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, November 30, 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 Definitions and Theorems

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
$[a, b]$	$\{x \in \mathbb{R} \mid a \leq x \leq b\}$
(a, b)	$\{x \in \mathbb{R} \mid a < x < b\}$
$[a, b)$	$\{x \in \mathbb{R} \mid a \leq x < b\}$
$u(\cdot, t)$	Funktion $\tilde{u}$ mit $\tilde{u}(x) = u(x, t)$
A^{T}	Transponierte der Matrix A
id	Identitätsabbildung
$a := b$	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
$\vert A\vert$	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 Mathematical Preliminaries

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.

2.1 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.

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

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.

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

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.

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

2.2 Number Theory and Finite Fields

3 Physical Simulations and Monte Carlo Methods

For our purposes, it is enough to explain the application of PRNGs to some given simulation procedures because there is no generic approach on how to randomize an arbitrary physical problem. Hence, we will not provide an excessive explanation on the theory of Monte Carlo methods and their application to general physical problems. Instead, the focus lies on the understanding of basic concepts and their implementation with respect to well-chosen examples.

As mentioned in the introduction, the simulation of physical and mathematical systems can be quite time intensive. Many degrees of freedom in a resulting partial differential equation makes the problem infeasible to solve deterministically (Landau and Binder 2014). This is typically called the "curse of dimensionality" (Müller-Gronbach, Novak, and Ritter 2012). As a consequence, we rely on probability theory to estimate the respective solutions and speed-up the simulation. Such randomized algorithms are in general called Monte Carlo methods (Landau and Binder 2014; Müller-Gronbach, Novak, and Ritter 2012).

DEFINITION 3.1: (Monte Carlo Method)

A Monte Carlo method is a random variable that computes its result based on given random variables according to an algorithm. We call the realization of a Monte Carlo method a run or its execution.

We will not give a rigorous definition of an algorithm but refer to Hromkovič (2011) for detailed information. With this definition, the output of an execution of a Monte Carlo method is interpreted as a realization of random variables. In contrast to a deterministic algorithm, calling a Monte Carlo method twice with identical input arguments will not necessarily produce the same output again. This behavior lets them overcome the curse of dimensionality and as a result they represent an efficient family of generalized algorithms to solve high-dimensional problems.

To get the idea behind Monte Carlo methods, the observation of direct simulations as given in Müller-Gronbach, Novak, and Ritter (2012) will serve perfectly. For some dimension $d \in \mathbb{R}$, we want to approximate a value $r \in \mathbb{R}^d$ by a Monte Carlo method. Direct simulation needs an already existent sequence of iid random variables with their expectation value equal to r which we interpret as random samples. But this does not impose strong restrictions because we are mostly able to find such random variables.

LEMMA 3.1: (Direct Simulation)

Let $d \in \mathbb{N}$, $r \in \mathbb{R}^d$ and $(X_n)_{n\in\mathbb{N}}$ a sequence of $\mathbb{R}^d$-valued iid random variables in $\mathrm{L}^2\left(\mathbb{R}^d, \lambda\right)$ with $\mathbb{E}\, X_n = r$ for all $n \in \mathbb{N}$. In this case, construct the following random variable for all $n \in \mathbb{N}$.

$$D_n := \frac{1}{n} \sum_{k=1}^{n} X_k$$

Then for arbitrary sample counts $n \in \mathbb{N}$ the random variable D_n is a Monte Carlo method which fulfills the following equations.

$$\mathbb{E}\, D_n = r \ , \qquad \sigma(D_n) = \frac{\sigma(X_1)}{\sqrt{n}} \ , \qquad \lim_{n\to\infty} \sigma(D_n) = 0$$

Furthermore, the following limit holds almost everywhere.

$$\lim_{n\to\infty} D_n = r$$

Again, we will give no proof of this lemma and instead refer to Müller-Gronbach, Novak, and Ritter (2012). Please note that the last limit follows from Theorem 2.3 the SLLN. The expectation value of the given method is always the result that we wanted to compute. This is not a special property. But looking at the standard deviation, the error of the direct simulation becomes smaller for a bigger sample count. Using a large number of samples will therefore estimate the actual result much more precisely. Additionally, the error is decreasing with $\frac{1}{\sqrt{n}}$. Hence, the error rate is independent of the given dimension which explains the overcoming of the curse of dimensionality.

3.1 Monte Carlo Integration and the Computation of π

Many simulations involve the calculation of multidimensional integrals. As a consequence, the so-called Monte Carlo integration forms the natural application of the direct simulation. We want to estimate the integral of a function. For given uniformly distributed random variables, we will construct a sequence of random variables such that their expectation value will coincide with the integral.

DEFINITION 3.2: (Monte Carlo Integration)

Let $d \in \mathbb{N}$ be the dimension, $U \subset \mathbb{R}^d$ be a measurable and bounded subset, such that $0 < \lambda(U) < \infty$, and $f \in \mathrm{L}^2(U, \lambda)$ the function to be integrated. Furthermore, let $(X_n)_{n\in\mathbb{N}}$ be a sequence of iid, U-valued, and uniformly distributed random variables. Then the Monte Carlo integration of f with n samples on the domain U is given by the following expression.

$$\mathrm{MCI}_n(f) := \frac{\lambda(U)}{n} \sum_{k=1}^{n} f \circ X_k$$

The domain of definition has to be restricted so that the method has a chance of reducing the overall estimation error. Additionally, the function f should be square-integrable such that we are able to get an upper bound on the standard deviation. The following lemma will show that Monte Carlo integration is indeed a Monte Carlo method with the properties of a direct simulation.

LEMMA 3.2: (Monte Carlo Integration Estimates Value of Integral)

Choose the same setting as in the above definition 3.2. In this case for all $n \in \mathbb{N}$, the Monte Carlo integration $\mathrm{MCI}_n(f)$ is a Monte Carlo method and the following statements for the expectation value and standard deviation are fulfilled.

$$\mathbb{E}\,\mathrm{MCI}_n(f) = \int_U f \,\mathrm{d}\lambda \ , \qquad \sigma\left[\mathrm{MCI}_n(f)\right] \leq \sqrt{\frac{\lambda(U)}{n} \int_U f^2 \,\mathrm{d}\lambda}$$

PROOF:
Let p be the probability density of X_n. Because the random variables are uniformly distributed on U, we can express it as follows.

$$p\colon U \to [0, \infty) \ , \qquad p(x) := \frac{1}{\lambda(U)}$$

By using substitution and chaining from propositions 2.1 and 2.2, the expectation value can be directly computed.

$$\begin{aligned}
\mathbb{E}\,\mathrm{MCI}_n(f) &= \mathbb{E}\left[\frac{\lambda(U)}{n} \sum_{k=1}^{n} f \circ X_k\right] = \frac{\lambda(U)}{n} \sum_{k=1}^{n} \mathbb{E}(f \circ X_k) \\
&= \lambda(U) \int_U f(x) p(x) \,\mathrm{d}\lambda(x) = \int_U f \,\mathrm{d}\lambda
\end{aligned}$$

For the standard deviation, first the variance will be observed. Since the sequence of random variables is stochastically independent, the sum can be taken out of the argument. Afterwards, we again apply substitution and chaining.

