LA-UR-23-25111

Approved for public release; distribution is unlimited.

Title:	Reassessing the MCNP Random Number Generator
Author(s):	Josey, Colin James
Intended for:	Report
Issued:	2023-09-01 (rev.1)

Los Alamos
NATIONAL LABORATORY

Reassessing the MCNP Random Number Generator

Colin Josey

Aug 24, 2023

1 Introduction

Random number generators are integral components to Monte Carlo codes. They provide the pseudorandom number sequence used to actually sample the distributions of interest. As a result, they are one of the most important components to the software.

The current recommended MCNP random number generator is a 63-bit linear congruential generator (LCG). This generator is quite fast, but it has some drawbacks. First, it only has a period of 2^{63}. Due to the necessarily non-optimal usage of random numbers to ensure parallel reproducibility, this amount is too few to guarantee random number sequences are not reused in all configurations the code runs under. As simulation size increases, users will need to be aware of the limitations of the generator and tune configuration variables to best suit their simulations, or they will need to assume that reuse is not negatively affecting their answers. Neither of these are optimal. Second, small LCGs are fairly weak in bit generation quality, and this can have an unknown impact on the quality of the simulation.

This paper is an investigation into whether or not more modern random number generators can supersede the current ones. The goal is to find a generator that is similar or superior in speed to the LCGs, has a state space large enough to make strong guarantees about random number reuse, and passes all modern random number test suites. If such a generator is found, it would eliminate the need for the user to even be aware of the limitations of the random number generator and would simplify the use of the code.

This paper will be broken into several parts. Sec. 2 will discuss the evolution of the random number generator within the MCNP code. Sec. 3 will go over what a Monte Carlo code needs from a generator to be reproducible and portable and how the current generator behaves in that light. Sec. 4 goes through how each generator was tested. Finally, Sec. 5 will discuss improvements that could be made to the code.

2 History

The generator within the MCNP code has gone through a long history of slow evolution. The initial random number generator in MCS [1], the earliest relative to MCNP, was a 35-bit multiplicative congruential generator (MCG - a subtype of Linear Congruential Generators with the add component set to zero), which took the form:

$$\text{seed} = (m \times \text{seed}) \mod 2^{35} \tag{1}$$

This 35-bit seed was then shifted right to generate a 27-bit integer and multiplied by 2^{-27} to create a value $[0, 1)$. The precise value of m is not documented. It was a user input value which, knowing what we know

now, was probably a bad decision. Given an ideal choice of m, this generator could have a period of 2^{33}. Rather interestingly, the counter for number of generated values was truncated to 15 bits, so at the time it may not have been expected for the method to generate more than $2^{16} - 1$ numbers.

In MCNP version 1a, the code uses a system-provided `frn` function to generate random values. In version 2a, this was replaced with the system-provided `ranf`. From [2], we know that `ranf` was a 48-bit multiplicative random number generator, identical to Eq. 1 with a modulo of 2^{48} and a resulting period of 2^{46}. `frn` appears to have been identical. When used with a multiplier of 5^{19}, this became what is now known as "generator 1" or the "default generator" in MCNP. In MCNP version 3 and 4C, the same generator was rewritten to be performed on machines with 32-bit integers, but otherwise this generator has persisted unchanged to this day.

In 2002, as 64-bit architectures became more common, 6 more generators were added [3]. The first 3 of these generators (generators 2-4) are LCGs with period of 2^{63}. The last 3 of these generators (generators 5-7) are MCGs with period of 2^{61}. While these generators are in almost all ways superior to generator 1, they are not the default.

3 Requirements of a Monte Carlo Random Number Generator

Let us now consider how the random number generators are used within the MCNP code in order to better understand its needs. For each particle history, the generator is started at the user-provided seed. Then, using the property that LCGs can skip through the random number sequence using an $\mathcal{O}(\log n)$ algorithm [4], it skips ahead a distance of (stride $\times$ i_h), where i_h is the index of the history. This approach provides a few useful properties:

1. The random number sequence is deterministic, both overall and for each history.
2. There is a single user input, the seed, which adjusts the sequence for all histories.
3. A history n uses no information from history $n - 1$.

Point 1 allows for easier verification and validation. It is often difficult to identify if anything has changed if the result is truly non-deterministic. Further, one can skip to a particular problematic history to observe it by telling the software to start on that history index. Point 2 is largely focused on user-friendliness in that an independent simulation can be started with a single change. Point 3 is critical to allow for a parallel simulation. If n relies on $n-1$, then $n-1$ must complete prior to n. The fact that the skip-ahead is $\mathcal{O}(\log n)$ also means each history can start relatively efficiently.

From this, one can surmise the following rules for an optimal Monte Carlo generator:

1. Each history must have a unique stream of random numbers.
2. This stream must solely be determined by information available at the start of the simulation and by properties of the history.
3. The generator must be fast, both in single evaluation, and in initial setup.
4. The generator must be portable. One should not rely on instructions restricted to some architectures.
5. The generator must support as many independent streams as possible.
6. A user must be able to change a single value and generate an independent simulation.
7. The random numbers must be as high quality as feasible.

