Journal of Modern Applied Statistical Methods

Volume 14 | Issue 1 Article 23

5-1-2015

# Pseudo-Random Number Generators for Vector Processors and Multicore Processors

Agner Fog
*Technical University of Denmark*, agfo@dtu.dk

Journal of Modern Applied Statistical Methods
May 2015, Vol. 14, No. 1, 308-334.

ISSN 1538 − 9472

# *JMASM Algorithms and Code*
# Pseudo-Random Number Generators for Vector Processors and Multicore Processors

**Agner Fog**
Technical University of Denmark
Ballerup, Denmark

Large scale Monte Carlo applications need a good pseudo-random number generator capable of utilizing both the vector processing capabilities and multiprocessing capabilities of modern computers in order to get the maximum performance. The requirements for such a generator are discussed. New ways of avoiding overlapping subsequences by combining two generators are proposed. Some fundamental philosophical problems in proving independence of random streams are discussed. Remedies for hitherto ignored quantization errors are offered. An open source C++ implementation is provided for a generator that meets these needs.

*Keywords:* Random number generation, SIMD, vector processors, multiprocessors, parallel generation, combination of generators, quantization errors, theoretical proofs, philosophy of science

## Introduction

The exponential increase in the computing power of mainstream microprocessors over several decades, known as Moore's Law, has made large scale Monte Carlo applications feasible and common. The current trend in microprocessor technology goes towards parallel processing of data in mainly two ways: 1) microprocessors have vector registers that can do arithmetic operations on a whole vector with a single CPU instruction (Single Instruction Multiple Data, SIMD), and 2) microprocessor chips have multiple CPU cores that can execute multiple threads simultaneously. The design of pseudo-random number generators (PRNGs) has been improved considerably in recent decades, but few of the published designs are suitable for utilizing the parallel processing capabilities of

*Dr. Fog is a Researcher at the Technical University of Denmark. Email him at agfo@dtu.dk.*

today's microprocessors in large scale computations (Manssen, et al., 2012; Passerat-Palmbach, Mazel and Hill, 2011). The construction of pseudo-random number generator software capable of utilizing both vector processing and multi-threading for the fast generation of large amounts of pseudo-random numbers of high quality, using the newest microprocessor technology are considered.

## Choice of hardware

Several hardware platforms are available for parallel processing:

### Mainstream CPUs for the PC market

These CPUs are quite powerful. They are universally available and cheap because of high production volumes. The size of vector registers in the common x86 family of microprocessors has grown exponentially in recent years, as illustrated in Table 1.

**Table 1.** Vector register size of x86 family microprocessors.

| Year introduced | Instruction set for integer vector operations | Vector size, bits |
|---|---|---|
| 1997 | MMX | 64 |
| 2001 | SSE2 | 128 |
| 2013 | AVX2 | 256 |
| expected 2017 | AVX-512 | 512 |

Vector sizes of 1024 bits and perhaps 2048 bits can be expected in mainstream CPUs in the coming years. However, the vector size will probably not keep growing exponentially because of diminishing returns and because the size of mask registers used for conditional execution is limited to 64 bits, corresponding to 64 elements of 32 bits each = 2048 bits, in current specifications from Intel (Intel, 2014a).

The high-end CPUs are currently available with 8 or more cores and a clock frequency of 3 – 4 GHz. Some models are capable of running two threads in each core, but this may not be useful for CPU-intensive code because both threads are competing for the same hardware resources (Fog, 2014a).

### Graphic processors.

Graphics Processing Units (GPUs) are included in many PCs and designed mainly for the purpose of computer games. Contemporary GPUs are available in many different configurations with hundreds or thousands of parallel streams and clock frequencies ranging from 200 to 1600 MHz. GPUs have increasingly been applied to general computation tasks that involve large amounts of parallel data. Software libraries for random number generation in GPUs are available (Manssen, et al., 2012; Demchik, 2011; Barash and Shchur, 2014; Nandapalan, et al., 2012).

A serious limitation of GPUs is that each stream has access to only a small amount of RAM memory, and communication between streams is expensive. We have to consider that random number generation is typically only a small part of an application, using only a small part of the total CPU time. The other parts of a typical application, the ones that consume the random numbers, will typically be running in the same units that produced the random numbers and be subject to the same limitations on memory use and communication between streams. This is limiting the usefulness of GPUs for large scale Monte Carlo applications.

### Many-core coprocessors

Intel's current Many Integrated Core (MIC) Xeon Phi coprocessor codenamed Knights Corner has up to 61 cores with 512-bit vector registers and a clock frequency of 1.2 GHz (Chrysos, 2012). The throughput per core is much lower than for a general purpose CPU, and the total throughput is rarely more than a few times the throughput of the best mainstream CPU configurations. In some cases, a mainstream CPU can even outperform the Knights Corner (Saule, Kamer and Çatalyürek, 2013; Chan, 2013; Karpiński 2014). The Knights Corner has its own instruction set, which makes it less attractive for portable software. The announced successor, codenamed Knights Landing, is expected to be faster and it will be using the same instruction set (AVX-512) as future mainstream CPUs (Anthony, 2013). This will make it possible to use the same software on MIC processors and mainstream CPUs.

Similar products from other vendors include Nvidia Tesla and AMD FireStream. These processors have more in common with GPUs.

### Large vector processors

For most applications, clusters of general microprocessors have largely replaced the large and expensive supercomputers that were used decades ago for demanding scientific purposes.

## Parallel generation of pseudo-random numbers in vector processors

A PRNG generally uses a generating function f of the form (L'Ecuyer, 1994)

$$x_i = \mathrm{f}\left(x_{i-1}, x_{i-2}, \ldots, x_{i-n}\right)$$

where each new value $x_i$ is a function of the previous $n$ values. The successive values $x_i$ may be used directly as random numbers, or they may be transformed by an output function g of the form

$$y_i = \mathrm{g}\left(x_i, x_{i-1}, \ldots, x_{i-n}\right)$$

Not all of the values $x_{i-1}$, $x_{i-2}$, ..., $x_{i-n}$ need to be included in f. We will say that f has a feedback path of length $\varphi$ if f depends on $x_{i-\varphi}$. The function f can be implemented in a vector processor with registers of size $v$ bits if $v \leq w\varphi$ for all feedback paths $\varphi$, where $w$ is the number of bits needed to represent each $x_i$. For example, for a vector size $v$ of 256 bits and a word size $w$ of 32 bits, the shortest feedback path $\varphi$ must be at least 8 for an efficient vectorized implementation of f. If $\varphi \geq 8$ and $n \geq 8$ then we can calculate 8 successive values of $x_i$ with a vectorized function **f** of the form:

$$\left(x_{i+7}, x_{i+6}, \ldots, x_i\right) = \mathbf{f}\left(x_{i-1}, x_{i-2}, \ldots, x_{i-n}\right)$$

If $v > w\varphi$ then the vectorized function **f** needs to implement multiple steps of the generating function f. This is usually so complicated that it offsets the advantage of vectorized calculation.