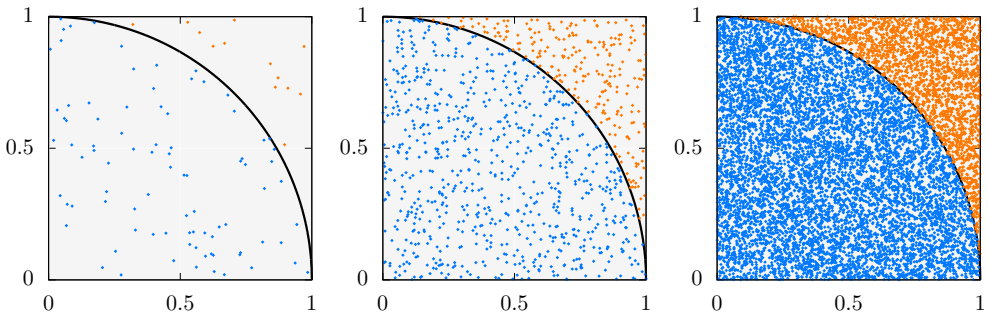
$$\begin{aligned}
\operatorname{var}\mathrm{MCI}_n(f) &= \operatorname{var}\left[\frac{\lambda(U)}{n} \sum_{k=1}^{n} f \circ X_k\right] = \frac{\lambda(U)^2}{n^2} \sum_{k=1}^{n} \operatorname{var}\left(f \circ X_k\right) \\
&= \frac{\lambda(U)^2}{n^2} \sum_{k=1}^{n} \mathbb{E}\left(f \circ X_k\right)^2 - \left[\mathbb{E}\left(f \circ X_k\right)\right]^2 \\
&\leq \frac{\lambda(U)^2}{n^2} \sum_{k=1}^{n} \mathbb{E}\left(f \circ X_k\right)^2 = \frac{\lambda(U)^2}{n} \int_U f^2(x) p(x) \,\mathrm{d}\lambda(x) \\
&= \frac{\lambda(U)}{n} \int_U f^2 \,\mathrm{d}\lambda
\end{aligned}$$

The inequality is now inferred by the definition of the standard deviation which proofs the lemma.

$$\sigma\left[\mathrm{MCI}_n(f)\right] = \sqrt{\operatorname{var}\mathrm{MCI}_n(f)} \leq \sqrt{\frac{\lambda(U)}{n} \int_U f^2 \,\mathrm{d}\lambda}$$

□

In the proof, we basically applied the lemma about the direct simulation. So we get the same convergence rate for the expectation value with respect to the standard deviation. To get a deeper understanding of this method, consider the estimation of π by Monte Carlo integration. For this, we would like to compute the area of a quarter of a circle which is strongly related to π.



(a) $n = 100$, $s = 87$, $\mu = 3.48$, $\varepsilon \approx 1.07 \cdot 10^{-1}$

(b) $n = 1000$, $s = 765$, $\mu = 3.06$, $\varepsilon \approx 2.60 \cdot 10^{-2}$

(c) $n = 10000$, $s = 7856$, $\mu = 3.1424$, $\varepsilon \approx 2.57 \cdot 10^{-4}$

Figure 1: The figures show the sample points for different realizations of the Monte Carlo integration $\mathrm{MCI}_n(f)$ for the computation of π. Points that lie in the quarter of the unit circle are shown in a blue color whereas other points are shown in orange color. The unit circle is represented by a black line. The result of the realizations is expressed by the sample mean μ and the relative error with respect to π is expressed by ε. The variable s names the count of samples that lie in the unit circle.

Figure 1 shows the execution of the following process for different realizations. Computing the area of a subset is in general done by integration. Therefore we choose $d = 2$ and $U := [0,1]^2$ with $\lambda(U) = 1$. Thus, the random variable X_n will be uniformly distributed on $[0,1]^2$ for all $n \in \mathbb{N}$. The last part consists of the construction of the function f. First, define the set which characterizes the quarter of the unit circle.

$$G := \left\{ x \in [0,1]^2 \;\middle|\; \|x\| \leq 1 \right\} , \qquad \lambda(G) = \frac{\pi}{4}$$

Based on this set, the function f can be expressed through the use of the characteristic function of G and by scaling its value.

$$f := 4 \cdot \mathbb{1}_G , \qquad \int_U f \,\mathrm{d}\lambda = 4 \int_{[0,1]^2} \mathbb{1}_G \,\mathrm{d}\lambda = 4 \cdot \lambda(G) = \pi$$

Simulating the integral of f will therefore give us an estimation of π. For a more detailed analysis, we will also compute the exact standard deviation of this Monte Carlo integration by using the above lemma 3.2.

$$\int_U f^2 \,\mathrm{d}\lambda = 16 \int_{[0,1]^2} \mathbb{1}_G \,\mathrm{d}\lambda = 16 \cdot \lambda(G) = 4\pi$$

$$\begin{aligned} \mathrm{var}\left(f \circ X_1\right) &= \mathbb{E}\left(f \circ X_1\right)^2 - \left[\mathbb{E}\left(f \circ X_1\right)\right]^2 = \int_U f^2 \,\mathrm{d}\lambda - \left(\int_U f \,\mathrm{d}\lambda\right)^2 \\ &= 4\pi - \pi^2 = \pi(4 - \pi) \end{aligned}$$

$$\sigma\left(\mathrm{MCI}_n(f)\right) = \sqrt{\frac{\pi(4-\pi)}{n}} \leq 2\sqrt{\frac{\pi}{n}}$$

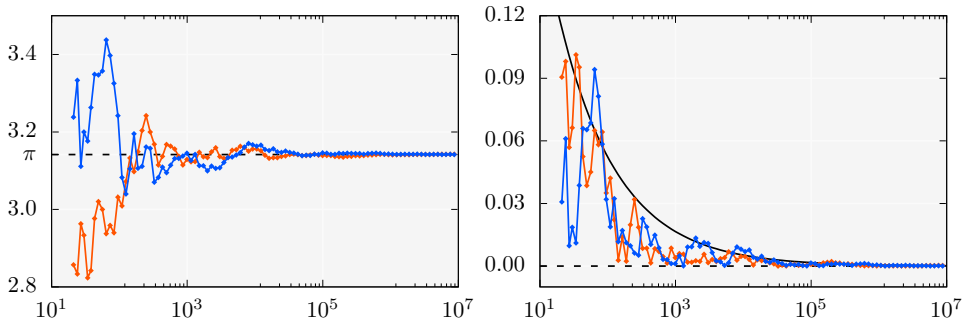


Figure 2: Each of the diagrams shows two different versions, colored in orange and blue, of realizations of the Monte Carlo integration $\mathrm{MCI}_n(f)$ for the computation of π for different values of n. The left one displays the estimated value of π and the right one displays its relative error with respect to π. Hereby the black line shows the exact relative standard deviation with respect to π of the Monte Carlo integration.

In figure 2, we can see this behavior for some actual simulations. By taking a larger amount of samples, the volume of the quarter of the unit circle becomes more occupied as can be seen in figure 1. As a consequence, the estimation of π will be more precise.

To use the described Monte Carlo integration for estimating π, an actual implementation in C++ is needed. The following code block provides a basic approach. It uses templates to generalize the usage of different RNGs and different real number types.

Code: monte_carlo_pi.hpp

```
#pragma once
#include <random>

namespace monte_carlo {

template <typename Real, typename Integer, typename RNG>
inline Real pi(RNG& rng, Integer samples) noexcept {
  std::uniform_real_distribution<Real> dist{0, 1};
  Integer samples_in_circle{};
  for (auto i = samples; i > 0; --i) {
    const auto x = dist(rng);
    const auto y = dist(rng);
    samples_in_circle += (x * x + y * y <= 1);
  }
  return static_cast<Real>(samples_in_circle) / samples * 4;
}

}  // namespace monte_carlo
```

The application of the given library function is shown in the following source file.

