Friedrich-Schiller-Universität Jena
Physikalisch-Astronomische Fakultät

Design and Implementation of Vectorized Pseudorandom Number Generators and their Application to Simulation in Physics

MASTER'S THESIS

for obtaining the academic degree

Master of Science (M.Sc.) in Physics

submitted by Markus Pawellek

born on May 7th, 1995 in Meiningen
Student Number: 144645

Primary Reviewer:	Bernd Brügmann
Primary Supervisor:	Joachim Gießen

Jena, September 18, 2019

Abstract

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum. Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Acknowledgements

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Contents

List of Figures

List of Abbreviations

Abbreviation	Definition
iid	independent and identically distributed
RNG	Random Number Generator
TRNG	True Random Number Generator
PRNG	Pseudorandom Number Generator
LCG	Linear Congruential Generator
MCG	Multiplicative Congruential Generator
MT	Mersenne Twister
MT19937	Mersenne Twister with period $2^{19937} - 1$
PCG	Permuted Congruential Generator
CPU	Central Processing Unit
GPU	Graphics Processing Unit
SIMD	Single Instruction, Multiple Data
SSE	Streaming SIMD Extensions
AVX	Advanced Vector Extensions

Symbol Table

Symbol	Definition
$x \in A$	x ist ein Element der Menge A.
$A \subset B$	A ist eine Teilmenge von B.
$A \cap B$	$\{x \mid x \in A$ und $x \in B\}$ für Mengen A, B — Mengenschnitt
$A \cup B$	$\{x \mid x \in A$ oder $x \in B\}$ für Mengen A, B — Mengenvereinigung
$A \setminus B$	$\{x \in A \mid x \notin B\}$ für Mengen A, B — Differenzmenge
$A \times B$	$\{(x, y) \mid x \in A, y \in B\}$ für Mengen A und B — kartesisches Produkt
$\emptyset$	$\{\}$ — leere Menge
$\mathbb{N}$	Menge der natürlichen Zahlen
$\mathbb{N}_0$	$\mathbb{N} \cup \{0\}$
$\mathbb{R}$	Menge der reellen Zahlen
$\mathbb{R}^n$	Menge der n-dimensionalen Vektoren
$\mathbb{R}^{n\times n}$	Menge der $n \times n$-Matrizen
$f\colon X \to Y$	f ist eine Funktion mit Definitionsbereich X und Wertebereich Y
$\partial\Omega$	Rand einer Teilmenge $\Omega \subset \mathbb{R}^n$
σ	Oberflächenmaß
λ	Lebesgue-Maß
$\int_\Omega f \,\mathrm{d}\lambda$	Lebesgue-Integral von f über der Menge Ω
$\int_{\partial\Omega} f \,\mathrm{d}\sigma$	Oberflächen-Integral von f über der Menge $\partial\Omega$
∂_i	Partielle Ableitung nach der i. Koordinate
∂_t	Partielle Ableitung nach der Zeitkoordinate
∂_i^2	Zweite partielle Ableitung nach i
∇	$\begin{pmatrix}\partial_1 & \partial_2\end{pmatrix}^\mathrm{T}$ — Nabla-Operator
Δ	$\partial_1^2 + \partial_2^2$ — Laplace-Operator
$\mathrm{C}^k(\Omega)$	Menge der k-mal stetig differenzierbaren Funktion auf Ω
$\mathrm{L}^2(\Omega)$	Menge der quadrat-integrierbaren Funktionen auf Ω
$\mathrm{H}^1(\Omega)$	Sobolevraum
$f\vert_{\partial\Omega}$	Einschränkung der Funktion f auf $\partial\Omega$
$\langle x, y\rangle$	Euklidisches Skalarprodukt
$[a, b]$	$\{x \in \mathbb{R} \mid a \le x \le b\}$
(a, b)	$\{x \in \mathbb{R} \mid a < x < b\}$
$[a, b)$	$\{x \in \mathbb{R} \mid a \le x < b\}$
$u(\cdot, t)$	Funktion $\tilde{u}$ mit $\tilde{u}(x) = u(x, t)$
A^T	Transponierte der Matrix A
id	Identitätsabbildung
$a \coloneqq b$	a wird durch b definiert
$f \circ g$	Komposition der Funktionen f und g
$\begin{vmatrix} a & b \\ c & d \end{vmatrix}$	Determinante der angegeben Matrix
$\operatorname{span}\{\ldots\}$	Lineare Hülle der angegebenen Menge
$\lvert A\rvert$	Anzahl der Elemente in der Menge A

1 Introduction

For various mathematical and physical problems, there exists no feasible, deterministic algorithm to solve them. Especially, the simulation of physical systems with many coupled degrees of freedom, such as fluids, seem to be difficult to compute due to their high dimensionality. Instead, a class of randomized algorithms, called Monte Carlo methods, are used to approximate the actual outcome. Monte Carlo methods rely on repeated random sampling to obtain a numerical result. Hence, they are not bound to the curse of dimensionality and are able to evaluate complex equations quickly.