The last $n$ values of $x_i$ are stored in a circular buffer, called the state buffer, which is updated by each call of the generating function f or **f**. The initial value of the state buffer is a function of an arbitrary number called the seed. This function is the so-called seeding procedure.

The size of the state buffer is at least $wn$ and often extended to the nearest multiple of the vector size $v$. The implementation is most efficient if $w\varphi$ and $wn$ are multiples of the vector size $v$.

Most of the commonly used PRNGs have a feedback path $\varphi = 1$, which makes them unsuited for vectorized calculation. Preferred generators are those with feedback paths corresponding to the largest vector size there is access to in

available vector processors. A generator designed to match 128-bit vector registers has been published under the name SIMD-oriented Fast Mersenne Twister (SFMT) (Saito and Matsumoto, 2008, 2009).

## Parallel generation of pseudo-random numbers in independent streams

The construction of generators suitable for vector processors has received relatively little attention in the literature, but the simultaneous generation of multiple pseudo-random streams has been discussed in several publications. Five different methods for producing independent streams have been proposed (L'Ecuyer, 1994; Salmon, 2011; L'Ecuyer, Oreshkin and Simard, 2014; Bauke and Mertens, 2007):

1. Use multiple instances of the same generator with different seeds. We want to avoid overlap between the generated subsequences. Assume that we are generating $k$ subsequences of length $\ell$ from a generator with total cycle length $\rho$. If the seeding procedure is sufficiently random then we can calculate the probability that any of the subsequences are overlapping as (L'Ecuyer, Oreshkin and Simard, 2014)

$$p \approx 1-\left(1-k\ell/\rho\right)^{k=1} \approx k^2\ell/\rho$$

   If the total cycle length $\rho$ is sufficiently long then this probability can be very small. For example, for a Mersenne Twister MT19937 (Matsumoto and Nishimura, 1998) with cycle length $\rho = 2^{19937}-1$, $k = 1000$ and $\ell = 10^{10}$, we have $p = 2\cdot 10^{-5986}$. This means that we can safely ignore the risk of overlapping subsequences in such cases.
2. Use a generator with a jump-ahead feature. We use this jump-ahead feature to generate each stream as a subsequence of the same generator at an offset $q \geq \ell$ relative to the preceding stream (L'Ecuyer, 1994; L'Ecuyer and Côté, 1991). The jump-ahead feature is usually quite complicated and requires a significant amount of computing resources. Regularly spaced starting points may cause inferior randomness for some generators (Durst, 1989).

3. A variant of the jump-ahead method is to put all the randomness in the output function g, while the generating function f is a simple counting $x_i = x_{i-1} + 1 \bmod 2^w$ (Salmon, 2011). This makes it trivial to generate non-overlapping subsequences. The output function g is borrowed from cryptology. Instructions for AES encryption are implemented in hardware in many computers, using a vector size of 128 bits, but not higher (Intel, 2014b).
4. Leapfrogging. The first of $k$ streams uses outputs $x_i$, $x_{i+k}$, $x_{i+2k}$, ... The next stream uses $x_{i+1}$, $x_{i+1+k}$, $x_{i+1+2k}$, ... and so on. This is useful when the $k$ streams form a vector generated by a single vector processor. It is more complicated to use leapfrogging when the streams are generated in separate processors. Known multiprocessor implementations use prime modulus (Bauke and Mertens, 2007), which leads to quantization errors (see below).
5. Use different generators based on the same principle but with different sets of parameters in the generating function. If we have many streams then we need to either store many pre-calculated parameter sets, or include the necessary code to search for good parameter sets on the fly (Matsumoto and Nishimura, 2000). This so-called dynamic creation method requires a lot of computational resources, possibly even more than the resources needed to generate the random number streams, and it has been reported to make inferior parameter sets in some cases (Passerat-Palmbach, Mazel, Mahul and Hill, 2010).

There is disagreement among theorists about whether method 5 can be recommended. One would intuitively assume that random streams generated by different generators with different parameter sets are statistically independent, but some have argued that we have no theoretical proof that there is no unwanted correlation between such random streams (Passerat-Palmbach, Mazel and Hill, 2011; L'Ecuyer, 1994). However, those who make this objection seem to ignore that the same argument can be made about subsequences from the same generator. Perhaps they rely on the implicit (and arguably false) assumption that the most recommended generators are perfect, and conclude that non-overlapping subsequences from the same generator are statistically independent.

However, if subsequences are spaced by an offset of e.g. $q = 10^{15}$ and experimental tests for randomness have included no sequences longer than $\ell = 10^{10}$ then we have no experimental proof that all subsequences are

independent, and no theoretical proof either (Bauke & Mertens, 2007). It is reasonable to assume that the probability of unwanted correlations between sequences from different generators (with different seeds) is not bigger than the probability of unwanted correlations between subsequences of the same generator. We will return to a more general discussion of theoretical proofs below.

6. A sixth method of making independent pseudorandom streams is now proposed. It involves the combination of two different PRNGs. We will have two different generators, G and H, and initialize them with seeds $s^1_G$ and $s^1_H$, respectively. G generates a pseudorandom sequence $x^1_{Gi}$ and H makes another sequence $x^1_{Hi}$, where each $x$ is an integer of $w$ bits, and $0 \leq i < \ell$. The two sequences are now combined into one stream by means of a bitwise XOR operation or addition modulo $2^w$, e.g. $x^1_i = x^1_{Gi} + x^1_{Hi} \bmod 2^w$. The combined stream $x^1_i$ now depends on both seeds $s^1_G$ and $s^1_H$. We can make a second combined stream (indicated by superscript 2) $x^2_i$ by changing the seed for G, $s^1_G$ to $s^2_G$ and keeping the seed for H constant: $s^1_G \neq s^2_G \wedge s^1_H = s^2_H$. The second combined stream is $x^2_i = x^2_{Gi} + x^2_{Hi} = x^2_{Gi} + x^1_{Hi} \bmod 2^w$. Now consider the unlikely event that the seed $s^2_G$ generates a sequence $x^2_{Gi}$ that is offset from $x^1_{Gi}$ by a distance $q < \ell$, perhaps because of a bad seeding procedure. In this case, the sequences $x^1_{Gi}$ and $x^2_{Gi}$ have a partial overlap of length $\ell - q$ because $x^2_{Gi} = x^1_{Gi+q}$. However, the contribution from H is $x^2_{Hi} = x^1_{Hi} \neq x^1_{Hi+q}$, except for random $i$-occurrences with expected frequency $2^{-w}$. Therefore, the first and second combined sequences $x^1_i$ and $x^2_i$ will be statistically independent, even in the unlucky event that the G component of the sequences have a partial overlap.
7. A variant of method 6 is to change both seeds: $s^1_G \neq s^2_G \wedge s^1_H \neq s^2_H$. To see if this method is safe from overlaps, consider the coincidence of three unlucky events: 1) The sequence $x^2_{Gi}$ is offset from $x^1_{Gi}$ by a distance $|q_G| < \ell$ so that the G-sequences have a partial overlap; 2) the sequence $x^2_{Hi}$ is offset from $x^1_{Hi}$ by a distance $|q_H| < \ell$ so that the H-sequences have a partial overlap; and 3) the two overlaps are equal $q_G = q_H$. The two combined sequences $x^1_i$ and $x^2_i$ have a partial overlap only in this contrived scenario. This is a theoretical possibility, but it can only happen at the coincidence of three unlucky events, all of which are extremely unlikely. The probability of this coincidence happening between any of $k$

