore be roughly iid standard normals — at least for large b and much larger n. Figure 3 shows a histogram of $X_1, X_2, \ldots X_b$, computed for the Mersenne Twister, as compared to the density function of the standard normal. The histogram has been scaled by a multiplicative factor to make it comparable to the normal density function, which is also shown. Here, there are fifty thousand subintervals and one billion simulations. The results are precisely as they should be. The corresponding figures for the Knuth 64 bit LCG and the MWC algorithms look similar.

Figure 3. Histogram for the X_j produced by the Mersenne Twister.
$n = 10^9$, $b = 5 \times 10^4$



The comparable data for the 32 bit LCG is shown in Figure 4. For this LCG, the X_j and, hence, the N_j, are too tightly packed about their respective means. This phenomenon occurs because the number of simulations, 1 billion, is almost one quarter of the LCG's cycle (recall the period is 4,294,967,296). To take this to the extreme, if we had taken the number of simulations n = 4,294,967,296, the LCG would have returned each of its possible values precisely once and all the N_j would be the same (plus or minus one because 4,294,967,296 is not evenly divisible by 50,000) and the X_j would exhibit no variability!

Figure 4. Histogram for the X_j produced by the LCG.
$n = 10^9$, $b = 5 \times 10^4$



I should point out that the same phenomenon afflicts MWC and the Mersenne

Twister for large n. As a practical matter it cannot be observed, however, because the periods of these algorithms are so astronomically large. Many of our applications will involve in excess of 4 billion simulations. We therefore reject the 32 bit LCG as our random number generator.

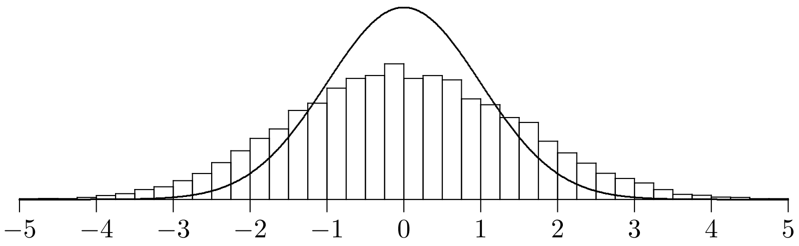
Permutation Test. In 1995 George Marsaglia introduced a battery of tests called the *Diehard* tests. Here we subject our RNGs to a version of Marsaglia's random permutation test. A vector $(U_1, \ldots, U_k)$ of random numbers can be used to generate a permutation of the numbers $\{1, 2, \ldots, k\}$ by ordering the k values of the U_i. For example: for $k = 4$, the vector

$$(U_1, U_2, U_3, U_4) \;=\; (.324, .167, .825, .634)$$

produces the permutation $(2, 1, 4, 3)$ since $U_2 < U_1 < U_4 < U_3$. To each vector $(U_1, \ldots, U_k)$, we assign an "order type" depending on the permutation it generates. The order type is a number j with $1 \le j \le k!$. If the numbers U_i are iid Uniform$\,(0,1)$, then all $k!$ order types are equally likely to result. Here we take $k = 8$, so there are $8! = 40{,}320$ different order types. Our version of the test works as follows. Generate a large number n of random vectors $(U_1, \ldots, U_8)$ and compute their order types j, where $1 \le j \le 40{,}320$. Let N_j denote the number of times order type j is observed among the n simulated vectors, so assuming uniformity and independence of the Us we would have that the N_js are (roughly) independent with each $N_j \sim$ Binomial$\,(n, p)$, where $p = \frac{1}{40{,}320}$. Let $\mu = np$ and $\sigma = \sqrt{np(1-p)}$. If n is very large, the numbers $X_j = (N_j - \mu)/\sigma$ would be approximately Normal$\,(0,1)$ by the Central Limit Theorem.

Figure 5 shows this for the MWC RNG with $n = 200$ million. Clearly the MWC RNG fails the random permutation test. The Knuth 64 bit LCG and the Mersenne Twister pass the test, producing histograms that look like Figure 3.

Figure 5. Histogram for the X_j produced by the MWC.
$n = 2 \times 10^8$, $k! = 8! = 40{,}320$



Conclusion. Both the Mersenne Twister and the Knuth 64 bit LCG do well under a panoply of statistical tests, three of which are illustrated here. As noted above, our 32 bit implementation of the 64 bit LCG is quite computationally inefficient, producing only 13 million Uniform$\,(0,1)$s per second. The Mersenne Twister is much more efficient, producing 63 million per second. We shall therefore use it as our random number generator.

Accompanying Code

The program `UniformTest.cpp` implements the uniformity test as described above for any of the four RNGs described in this chapter. It creates a histogram (as in Figure 2) to present the results. `BumpinessTest.cpp` implements the bumpiness test for any of the four RNGs, also creating a normal histogram (as in Figures 3 and 4) for viewing the results. `RandomPermTest.cpp` implements the random permutation test for any of the four RNGs, again creating a normal histogram (as in Figure 5) for viewing the results. All histograms may be viewed with the TeX software `Histogram.tex`.