3.1 How the MCNP Generators Fail

Now, the rules listed above are fairly strict. In fact, the MCNP generators do not fully meet all of the parameters. Rules 2, 3, and 4 are all met, but there are issues with the remainder. This section will talk about rules 1, 5, and 6, which all are related to one another. Rule 7 will be discussed later in Sec. 4.1, as it is best discussed relative to other generators.

Let us start with rule 1, that each history must have a unique stream of numbers. As mentioned prior, the MCNP generators achieve this by skipping a distance (stride $\times\, i_h$) through the random number sequence. By default, this stride is 152917. This value appears to have been selected because it was longer than the previous value of 4297, and possibly because it had a nice pattern in binary: `100101010101010101` [5]. Unfortunately, it is easy to overwhelm this value. For example, one of the test problems for the code, which are tuned to run relatively fast, requests 56 million random numbers in a single history. This is 368 times more than the stride and so the following 368 histories will reuse these values.

The argument used to ignore the impact of number reuse was that these random numbers were used in different portions of the code at any given time. Unfortunately, simplified test problems have demonstrated that there can be an effect due to too short a stride [6], and that this effect is amplified as the stride decreases. More recent research indicates that there can be spatial tilts in real reactor models when using very short strides [7]. The threshold for this effect to be significant is poorly researched. While it is not expected that the effect is statistically significant in regular problems run with the MCNP default configuration, it is currently impossible to detect issues a priori.

While we are not yet done with the issue of stride, it is necessary to pivot to rule 5: a generator must support as many independent streams as possible. The default generator can generate 460 million histories before the sequence begins to overlap using the default stride. The best generator can generate 60.3 trillion histories before the sequence begins to overlap. While the latter may seem sufficient, the author has witnessed simulations approaching a trillion histories at the time of writing this, so such a value is not insurmountable. If one needs more histories, one can decrease the stride. Similarly, if one needs a longer stride, it will result in a decrease in the number of histories prior to overlapping. This results in a tradeoff that can quickly result in no winners in large simulations.

What then happens if the period is exceeded? After n wraparounds, the next history will begin at $\mathrm{mod}\,(n \times \mathrm{period}, \mathrm{stride})$. Tracking where these histories begin, one finds that the minimum distance in the random number sequence between histories (an "effective" stride) follows that of Table 1. In the MCNP code's default configuration, the stride is effectively 1 after 631 billion histories, well within the reach of computers today.

Finally, let us consider rule 6, that a single user input must create an independent stream of results. Unfortunately, this even further compounds the issues just discussed. The user is requested to input a seed, but this seed is just the position in the random number sequence to start at. Poor choice of seed could easily result in the same simulation shifted over by one history, or a pair of simulations in which the stride between particles is small.

To summarize, the current generators make it easy to get into a situation in which sequences are reused. Too short a stride will increase the chances that histories are no longer independent and identically distributed. Too long a stride, and the maximum number of histories that can be simulated with said stride goes down. Generator 1, with its 2^{46} period is extremely susceptible to these issues. Generator 2, with its 2^{63} period, is somewhat more manageable.

3.2 How Large Should the Period Be?

Based on Sec. 3.1, it is clear that a period of 2^{46} is insufficient for modern simulations. So, how long should it be? If we consider contemporary computers operating at 5 GHz, generating one random number per core per

Table 1: Minimum Effective Strides

Overlaps	Effective Stride
0	152917
1	71443
2	10031
15	1226
122	223
625	111
1372	1

(a) Effective Stride for a 2^{46} Period Generator

Overlaps	Effective Stride
0	152917
1	1433
106	1019
107	414
320	191
747	32
4055	31
4802	1

(b) Effective Stride for a 2^{63} Period Generator

clock (optimistically), it would take 5.1×10^5 core-hours to exhaust 2^{63} random numbers. This can readily be achieved on the largest of supercomputers today, and is thus likely insufficient for the long-term future. On the other end, a period of 2^{128} would take 1.9×10^{25} core-hours to exhaust, which is clearly enough for all machines imagined to be built in the author's lifetime.

Unfortunately, we cannot efficiently use all these random numbers if we want to start making guarantees that numbers are not reused. Random numbers need to be chunked into 3 dimensions. The first is the number of generated values per history, the stride in previous discussion. The second is the number of histories per simulation. The third is the number of independent simulations run for the same problem. To simultaneously guarantee no reuse and to eliminate the user's need to be aware of the generator, all dimensions must exceed the needs of any configuration a user is interested in. The resulting generator must then support generating numbers amounting to the product of these dimensions, which is likely many orders of magnitude larger than the actual quantity of numbers that will ever be used.

Let us consider three cases. The first is based on the simulations observed today and would form the minimum requirements. For the stride, test problems have shown the need for at least 2^{26} random numbers in a single history. For problem size, 2^{40}-history simulations have been run. Finally, independent simulations are often used to compute the true variance for problems with uncertain input. One often does not need more than a few hundred simulations to have confidence in the results, so 2^{10} is probably safe here. This results in a minimum generator of size $2^{26+40+10} = 2^{76}$.