combined sequences is approximately $k^2\ell$ / ($\rho_G\rho_H$) where $\rho_G$ and $\rho_H$ are the cycle lengths of G and H, respectively. With large cycle lengths, this probability is so low that there is room for human errors. Even in the event that both seeding procedures are seriously flawed, the coincidence of the three unlikely events seems no more than a theoretical possibility.

Method 7 has the advantage that the difference between two combined streams $d_i = x^2_i - x^1_i$ depends on both generators G and H, while $d_i$ depends only on G if method 6 is used. This gives improved randomness in applications where differences between streams are involved. The possible improvement in randomness by combining two different generators is discussed in the next section.

## Advantages of combined generators

The technique of combining two or more PRNGs is often used in order to improve randomness and cycle length. The cycle length of a combined generator is the least common multiple of the cycle lengths of the individual generators.

There are different opinions on the merits of combining two or more PRNGs. L'Ecuyer has argued that the combined output of two generators may conceivably be less random than the individual sequences (L'Ecuyer, 1990, 1994), while the acknowledged handbook *Numerical Recipes* emphasizes: "An acceptable random generator must combine at least two (ideally unrelated) methods" (Press, 2007, p. 342).

The combination of two random streams can only be less random than its components if the two streams are correlated in a certain way. The next section will discuss whether it is possible to prove that such an unfortunate correlation between two random streams does not exist.

It has been observed that the combination of two or more PRNGs produces a stream that is more random than either component. In fact, many good random generators have been made by combining inferior ones. Pragmatically speaking, we may say that if generator G has some defects and generator H has some other defects, then the combination of G and H has neither of these defects, as long as the defects of G and H are of different kinds. This is not a universal law of nature, of course, and it requires a more specific analysis to determine whether a particular kind of defect can be eliminated by combination of generators. There is plenty of theoretical evidence that various defects in random generators can be eliminated by combining with other generators that do not have the same kind of

defects (Matsumoto and Nishimura, 2000; Deng, Lin, Wang and Yuan, 1997; L'Ecuyer and Granger-Piché 2003; Marsaglia, 1985). Experience shows that combining two generators is a very efficient way of improving randomness. For example, if generator G has a bias that makes certain values more frequent than others, and generator H has no such bias, then the combined output of G and H will have no bias. If Generator H has a correlation between subsequent numbers and generator G has no such correlation, then the combined output will be free from such correlations. The two generators should preferably be very different in their design in order to avoid that they both have the same kinds of defects (Press, 2007).

Combining two or more generators is also useful in applications where security is important. It is possible to reconstruct a complete sequence from a subsequence in many generators. This becomes very difficult or impossible when multiple generators are combined and only the combined output is accessible to the attacker.

## How much can be proven?

It has been argued above that it is unreasonable to demand a theoretical proof that streams from different PRNGs are uncorrelated as long as we cannot even prove the same thing for different substreams of the same generator. This opens up a much more general discussion about what kind of proofs are actually possible in relation to PRNGs. There are three kinds of claims that we would like to prove for generators:

a) A particular generator G has no unwanted correlation with an application A, i.e. a correlation that would make A produce results that are significantly different from what perfectly random numbers would give.
b) There is no correlation between non-overlapping subsequences from the same generator G.
c) There is no correlation between the outputs of two different generators G and H.

Claims of type (a) are made implicitly or explicitly whenever a particular PRNG is recommended. Such claims may later be falsified when a particular weakness in a generator is discovered. For example, Linear congruential generators which have been widely used in commercial software were found after

many years to have serious defects (Entacher, 1998). The popular and often recommended Mersenne Twister has the flaw that it can produce long sequences with more 0's than 1's if it comes into a state where the state buffer contains mostly 0's. This flaw was reported only after the Mersenne Twister had been the preferred generator for several years (Saito and Matsumoto, 2008). A tiny bias in the Multiply-with-carry generators was discovered a few years after this kind of generators had been recommended (Couture and L'Ecuyer, 1997). In fact, one defect reported by Bauke and Mertens (2004) applies to a large part of all known PRNGs.

The possibility cannot be ruled out that more such discoveries will be made in the future, no matter how good we believe that our generators are. Claims that a PRNG is good should therefore be regarded as falsifiable propositions in accordance with Popper's (1963) philosophy of science. The claim that a generator produces random output is never true in the strictest sense, because the output is deterministic. It may be proven experimentally that the output of a PRNG passes certain tests for randomness, but the possibility that it will fail some test if a larger sample size is used cannot be ruled out. If the sample size is increased to the entire cycle length then the total sample is no longer random because, typically, all output values occur the same number of times in a full cycle.

In science, theoretical proofs are often regarded as stronger than experimental proofs. However, for PRNGs there is a dilemma. If it is possible to prove theoretically that a PRNG has a certain desirable property, then the theoretical insight that allowed this analysis may also be used in the construction of an experimental test that defeats the same generator. For example, the construction of generators in the Mersenne Twister family usually relies on the Berlekamp-Massey algorithm for verification of the cycle length (Saito and Matsumoto, 2008). Therefore, it is no surprise that the Mersenne Twisters fail a test based on the Berlekamp-Massey algorithm, the so-called linear complexity test (L'Ecuyer and Simard, 2007). If a chaotic behavior with no recognizable mathematical structure is what characterizes a good PRNG, then perhaps the best generators are the ones that are most difficult to prove good (Fog, 2001). On the other hand, attempts to produce PRNGs without any theory have led to very bad results (Knuth, 1998).

Claims of type (a) are generally the easiest to falsify. Most of the generators described in the literature have weaknesses that have been discovered by either experimental of theoretical methods.

Claims of type (b) have occasionally been falsified. Durst (1989) demonstrated a correlation between regularly spaced subsequences of linear congruential generators.

Claims of type (c) are the most difficult to falsify. The more different two generators are, the more difficult it is to construct a mathematical framework that allows the simultaneous analysis of both, and the more unlikely it is that they have a common structural property that can produce a correlation (Press, 2007). A given generator is more likely to correlate with an application, which can have a lot of regularity, than with another generator that was designed with the goal of avoiding correlations.