To obtain precise answers with a small relative error, Monte Carlo algorithms have to use a tremendous amount of random numbers. But the usage of truly random numbers generated by physical processes consists at least of two drawbacks. First, the output of the algorithm will be non-deterministic and, as a result, untestable. Second, the generation of truly random numbers is typically based on a slow process and consequently reduces the performance of the entire program. For that reason, Monte Carlo algorithms usually use so-called pseudorandom number generators. PRNGs generate a sequence of numbers based on a deterministic procedure and a truly random initial value as seed. The sequence of numbers is not truly random but fulfills several properties of truly random sequences.

The structure of Monte Carlo methods causes a program to spend most of its time with the construction of random numbers. Even the application of PRNGs does not change that. Today's computer processors provide functionality for the parallel execution of code in different ways, mainly SIMD and MIMD. Hence, to efficiently use the computing power of a CPU for Monte Carlo algorithms PRNGs have to be vectorized and parallelized to exploit such features. Whereas parallelization takes place at a high level, vectorization has to be done by the compiler or manually by the programmer at a much lower level. The implementation of PRNGs constraints automatic vectorization due to internal flow and data dependencies. To lift this restriction, a manual vectorization concerning data dependence and latencies appears to be the right way.

The C++ programming language is a perfect candidate for the development of vectorized PRNGs. It is one of the most used languages in the world and can be applied to small research projects as well as large enterprise programs. The language allows for the high-level abstraction of algorithms and structures. On the other hand, it is capable of accessing low-level routines to exploit special hardware features, like SSE, AVX, and threads. A typical C++ compiler is able to optimize the code with respect to such features automatically. But we as programmers are not bound to this and can manually optimize the code further. Every three years, a new standard is published, such as the new C++20 language specification. The language is evolving by its communities improvements and therefore it keeps to be a modern language. On top of this, other languages, such as Python, usually provide an interface to communicate with the C programming language. Through the design of an efficient implementation in C++, we can easily add support for other languages as well by providing a standard C interface.

Lots of PRNGs have been implemented by different libraries with different APIs. For example, STL, Boost, Intel MKL, RNGAVXLIB, Lemire, tinyrng,... STL, Boost and ... provide a large set of robust PRNGs which are not vectorized but well documented. Their API makes them likely to be used but shows many flaws. It does not allow to explicitly use the vectorization capabilities of a PRNG, gives you a bad default seeding and makes use of standard distributions difficult and not adjustable. Lemire and RNGAVXLIB provide open-source, vectorized implementations with bad documentation and difficult-to-use code. Intel MKL as well provides vectorized PRNGs but is not available open-source and uses difficult interfaces. There is not any easily-accessible, portable, open-source library which gives a coherent, easy-to-use and consistent interface for vectorized PRNGs.

In this thesis, we develop a new library, called pxart, in the C++ programming language. pxart vectorizes a handful of already known PRNGs which partly do not exist as vectorized versions and provides a new API for their usage to accommodate the disadvantages of the standard random library of the STL. The library itself is header-only, open-source, and can be found on GitHub. It is easily installable on every operating system. Additionally, we compare the performance of our vectorized PRNGs to other already accessible implementations in Boost, Intel MKL, Lemire, RNGAVXLIB and others. The performance is measured by speed, code size, memory size, complexity, and random properties. Meanwhile, we apply the implementations to an example Monte Carlo simulation. For this, a small test framework is implemented which allows us to easily test and evaluate PRNGs with respect to stated measures.

2 Background

To systematically approach the implementation of PRNGs, basic knowledge in the topics of stochastics and finite fields is administrable. Together, these topics will give a deeper understanding of randomness in deterministic computer systems, a formal description of pseudorandom sequences and generators, and the mathematical foundation of Monte Carlo algorithms. Based on them, we are capable of scientifically analyzing PRNGs concerning their randomness properties. Vectorization techniques can be conceptualized by the architecture of modern SIMD-capable multiprocessors and their instruction sets. Especially the knowledge of typical instructions will make the design of a new API and its application to Monte Carlo simulations clear. The following sections will give an overview of the named topics.