Code: monte_carlo_pi.cpp

```
#include "monte_carlo_pi.hpp"

#include <cmath>
#include <iostream>
#include <random>
#include <sstream>

template <typename Real>
constexpr Real pi = 3.14159265358979323846264 33;

using Real = double;
using namespace std;

int main(int argc, char** argv) {
  if (argc != 2) {
    cout << "usage: " << argv[0] << " <max sample count>\n";
    return -1;
  }

  stringstream input{argv[1]};
  int n;
  input >> n;

  random_device rng{};

  cout << "#samples\testimation\terror\trelative error\n";
  for (int i = 10; i <= n; i *= 10) {
    const auto monte_carlo_pi = monte_carlo::pi<Real>(rng, i);
    const auto error = abs(monte_carlo_pi - pi<Real>);
    const auto relative_error = error / pi<Real>;
    cout << i << "\t" << monte_carlo_pi << "\t" << error << "\t"
         << relative_error << "\n";
  }
}
```

4 SIMD-Capable Processors

According to Hennessy and Patterson (2019, pp. 10–11), in the year 1966, Flynn classified parallel architectures of computers with respect to their data-level and task-level parallelism. Based on this classification, a conventional uniprocessor has a single instruction stream and single data stream, also known as single instruction single data (SISD) architecture (Patterson and Hennessy 2014, pp. 509–510). The single instruction multiple data (SIMD) architecture exploits data-level parallelism by applying the same operations to multiple items of independent data at the same time (Hennessy and Patterson 2019) which, from the programmer's perspective, is close to the SISD mode of operation (Patterson and Hennessy 2014). In contrast to the multiple instruction multiple data (MIMD) architecture, SIMD only has to fetch one instruction to launch several data operations potentially reducing the power consumption. The application of SIMD ranges from matrix-oriented algorithms in scientific computing to media-oriented image and sound processing, as well as machine learning algorithms (Hennessy and Patterson 2019, pp. 10–11). Modern Intel processors typically provide SIMD utilities through special vector registers and a richer instruction set, like the Streaming SIMD Extensions (SSE) and the Advanced Vector Extensions (AVX) (Fog 2019a,b,c,d,e; *Intel Intrinsics Guide*). At the same time, MIMD utilities are implemented through multiple processor cores and multithreading. In a modern processor, SIMD and MIMD are orthogonal features of its design and can therefore be discussed independently. Hence, we will not focus on the exploitation of the MIMD architecture. To be able to design and implement vectorized algorithms for an SIMD architecture, we have to explain how data-level and instruction-level parallelism can be used to raise the performance of a computer program. Especially the knowledge of typical instructions will make the design of a new API and its application to Monte Carlo simulations clear. Therefore, we will briefly introduce the fundamentals of computer architecture and refer to Patterson and Hennessy (2014) and Hennessy and Patterson (2019) for a more detailed observation. In reality, there are several different SIMD-capable CPU architectures. Here, we will restrict our discussions to the SSE and AVX instruction set architectures from modern Intel processors, like the *Intel® Core™ i7-7700K Processor* and the *Intel® Core™ i5-8250U Processor* used to test implementations of described PRNGs (*Intel® Core™ i5-8250U Processor*; *Intel® Core™ i7-7700K Processor*).

4.1 Fundamentals of Computer Architecture

The Von Neumann architecture still describes the basic organization of a modern computer. Besides external mass storage, like hard disk drives (HDDs), and input/output (IO) mechanisms, the model consists of two main parts — the central processing unit (CPU), also called the processor, to execute instructions from a computer program, and the memory to store the respective data and instructions (Hennessy and Patterson 2019). Today, both, data and instructions, are encoded as binary numbers with fixed length which has proven to make the building and functioning of a computer much more efficient (Patterson and Hennessy 2014).

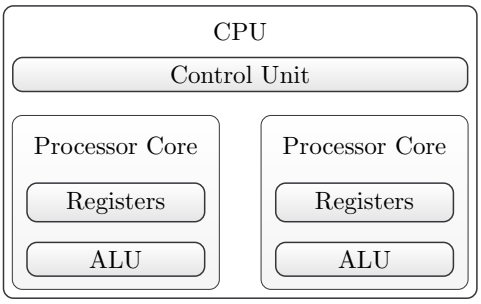


Figure 3: The figure shows the basic components of a typical CPU with multiple processing cores in an hierarchical order. There is only one control unit which is handling communication between different cores by coordinating the execution of program instructions. Every processor core employs its own registers and ALUs to provide an MIMD architecture.

The Processor

The processor in general consists of multiple arithmetic logic units (ALU) or execution units, a small number of registers and a control unit. The ALU performs arithmetic and logic operations and stores its results in registers. These registers also supply the operands for ALU operations. To fetch program instructions from memory and executing them, the control unit directs the coordinated operations of the ALU, registers and other components. Today, nearly every processor consists even of multiple processing cores each connected by a global control unit and containing its own registers and ALUs to provide an MIMD architecture. In figure 3, all the named components are shown schematically in a hierarchy to support the understanding. Here, we will focus on a single processing core.

The set of instructions a CPU is able to execute is called its architecture. Usually, processor architectures provide commands to move data between memory and registers and simple arithmetic and logic operations, like addition and multiplication of integral or floating-point numbers, to actually compute results of algorithms. The actual implementation of an instruction set in form of a processor circuit is called the microarchitecture. It defines how instructions are executed in reality by providing different execution units and modes of operation. Hence, with respect to the microarchitecture instructions suddenly exhibit physical properties, like the time to execute the given instruction, that were not considered by the abstract processor architecture. (Hennessy and Patterson 2019; Patterson and Hennessy 2014)

Executing an instruction in the processor is done in several stages. The number and kind of these stages depend on the type of the instruction and the underlying microarchitecture of the CPU. For example, an instruction first has to be fetched from memory. Afterwards, the bits of the instruction will be decoded and all referenced registers will be read. Finally, the ALU computes the actual operation and stores the result in the target register. Basically we can say, each stage is completed after one CPU cycle. The number of CPU cycles an instruction needs to be finished and provide its result to the next instruction is called its latency. The throughput of an instruction is measured in cycles per instruction and specifies the number of cycles an

A_1	A_2	A_3	A_4			
	B_1	B_2	B_3	B_4		
		C_1	C_2	C_3	C_4	
			D_1	D_2	D_3	D_4

Figure 4: This figure shows the functioning of a pipeline.

A_1	A_2	A_3	A_4			
B_1	B_2	B_3	B_4			
	C_1	C_2	C_3	C_4		
	D_1	D_2	D_3	D_4		
		E_1	E_2	E_3	E_4	
		F_1	F_2	F_3	F_4	
			G_1	G_2	G_3	G_4
			H_1	H_2	H_3	H_4

Figure 5: This figure shows the functioning of a pipeline.

instruction needs to reside in the execution unit.

To speed up the execution of independent instructions, in nearly all modern CPUs a so-called pipeline is used. Independent instructions do not have to wait for the results of other immediate instructions and therefore do not need to stall the execution unit for their complete latency. Instead, the processor is performing the different stages of different instructions concurrently according to their throughput. For a better understanding, figure 4 shows a schematic example of this pipeline process for four independent instructions with a latency of four and a throughput of one. As a consequence, a pipeline does not reduce the latency of an instruction but increases its throughput. It is therefore a form of instruction-level parallelism. To further decrease the throughput of instructions, the processor core typically uses multiple execution units to run independent instructions directly in parallel. This enhancement is shown in figure 5 with eight independent again with a latency of four and a throughput of one.

The theoretical performance of the CPU pipeline is reduced if it has to be stalled. These situations, also known as hazards, happen due to hardware resource conflicts, data dependencies and control instructions, like branches. To handle control instructions and especially branches much more efficiently, the CPU uses branch prediction. The processor tries to guess the outcome of the branch condition to keep the pipeline filled. Should the estimated value proven to be wrong the pipeline has to be stalled and cleared. This process is called a branch miss and introduces an execution time penalty. Hence, we will strive for branchless code or for easy-to-predict branches if we have to insert them.