The dilemma that mathematical tractability is good for theoretical analysis but bad for randomness seems to prevent us from making the best random generators, or at least from knowing which generators are best. Fortunately, we can get along with less than perfect generators as long as we can eliminate known defects by combining two different generators. This means that we can live with minor imperfections in (a) and (b) as long as we can rely on claims of type (c).

It is unreasonable to demand a theoretical proof of type (c) for three reasons. The first reason is that it is not clear what kind of theoretical proof is expected to prove the randomness of a pseudo-random sequence of numbers. The second reason is that the philosophy of science does not allow absolute proofs of this kind, only evidence and falsifiable hypotheses. And the third reason is that the mathematical tractability that would allow such a proof, would also defeat it.

All evidence, theoretical as well as experimental, supports the claim that we can improve randomness by combining the outputs of two or more very different generators. We will rely on this claim as long as it has not been falsified, because it is the best method we have so far for producing deterministic pseudo-random numbers. A more general philosophical discussion is needed about what kind of proofs are possible or desirable in relation to PRNGs.

## Quantization effects

The minimum difference between two floating point numbers in the interval [½, 1] is $\delta = 2^{-24}$ for single precision, and $2^{-53}$ for double precision according to the IEEE-754 standard, which all modern computers support (IEEE Computer Society, 2008). The minimum difference for single precision is $2^{-25}$ in [¼, ½], $2^{-26}$ in [⅛, ¼], and so on. Many applications require random floating point numbers with uniform distribution in the interval [0,1). If we require equidistant points with the best possible resolution in single precision, then we will have $2^{24}$ possible

values in the interval [0,1). For this, we need a generator capable of giving $2^{24}$ different values, all with the same frequency. If the generator outputs e.g. a 32-bit word then we can simply use 24 of these bits and discard the remaining 8 bits.

For most generators, the generating function f gives an integer output $x_i$ in an interval [0, m). Typically f is some arithmetic function modulo $m$. If $m$ is a power of 2 then we can easily extract the desired number of random bits. Unfortunately, many of the generators that are described in the literature have a modulus $m$ which is not a power of 2. Often $m$ is a prime because functions with prime modulus have advantageous mathematical properties. When converting a pseudorandom integer $x_i$ modulo $m$ to a floating point number in [0,1) it is common to just divide $x_i$ by $m$. Unfortunately, this does not give equidistant points with equal frequency. If $m < 2^{24}$ then there will be some of the possible values that never occur. If $m > 2^{24}$ then some values between 0.5 and 1 will occur more frequently than other, and values less than 0.5 can be spaced less than $\delta = 2^{-24}$ apart. Such quantization effects can lead to systematic errors in applications that depend on the probability that a random number falls within a certain narrow interval.

For example, consider a generator with prime modulus $m = 2^{32}-5$ (e.g. L'Ecuyer, 1999). A floating point output from this generator will have the value 0.6 with frequency $255/m$, while the next value $0.6 + \delta$ occurs with frequency $256/m$. The value 0.2 occurs with frequency 63/m while the next value $0.2 + \delta/4$ occurs with frequency $64/m$.

Such inaccuracies may be unimportant in small applications, but in large applications that use billions of random numbers, the accumulated errors may actually be statistically significant. It is possible to eliminate the quantization errors by means of a rejection method, but this is quite costly in terms of efficiency (See below for an example of a rejection method). Alternatively, the quantization error may be tempered by an appropriate output function that uses multiple elements in the state buffer.

## Why is the output interval half open?

The half-open intervals [0,1) and (0,1] can both be divided into $2^{24}$ equidistant points with the maximum resolution $\delta = 2^{-24}$ for single precision floating point numbers. This makes it easy to generate a uniformly distributed variable from 24 random bits. We will have quantization errors, as explained above, if we map a 24-bit random number to one of the symmetric intervals [0,1] and (0,1), which have $2^{24} + 1$ and $2^{24} - 1$ equidistant points, respectively.

A Monte Carlo application can generate an event with probability $p \in [0,1]$ by testing $x < p$, where $x \in [0,1)$ is a uniform random variable. If $x$ is quantized as $2^{24}$ equidistant points in [0,1) with equal frequency and $p$ is similarly quantized by $\delta = 2^{-24}$ then the event $x < p$ will occur with the exact frequency $p$. If $x \in (0,1]$ then $x \leq p$ will also occur with the exact frequency $p$. A uniformly distributed $x$ in one of the symmetric intervals [0,1] or (0,1) will give rise to tiny rounding errors in the frequency of $x < p$.

A disadvantage of the half-open intervals is that the mean is not exactly ½, but $(1-\delta)/2$ and $(1+\delta)/2$, respectively. This is acceptable for most purposes since it will take a sample size of $8 \cdot 10^{14}$ to estimate the mean of $x$ with enough precision to get a statistically significant error of 3 standard deviations.

## Requirements for good generators

Consider some requirements that are important for the choice of PRNGs for large applications using vector processors, multicore processors and CPU clusters.

1. The generator should pass experimental tests for randomness.
2. The cycle length should be so high that the risk of overlapping subsequences is negligible, but not so high that the state buffer uses an excessive amount of data cache.
3. Good equidistribution, as determined by theoretical or experimental methods (L'Ecuyer, 1994).
4. Good diffusion. This is obtained if each bit in the state buffer depends on multiple bits in the previous state (Panneton, L'Ecuyer and Matsumoto, 2006). Diffusion is closely related to the concept of bifurcation in chaos theory (Fog, 2001; Černák, 1996). A good diffusion means highly chaotic behavior, which is a desirable property for a PRNG.
5. The shortest feedback path should be long enough to fit the largest available vector register. However, a long feedback path means poor diffusion. Therefore, the shortest feedback path should not be longer than necessary.
6. The modulus $m$ should be a power of 2 to avoid quantization effects and rounding errors.
7. The generator should be reasonably fast.
8. It should be possible to generate independent streams from multiple instances of the generator.

## Construction of a generator satisfying these requirements

There are many PRNGs described in the literature, but few that satisfy all the requirements listed above. Parallel generation has relied more on multiprocessors than on vector processors (L'Ecuyer, Oreshkin and Simard, 2014). The only generator explicitly designed for vector processors is the "SIMD-oriented Fast Mersenne Twister" (SFMT), which relies on 128-bit vectors (Saito and Matsumoto, 2008, 2009). Unfortunately, the feedback path of this generator does not allow implementations in larger vector registers, and there are no plans for an extended version (Saito, 2014). The general Mersenne Twisters have long feedback paths (Matsumoto and Nishimura, 1998; Nishimura, 2000) so that they can easily be implemented in vector processors. These generators have poor diffusion and slow recovery from a state of mostly 0's. The recently published variant "Mersenne Twister for Graphic Processors" (MTGP) (Saito and Matsumoto, 2013) has somewhat improved diffusion properties, and this appears to be the best choice. The chosen version has the Mersenne exponent 11213, which gives a state buffer size of 351 x 32 bits. The cycle length is $\rho = 2^{11213}-1$. This is more than enough to avoid overlapping subsequences, and higher values would be a waste of data cache. Smaller versions have not been published. The shortest feedback path is 84 x 32 bits, which makes implementation in large vector registers possible.