2.1 Mathematical Preliminaries

Probability Theory

The observation of random experiments resulted in the construction of probability theory. But probability theory itself does not use a further formalized concept of randomness (Schmidt 2009). In fact, it allows us to observe randomness without defining it (Volchan 2002). Hence, we will postpone an examination of truly random sequences to the next section.

According to Schmidt (2009), Kolmogorov embedded probability theory in the theory of measure and integration. Albeit it heavily relies on measure-theoretical structures, probability theory is one of the most important applications of measure and integration theory. Therefore we will assume basic knowledge in this topic and refer to Schmidt (2009) and Elstrodt (2018) for a more detailed introduction to measure spaces, measurable functions, and integrals. Propositions and theorems will be given without proof.

The underlying structure of probability theory, which connects it to measure theory, is the probability space. It is a measure space with a finite and normalized measure. This gives access to all the usual results of measure theory and furthermore unifies discrete and continuous distributions. (Schmidt 2009, p. 193 ff.)

Definition 2.1: (Probability Space)

A probability space is a measure space $(\Omega, \mathcal{F}, P)$ such that $P(\Omega) = 1$. In this case, we call P the probability measure, $\mathcal{F}$ the set of all events, and Ω the set of all possible outcomes of a random experiment.

Due to the complex definition of a measure space, it is convenient to not have to explicitly specify the probability space when analyzing random experiments. Instead, we use random variables which are essentially measurable functions on a probability space (Schmidt 2009, p. 194). For complicated cases, these will serve as observables for specific properties and will make the analysis much more intuitive.

DEFINITION 2.2: (Random Variable)

Let $(\Omega, \mathcal{F}, P)$ be a probability space and $(\Sigma, \mathcal{A})$ a measurable space. A measurable function $X\colon \Omega \to \Sigma$ is called a random variable.
In this case, we denote with $P_X := P \circ X^{-1}$ the distribution and with $(\Sigma, \mathcal{A}, P_X)$ the probability space of X. Two random variables are identically distributed if they have the same distribution. Additionally, we say that X is a real-valued random variable if $\Sigma = \mathbb{R}$ and $\mathcal{A} = \mathcal{B}(\mathbb{R})$.

From now on, if a random variable is defined then, if not stated otherwise, it is assumed there exists a proper probability space $(\Omega, \mathcal{F}, P)$ and measurable space $(\Sigma, \mathcal{A})$.

Another important concept of stochastics is known as independence. In Schmidt (2009) it is defined for a family of events, a family of sets of events, and a family of random variables. If we think of random variables as observables then their independence means that their outcomes do not influence each other. For our purposes, the general definition of all three forms of independence is distracting. In a computer, it makes no sense to talk about infinite sequences. Therefore the following definition of independence takes only a finite sequence of random variables into account. Furthermore, to make it more understandable, this definition uses a theorem from Schmidt (2009, p. 238) which characterizes the independence of random variables.

DEFINITION 2.3: (Independence)

Let $n \in \mathbb{N}$ and X_i be a random variable for all $i \in \mathbb{N}$ with $i \leq n$. We denote the respective random vector with $X := (X_i)_{i=1}^n$. Then these random variables are independent if the following equation holds.

$$P_X = \bigotimes_{i=1}^{n} P_{X_i}$$

Typical observations of random sequences include the estimation of the expectation value and the variance. Both of these values are needed for analyzing PRNGs and the development of Monte Carlo simulations (Landau and Binder 2014, p. 30 ff.). Due to their deep connection to the integral, both of these moments are defined for real-valued random variables. We give the usual definitions based on Schmidt (2009, p. 274 ff.) in a simplified form.

DEFINITION 2.4: (Expectation Value and Variance)

Let X be a real-valued random variable such that $\int_\Omega |X| \,\mathrm{d}P < \infty$. Then the expectation value $\mathbb{E}X$ and variance $\operatorname{var} X$ of X is defined in the following way.

$$\mathbb{E}X := \int_\Omega X(\omega)\,\mathrm{d}P(\omega)\ , \qquad \operatorname{var} X := \mathbb{E}\left(X - \mathbb{E}X\right)^2$$

To not rely on the underlying probability space directly, we want to be able to compute the expectation value through the respective distribution of the random variable. The theory of measure and integration gives the following proposition, also known as rule of substitution (Schmidt 2009, p. 276).