In the next case, let us investigate the minimum required for a generator to be sufficient for decades. For problem size, the code accepts as a maximum $2^{63} - 1$ for the number of histories, which we can round to 2^{64}. Any increase beyond this would require major architectural changes to the code. For the number of independent simulations, 2^{16} would likely give the shape of the distribution of any parameter of interest in high detail. A good value for the stride is the least clear of the three, but a sequence of length 2^{48} would increase the space a bit over 6 orders of magnitude and brings the total to a nice round $2^{48+64+16} = 2^{128}$.

Finally, in the last case would be the maximum conceivable simulation size. A good example of this would be the direct numerical simulation of every neutron that exists during the lifetime of a nuclear reactor. A reactor running at 5 GWth for 50 years with 200 MeV per fission would have $2^{97.6}$ fissions over its lifespan. For the number of independent simulations, the current MCNP user interface allows up to $2^{63} - 1$. As for the stride, 2^{64} would be safe as such a huge single history is intractable on expected hardware. This results in $2^{64+97.6+63} = 2^{224.6}$.

From this we can see that both the minimum and long-term examples could use a generator with 2^{128} space, whereas a generator of size 2^{256} would never need to be replaced for state space reasons[1]. As such, this investigation will focus on generators of this order of magnitude.

4 Testing Methodology

There are two tests that must be performed for each alternate generator, both of equal importance. The first is to test the generator quality. The methods used in this work are discussed in Sec. 4.1. Second, performance testing is discussed in Sec. 4.2.

4.1 Generator Quality

One of the biggest issues with random number generators is that it is impossible to determine if a generator is good. Available tests can only indicate if a generator is flawed in some way. As a result, it is imperative to test a random number generator with as many tests as possible to minimize avenues of concern. The two testing suites that will be used are TestU01 [8] version 1.2.3 and PractRand [9] version 0.95-pre.

For TestU01, the "BigCrush" test suite was used, which involves 106 tests. As this test needs 32-bit inputs, the upper and lower 32 bits of the 64-bit generators were passed, along with the bit pattern reversed for both, for a total of four BigCrush runs. The 63-bit LCG was only tested for the upper 32 bits. As was done in the PCG paper [10], any p-value worse than 1 in 1000 was rerun five times with different seeds. If it occurred more than once it was considered systematically failed.

PractRand is a convenient series of tests in that it will generate n bytes, run tests, then double n until the tests fail or it hits the test cutoff which defaults to 32 TiB. The number of bytes at which a generator fails can be used to roughly estimate relative generator quality. As 32 TiB takes several days with most generators, the tool was not run past that. PractRand takes as input 32- or 64-bit values, so the 63-bit LCG was truncated to the upper 32 bits.

4.2 Performance

In order to quantify performance, we will need a test problem. The problem mentioned in Sec. 3.1 with the longest known stride would be a good candidate, as it spends 8.8% of simulation time in the random number generator. Unfortunately, the change in the random number sequence is liable to significantly affect the longevity of particle histories in this problem due to the large number of produced secondary particles. As such, a Godiva sphere [11] was run within the Intel VTune tool. The time spent generating random numbers was recorded, as well as the number of random numbers generated. This approach is somewhat noisy, but it does include low-level hardware behavior.

All simulations were built with the Intel LLVM 2023.2 C++ and Intel Classic 2023.2 Fortran compilers with the MCNP code's default compiler options. One modification was made in that the target architecture was set to support SSE4.2 and prior instruction sets. This was necessary for some of the vectorizable generators and was used everywhere for consistency. It is not believed that this falls afoul of the rule 4 involving specific architecture instructions, as vector arithmetic of some form has been supported in almost all performant hardware for well over a decade. The hardware under test was an Intel Xeon Gold 5218R, and the operating system was Linux.

[1]The author is aware that 150 years from now this statement has a non-zero chance of being mocked for lack of foresight.

The random number generator as implemented within the MCNP code has three issues that affect performance in ways that prevent direct comparison with other work. First, the generator is held on the C++ side and not on the Fortran side predominantly due to not having unsigned integers on the Fortran side. This results in a cross-language function call in most situations, which can harm optimization. Second, the generator is encapsulated in a C++ 17 variant to provide support for the current 7 generators with low overhead[2]. Third, each generator tracks the number of values generated. This gives a slight leg up for counter-based generators, as this becomes effectively free. None of these are strict requirements like those in Sec. 3. For example, the variant may be removed once the LCGs are formally removed from the code. As the removal of the variant is the most likely to occur first, performance numbers in the following sections are shown with and without runtime polymorphism.

Finally, not discussed in much detail is the conversion of the bitstream into 64-bit floating point values. For fair comparison, all algorithms used the method currently used in the MCNP code. However, in Sec. A, an alternative method with better statistical and performance characteristics is shown, which will be used in the final implementation.

5 Alternative Generators

Here, the focus turns to other random number generators. Some of these generators do not handle parallelism in the same fashion as the LCG currently in the code, so some discussion is made in Sec. 5.1. Finally, the list of tested generators is in Sec. 5.2.

5.1 Modes of Parallelism

Support for parallelism is the hardest of the generator requirements to meet. The generators that will be described below all fall into one of three ways of handling parallelism: skip-ahead, incrementing a counter, or brute force.