This generator has known weaknesses, which are common to the Mersenne Twister family: It is vulnerable to tests based on $\mathbb{F}_2$ algebra; it has relatively poor diffusion; and it has subsequences with more 0's than 1's. These weaknesses should be eliminated by combination with a second generator that does not have the same weaknesses.

Other generators with long feedback paths are difficult to find in the literature. The RANROT generator is a lagged Fibonacci generator with bit rotation (Fog, 2001). This generator is simple and fast, it can be constructed with any feedback path length, and most versions pass all tests for randomness. However, this is an example of a generator that is difficult to analyze theoretically. Assumptions about the cycle lengths of RANROT generators are based on extrapolations from experimental measurements on very small generators. The RANROT may be a good generator, but more research is needed before we can rely on this generator for demanding applications.

No other generator was found with a sufficiently long feedback path suitable for our purpose. Multiply-with-carry generators with lag have been described, but they have an extra feedback path of length 1 in the carry (Marsaglia, 2003). It

may be possible to construct a multiply-with-carry generator where the carry feedback is also lagged.

Because no suitable candidate for the second generator has been found with a feedback path that allows vectorization, we have instead to rely on multiple parameter sets for the same kind of generator (method 5). Each vector position will have its own independent generator with different parameters for each. After rejecting generators with prime modulus, the best candidate we found was a multiply-with-carry (MWC) generator (Goresky and Klapper, 2003). This generator is relatively simple, it has excellent randomness and very high diffusion or bifurcation. Nine good multipliers for MWC are listed by Press (2007). Eight of these are used in order to implement eight generators of 64 bits each in a 512 bit vector. The output function is a 64-bit XOR-shift method as recommended by Press (2007). Unfortunately, there are not enough good multipliers for future implementations in larger vector registers. Each MWC generator delivers a 64-bit output which is divided into two 32-bit random numbers.

The eight MWC generators have different cycle lengths, ranging from $5 \cdot 10^{18}$ to $9 \cdot 10^{18}$. This is not enough to completely rule out overlapping subsequences in large applications when the MWC generator is used alone, but the MTGP generator has prime cycle length so that the cycle lengths are multiplied when the MWC and MTGP generators are combined.

The MWC generator has a very slight bias in the upper bits (Couture and L'Ecuyer, 1997). The bias is too small to have practical significance, and it is removed by the output function or by the combination with the MTGP generator anyway.

It can be concluded that the MTGP and MWC generators both have known defects, but they have no defects in common. There are no known defects in any of these two generators that cannot be removed by combination with the other generator. Therefore, it is expected that the combined output of these two generators is suitable for even the most demanding applications. Multiple independent streams can be generated from multiple instances of the combined generator by changing the seed of one or both generators, in accordance with method 6 or 7.

## Practical implementation

It was decided to make an implementation that is suitable for the forthcoming AVX-512 instruction set, which will be common to the most relevant hardware platforms in a near future. Existing instruction sets with vector sizes smaller than

512 bits are supported by dividing the data into smaller vectors. C++ is the obvious choice of programming language for code that needs to be portable to several platforms and operating systems, highly optimized, and needs overloaded operators for vector operations. The code is integrated into the vector class library (VCL. Fog, 2014b) which provides efficient vector operators for the generator as well as for the application that uses it. Supported platforms include Windows, Linux and Mac OS with Microsoft, Intel, Gnu and Clang compilers.

The generator, named RANVEC1, is implemented as a C++ class so that an application can make a separate instance for each thread in a multiprocessor environment. Each instance can deliver random number vectors of up to 512 bits with integer or floating point elements.

The fastest way of generating a uniform floating point output with equidistant points from random bits is to set the exponent of a single precision floating point number in the IEEE-754 representation to (0+bias) and set the mantissa to 23 random bits. This gives a uniform random number in the interval [1,2). Subtracting 1 then gives a number in the desired interval [0,1) (Saito and Matsumoto, 2009). This method gives a resolution of $2^{-23}$. The maximum resolution of $\delta = 2^{-24}$ can be obtained from 24 random bits by first using 23 bits to make a random number in the interval [1,2) as above, and then subtracting either 1 or $(1-\delta)$ depending on whether the last bit is 0 or 1. It is possible to make a double precision random number with the maximum resolution of $2^{-53}$ by the same method, but the current implementation gives only a resolution of $2^{-52}$ for double precision because it was decided that the last bit will have no significance for applications with a realistic sample size.

Many applications need a random integer $u$ with uniform distribution in an interval $[a,b]$ of length $d$ = b-a + 1. This can be obtained from a random 32-bit unsigned integer $x$ by a 64-bit multiplication: $u = a + \lfloor xd / 2^{32} \rfloor$. However, this method is subject to a bias similar to the quantization error discussed above when the interval length $d$ is not a power of 2. Floating point calculation methods give the same error because of the mapping of an interval of a power-of-2 length to another interval of incommensurable length $d$. Most standard random generator libraries have this error. The error may be negligible when $d$ is small, but it can be quite serious for large $d$. The worst case is $d = 3 \cdot 2^{30}$. In this case, values of $(u - a)$ that are divisible by 3 occur twice as frequent as other values. This can obviously lead to serious errors in applications that happen to depend on $u$ mod 3. This error can be eliminated by using a rejection method. Confine $x$ to $r$ possible values

where $r$ is a multiple of $d$. $r = \lfloor 2^{32} / d \rfloor \cdot d$. If $xd \bmod 2^{32} \geq r$ then reject the value and generate a new $x$.

Rejection methods are also used for generating random variables with other distributions than uniform (Devroye, 1986). Algorithms that involve rejection methods may be implemented in vector processors as follows. First generate a random vector and execute the steps in the algorithm necessary to determine rejection. If any elements of the vector are rejected, then generate another random vector and repeat the calculations. Replace any rejected elements in the first vector by accepted elements from the second vector. Continue like this until we have a vector of only accepted elements. If calculations are expensive and not dependent on changing parameters then we may save any remaining accepted elements for the next round. If exact reproducibility across platforms is required then we must keep the vector size constant.

## Tests of the constructed generator

The randomness of the generator outputs were tested using the powerful BigCrush battery of tests in the TestU01 software suite of experimental tests for randomness (L'Ecuyer and Simard, 2007). The MWC generators were tested in various configurations: each of the eight generators separately, the lower or upper 32-bit half of each generator output, as well as all eight generators in a round robin fashion. All tests were passed. The MWC generators failed several tests when the XOR-shift output function was removed.