PROPOSITION 2.1: (Substitution)

Let X be real-valued random variable and $f\colon \mathbb{R} \to \mathbb{R}$ a measurable function such that $\int_\Omega |f| \,\mathrm{d}P_X < \infty$. Then the following equation holds.

$$\mathbb{E}(f \circ X) = \int_{\mathbb{R}} f(x) \,\mathrm{d}P_X(x)$$

In particular, if $\mathbb{E}\,|X| < \infty$ then the above equation can be reformulated as follows.

$$\mathbb{E}X = \int_{\mathbb{R}} x \,\mathrm{d}P_X(x)$$

The distribution of real-valued random variables is univariate and as a result can be described by so-called cumulative distribution functions (CDFs). The CDF intuitively characterizes the distribution and simplifies the analysis. Further, it can be proven that every CDF belongs to a univariate distribution. According to Schmidt (2009, p. 246), this is the theorem of correspondence. Sometimes it is even possible to define a probability density; a function that is the Lebesgue density of the respective distribution (Schmidt 2009, p. 255).

DEFINITION 2.5: (Probability Density and Cumulative Distribution Function)

Let X be a real-valued random variable. Then the respective cumulative distribution function is defined as follows.

$$F_X : \mathbb{R} \to [0,1] \;, \qquad F_X(x) := P_X((-\infty, x])$$

We call the function $p\colon \mathbb{R} \to [0,\infty)$ a probability density of X if for all $A \in \mathcal{B}(\mathbb{R})$

$$P_X(A) = \int_A p(x) \,\mathrm{d}\lambda(x) \;.$$

As well as CDFs, probability densities can greatly simplify computations which are based on absolute continuous random variables. The following proposition, obtained from Schmidt (2009), shows the simplified computation of an expectation value through a Lebesgue integral.

PROPOSITION 2.2: (Chaining)

Let X be a real-valued random variable with p as its probability density. If $f\colon \mathbb{R} \to$

$\mathbb{R}$ *is a measurable function such that* $\mathbb{E}\,|f \circ X| < \infty$ *then*

$$\mathbb{E}\,(f \circ X) = \int_{\mathbb{R}} f(x)p(x)\,\mathrm{d}\lambda(x)\ .$$

A last important theorem to name is the strong law of large numbers (SLLN). According to Graham and Talay (2013, p. 13), the principles of Monte Carlo methods are based on this theorem. Please note, there exist many more variations of this theorem. We will again use a simplified version from Graham and Talay (2013).

THEOREM 2.3: (Strong Law of Large Numbers)

Let $(X_n)_{n\in\mathbb{N}}$ *be a sequence of iid real-valued random variables with finite expectation value* μ*. Then the following equation holds* P*-almost everywhere.*

$$\lim_{n\to\infty} \frac{1}{n}\sum_{i=1}^{n} X_i = \mu$$

Finite Fields

2.2 Pseudorandom Number Generators

Random Sequences

In the above subsection 2.1 the theory of probability was introduced to make an examination of randomness possible. Randomness is a difficult concept and drives many philosophical discussions. According to Volchan (2002) and Kneusel (2018, pp. 10–11), humans have a bad intuition concerning the outcome of random experiments. But for our purposes, it would suffice to find a formal mathematical definition applicable to RNGs. However, such a formal concept, which is also widely accepted and unique, has not been found yet (Volchan 2002).

The first problem about randomness is the word itself. It is unclear and vague because there is no intentional application. To be more specific, we will observe randomness in form of random sequences of real numbers. But as stated in Volchan (2002) the question if a sequence is random decides at infinity. As long as we are only observing finite sequences, we cannot decide if such a sequence is the outcome of a truly random experiment or the result of a non-random algorithm. Following his explanation, Volchan makes clear that typical characterizations of a random sequence are closely associated with noncomputability. So even if we would be able to algorithmically produce an infinite amount of numbers, the resulting sequence could not be seen as truly random. Furthermore, the question if a sequence is random cannot be decided by an algorithm. Hence, the existing formal concepts for truly random sequences are not applicable to computer systems. Instead, Volchan proposed a more pragmatic principle: “if it acts randomly, it is random” (Volchan 2002) — the use of pseudorandom sequences.

A computer is only capable of using finite sequences of values and for the development of RNGs, it is enough to measure and compare different properties of truly random sequences to a sequence of real numbers. For this, we rely on probability theory and first define an abstract random sequence drawn from a random experiment. The definition will use realizations of random variables to model the samples of a random experiment. We make sure that these variables are identically and independent distributed (iid). This makes analyzing other sequences simpler and imposes no boundary because every important distribution can be generated out of iid random variables (Kneusel 2018, pp. 81–111).

DEFINITION 2.6: (Random Sequence)

Let I be a countable index set and $(X_n)_{n\in I}$ be a sequence of iid real-valued random variables. Then a realization of $(X_n)_{n\in I}$ is called a random sequence.