Skip-Ahead

This is the approach currently used within the MCNP code. At the start of each history, the generator is reset to the initial seed and moved forward (stride $\times i_h$) steps. The algorithm currently used is $\mathcal{O}(\log n)$ [4]. Approaches that are $\mathcal{O}(n)$ are unacceptable, as n may be in the trillions. As will be shown in Sec. 5.2, even $\mathcal{O}(\log n)$ has negative performance implications.

Counter Generators

These generators have, within their construction, a counter. The counter is set to zero and incremented by one at each random number call. Most of these generators as a result generate entirely unique, independent sequences for each initial seed. If the seed is a combination of the user-defined seed and a history identifier, this suits all our parallel needs in an $\mathcal{O}(1)$ way.

Some special generators have states that are entirely constant except for the counter. These generators are ideal for generating very long strings of random numbers in loops or for use with event-based Monte Carlo. Within the MCNP code, numbers are often only used one at a time which does not benefit significantly from vectorized generation.

[2]The variant allows the multipliers and adders to be constants and eliminates a branch in the conversion to real where the 48-bit generator needs a left shift and the rest needs a right shift, at the cost of moving the branch elsewhere.

Brute Force

Some algorithms do not particularly support generating independent random sequences. Parallelism is only supported by generating a new seed for each history and hoping that any two sequences do not touch. Given a period of p, a required stride of s, a number of histories n, and a uniform probability to start a random sequence in the entirety of the period, the probability of sequence reuse is [12]:

$$P_{\text{reuse}} = 1 - \frac{(p - ns - 1)!}{p^{n-1}\,(p - n\,(s+1))!}$$

For $p = 2^{63}$ and $s = 152917$, which is the MCNP generator 2 configuration, $P_{\text{reuse}} > 0.5$ at $n = 6.47 \times 10^6$. Here, n is much lower than the 7.0×10^{13} that perfect spacing would permit. Generators that rely on this approach must thus have a much larger state space than the other two to still meet the requirements discussed in Sec. 3.2. Results for n in the following calculations will be based on $P_{\text{reuse}} > 10^{-10}$.

5.2 The Generator List

This list of generators is not exhaustive in the slightest. Many were excluded out-of-hand for one reason or another. For example, using AES-128 as an encryption function in a form similar to SPECK-128/128 below was excluded as efficient implementations require hardware instruction support, which is not guaranteed. Other generators were excluded for having too small a state (below 2^{128}), or were optimized with different goals in mind. Lastly, the author may simply not have been aware of any given method.

The listed storage requirement is the minimum for the generator to function, and does not include auxiliary functionality that the MCNP code needs. Timing values given are from the best-optimized implementations (limited to SSE instructions) available, and someone more skilled could probably reduce these numbers.

5.2.1 The Current 63-bit LCG

Reference	[3]
Period	2^{63}
Streams	As used, 1.
Bits Output	63
Parallelism	$\mathcal{O}(\log n)$ skip-ahead
Storage Required	8 bytes
TestU01 Result	Failed, `SerialOver` (test 1), `CollisionOver` (tests 7, 9, 11), `BirthdaySpacings` (tests 13, 15, 17, 18, 19, 21)
PractRand Result	Failed at 32 MiB
Time Per Double	3.0 ns (polymorphic), 2.5 ns (non-polymorphic)
Time Initializing History	115 ns

During the non-polymorphic case, the code spent more time skipping ahead than it did generating random numbers. This problem in particular has few random numbers used per history exacerbating the issue, but it is still a disadvantage in comparison to generators that have cheaper initialization functions.

It is possible to generate multiple streams by using a different adder than 1, but the bits will be a constant offset from either the $m \times \text{seed} + 1$ or $m \times \text{seed} + 3$ sequence [13]. This is too correlated to be useful. Generating a series of good multipliers would be impractical due to the sheer number required.

5.2.2 Mersenne Twister

Reference	[14] - the implementation used is `mt19937_64` in C++.
Period	$2^{19937} - 1$
Streams	1
Bits Output	64
Parallelism	There exists a skip-ahead [15], but execution time is measured in milliseconds, so brute force was used for testing instead. For a stride of 2^{64}, probability of reuse exceeds 10^{-10} after roughly $2^{9919.9}$ histories if randomly initialized. Default initializer provides only 2^{64} independent streams.
Storage Required	2500 bytes
TestU01 Result	Failed, `LinearComp` (tests 80, 81)
PractRand Result	Failed at 512 GiB
Time Per Double	3.8 ns (polymorphic), 3.4 ns (non-polymorphic)
Time Initializing History	643 ns

While this generator is quite popular, it has issues with output quality and performance. In all cases, the simulation spent more time initializing the generator than generating random numbers due to the costly seeding mechanism and the simplicity of the test problem. The Intel library implementation used here benefits significantly from allowing SSE vector instructions, running nearly four times faster than the scalar form (which took around 12 ns each).