For data-level parallelism, we want to focus on SIMD architectures. Intel CPUs establish this feature by using so-called vector registers of a fixed length. Vector registers contain

more than one value at the same time. For example, a $256\,\mathrm{bit}$ register can contain four $64\,\mathrm{bit}$ values or eight $32\,\mathrm{bit}$ values. One operation, like addition or multiplication, is then performed on all contained elements simultaneously. The choice which pattern to use is based on the trade-off between precision and throughput. If an application demands a high precision from the underlying floating-point operations, it will be more efficient to use four $64\,\mathrm{bit}$ double precision values reducing the throughput instead of eight single precision values.

The Memory

Memory can be described as a finite sequence of bits, whereby each bit anytime represents either the value 0 or 1. Eight bits are grouped into a byte and enumerated with a natural number starting from zero. These numbers are called memory addresses and make it possible to specify the location of variables in memory. This basic interpretation is visualized in figure 6. Fetching instructions from memory or transferring data between the CPU and memory, therefore requires the usage of those memory addresses to be able to reference data in the sequence of bytes. Each byte can be altered by program execution through storing instructions. (Patterson and Hennessy 2014)

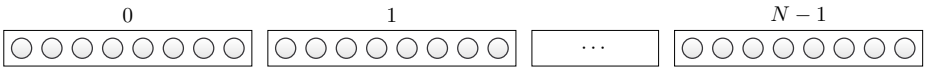


Figure 6: This figure visualizes memory with $N \in \mathbb{N}$ bytes as a sequences of bits where each byte can be referenced by its memory address.

Because physically there is no possibility to provide an unlimited amount of fast memory, computer designers found a more economical solution. In the majority of cases, faster memory means reduced storage capabilities and vice versa. Hence, memory is built to be a hierarchy of several levels — each smaller, faster, and more expensive per byte than the next lower level, which is farther from the processor. Interleaving levels are called caches and with caching we mean the process of loading data into the next cache level. If the processor wants to load some data from memory which cannot be found in the first level cache, data has to be fetched from a lower level in the hierarchy. This is called a cache miss. If on the other hand the data can be found in the cache, it can be directly used by the higher level cache or the processor itself. We call this a cache hit. A cache miss introduces a so-called miss penalty to the memory access time and should therefore be avoided to reduce the latency for fetching instructions and data. Figure 7 shows a schematic view of a usual memory hierarchy found in today's laptops and desktop computers. In modern processor architectures, like the Kaby Lake microarchitecture from Intel, each processing core of the CPU features its own level one cache which is further split into an instruction cache and a data cache (*7th Generation Intel Processor Families for S Platforms and Intel Core X-Series Processor Family*). This reduces the overall complexity of level one caches and as a result decreases the cache access time. (Hennessy and Patterson 2019, pp. 78–83)

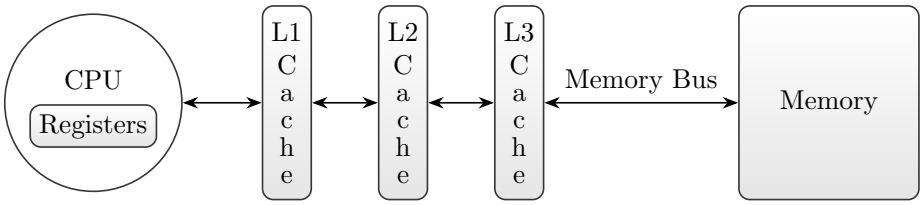


Figure 7: The figure shows a scheme of the non-persistent part of the memory hierarchy for a modern laptop or desktop computer. Modeled after Hennessy and Patterson (2019, p. 79).

Let $n := 2^k$ for some $k \in \mathbb{N}_0$ be a power of two. We say that a variable in memory is $n\,\mathrm{Byte}$ aligned if its starting address is divisible by n without remainder. Modern systems typically provide a $16\,\mathrm{Byte}$ alignment as default. SSE vector registers have a size of $128\,\mathrm{bit}$ and even demand that variables to be loaded from memory exhibit a $16\,\mathrm{Byte}$ alignment. The AVX architecture of Intel CPUs is working with $256\,\mathrm{bit}$ or $32\,\mathrm{Byte}$ long vector registers which do not have to be aligned but should provide a slightly improved performance otherwise. (Fog 2019c)

4.2 Usage in C++

To force the usage of the SSE or AVX instructions which exploit the SIMD capabilities on modern Intel processors, assembler code resulting from compiling a given program has to explicitly call the according instructions of the microarchitecture. To achieve this behavior in C++, there are several variants.

First, we could use inline assembly code to directly call the appropriate instructions. This was done in Barash, Guskova, and Shchur (2017) and Guskova, Barash, and Shchur (2016). But developing modules with inline assembly statements tends to be error-prone, complicated, unmaintainable and often results in code bloat.

The second variant uses the automatic vectorization of the compiler. Typically, this process should be preferred in contrast to manually optimizing the code by introducing SSE or AVX instructions to provide machine independent code. Due to the compiler's knowledge of the underlying hardware, automatic vectorization often generates code that is superior to other variants. But sometimes the complexity of problems exhibits several data and instruction dependencies by using non-trivial branches with different code paths or long chains of dependent calculations. In such cases, the compiler will not be able vectorize the code and we as programmers have to fall back to a manual alternative.

To get the best of both worlds, Intel provides so-called SIMD intrinsics for the SSE and AVX instruction sets. Theses intrinsics describe abstract functions in the C++ language working with data types representing vector registers and are not defined by the language itself. Each intrinsic is basically substituting an assembler instruction. With these utilities, it is possible to manually vectorize the code without the need to use inline assembly statements. Therefore, we are able to use high-level abstraction features of C++ to create a usable API

and make the code maintainable while inserting low-level routines to improve its performance. Usually, this approach is easier to understand, less error-prone and results in less code than inline assembly statements. Above all, taking care of alignment, latency and throughput of operations is made much simpler due to the abstraction of registers to variables. The latencies and throughputs of specific intrinsics for different microarchitectures is given by Intel (*Intel Intrinsics Guide*) and Fog (2019b). As a consequence, we rely on this variant to develop the vectorized implementations of PRNGs and algorithms.

The AVX intrinsic data types are given by `__m256`, `__m256d`, `__m256i` representing eight single precision floating-point values, four double precision floating-point values and a different amount of integer numbers with different sizes. The name of each intrinsic is based on the same pattern which first encodes the instruction set to use, followed by the code name of the operation, finished by identification of the storage pattern all separated by underscores. The SSE instruction set is encoded by `_mm` and the AVX instruction set by `_mm256`. For example, the name of the AVX intrinsic to add every element of two other vectors each containing eight single precision floating-point values is given by `_mm256_add_ps`.

5 Pseudorandom Number Generators

5.1 Random Sequences

In the above section 2 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. A modified version of this idea which is easier to understand is given in Kneusel (2018), where a sequence of values $(x_n)_{n\in\mathbb{N}}$ is truly random if there exists no algorithm such that for all $n \in \mathbb{N}$ the value x_{n+1} can be computed as a function of all x_i with $i \in \mathbb{N}$ and $i \leq n$. Put more simply, knowing finitely many elements of a truly random sequence does not enable us to predict the next values within a computer. 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 5.1: (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).