Generating a truly random sequence in a deterministic computer system is impossible. An RNG which is able to generate such a sequence is called a true random number generator (TRNG) and is typically implemented as a device drawing random samples from an essentially non-deterministic physical process, like temperature fluctuations (Intel 2018).

Pseudorandom Sequences

The given abstract definition of a random sequence in terms of probability theory helps to assess the randomness properties of a given sequence produced by a computer. Typically, a computer-generated sequence which fulfills various conditions about randomness will be called a pseudorandom sequence. The respective structure and algorithm which produced the sequence is then called a PRNG.

For computer programming and simulations, the usage of a TRNG would introduce severe disadvantages in contrast to a PRNG. Concerning program verification, debugging, and the comparison of similar systems, the reproducibility of results is essential (L'Ecuyer 2015). A truly random sequence produced by physical devices, such as thermal noise diodes or photon trajectory detectors, is not reproducible and can therefore not be conveniently used for mathematical and physical simulations (L'Ecuyer 2015). According to L'Ecuyer (2015), a given simulation should produce the same results on different architectures for every run. This property becomes even more important if parallel generation of random numbers with multiple streams is taken into account. Additionally, considering the performance of random number generation PRNGs tend to be much faster than TRNGs (Intel 2018). Thus, especially for Monte Carlo methods, PRNGs are a key resource for computer-generated random numbers (Bauke and Mertens 2007).

For a detailed discussion about its mathematical properties, design, and implementation, the concept of a PRNG has to be formalized. In this thesis, we use the following slightly modified variation of L'Ecuyer's definition (Barash, Guskova, and Shchur 2017; Bauke and Mertens 2007; L'Ecuyer 1994, 2015). It assumes a finite set of states and a transition function

$$\begin{tikzcd} s_0 \arrow[r, "T"] & s_1 \arrow[r, "T"] \arrow[d, "G"'] & s_2 \arrow[r, "T"] \arrow[d] & \dots \\ & u_1 & u_2 & \dots \end{tikzcd}$$

Figure 1: The figure shows a scheme about the generation of a pseudorandom sequence for a given PRNG $\mathcal{G} := (S, T, U, G)$ and seed value $s_0 \in S$. The internal state is advanced by the transition function T through a recurrence relation. To get an output value for the pseudorandom sequence the generator function G is used.

which advances the current state of the PRNG by a recurrence relation. For the output, a finite set of output symbols and a generator function which maps states to output symbols is chosen. As of Bauke and Mertens (2007), almost all PRNGs produce a sequence of numbers by a recurrence. Hence, the given formalization is widely accepted and builds the basis for further discussions about pseudorandom numbers (Barash, Guskova, and Shchur 2017; Bauke and Mertens 2007; L'Ecuyer 1994, 2015).

Definition 2.7: (Pseudorandom Number Generator)

Let $\mathcal{G} := (S, T, U, G)$ be a tuple consisting of a non-empty, finite set of states S, a transition function $T\colon S \to S$, a non-empty, finite set of output symbols U and an output function $G\colon S \to U$. In this case $\mathcal{G}$ is called a PRNG.

Given a PRNG and a seed value as an initial state, producing a sequence of pseudorandom numbers can be done by periodically applying the transition function on the current state and then extracting the output through the generator function (Barash, Guskova, and Shchur 2017; L'Ecuyer 1994, 2015). Here, we will use this method as the generalization of a pseudorandom sequence. Figure 1 shows this process schematically.

Definition 2.8: (Pseudorandom Sequence of PRNG)

Let $\mathcal{G} := (S, T, U, G)$ be a PRNG and $s_0 \in S$ be the initial state, also called the seed value. The respective sequence of states $(s_n)_{n\in\mathbb{N}}$ in S is given by the following equation for all $n \in \mathbb{N}$.

$$s_{n+1} := T(s_n)$$

The sequence $(u_n)_{n\in\mathbb{N}}$ in U given by the following expression for all $n \in \mathbb{N}$ is then called the respective pseudorandom sequence of $\mathcal{G}$ with seed s_0.

$$u_n := G(s_n)$$

Using a TRNG in a computer system is like consulting an oracle. We are calling a function with no arguments which returns a different value for every call. Let $(u_n)_{n\in\mathbb{N}}$ be the respective pseudorandom sequence of a PRNG $\mathcal{G}$ with a given seed. Then in a computer $\mathcal{G}$ can be interpreted as a function with no parameters which produces the pseudorandom sequence $(u_n)_{n\in\mathbb{N}}$ in the following way. Hereby, we understand $\leftarrow$ as the assignment operator that assigns a value given on the right-hand side to the variable given on the left-hand side.

$$u_1 \leftarrow \mathcal{G}() \,, \qquad u_2 \leftarrow \mathcal{G}() \,, \qquad u_3 \leftarrow \mathcal{G}() \,, \qquad \ldots$$

A PRNG has to artificially model this behavior by an internal state. Every function call must change this state according to the transition function. Consequently, if a PRNG should be used as an oracle in that sense, the set of states and the transition function in its definition are obligatory.