5.2.3 Xoshiro256**

Reference	[16]
Period	$2^{256} - 1$
Streams	1
Bits Output	64
Parallelism	Has skip-ahead of 2^{128} or 2^{192}. As this is $\mathcal{O}(n)$, would need to use brute force. For a stride of 2^{64}, probability of reuse exceeds 10^{-10} after roughly $2^{79.4}$ histories if randomly initialized. $\mathcal{O}(\log n)$ should be possible [17] by deriving $\mathcal{O}(1)$ skip-aheads of 2^n for all n, but such an implementation is expected to be slow.
Storage Required	32 bytes

TestU01 Result	Passed
PractRand Result	Passed to at least 32 TiB
Time Per Double	3.6 ns (polymorphic), 3.4 ns (non-polymorphic)
Time Initializing History	11.7 ns (using default SplitMix generator)

This generator has a minor issue with recovering from all-zero state [18], but the probability of this affecting the calculation is exceedingly low. Otherwise, its statistical behavior is good. The seed-space of the auxiliary generator used to initialize the state determines how many sequences this method supports.

5.2.4 PCG DXSM 128/64

Reference	Original at [10] with improvements at [19]
Period	2^{128}
Streams	As used, 1.
Bits Output	64
Parallelism	$\mathcal{O}(\log n)$ skip-ahead
Storage Required	16 bytes
TestU01 Result	Passed
PractRand Result	Passed to at least 32 TiB
Time Per Double	5.4 ns (polymorphic), 4.7 ns (non-polymorphic)
Time Initializing History	165 ns

This is a descendent of LCGs with a permutation function on output that improves results. This particular one has a 128-bit state and 64-bit output. The DXSM form fixes an issue found after PCG XSL RR 128/64 generator was added to NumPy [20] as its default generator. The history initialization overhead was quite large for this generator.

Like the LCG this is based off of, modifying the adder generates a related stream. On one hand, the permutation function prevents the bit streams from being as simply related to one another as an offset. On the other, the permutation function is not designed to be a maximal avalanche, so traces of correlation can still be found if examined closely. As such, PCG's streams are far superior to the LCG form but perhaps weaker than streams in many of the other methods discussed.

5.2.5 SPECK-128/128

Reference	[21]
Period	2^{128}
Streams	2^{128}
Bits Output	128
Parallelism	Incrementing counter, can $\mathcal{O}(1)$ skip-ahead within a sequence

Storage Required	Minimal: 16 bytes for the particle ID and counter SSSE3 Vectorized: 16 bytes for the counter, 16 bytes to store round keys per round, 128 bytes to store cyphertexts for a total of 656 bytes (32-rounds).
TestU01 Result	Passed
PractRand Result	Passed to at least 32 TiB
Time Per Double	32-round: 9.9 ns (polymorphic), 9.7 ns (non-polymorphic) 12-round: 6.1 ns (polymorphic), 5.8 ns (non-polymorphic)
Time Initializing History	32-round: 43.3 ns 12-round: 17.0 ns

This algorithm is applying the SPECK-128/128 encryption algorithm on an incrementing 128-bit counter. SPECK was designed for efficiency on small devices. As such, it is generally performant with many space-efficiency tradeoff options available. In the implementation used, 1024 bits were encrypted at a time with precomputed round keys.

If one has an event-based Monte Carlo method, in which many histories are in flight at once but not all being actively simulated, it would be worth considering not precomputing round keys and composing the key on-the-fly. Then, one only needs to store two integers, the history ID and the counter, for each history. For this algorithm, not computing round keys negatively affects performance.

With many cryptographic generators, it is possible to exchange quality for performance. SPECK is a 32-round generator in that its inner loop applies its base operation 32 times before returning. Reducing the round count reduces runtime. However, at some point, the generator ceases to pass tests. The smallest number of rounds that still passed all tests was a 12-round generator. Performance improvements are not linear due to the overhead of storing and re-loading multiple cyphertexts.

The history initialization involves setting the first 64 bits of the encryption key to the user-defined seed and the last 64 bits to the history ID. Then, the round keys are generated and stored.

5.2.6 ChaCha

Reference	[22] - using the SSSE3 implementation by Orson Peters and Melissa E. O'Neill
Period	2^{128}
Streams	2^{256}
Bits Output	512
Parallelism	Incrementing counter, can $\mathcal{O}(1)$ skip-ahead within a sequence
Storage Required	Minimal: 16 bytes for the particle ID and counter SSSE3 Vectorized: 32 bytes for the key, 16 bytes for the counter, 64 bytes to store cyphertexts for a total of 112 bytes.
TestU01 Result	Passed
PractRand Result	Passed to at least 32 TiB
Time Per Double	20 round: 31.1 ns (polymorphic), 30.3 ns (non-polymorphic) 6 round: 14.3 ns (polymorphic), 14.7 ns (non-polymorphic) 4 round: 12.0 ns (polymorphic), 11.9 ns (non-polymorphic)

Time Initializing History ~ 1 ns

This algorithm is applying the ChaCha stream cypher on an incrementing 128-bit counter. The full algorithm, known as ChaCha20, has 20 iteration rounds. The fewest number of rounds that still passes PractRand to 32 TiB is 4.

Initialization of a history is simply setting a few values to zero and setting a new key. Performance was such that there was a high degree of stochastic uncertainty in how long the evaluation took. For all purposes, this time is negligible.