5.2 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 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 5.2: (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

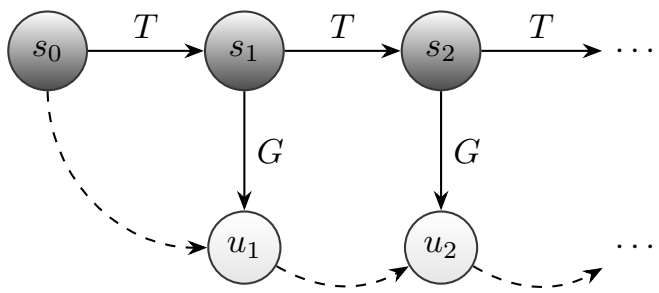


Figure 8: The figure shows a scheme about the generation of a pseudorandom sequence for a given PRNG $\mathcal{G} := (S, T, U, G)$ and seed value $s_0 \in S$. The internal state is advanced by the transition function T through a recurrence relation. To get an output value for the pseudorandom sequence the generator function G is used.

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 8 shows this process schematically.

DEFINITION 5.3: (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.

$$u_n := G(s_n)$$

In the definition we have used a recursive formulation. For theoretical discussions and the initialization of multiple streams of pseudorandom numbers an explicit variation seems to be more adequate. The following lemma will be given without a proof, but it can be shown by mathematical induction.

LEMMA 5.1: (Explicit Formulation of Pseudorandom Sequence)

Let $\mathcal{G} := (S, T, U, G)$ be a PRNG and $s_0 \in S$ its initial state. Then the respective pseudorandom sequence $(u_n)_{n\in\mathbb{N}}$ is given by the following formula for all $n \in \mathbb{N}$.

$$u_n = G \circ T^n(s_0)$$

5.3 Explanation of the Concept

Using a TRNG in a computer system is like consulting an oracle (Müller-Gronbach, Novak, and Ritter 2012). 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 function in the definition of PRNGs is as important as the set of states and the transition function.

In the majority of cases, the transition function T of a PRNG $\mathcal{G}$ should be injective (L'Ecuyer 1994, 2015; O'Neill 2014; Widynski 2019). Because we have a finite set of states this is equivalent to the proposition that T is a permutation and therefore bijective (Waldmann 2017, pp. 201–202). The property makes sure that every state is reached at a certain point in a sequence without introducing bias in the resulting distribution (O'Neill 2014). The generator function G cannot be a permutation but should not distort the distribution either. Hence, a uniform function which maps to every output value the same number of input values is a perfect candidate (O'Neill 2014).

5.4 Randomization

The goal of PRNGs is to imitate the properties of TRNGs as much as possible (L'Ecuyer 1994) and at the same time retaining executability by a computer system and reproducibility for a

given seed (L'Ecuyer 2015). These restrictions make a pseudorandom sequence completely predictable and characterizable by its seed. So up until now, we have not introduced any kind of randomness to the definition of a PRNG. But to extend the process of generating a pseudorandom sequence with true randomness, the seed will be chosen to be a truly random number produced by a TRNG. L'Ecuyer (1994) states that receiving such a seed is much less work and more reasonable than acquiring a long sequence of truly random values. A generator with a truly random seed can be seen as an extensor of randomness. Even today, Intel uses hardware-implemented PRNGs repeatedly seeded by a high-quality entropy source in their CPUs to provide a high-performance hardware module for producing random numbers with good statistical quality and protection against attacks (Intel 2018).

DEFINITION 5.4: (Randomized Pseudorandom Sequence)

Let $\mathcal{G} := (S, T, U, G)$ be a PRNG and X be an S-valued random variable with distribution P_X. Then the randomized pseudorandom sequence $(X_n)_{n \in \mathbb{N}}$ of $\mathcal{G}$ with respect to P_X is defined by the following expression for all $n \in \mathbb{N}$.

$$X_n := G \circ T^n \circ X$$

As with abstract random sequences, a truly random seed value is again modeled by a realization of the random variable X. As a result, the randomized pseudorandom sequence becomes a sequence of random variables which all depend on X. For the definition the explicit formulation in Lemma 5.1 was used. Typically, the distribution of seed values P_X is chosen so that it is uniformly distributed in a certain subset of S (Bauke and Mertens 2007; L'Ecuyer 1994, 2015; Matsumoto and Nishimura 1998; O'Neill 2014). This makes sure that no bias will be introduced by the randomization.

5.5 Distributions

5.6 Limitations and Mathematical Properties

As was already discussed, PRNGs have certain advantages in comparison with TRNGs. But they are also yielding essential and intrinsic limitations. From the previous subsubsection, it becomes clear that all the samples of a randomized pseudorandom sequence are not stochastically independent. In general, this means the output of a PRNG can consist of certain regular patterns or artifacts (L'Ecuyer 1994; O'Neill 2014). In L'Ecuyer (1994) these artifacts are also called the lattice structure. For applications that are using a large amount of random numbers, such patterns will introduce bias in the evaluated outputs. Hence, we will discuss a few mathematical properties a PRNG should fulfill to reduce the lattice structure as much as possible.

5.6.1 Periodicity

Since the set of states in a PRNG is finite, every respective pseudorandom sequence has to be periodic or ultimately periodic (Bauke and Mertens 2007; L'Ecuyer 1994). First, a rigorous definition of this concept should be given.

DEFINITION 5.5: (Periodic and Ultimately Periodic Sequences)

Let U be a non-empty set and $(u_n)_{n\in\mathbb{N}}$ be a sequence in U. Assume there exist $\rho, \tau \in \mathbb{N}$ such that for all $n \in \mathbb{N}_0$ the following holds.

$$u_{\tau+n+\rho} = u_{\tau+n}$$

Then (u_n) is called ultimately periodic. The smallest possible values for ρ and τ, such that the equation holds, are called period and transient respectively. In particular, if τ equals to 1 we call (u_n) periodic with period ρ.

This means an ultimately periodic sequence will be periodic after it reached its transient. Every periodic sequence is therefore ultimately periodic but not vice versa and as another consequence, the given concept is more general than the typical one of a periodic sequence. Please note that the values for ρ and τ are not unique. Let ρ^* be the period and τ^* be the transient. Then the equation given in the above definition holds for all values ρ and τ with respect to $m \in \mathbb{N}$ and $n \in \mathbb{N}_0$ in the following sense.

$$\rho = m\rho^* \,, \qquad \tau = \tau^* + n$$

Choosing the minimal values allows us to talk about a unique transient and a unique period. In the following lemma we show the application of the definition to pseudorandom sequences.

LEMMA 5.2: (Pseudorandom Sequences are Ultimately Periodic)

Let $\mathcal{G} := (S, T, U, G)$ be a PRNG and $s_0 \in S$ its initial state. Then the respective pseudorandom sequence $(u_n)_{n\in\mathbb{N}}$ is ultimately periodic. In this case, for the period ρ and the transient τ the following holds.

$$1 \leq \rho + \tau - 1 \leq \#S$$

In particular, if T is bijective (u_n) will be periodic.

PROOF:
Let $(s_n)_{n\in\mathbb{N}}$ be the respective sequence of states and $N := \#S$ the number of different states. T maps all elements of S to at most N other elements of S. Therefore at least the element s_N has to be mapped to an element s_k for $k \in \mathbb{N}$ with $k \leq N$ which was already reached. Hence, we conclude the following.

$$\exists n, k \in \mathbb{N}, k \leq n \leq N : \quad T(s_n) = s_k$$

We choose n and k appropriately and define the following values.

$$\rho := n - k + 1 \,, \qquad \tau := k$$

Now let $i \in \mathbb{N}_0$ be arbitrary and apply the definition. We get the following chain of equations which show that (u_n) is ultimately periodic.

$$\begin{aligned} u_{\tau+i+\rho} &= u_{n+1+i} = G \circ T^{n+1+i}(s_0) = G \circ T^i \circ T^{n+1}(s_0) \\ &= G \circ T^i(s_k) = G \circ T^i \circ T^k(s_0) = G \circ T^{i+k}(s_0) = u_{k+i} = u_{\tau+i} \end{aligned}$$

The inequality can be shown by directly inserting the values into the definition.

$$1 \leq \rho + \tau - 1 = n \leq N = \#S$$

This proofs the given lemma. □

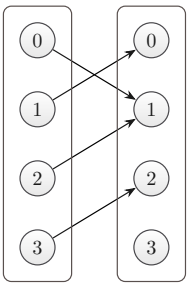
Thus, every pseudorandom sequence will repeat itself after it reached a certain point. The period and the transient are greatly affected by the number of states and the transition function of the PRNG. To get a better insight, we will examine the following idealized examples with different transition functions. Let $\mathcal{G} := (S, T, U, G)$ be a PRNG defined as follows.

$$S := U := \mathbb{Z}_4 \,, \qquad G := \mathrm{id}$$

For a seed $s_0 \in S$ the respective pseudorandom sequence $(u_n)_{n\in\mathbb{N}}$ with period ρ and transient τ will be shown in the following way. Hereby, all elements of the sequence up to the end of the first period are written consecutively and the periodic part is marked by an overline.