It will be shown that the number of different states a PRNG can reach greatly affects the randomness of a respective pseudorandom sequence. A larger set of states is not a guarantee that the output of a PRNG will look more like a truly random sequence, but at least gives the opportunity to better mask its deterministic nature (O'Neill 2014). Therefore the number of states in general is much bigger than the number of different outputs. Through the usage of output symbols together with a generator function a PRNG can take advantage of a large set of states while returning only a few different values. This idea has two important implications. A generator function which shrinks the set of states to a smaller space of output symbols makes the PRNG less predictable and more secure (O'Neill 2014). The generator function would not be bijective and as a result we as consumers would not be able to draw conclusions about the current state of the PRNG based on its given output. Both properties are highly appreciated because they mimic the behavior of TRNGs. Hence, the set of output symbols and the generator in the definition of PRNGs is as important as the set of states and the transition function.

The following paragraphs will provide examples of an application of this definition to deepen the understanding.

In the above definition, the initial state was deterministically given. As discussed, this behavior is intended to be able to reproduce the complete pseudorandom sequence based on its initial state. But to extend this process with true randomness, in general the seed will be chosen to be a truly random number generated by some physical device. L'Ecuyer (1994) states that generating such a seed is much less work and more reasonable than generating a long sequence of truly random values. A generator with a truly random seed can be seen as an extensor of randomness.

Linear Congruential Generator

Definition 2.9: (Linear Congruential Generator)

Let $m \in \mathbb{N}$ with $m \geq 2$ and $a, c \in \mathbb{Z}_m$. We define the PRNG $\mathrm{LCG}(m, a, c) :=$

(S, T, U, G)

$$S \coloneqq U \coloneqq \mathbb{Z}_m \ , \qquad G \coloneqq \mathrm{id}_{\mathbb{Z}_m}$$

$$T\colon S \to S \ , \qquad T(x) \coloneqq ax + c$$

Multiplication and addition are understood in the sense of $\mathbb{Z}_m$. We call $\mathrm{LCG}(m, a, c)$ the linear congruential generator with modulus m, multiplier a and increment c.

Mersenne Twister

Definition 2.10: (Mersenne Twister)

Let $w, n, m \in \mathbb{N}$ and $r \in \mathbb{N}_0$ with $m \leq n$ and $r < w$. Further, let $a, b, c \in \mathbb{Z}_2^w$ and $u, s, t, l \in \mathbb{Z}_w$. Then the Mersenne Twister $\mathrm{MT}(w, n, m, r, a, b, c, u, s, t, l) \coloneqq (S, T, U, G)$ is defined as a PRNG in the following way.

$$S \coloneqq \mathbb{Z}_n \times \mathbb{Z}_2^{w\times n} \ , \qquad U \coloneqq \mathbb{Z}_2^w$$

$$T\colon S \to S$$

$$\forall i \in \mathbb{Z}_{n-1} : T(i, x) \coloneqq (i + 1, x)$$

$$T(n - 1, x) \coloneqq (0, y)$$

$$\begin{aligned} \forall i \in \mathbb{Z}_{n-m} : y_i &\coloneqq x_{m+i} \oplus \left(x_i^u \middle| x_{i+1}^l\right) A \\ \forall i \in \mathbb{Z}_{m-1} + (n-m) : y_i &\coloneqq y_{i-(n-m)} \oplus \left(x_i^u \middle| x_{i+1}^l\right) A \\ y_{n-1} &\coloneqq y_{m-1} \oplus \left(x_{n-1}^u \middle| y_0^l\right) A \end{aligned}$$

$$xA \coloneqq \begin{cases} x \gg 1 & : x_0 = 0 \\ (x \gg 1) \oplus a & : x_0 = 1 \end{cases}$$

$$f_1(x) \coloneqq x \oplus (x \gg u)$$

$$f_2(x) \coloneqq x \oplus ((x \ll s) \odot b)$$

$$f_3(x) \coloneqq x \oplus ((x \ll t) \odot c)$$

$$f_4(x) \coloneqq x \oplus (x \gg l)$$

$$G\colon S \to U \ , \qquad G(i, x) \coloneqq f_4 \circ f_3 \circ f_2 \circ f_1(x_i)$$