Unlike SPECK above, ChaCha does not have round keys that need to be computed for maximum performance. As a result, there is essentially no performance loss to only storing the particle ID and counter so long as 8 random numbers are used at a time. However, it does perform worse in comparison to SPECK, as implemented in this testing.

5.2.7 SFC64

Reference	[9]
Period	2^{64} minimum, 2^{255} expected in each stream [23]
Streams	2^{192}
Bits Output	64
Parallelism	Incrementing counter, cannot $\mathcal{O}(1)$ skip-ahead within a sequence
Storage Required	32 bytes
TestU01 Result	Passed
PractRand Result	Passed to at least 32 TiB. Algorithm author has noted that the theoretically weaker SFC16 failed at 512 TiB.
Time Per Double	3.1 ns (polymorphic), 2.6 ns (non-polymorphic)
Time Initializing History	23 ns

This generator has some interesting properties. A portion of the internal state of the generator is an incrementing counter, which allows each seed to generate an independent sequence of a length of at least 2^{64}. Unlike cryptographic generators, however, the rest of the internal state is mutated and fed forward, which reduces the computational cost required to generate high quality bits. This results in excellent performance.

History initialization is performed by setting bits 1-64 to the user-defined seed, bits 129-192 to the history index, and the rest to zero. Then, the generator is called 18 times to mix the state. The author recommends either 12 or 18 iterations when used in this fashion, so 18 was chosen out of caution.

PractRand was also run on this history initialization approach. An augmented generator was implemented in which the history index was incremented, the history initialization was called, and then the first generated value was returned to PractRand. This pure counter generator also passed PractRand to 32 TiB, indicating that this approach to handling multiple histories was valid.

There is a structurally related generator, Romu [24], whose paper provides a great deal of the theory on generators of this form. The RomuTrio generator is faster than SFC64 with similar bit quality, but it has a drawback. The generator does not have a counter in its structure. This means the minimum period is much smaller than SFC64, although the probability of being in a short loop is exceedingly small. Additionally, the MCNP code needs to track random number usage and adding a counter alongside to do so interferes with Romu's careful optimization. For this reason, only SFC64 was tested.

5.3 Generator Summary

Of the generators considered, it is easy to eliminate a few of the generators. Mersenne Twister has poor quality number generation and an exceptionally inefficient seeding method. The large state is also a concern for larger simulations, in which the chance the generator stays in cache decreases.

The cryptographic generators, SPECK-128/128 and ChaCha are also easy to eliminate for several reasons. First is performance. In full form, ChaCha20 is over 10 times slower than the current LCG, and the fastest SPECK version is still half as fast. Second is portability. While almost all architectures have vector instructions, this performance was only achieved with hand-tuned vector intrinsics which would need tuning for each target architecture. The reliability of compiler auto-vectorization was not investigated thoroughly. Third, there are concerns about de-rating an algorithm until it just passes tests. However, for codes that target only a few architectures, need the $\mathcal{O}(1)$ traversal through the stream, and need a small state, ChaCha4 with careful optimization may be the best choice there.

That then leaves Xoshiro256**, PCG DXSM 128/64, and SFC64. All three would make an excellent generator for a Monte Carlo transport code. These three methods have excellent bit quality, perform very closely to the current 63-bit LCG, and would expand the number of available states tremendously. In the interest of having only one generator to minimize the complexity for the user and for performance reasons, slight differences must be used to decide. PCG is the slowest of the three and only has 2^{128} states without using its stream functionality. It is also somewhat complicated to implement in an efficient and portable way due to needing 128-bit integer arithmetic. Xoshiro256** does not have an already-derived mechanism to generate uniformly spaced sequences and must rely on an auxiliary generator. That then leaves SFC64, which was the fastest new generator tested. It also has the ability to generate 2^{192} independent streams, which far exceeds any imagined use case.

While it is expected that the generator tests alone are sufficient to indicate that SFC64 is as good or better than the current default generator in all circumstances, the MCNP expanded criticality test suite was run with both generators and compared. The difference in k-eigenvalue was divided by the combined estimated standard deviations between each pair of simulations. 58.0% of simulations were within 1 standard deviation of each other, 33.6% were between 1 and 2 standard deviations, and 7.5% were between 2 and 3. The largest difference was 3.07 standard deviations. Due to the underprediction of the variance in k-eigenvalue calculations due to inter-cycle autocorrelation, this result is as expected and does not indicate any issues.

6 Summary

The overall recommendation is to add SFC64 to the code as a "generator 8" and make it the default. In the next MCNP version, the LCGs will still be supported to enable users to perform comparisons. If there are no indications of issues, it can be made non-polymorphic for the performance improvement. The advantages and general usability of this generator will hopefully eliminate any concerns about MCNP's random number generation going forward. For SFC64 specifically, the modified integer to real conversion described in Sec. A will be used, but this will not be ported to the older generators for reproducibility reasons.