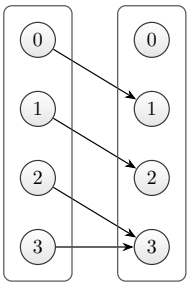
$$(u_n) = u_1 \dots u_{\tau-1}\overline{u_\tau \dots u_{\tau+\rho-1}}$$

To the left of the examples, a scheme of their respective transition function is displayed to make the originating sequences together with their periods and transients more understandable. The boxes are used in place of the set of states S whereas arrows characterize the transition function T.



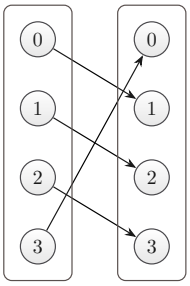
$$T(x) := \begin{cases} 1 & : x \in \{0,2\} \\ 0 & : x = 1 \\ 2 & : x = 3 \end{cases} \,, \qquad (u_n) = \begin{cases} \overline{10} & : s_0 \in \{0,2\} \\ \overline{01} & : s_0 = 1 \\ 2\overline{10} & : s_0 = 3 \end{cases}$$

The first example shows a transition function which is not bijective but does not map any element of S to itself. Hence, in all cases we get a period of 2. The transient varies between 1 and 2 and depends on the seed value.



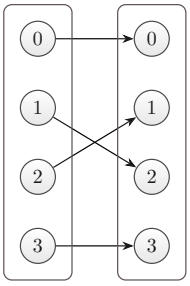
$$T(x) := \begin{cases} x+1 & : x < 3 \\ 3 & : x = 3 \end{cases}, \qquad (u_n) = \begin{cases} 12\overline{3} & : s_0 = 0 \\ 2\overline{3} & : s_0 = 1 \\ \overline{3} & : s_0 \geq 2 \end{cases}$$

In the second example, again a non-bijective transition function is used. This time the value 3 is mapped to itself and as a consequence the period for all possible sequences is 1. As before, the transient varies with respect to the seed value.



$$T(x) := x + 1 \mod 4\,, \qquad (u_n) = \begin{cases} \overline{1230} & : s_0 = 0 \\ \overline{2301} & : s_0 = 1 \\ \overline{3012} & : s_0 = 2 \\ \overline{0123} & : s_0 = 3 \end{cases}$$

In the third example, a bijective transition function is used. The period is maximized and reaches the number of states. In all cases the transient is 1 and as a result all sequences are periodic.



$$T(x) := \begin{cases} x & : x \in \{0, 3\} \\ 2 & : x = 1 \\ 1 & : x = 2 \end{cases}, \qquad (u_n) = \begin{cases} \overline{0} & : s_0 = 0 \\ \overline{21} & : s_0 = 1 \\ \overline{12} & : s_0 = 2 \\ \overline{3} & : s_0 = 3 \end{cases}$$

The last example shows again a bijective transition function T. The transient is again always 1 and all possible pseudorandom sequences are purely periodic. But this time, T maps the values 0 and 3 to themselves. Hence, the period becomes dependent on the initial value and differs between the smallest possible value 1 and 2.

The periodic behavior of pseudorandom sequences greatly constrains the possible randomness of a PRNG. Especially for simulations, using a PRNG which is repeating itself while in use introduces unwanted regularities resulting in an incorrect output. As a consequence, developers of PRNGs try to construct a large period by adjusting the number of states and the transition function. For example, the MT19937 is a PRNG with an extremely large period of $2^{19937} - 1$ if not used with a seed value of zero (Matsumoto and Nishimura 1998). The use of a bijective transition function is not enough to ensure the maximal period. Values that are mapped to

themselves result in the smallest possible period even if the transient of the sequence could be large. Especially for linear PRNGs that are mapping 0 to itself, developers tend to exclude such states from the seeding process to always obtain the maximal period (Blackman and Vigna 2019; Marsaglia et al. 2003). As a counter-example, the so-called "Middle Square RNG" which was developed by Von Neumann in the early days of computer science should be named (Kneusel 2018, pp. 12–15; Widynski 2019). This PRNG computed the square of its current state and returned the middle digits as next random number. It was well known to suffer from the "zero mechanism" — once some digits become zero, all following return values would be zero as well (Kneusel 2018, pp. 12–15; Widynski 2019). So besides a large state space and a bijective transition function, the largest possible permutation cycle should be reached when advancing the state of a PRNG.

5.6.2 Equidistribution

Pseudorandom sequences should mimic the behavior of truly random sequences. And for that reason, we want them to be uniformly distributed on the set of output values in some sense. This property will make it possible to generate every important distribution of random numbers by applying special transformations based on stochastics. Such distributions can then be used by Monte Carlo simulations to estimate solutions more efficiently. But because we are dealing with actual values instead of random variables, we have to clarify what uniformly distributed means. Consequently, we will again rely on probability theory to elaborate on the details without a deeper understanding of randomness (Eisner and Farkas 2019). To be able to always distinct these two different concepts, we will call a sequence of actual values with the desired properties equidistributed.

DEFINITION 5.6: (Equidistributed Sequence)

Let U be a non-empty, finite set of values and μ be a probability measure on the measurable space $(U, \mathcal{P}(U))$. A sequence $(u_n)_{n\in\mathbb{N}}$ in U is equidistributed with respect to μ if for every measurable function $X\colon U \to \mathbb{R}$ the following is true.

$$\lim_{n\to\infty} \frac{1}{n} \sum_{k=1}^{n} X(u_k) = \int_U X \,\mathrm{d}\mu$$

If μ is not specified, we assume it to be the uniform distribution on U.

The idea is that every possible output value should essentially be reached the same amount of times when advancing the state. For pseudorandom sequences generated by a non-bijective transition function the transient part should be ignored as it can be seen as non-recurring "warm-up" time. Therefore equidistribution will be evaluated at infinity in the sense of a limit. Because we wanted to use probability theory to observe randomness, we had to generalize the idea of counting how often different output values would be reached. Instead we use arbitrary measurable functions as observables to estimate their expectation value with respect to the

given sequence and to compare it to their actual expectation value (Eisner and Farkas 2019). Please note that for our needs we have chosen a finite set of elements to simplify the definition of equidistribution. A more general alternative where U has to be a compact metric space with Borel probability measure μ can be found in Eisner and Farkas (2019). Here, measurable functions are interchanged with continuous functions. Because of this, we can further simplify the right-hand side of the definition.

$$\int_U X \,\mathrm{d}\mu = \mathbb{E}\, X = \sum_{u \in U} f(u)\mu(\{u\})$$

To make sure the generalization is working properly, we proof the following lemma which states that, while observing pseudorandom sequences, the relative frequency in one period of an arbitrary element must be given by its probability.