Permuted Congruential Generator

DEFINITION 2.11: (Permuted Congruential Generator)

Given $\mathcal{G} \coloneqq \mathrm{LCG}(b,a,c)$ *with transition function* T*. Let* $t \in \mathbb{Z}_b$ *and* $f_c \colon \mathbb{Z}_{2^{b-t}} \to \mathbb{Z}_{2^{b-t}}$ *be a permutation for all* $c \in \mathbb{Z}_{2^t}$*.*

$$S \coloneqq \mathbb{Z}_{2^b} \ , \qquad U \coloneqq \mathbb{Z}_{2^{b-t}}$$

$$G \coloneqq \pi_2 \circ f_* \circ \mathrm{split}_t$$

$$f_*(a,b) \coloneqq (a, f_a(b))$$

$$\mathrm{PCG}(\mathcal{G}, t, \{f_c \mid c \in \mathbb{Z}_{2^t}\}) \coloneqq (S,T,U,G)$$

Xorshift and Variants

DEFINITION 2.12: (Xoroshiro128+)

Let $a,b,c \in \mathbb{Z}_{64}$*.*

$$S \coloneqq \mathbb{Z}_{2^{64}}^2 \ , \qquad U \coloneqq \mathbb{Z}_{2^{64}}$$

$$T(x,y) \coloneqq (x \circlearrowleft a \oplus f(x,y) \oplus (f(x,y) \leftarrow b), f(x,y) \circlearrowleft c)$$

$$f(x,y) \coloneqq x \oplus y$$

$$G(x,y) \coloneqq x + y$$

Middle Square Weyl Sequence RNG

DEFINITION 2.13: (Middle Square Weyl Sequence RNG)

Let $s \in \mathbb{Z}_{2^{64}}$ *be an odd constant. The middle square Weyl sequence RNG* $\mathrm{MSWS}(s) \coloneqq (S,T,U,G)$ *is defined as a PRNG in the following way.*

$$S \coloneqq \mathbb{Z}_{2^{64}}^2 \ , \qquad U \coloneqq \mathbb{Z}_{2^{32}}$$

$$T \colon S \to S \ , \qquad T(w,x) = (w+s, f(x^2+w+s))$$

$$f \colon \mathbb{Z}_{2^{64}} \to \mathbb{Z}_{2^{64}} \ , \qquad f(x) \coloneqq (x \gg 32)\mathrm{or}(x \ll 32)$$

$$G \colon S \to U \ , \qquad G(w,x) \coloneqq x \mod 2^{32}$$

2.3 Simulation in Physics and Mathematics

Mathematical and Physical Preliminaries

Baseline Model Problems

2.4 SIMD-Capable Processors

Architecture of Modern Central Processing Units

SIMD Instruction Sets and Efficiency

SSE, AVX, AVX512

2.5 Summary

Kneusel (2018) and Volchan (2002) (Volchan 2002, p. 1; Kneusel 2018, p. 2)

3 Previous Work

3.1 The C++ API and Further Progressions

3.2 Techniques for Vectorization and Parallelization

3.3 Summary

The topic of PRNGs consists of several smaller parts. From a mathematical point of view, one has to talk about their definition and construction as well as methods on how to test their randomness. There have been a lot of publications concerning these issues. Hence, I am not able to give you a detailed overview. Instead, I will focus on the most relevant PRNGs and test suites, as well as some modern examples.

The creation of new PRNGs is sometimes understood to be black magic and can be hard since basically, one has to build a deterministic algorithm with a nearly non-deterministic output. In Kneusel (2018) one can find numerous different families of PRNGs. The most well-known ones are Linear Congruential Generators, Mersenne Twisters and Xorshift with its Variants. Whereas LCGs tend to be fast but weak generators in O'Neill (2014), one can find a further developed promising family of algorithms, called PCGs. Widynski (2019) describes another RNG based on the so-called middle square Weyl sequence. All of these generators have certain advantages and disadvantages in different areas such as security, games, and simulations.

After building a PRNG, one has to check if the generated sequence of random numbers fulfills certain properties. In general, these properties will somehow measure the randomness of our RNG. Typically, there are a lot of tests bundled inside a test suite such as TestU01 and Dieharder.

4 Design of the API

What do we want from the interface of our RNG? It should make testing with given frameworks like TestU01, dieharder, ent and PractRand easy. Benchmarking should be possible as well. Therefore we need a good API and a good application interface. Most of the time we want to generate uniform distributed real or integer numbers. We need two helper functions. So we see that the concept of a distribution makes things complicated. We cannot specialize distributions for certain RNGs. We cannot use lambda expressions as distributions. Therefore we want to use only helper functions as distributions and not member functions. So we do not have to specify a specialization and instead use the given standard but we are able to do it. Therefore functors and old-distributions are distributions as well and hence we are compatible to the standard.