References

[1] R. R. Johnston, "A General Monte Carlo Neutronics Code," Los Alamos Scientific Laboratory, Los Alamos, NM, USA, Tech. Rep. LAMS-2856, Mar. 1963. 1

[2] K. Entacher, "On the CRAY-System Random Number Generator," *SIMULATION*, vol. 72, no. 3, pp. 163–169, 1999. DOI: 10.1177/003754979907200308 2

[3] F. B. Brown and Y. Nagaya, "The MCNP5 Random Number Generator," in *Transactions of the American Nuclear Society*, vol. 87, 2002, pp. 230–232. 2, 7

[4] F. B. Brown, "Random Number Generation with Arbitrary Strides," in *Transactions of the American Nuclear Society*, vol. 71, 1994, p. 202. 2, 6

[5] J. S. Hendricks, "Random Number Stride in Monte Carlo Calculations," Los Alamos National Laboratory, Los Alamos, NM, USA, Tech. Rep. LA-UR-90-1845, Nov. 1990. 3

[6] T. E. Booth, "Bad Estimates as a Function of Exceeding the MCNP Random Number Stride," Los Alamos National Laboratory, Los Alamos, NM, USA, Tech. Rep. LA-UR-14-23159, May 2014. DOI: 10.2172/1130484 3

[7] A. R. Hakim and D. A. Fynan, "Challenges of Near Critical-Fixed Source Monte Carlo Simulations of CANDU-6 Reactor: Bundle Power Tally Bias and Error Autocorrelation from Exceeding Random-Number Stride," in *The International Conference on Mathematics and Computational Methods Applied to Nuclear Science and Engineering (M&C 2023)*, Niagara Falls, ON, CA; Aug 13–17, 2023. 3

[8] P. L'Écuyer and R. Simard, "TestU01: A C Library for Empirical Testing of Random Number Generators," *ACM Trans. Math. Softw.*, vol. 33, no. 4, aug 2007. DOI: 10.1145/1268776.1268777 5

[9] C. Doty-Humphrey, "PractRand," 2019. URL: https://pracrand.sourceforge.net/ 5, 11

[10] M. E. O'Neill, "PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation," Harvey Mudd College, Claremont, CA, Tech. Rep. HMC-CS-2014-0905, Sep. 2014. 5, 9

[11] "International Criticality Safety Benchmark Evaluation Project Handbook 2015," OECD Nuclear Energy Agency, Paris, France, Tech. Rep. ICSBEP-2015, 2015. DOI: 10.1787/936dd0d6-en 5

[12] M. Abramson and W. Moser, "More Birthday Surprises," *The American Mathematical Monthly*, vol. 77, no. 8, pp. 856–858, 1970. DOI: 10.1080/00029890.1970.11992600 7

[13] M. E. O'Neill, "Critiquing PCG's Streams (and SplitMix's Too)," 2017. URL: https://www.pcg-random.org/posts/critiquing-pcg-streams.html 8

[14] T. Nishimura, "Tables of 64-bit Mersenne Twisters," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 10, no. 4, pp. 348–357, 2000. DOI: 10.1145/369534.369540 8

[15] H. Haramoto, M. Matsumoto, T. Nishimura, F. Panneton, and P. L'Écuyer, "Efficient Jump Ahead for $\mathbb{F}_2$-Linear Random Number Generators," *Les Cahiers du GERAD ISSN*, vol. 711, p. 2440, 2006. 8

[16] D. Blackman and S. Vigna, "Scrambled Linear Pseudorandom Number Generators," *ACM Transactions on Mathematical Software (TOMS)*, vol. 47, no. 4, pp. 1–32, 2021. DOI: 10.1145/3460772 8

[17] S. Vigna, "Further Scramblings of Marsaglia's Xorshift Generators," *Journal of Computational and Applied Mathematics*, vol. 315, pp. 175–181, 2017. DOI: 10.1016/j.cam.2016.11.006 8

[18] M. E. O'Neill, "Exploring Xoshiro's Close-Repeats Flaw," 2018. URL: https://www.pcg-random.org/posts/xoshiro-repeat-flaws.html 9

[19] M. E. O'Neill, "PCG – C++ Implementation." URL: https://github.com/imneme/pcg-cpp 9

[20] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array Programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. DOI: 10.1038/s41586-020-2649-2 9

[21] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers, "The SIMON and SPECK Families of Lightweight Block Ciphers," *Cryptology ePrint Archive*, 2013. 9

[22] D. J. Bernstein *et al.*, "ChaCha, a Variant of Salsa20," in *Workshop record of SASC 2008: The State of the Art of Stream Ciphers*, vol. 8, no. 1. Citeseer, 2008, pp. 3–5. 10

[23] M. E. O'Neill, "Random Invertible Mapping Statistics," 2018. URL: https://www.pcg-random.org/posts/random-invertible-mapping-statistics.html 11

[24] M. A. Overton, "Romu: Fast Nonlinear Pseudo-Random Number Generators Providing High Quality," 2020. DOI: 10.48550/arXiv.2002.11331 11

[25] F. Goualard, "Generating Random Floating-Point Numbers by Dividing Integers: A Case Study," in *Computational Science – ICCS 2020*, V. V. Krzhizhanovskaya, G. Závodszky, M. H. Lees, J. J. Dongarra, P. M. A. Sloot, S. Brissos, and J. Teixeira, Eds. Cham: Springer International Publishing, 2020, pp. 15–28. 14