LEMMA 5.3: (Equidistributed Pseudorandom Sequences)

Let $\mathcal{G} := (S, T, U, G)$ be a PRNG with $s_0 \in S$ as its seed value and $(u_n)_{n\in\mathbb{N}}$ the respective pseudorandom sequence with transient τ and period ρ. Furthermore, let μ be a probability measure on $(U, \mathcal{P}(U))$. Then the following statements are equivalent.

(i) (u_n) is equidistributed with respect to μ.

(ii) For all $u \in U$ the following is true.

$$\frac{1}{\rho} \cdot \#\left\{n \in \mathbb{N} \mid \tau \leq n < \rho + \tau, u_n = u\right\} = \mu(\{u\})$$

PROOF:
Because U is a finite set, every measurable function $X : U \to \mathbb{R}$ can be described as a linear combination of characteristic functions with respect to some real coefficients α_u for all $u \in U$ in the following way.

$$X = \sum_{u \in U} \alpha_u \mathbb{1}_{\{u\}}$$

Hence, without loss of generality, it suffices to take only characteristic functions into account. Let $u \in U$ be arbitrary. The right-hand side of the definition will then result in the following.

$$\int_U \mathbb{1}_{\{u\}} \,\mathrm{d}\mu = \mu(\{u\})$$

Applying the characteristic function together with the properties of a periodic sequence to the left-hand side of the definition, looks as follows.

$$\begin{aligned}
\lim_{n\to\infty} \frac{1}{n} \sum_{k=1}^{n} \mathbb{1}_{\{u\}}(u_k) &= \lim_{n\to\infty} \frac{1}{n} \sum_{k=1}^{\tau-1} \mathbb{1}_{\{u\}}(u_k) + \lim_{n\to\infty} \frac{1}{n} \sum_{k=\tau}^{\tau+n-1} \mathbb{1}_{\{u\}}(u_k) \\
&= \frac{1}{\rho} \sum_{k=\tau}^{\tau+\rho-1} \mathbb{1}_{\{u\}}(u_k) \\
&= \frac{1}{\rho} \cdot \#\left\{n \in \mathbb{N} \mid \tau \leq n < \rho + \tau, u_n = u\right\}
\end{aligned}$$

This shows the desired equivalence and proofs the lemma. □

Based on this lemma, it directly follows that for equidistributed, pseudorandom sequences with a maximal period the number of different states has to be a multiple of the number of output values.

COROLLARY 5.4: (Equidistributed Pseudorandom Sequence with Maximal Period)

Let $\mathcal{G} := (S, T, U, G)$ be a PRNG with $s_0 \in S$ as its initial state and $(u_n)_{n\in\mathbb{N}}$ the respective pseudorandom sequence. If (u_n) is equidistributed and periodic with maximal period $\#S$ then the following is true.

$$\exists k \in \mathbb{N} : \quad \#S = k \cdot \#U$$

5.6.3 Multidimensional Equidistribution

In physical problems, we typically have to deal with partial differential equations in many dimensions. Finding deterministic, numerical solutions through iterated integrals becomes infeasible due to the resulting degrees of freedom. This is called the "curse of dimensionality". With the use of Monte Carlo integration, we can overcome this burden so that for high-dimensional problems we are able to reduce the error of the estimated solutions for every iteration much faster. As a consequence, successive pseudorandom numbers generated by a PRNG should be interpretable as a pseudorandom vector. But due to the shown dependence of successive values in a pseudorandom sequence, again regular patterns and artifacts can arise which can only be observed by some advanced testing techniques for statistical performance. However, a PRNG that is used in more than one dimension should at least provide an equidistribution over all possible multidimensional output values. For a rigorous definition of this concept, we will first clarify how to use a pseudorandom sequence as a sequence of pseudorandom vectors.

DEFINITION 5.7: (Corresponding Vector Sequence)

Let U be a non-empty set of values and $(u_n)_{n\in\mathbb{N}}$ be a sequence in U. Choose $k \in \mathbb{N}$ and $t \in \mathbb{N}_0$ and define the following for all $n \in \mathbb{N}$.

$$v_n := (u_i)_{i\in I_n} \ , \qquad I_n := \{t + (n-1)k + p \mid p \in \mathbb{N}, p \leq k\}$$

We call the sequence $(v_n)_{n\in\mathbb{N}}$ in U^k the corresponding k-dimensional vector sequence with translation t with respect to (u_n).

Transforming a sequence of values into a sequence of vectors consists of interpreting successive values as coordinates of vectors. Figure 9 shows this process schematically. Corresponding vector sequences inherit the property of being ultimately periodic.

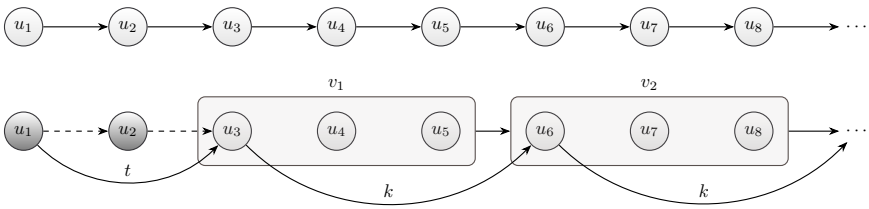


Figure 9: The upper part of the figure shows a schematic view of an arbitrary sequence of values $(u_n)_{n\in\mathbb{N}}$ in an arbitrary non-empty set U. The lower part visualizes the corresponding k-dimensional vector sequence $(v_n)_{n\in\mathbb{N}}$ with translation t, whereby $k = 3$ and $t = 2$. The first two values of (u_n) are skipped due to the translation. Afterwards the elements of (v_n), marked through boxes, emerge from interpreting successive values of (u_n) as their coordinates.

LEMMA 5.5: (Corresponding Vector Sequences are Ultimately Periodic)

Let U be a non-empty set of values and $(u_n)_{n\in\mathbb{N}}$ be an ultimately periodic sequence in U with period ρ and transient τ. In this case, every corresponding k-dimensional vector sequence $(v_n)_{n\in\mathbb{N}}$ with translation t is ultimately periodic with period ρ' and transient τ' defined as follows.

$$\rho' := \frac{\rho}{\gcd(\rho, k)} \ , \qquad \tau' := \left\lceil \frac{\max(0, \tau - 1 - t)}{k} \right\rceil + 1$$

PROOF:

Choose $n \in \mathbb{N}_0$ and $i \in \mathbb{N}$ with $i \leq k$ to be arbitrary. We denote with $v_n^{(i)}$ the i. coordinate of the n. vector. By definition the following equality holds.

$$v^{(i)}_{\tau'+n+\rho'} = u_{t+(\tau'+n+\rho'-1)k+i}$$

Observing the index, we separate it into three parts. One for the index, one for the transient one for the period.

$$t + (\tau' + n + \rho' - 1)k + i = \underbrace{(t + \tau' k - k + 1)}_{=:\tilde{\tau}} + \underbrace{(nk + i - 1)}_{=:\tilde{n}} + \underbrace{\rho' k}_{=:\tilde{\rho}}$$

The period part has to be a multiple of the period ρ of (u_n) as can be seen in the following. Hence, $\tilde{\rho}$ has the property of a period.

$$\tilde{\rho} = \rho' k = \frac{\rho k}{\gcd(\rho, k)} = \rho \frac{k}{\gcd(\rho, k)}$$

To apply the periodicity of (u_n), the transient part has to be bigger or equal to the transient τ of (u_n).

$$\tilde{\tau} = t + \tau' k - k + 1 = 1 + t + k \left\lceil \frac{\max(0, \tau - 1 - t)}{k} \right\rceil \geq \tau$$

Inserting the results and applying the periodicity of (u_n), we can conclude that the corresponding vector sequence has to be ultimately periodic as well.

$$v^{(i)}_{\tau'+n+\rho'} = u_{\tilde{\tau}+\tilde{n}+\tilde{\rho}} = u_{\tilde{\tau}+\tilde{n}} = u_{t+(\tau'+n-1)k+i} = v^{(i)}_{\tau'+n}$$

Due to the shown statements, ρ' and τ' are indeed the smallest possible values such that this equation holds and can therefore be denoted as period and transient of (v_n) respectively. □

The given concept shall now be applied to define the equidistribution of a sequence in more than one dimension. As a result, the following property, called multidimensional equidistribution, becomes a generalization of equidistribution and quantifies in how many dimensions a PRNG can be used. We do not follow the typical definitions from L'Ecuyer (1994) and Matsumoto and Nishimura (1998).