The MTGP generator failed the linear complexity test as expected, but passed all other tests in the BigCrush battery of tests. The MTGP generator also failed a binary matrix rank test where the matrix size was increased to 12000×12000. The test results were the same when the output function (so called tempering) was removed. The combination of the MWC and MTGP generator passed all tests, with or without tempering.

The speed of the random generators were tested after compiling with different compilers and different vector register sizes. The test measured the time required to generate $2^{14}$ random 32-bit integers and computing their sum. The calculation time depends on the CPU clock frequency, which varies a lot due to the power-saving features of the CPU. In order to get consistent and reproducible time measurements, it was decided to use the core clock count as time unit. This time unit is defined by the frequency that the execution unit in the CPU is actually running at. Core clock counts were measured using the TESTP test program (Fog,

2014c). The calculation speed was measured for the MWC and MTGP generators as well as for the SFMT generator and the original Mersenne Twister (MT). The results are given in Table 2.

**Table 2.** Random number generation times for various generators using different compilers and register sizes. The unit is core clock cycles per 32 bits, single thread.

| | | Compiler | | | |
|---|---|---|---|---|---|
| **Generator** | **Register size bits** | **Gnu** | **Clang** | **Intel** | **Microsoft** |
| MWC | 128 | 4.1 | 4.0 | 3.6 | 3.0 |
| | 256 | 1.8 | 2.2 | 2.6 | 3.1 |
| MTGP | 128 | 8.9 | 10.3 | 8.8 | 18.4 |
| | 256 | 4.0 | 4.5 | 4.5 | 43.1 |
| MTGP w/o tempering | 256 | 3.1 | 3.5 | 3.6 | 18.9 |
| MWC + MTGP | 128 | 10.4 | 12.4 | 10.4 | 20.3 |
| | 256 | 5.0 | 5.7 | 6.1 | 46.4 |
| MWC + MTGP w/o tempering | 256 | 3.9 | 4.6 | 5.1 | 20.7 |
| SFMT | 128 | 2.0 | 1.8 | 2.0 | 1.9 |
| MT | 32 | 9.3 | 14.2 | 8.5 | 12.8 |

*Configuration:* Intel Haswell microprocessor, 3.4 GHz. Windows 7, 64 bits. Gnu C++ compiler v. 4.8.3 Cygwin. Clang C++ compiler v. 3.4.2 Cygwin. Intel C++ compiler v. 15.0. Microsoft C++ compiler v. 17.0.

Notice that the combined generator takes $5-6$ clock cycles per random number using a vector size of 256 bits when the Gnu, Clang or Intel compiler is used. This corresponds to approximately $6 \cdot 10^8$ random numbers per second per thread on a 3.4 GHz processor. This number can be multiplied by the number of cores in the CPU when each core is running one thread. It is possible to run two threads per core on some CPUs, but this may not be optimal if the two threads are competing for the same execution resources (Fog, 2014a).

Most Monte Carlo applications take much more time than this to process the random numbers, so that the random number generation will account for only a small fraction of the total execution time. A few clock cycles more or less is hardly important in this context. Therefore, we can afford the luxury of using a combined generator of very high quality. The convenient availability of random numbers as vectors can make it easier to vectorize the applications that use the

random numbers, possibly leading to very significant speed gains for some applications.

The RANVEC1 code also supports a register size of 512 bits. This was verified using Intel Software Emulator version 7.1.0, but no meaningful speed measurement was possible because no microprocessor with the AVX-512 instruction set is available yet.

The SFMT generator is faster than the MTGP generator because the former is designed specifically for vector processing while the MTGP is designed for graphics processors. Unfortunately, the SFMT generator cannot be implemented with vector sizes higher than 128 bits.

## Conclusion

There are two main principles for parallel processing: vector processing and multicore processing. Large Monte Carlo applications need to utilize both in order to get the maximum performance out of modern computers. A literature search revealed only one generator specifically designed for vector processing, and none that fits the growing vector size of modern processors. Fortunately, it is possible to utilize vector processors by adapting other generators with sufficiently long feedback paths or by implementing multiple similar generators in parallel. The combined generator described here (RANVEC1) utilizes both methods. A C++ implementation of this combined generator is available as part of the vector class library (VCL) at http://www.agner.org/optimize/#vectorclass.

As Monte Carlo applications get larger they also put higher demands on the quality of random number generators. The following qualities must be considered:

1. Quality of randomness.
2. Speed.
3. Avoid overlapping sequences.
4. Equidistant points with perfectly uniform distribution.
5. Portability among platforms.
6. Reproducibility.

The quality of randomness (1) can be improved by combining two generators with fundamentally different design. This enables us to overcome the flaws caused by the unsolvable dilemma between the need for mathematical tractability and the desire for chaotic behavior.

The speed (2) of the available generators is so high that the generation of random numbers accounts for only a small fraction of the total calculation time of a typical application. However, there is a pitfall when measuring the speed of a generator in isolation. The larger Mersenne Twister generators are consuming considerable amounts of data cache whereby they may slow down the applications that use them. The size of the state buffer should be a compromise between long cycle length and low data cache use.

The risk of overlapping sequences (3) gets higher as the number of simultaneous random streams is increasing. This risk can be made negligible by using a generator with an extremely long cycle length, or we can eliminate it completely by combining two different generators.

Quantization effects are often ignored in standard PRNG libraries, which makes them deviate from the perfectly uniform distribution (4). Undesired quantization effects are seen when the output of a generator with prime modulus is mapped onto an interval with power-of-2 modulus and when the output of any generator is used for generating a random integer in an interval of arbitrary (incommensurable) length. These undesired effects can be eliminated by avoiding generators with prime modulus or by using a rejection method.

Portability (5) is generally obtained by using a standardized programming language. The RANVEC1 generator is designed for the vector extensions to the x86 instruction set. This fits the most commonly used computer platforms today, as well as prospected future processors with 512-bit vectors. It cannot be used on platforms with other instruction sets without major reprogramming, and the target platform must have similar vector processing capabilities.

Reproducibility (6) is useful for replaying an interesting simulation event, for verifying results and for debugging. It is always possible to reproduce a random number stream by using the same generator again with the same seed. However, problems may arise when vector sizes change. For example, consider a simulation application that uses both integer and floating point random number vectors. First, it generates a vector of 8 integers, then a vector of 8 floats, then 8 integers, 8 floats, etc. If we now update the hardware to a processor that supports bigger vectors, we may generate first 16 integers and then 16 floats, etc. This means that the numbers are generated in a different order so that the simulation results will be different even though we have used the same seed. A remedy against this problem is to generate numbers in batches that correspond to the biggest possible vector size. The RANVEC1 software uses batches of 512 bits to fit the future AVX-512 instruction set, but the reproducibility will be lost in case

of future extensions to 1024 bits or more. Reproducibility can also be lost in case of outputs that use a rejection method when the vector size is changed.