[26] "IEEE Standard for Floating-Point Arithmetic," Institute of Electrical and Electronics Engineers (IEEE), Tech. Rep. IEEE Std 754-2019 (Revision of IEEE 754-2008), 2019. DOI: 10.1109/IEEESTD.2019.8766229 15

[27] T. R. Campbell, "Uniform Random Floats: How to Generate a Double-Precision Floating-Point Number in $[0, 1]$ Uniformly at Random Given a Uniform Random Source of Bits." 2014. URL: https://prng.di.unimi.it/random_real.c 15

A Conversion to Real

There are many ways to convert a bitstream into floating point numbers [25]. This appendix will compare and contrast a few of them.

Within the MCNP code, the random number output is interpreted as if it was a sample from the continuous uniform distribution with support $(0, 1)$. This distribution deliberately excludes zero and one to simplify coding, as asymptotic behavior at both ends neither needs to be implemented or tested if the generator cannot emit the value.

The method currently used within the code is to take a single integer output of the bit generator, shift the integer such that it spans $\left[0, 2^{53} - 1\right]$ and then multiply it by 2^{-53}. Then, to prevent generation of zero, the maximum of 2^{-53} and the generated value is returned. The end result is a floating point value $(0, 1)$ with 2^{-53} twice as likely as all other values. Floating point values that are not integer multiples of 2^{-53} cannot be generated. If one computes the mean and the variance of this discrete approximation to the uniform distribution, one finds that there is a slight error in both. The values are shown in Table 2.

Table 2: Statistical Properties of the Discrete Float Distributions

Method	Domain of n	Mean Error	Variance Error	Minimum	Maximum
$n \times 2^{-53}$	$\left[0, 2^{53} - 1\right]$	5.6×10^{-17}	1.0×10^{-33}	0	$1 - 2^{-53}$
$\max\left(2^{-53}, n \times 2^{-53}\right)$	$\left[0, 2^{53} - 1\right]$	5.6×10^{-17}	1.3×10^{-32}	2^{-53}	$1 - 2^{-53}$
$n \times 2^{-52} + 2^{-53}$	$\left[0, 2^{52} - 1\right]$	0	4.1×10^{-33}	2^{-53}	$1 - 2^{-53}$

At the cost of one bit of output, it is possible to improve on this method by sampling as follows:

$$\xi = n \times 2^{-52} + 2^{-53}, n \in \left[0, 2^{52} - 1\right]$$

This form eliminates the error in the mean and reduces the variance error by a factor of four relative to the current method. This also has the benefit that no branching is required to prevent generating zero. This approach can be implemented in one of two ways. The first assumes only that the generator made 64 bits:

```
double sampleReal(uint64_t bitstream) {
  return static_cast<double>(bitstream >> 12) * 0x1.0p-52 + 0x1.0p-53;
}
```

The second further requires that double is 64 bits in IEEE-754 [26] format:

```
double sampleReal(uint64_t bitstream) {
  uint64_t float_bits = (bitstream >> 12) + 0x3FF0000000000000;
  auto float_val = std::bit_cast<double>(float_bits);
  return float_val - 0x1.FFFFFFFFFFFFFp-1;
}
```

It depends on compiler optimization levels, architecture, and the generator itself on which of the two is more efficient, but both generate identical results for a given input bitstream. In testing, the latter form sped up SFC64 by 0.3 ns per evaluation to 2.3 ns.

A few other methods were investigated, but were found to be problematic for one reason or another. Multiplying the entire bitstream by 2^{-64} (or the far slower equivalent, `std::ldexp(bitstream, -64)`) allows up to 12 more bits to be used for small values. This has two drawbacks. First it allows the generation of both zero and one. Second, the rounding that occurs during the conversion of the bitstream to a double very slightly distorts the uniformity of the distribution. It is possible to fix the rounding issue, either by changing the floating point environment to round upward or by setting the last bit of the bitstream to 1, but a branch would still have to be added to prevent generating one to meet all the requirements.

A method by Taylor Campbell [27] instead starts with an exponent of -64, generates bits, counts the leading zeros, adjusts the exponent, and generates more bits to fill the zeros until there is 64 significant bits. Then, it multiplies the resulting bitstream by 2^{-exp}. As the last 11 bits are always discarded in this operation, it can set the last bit to 1 to ensure correct rounding without loss of precision or adjusting the floating point environment. This method is capable of generating all possible floating point values in $[0, 1]$ inclusive to correct probability. However, the branching and extra bit generation greatly slows down the method, slowing SFC64 down to 9.8 ns each.

Of the four methods discussed, the 52-bit algorithm will be used for future MCNP generators. While it has the lowest precision of all methods, it is fastest by far, guarantees zero and one are not generated, and generates a high quality approximation to the uniform distribution. While other methods yield more bits, it is unclear of any circumstance where simulations that are often accurate to only 10^{-5} would be sensitive to values on the order of 10^{-16} without also needing to go to higher precision than double. For other types of work particularly sensitive to values near zero, Campbell's method would be the only one recommended.