Additionally, we have to be more specific about the concept of a random number engine. The output of a random number engine of the current concept is magical unsigned integer which should be uniformly distributed in the interval [min,max]. But these magic numbers can result in certain problems if used the wrong way, see Melissa O'Neill Seeding Surprises. Therefore the general idea is to always use the helper functions as new distributions which define min and max explicitly and make sure you really get those values. This is also a good idea for the standard. And it is compatible with the current standard.

Now think of vector registers and multiprocessors. The random number engine should provide ways to fill a range with random numbers such that it can perform generation more efficiently. Think about the execution policies in C++17. They should be provided as well.

5 Testing Framework

6 Implementation of Vectorized PRNGs

6.1 Linear Congruential Generatiors

6.2 Mersenne Twister

6.3 Permuted Congruential Generators

6.4 Xoroshiro

6.5 Middle Square Weyl Generator

6.6 Summary

7 Application to Simulations

8 Evaluation and Results

godbolt google benchmark intel vtune amplifier testu01 dieharder

9 Conclusions

References

Barash, L. Yu., Maria S. Guskova, and Lev. N. Shchur (2017). “Employing AVX Vectorization to Improve the Performance of Random Number Generators”. In: *Programming and Computer Software* 43.3, pp. 145–160. DOI: 10.1134/S0361768817030033.

Bauke, Heiko and Stephan Mertens (2007). “Random Numbers for Large-Scale Distributed Monte Carlo Simulations”. In: *Physical Review E* 75.6, p. 066701. DOI: 10.1103/PhysRevE.75.066701.

Elstrodt, Jürgen (2018). *Maß- und Integrationstheorie*. Achte Auflage. Springer Spektrum. ISBN: 978-3-662-57938-1. DOI: 10.1007/978-3-662-57939-8.

Graham, Carl and Denis Talay (2013). *Stochastic Simulation and Monte Carlo Methods. Mathematical Foundations of Stochastic Simulation*. Springer. ISBN: 978-3-642-39362-4. DOI: 10.1007/978-3-642-39363-1.

Intel (2018). *Intel Digital Random Number Generator (DRNG) Software Implementation Guide*. Revision 2.1. URL: https://software.intel.com/en-us/articles/intel-digital-random-number-generator-drng-software-implementation-guide (visited on 09/17/2019).

Kneusel, Ronald T. (2018). *Random Numbers and Computers*. Springer. ISBN: 978-3-319-77697-2. DOI: 10.1007/978-3-319-77697-2.

Landau, David P. and Kurt Binder (2014). *A Guide to Monte Carlo Simulations in Statistical Physics*. Fourth Edition. Cambridge University Press – University of Cambridge. ISBN: 978-1-107-07402-6. DOI: 10.1017/CBO9781139696463.

L’Ecuyer, Pierre (December 1994). “Uniform Random Number Generation”. In: *Annals of Operations Research* 53, pp. 77–120. DOI: 10.1007/BF02136827.

— (2015). “Random Number Generation with Multiple Streams for Sequential and Parallel Computing”. In: *2015 Winter Simulation Conference (WSC)*. IEEE, pp. 31–44. DOI: 10.1109/WSC.2015.7408151.

O’Neill, Melissa E. (2014). *PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation*. Tech. rep. HMC-CS-2014-0905. Claremont, CA: Harvey Mudd College. URL: https://www.cs.hmc.edu/tr/hmc-cs-2014-0905.pdf (visited on 08/28/2019).

Schmidt, Klaus D. (2009). *Maß und Wahrscheinlichkeit*. Springer. ISBN: 978-3-540-89729-3. DOI: 10.1007/978-3-540-89730-9.

Volchan, Sérgio B. (2002). “What is a Random Sequence?” In: *The American Mathematical Monthly* 109.1, pp. 46–63. DOI: 10.2307/2695767.

Widynski, Bernard (July 31, 2019). “Middle Square Weyl Sequence RNG”. In: *arXiv.org*. URL: https://arxiv.org/abs/1704.00358v4 (visited on 08/28/2019).

Statutory Declaration

I declare that I have developed and written the enclosed Master's thesis completely by myself, and have not used sources or means without declaration in the text. Any thoughts from others or literal quotations are clearly marked. The Master's thesis was not used in the same or in a similar version to achieve an academic grading or is being published elsewhere.
On the part of the author, there are no objections to the provision of this Master's thesis for public use.

Jena, September 18, 2019

Markus Pawellek