## Scope for future research

We have found an acceptable solution to our needs for a good PRNG that utilizes both vector processing and multiprocessing, but we can predict the future need for a generator that fits larger vector sizes. We would also like a more efficient solution even though the speed is acceptable for current purposes.

The vector implementation of the MTGP is slower than the SFMT even though it can use a larger vector size. The difference in speed can be explained by the following factors.

- The size of the state buffer in the MTGP is not divisible by the vector size. Extra code is needed to handle the wrap-around situation where a vector spans part of the end of the buffer and part of the beginning. Memory access is misaligned for the same reason.
- The output function in the MTGP, called tempering, consumes a large fraction of the code and CPU time. The purpose of the tempering is to improve equidistribution, but this improvement is not visible in the test results. The SFMT generator obtains good equidistribution by an appropriate choice of parameters without a tempering function.
- The MTGP algorithm has longer dependency chains than the SFMT.
- The SFMT can use the state buffer also as output buffer in a block generation scheme. This is not possible with the MTGP because its tempering function needs to read two parts of the state buffer for each output value.

A better solution would have a state buffer size that is a multiple of the largest vector size we expect to be available in a reasonable future. It is possible to increase the state buffer size beyond the Mersenne exponent either by having some bits without feedback or by using the same method as the SFMT (Saito and Matsumoto, 2008, 2009). The state buffer size should not be excessive because of the data cache use. Parameters should be adjusted to give satisfactory equidistribution in order to eliminate the need for a tempering function.

The shortest feedback path should be at least as long as the largest possible vector size. There is a tradeoff here because a large feedback path is reducing the

diffusion in the generator. The diffusion is already low in many variants of Mersenne Twisters because they use sparse matrixes in the algorithm. There are various ways to make more dense matrixes without excessive computation time. It is possible to implement a 4×32 bit $\mathbb{F}_2$ matrix multiplication with a single 512-bit vector permutation instruction, and this method is used in the RANVEC1 code. Another possibility, which has not been utilized so far, is to use carry-less multiplication. Modern x86 processors have such an instruction. The carry-less multiplication instruction multiplies two 64-bit vectors to give a 127-bit product (Intel, 2014b), and this corresponds to a dense matrix multiplication in $\mathbb{F}_2$. Unfortunately, there is no version of this instruction with larger vectors, but the result can easily be broadcast into a larger vector in order to increase diffusion.

The second generator in our combination, the MWC, cannot easily be expanded to larger vectors than 512 bits. There are nine known good multipliers for a 64-bit MWC (Press, 2007) and we have used eight of these for implementing eight parallel MWC generators. Future implementations with larger vector sizes need another generator with more good parameter sets—perhaps a variant of MWC with an addend, an extra term or a short lag.

These are very practical problems, which can definitely be solved. On a more philosophical level, we need a clarification of the role of proofs in PRNG research. Is it possible to prove that a generator has no defects? What kind of evidence can we accept? If all we have is falsifiable propositions, does it make sense to say that some propositions have more value than others if it is more difficult to find examples that falsify them? Does it make sense to require theoretical proofs, e.g. that two random number streams are statistically independent, when it is impossible to even prove the more fundamental assumptions about randomness of a single stream?

## References

Anthony, S. (2013). *Intel unveils 72-core x86 Knights Landing CPU for exascale supercomputing* [Blog post]. Retrieved from http://www.extremetech.com/extreme/171678-intel-unveils-72-core-x86-knights-landing-cpu-for-exascale-supercomputing

Barash, L. Yu., & Shchur, L. N. (2014). PRAND: GPU accelerated parallel random number generation library: using most reliable algorithms and applying parallelism of modern GPUs and CPUs. *Computer Physics Communications*, *185*(4), 1343–53. doi:10.1016/j.cpc.2014.01.007

Bauke, H., & Mertens, S. (2004). Pseudo random coins show more heads than tails. *Journal of Statistical Physics*, *114*(3–4), 1149–69. doi:10.1023/B:JOSS.0000012521.67853.9a

Bauke, H., & Mertens, S. (2007). Random numbers for large-scale distributed Monte Carlo simulations. *Physical Review E*, *75*(6), 066701. doi:10.1103/PhysRevE.75.066701

Černák, J. (1996). Digital generators of chaos. *Physics Letters A*, *214*(3), 151–60. doi:10.1016/0375-9601(96)00179-X

Chan, E. Y. K. (2013). *Benchmarks for Intel MIC Architecture*. Retrieved from http://www.clustertech.com/wp-content/uploads/2014/01/MICBenchmark.pdf

Chrysos, G. (2012). *Intel Xeon Phi Coprocessor - the Architecture*. Retrieved from http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner

Couture, R., & L'Ecuyer, P. (1997). Distribution properties of multiply-with-carry random number generators. *Mathematics of Computation*, *66*(218), 591–607.

Demchik, V. (2011). Pseudo-random number generators for Monte Carlo simulations on ATI graphics processing units. *Computer Physics Communications*, *182*(3), 692–705. doi:10.1016/j.cpc.2010.12.008

Deng, L. Y., Lin, D. K. J., Wang, J., & Yuan, Y. (1997). Statistical justification of combination generators. *Statistica Sinica*, *7*, 993–1003.

Devroye, L. (1986). *Non-uniform random variate generation*. New York: Springer.

Durst, M. J. (1989). Using linear congruential generators for parallel random number generation. In E. A. MacNair, K. J. Musselman, & P. Heidelberger (Eds.), *WSC '89 Proceedings of the 21st conference on Winter simulation* (pp. 462–66). New York: ACM. doi:10.1145/76738.76798

Entacher, K. (1998). Bad subsequences of well-known linear congruential pseudorandom number generators. *ACM Transactions on Modeling and Computer Simulation*, *8*(1), 61–70.

Fog, A. (2001). *Chaotic random number generators with random cycle lengths*. Publisher: Author. Retrieved from http://www.researchgate.net/publication/245642152

Fog, A. (2014a). *Optimizing software in C++. An optimization guide for Windows, Linux and Mac platforms*. Publisher: Author. Retrieved from http://www.agner.org/optimize/optimizing_cpp.pdf

Fog, A. (2014b). *C++ vector class library* [Software library]. Publisher: Author. Retrieved from http://www.agner.org/optimize/#vectorclass

Fog, A. (2014c). *Test programs for measuring clock cycles and performance monitoring* [Software library]. Publisher: Author. Retrieved from http://www.agner.org/optimize/#testp

Goresky, M., & Klapper, A. (2003). Efficient multiply-with-carry random number generators with maximal period. *ACM Transactions on Modeling and Computer Simulation*, *13*(4), 310–21. doi:10.1145/945511.945514

IEEE Computer Society. (2008). *IEEE Standard for Floating-Point Arithmetic* (IEEE Std. 754-2008) New York: IEEE.

Intel. (2014a). *Intel architecture instruction set extensions programming reference* (Doc. 319433-021). Retrieved from http://software.intel.com/en-us/intel-isa-extensions

Intel. (2014b). Intel 64 and IA-32 architectures software developer's manual (Doc. 325462-052US). http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

Karpiński, P. (2014). *Evaluation of Intel Xeon Phi (Knight's Corner) coprocessor's core performance using VCL*. Manuscript submitted for publication.

Knuth, D. E. (1998). *The art of computer programming, volume 2: seminumerical algorithms* (3rd Ed). Boston, MA: Addison-Wesley Professional.

L'Ecuyer, P. (1990). Random numbers for simulation. *Communications of the ACM, 33*(10), 85–97. doi:10.1145/84537.84555

L'Ecuyer. P. (1994). Uniform random number generation. *Annals of Operations Research, 53*(1), 77–120. doi:10.1007/BF02136827

L'Ecuyer, P. (1999). Tables of linear congruential generators of different sizes and good lattice structure. *Mathematics of Computation, 68*(225), 249–60. doi:10.1090/S0025-5718-99-00996-5

L'Ecuyer, P., & Côté, S. (1991). Implementing a random number package with splitting facilities. *ACM Transactions on Mathematical Software, 17*(1), 98–111. doi:10.1145/103147.103158

L'Ecuyer, P., & Granger-Piché, J. (2003). Combined generators with components from different families. *Mathematics and Computers in Simulation, 62*(3–6), 395–404. doi:10.1016/S0378-4754(02)00234-3

L'Ecuyer, P., Oreshkin, B., & Simard, R. (2014). *Random numbers for parallel computers: requirements and methods*. Manuscript submitted for publication.

L'Ecuyer, P., & Simard, R. (2007). TestU01: a C library for empirical testing of random number generators. *ACM Transactions on Mathematical Software, 33*(4), 22. doi:10.1145/1268776.1268777

Manssen, M., Weigel, M., & Hartmann, A. K. (2012). Random number generators for massively parallel simulations on GPU. *European Physical Journal: Special Topics, 210*(1), 53–71. doi:10.1140/epjst/e2012-01637-8

Marsaglia, G. (1985). A current view of random number generators. In L. Billard (Ed.), *Computer science and statistics: proceedings of the Sixteenth Symposium on the Interface, Atlanta, Georgia*, (pp. 3–10). Amsterdam: North-Holland.

Marsaglia, G. (2003). Random number generators. *Journal of Modern Applied Statistical Methods, 2*(1), 2–13. http://digitalcommons.wayne.edu/jmasm/vol2/iss1/2/

Matsumoto, M., & Nishimura, T. (1998). Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. *ACM Transactions on Modeling and Computer Simulation, 8*(1), 3–30.

Matsumoto, M., & Nishimura, T. (2000). Dynamic creation of pseudorandom number generators. In H. Niederreiter & J. Spanier (Eds.), *Monte Carlo and Quasi-Monte Carlo Methods, 1998: Proceedings of a Conference Held at the Claremont Graduate University, Claremont, California, USA* (pp. 55–69). New York: Springer-Verlag, Inc.

Nandapalan, N., Brent, R.P., Murray, L. M., & Rendell, A. P. (2012). High-performance pseudo-random number generation on graphics processing units. In R. Wyrzykowski, J. Dongarra, K. Karczewski, & J. Waśniewski (Eds.), *Lecture Notes in Computer Science 7203: Parallel Processing and Applied Mathematics (9th International Conference, PPAM 2011, Torun, Poland, September 11-14, 2011, Revised Selected Papers, Part I)* (pp. 609–18). New York: Springer-Verlag. doi:10.1007/978-3-642-31464-3_62

Nishimura, T. (2000). Tables of 64-Bit Mersenne twisters. *ACM Transactions on Modeling and Computer Simulation, 10*(4), 348–57. doi:10.1145/369534.369540

Panneton, F., L'Ecuyer, P., & Matsumoto, M. (2006). Improved long-period generators based on linear recurrences modulo 2. *ACM Transactions on Mathematical Software, 32*(1), 1–16. doi:10.1145/1132973.1132974

Passerat-Palmbach, J., Mazel, C., Mahul, A., & Hill, D. (2010). Reliable Initialization of GPU-Enabled Parallel Stochastic Simulations Using Mersenne Twister for Graphics Processors. In G. K. Janssens, K. Ramaekers, & A. Caris (Eds.), *European Simulation and Modelling 2010, Essen, Belgium* (pp. 187–95). Ostend, Belgium: Eurosis.

Passerat-Palmbach, J., Mazel, C., & Hill, D. R. C. (2011). Pseudo-random number generation on GP-GPU. In S. Strassburger (Ed.), *2011 IEEE Workshop on Principles of Advanced and Distributed Simulation (PADS)*, (pp. 1–8). New York: IEEE. doi:10.1109/PADS.2011.5936751

Popper, K. (1963). *Conjectures and refutations: the growth of scientific knowledge*. London, U.K.: Routledge.

Press, W. H. (2007). *Numerical recipes: the art of scientific computing* (3rd Ed.). Cambridge, U.K.: Cambridge University Press.

Saito, M. (2014). Personal communication.

Saito, M., & Matsumoto, M. (2008). SIMD-oriented fast Mersenne twister: a 128-Bit pseudorandom number generator. In A. Keller, S. Heinrich, & H. Niederreiter (Eds.), *Monte Carlo and Quasi-Monte Carlo Methods 2006*, (pp. 607–22). New York: Springer. doi:10.1007/978-3-540-74496-2_36

Saito, M., & Matsumoto, M. (2009). A PRNG specialized in double precision floating point numbers using an Affine transition. In P. L'Ecuyer and A. B. Owen (Eds.), *Monte Carlo and Quasi-Monte Carlo Methods 2008*, (pp. 589–602). New York: Springer. doi:10.1007/978-3-642-04107-5_38

Saito, M., & Matsumoto, M. (2013). Variants of Mersenne twister suitable for graphic processors. *ACM Transactions on Mathematical Software, 39*(2), 12. doi:10.1145/2427023.2427029

Salmon, J. K. (2011). Parallel random numbers: as easy as 1, 2, 3. In J. Costa & W. Kramer (Program Chairs), *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, Seattle, WA*. doi:10.1145/2063384.2063405

Saule, E., Kamer, K., & Çatalyürek, Ü. V. (2013). *Performance Evaluation of Sparse Matrix Multiplication Kernels on Intel Xeon Phi* (arXiv:1302.1078). http://arxiv.org/abs/1302.1078.