DEFINITION 5.8: (Multidimensional Equidistributed Sequence)

Let U be a non-empty, finite set of values, $k \in \mathbb{N}$ and μ be a probability measure on $\left(U^k, \mathcal{P}\left(U^k\right)\right)$. A sequence $(u_n)_{n\in\mathbb{N}}$ in U is k-dimensional equidistributed with respect to μ if for all $t \in \mathbb{N}_0$ the corresponding k-dimensional vector sequence with translation t is equidistributed with respect to μ. If μ is not specified, we assume it to be the uniform distribution on U^k.

In comparison to the one-dimensional equidistribution, the general idea of multidimensional equidistribution is straightforward. For corresponding vector sequences, it reduces to the application of equidistribution. Especially for pseudorandom sequences, we can get a more precise result which will serve as an easily testable criterion for multidimensional equidistribution.

COROLLARY 5.6: (Multidimensional Equidistributed Pseudorandom Sequence)

Let $\mathcal{G} := (S, T, U, G)$ be a PRNG, $s_0 \in S$ its initial state and $(u_n)_{n\in\mathbb{N}}$ be the respective pseudorandom sequence with period ρ. Furthermore, let $k \in \mathbb{N}$ and (u_n) be k-dimensional equidistributed. In this case the following statement is true.

$$\exists a \in \mathbb{N}: \quad \rho = a \cdot \gcd(\rho, k) \cdot \#U^k$$

As a consequence, multidimensional equidistribution is greatly affected by the set of output symbols and the generator function. Furthermore, according to the formula, for k-dimensional equidistribution with $k \geq 2$, the set of output symbols has to be smaller than the set of states. In practice, the seed of a pseudorandom sequence defines the translation of its corresponding vector sequence. For the full period, the definition of multidimensional equidistribution given here is equivalent to the typical definition given in L'Ecuyer (1994). If the corresponding vector sequence consists of a smaller period then the given concept is stronger than the typical one. We will again show some idealized examples to explain the details of the result and to understand its principles. For this, let $\mathcal{G} := (S, T, U, G)$ again be a PRNG, $s_0 \in S$ its initial state and $(u_n)_{n\in\mathbb{N}}$ the respective pseudorandom sequence with period ρ. The corresponding k-dimensional vector sequence with translation t will be called $(v_n)_{n\in\mathbb{N}}$. Sequences will again be denoted by writing their elements consecutively with their periodic part marked by an overline. G will be shown as table which maps values from the first line to values in the second line.

The first example will use a PRNG with a state size of 8 and a trivial, bijective transition function with full period. The generator function G is chosen so that the resulting pseudorandom sequence is k-dimensional equidistributed for $k = 3$.

$$S := \mathbb{Z}_8 \,, \qquad G := \mathbb{Z}_2 \,, \qquad T(x) := x + 1 \mod 8 \,, \qquad s_0 = 7$$

$$G := \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \end{pmatrix}$$

Choosing $k = 3$ and setting the translation to $t = 0$, the corresponding vector sequence will have a maximal period and will reach every element in U^3. A different translation would only result in a cyclic permutation of the periodic part.

$$(v_n) = \overline{(000)(111)(010)(001)(110)(100)(011)(101)}$$

For $k = 2$ and $t = 0$ the corresponding vector sequence must have the half period. In this case, the sequence is not equidistributed in U^2 because the element (10) is not reached and (01) is reached twice.

$$(v_n) = \overline{(00)(01)(11)(01)}$$

Shifting the sequence by setting $t = 1$, we get its complement. This time, it does not reach (01) but (10) twice instead. Again, the period is 4 and the sequence is not equidistributed.

$$(v_n) = \overline{(00)(11)(10)(10)}$$

Putting both sequences for $t = 0$ and $t = 1$ together results in an two-dimensional equidistributed sequence. Note that the weaker definition of multidimensional equidistribution given in L'Ecuyer (1994) would therefore call the given sequence two-dimensional equidistributed. In the second example, we have chosen a doubled state size and adjusted the generator function to achieve k-dimensional equidistribution for $k = 2$ and $k = 3$.

$$S := \mathbb{Z}_{16} \,, \qquad G := \mathbb{Z}_2 \,, \qquad T(x) := x + 1 \mod 16 \,, \qquad s_0 = 15$$

$$G := \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \end{pmatrix}$$

The greatest common divisor of 2 and 16 is again 2. Hence, the corresponding vector sequence has half period. But this time every element is reached. Changing the translation to $t = 1$ would permute the sequence.

$$(v_n) = \overline{(00)(01)(11)(01)(00)(11)(10)(10)}$$

In three dimensions, we get the full period and an equidistribution in U^3. A different translation would again only result in a cyclic permutation of the periodic part.

$$\begin{aligned}(v_n) = {} & \overline{(000)(111)(010)(011)(101)(000)(011)(101)} \\ & \overline{(001)(110)(100)(001)(110)(100)(111)(010)}\end{aligned}$$

Note that for $k = 4$, according to corollary 5.6, we cannot achieve k-dimensional equidistribution because for all $a \in \mathbb{N}$ we get the following inequality.

$$2^4 = 16 = \rho \neq a \cdot \gcd(\rho, k) \cdot \#U^k = a \cdot 4 \cdot 2^4 = a \cdot 2^6$$

Hence, for the given PRNG we have reached maximal equidistribution by keeping its maximal period.

5.6.4 Linearity

According to L'Ecuyer (2015), the transition function of most PRNGs can be viewed as linear recurrence modulo some prime number. Thus, such PRNGs are called linear and exhibit certain advantages and disadvantages in comparison to non-linear PRNGs. The linearity property makes a theoretical and statistical analysis of respective pseudorandom sequences much easier (Bauke and Mertens 2007; Blackman and Vigna 2019; L'Ecuyer 2015). Hence, linear PRNGs are mathematically well-founded and understood. But exactly this makes them vulnerable to certain empirical tests and applications, such as the linear-complexity and the matrix-rank tests, which exploit linearity (L'Ecuyer 2015; Lemire and O'Neill 2019; O'Neill 2014). In general, linear PRNGs suffer from too much regularity in their output. Nevertheless, many well-known and widely used PRNGs are linear. This is due to the fact that while offering more features they tend to be faster and easier to implement than their counterparts (Blackman and Vigna 2019; L'Ecuyer 2015). To name a few examples, the MT19937 (Matsumoto and Nishimura 1998), Xorshift RNGs (Marsaglia et al. 2003; Vigna 2016, 2017), and WELL generators (Panneton, L'ecuyer, and Matsumoto 2006) are all linear PRNGs. We will first introduce a rigorous concept of linearity to understand their underlying theory.

DEFINITION 5.9: (Linear and Scrambled Linear PRNG)

Let $m \in \mathbb{P}$ and $\mathcal{G} := (S, T, U, G)$ be a PRNG. We call $\mathcal{G}$ linear modulo m if S and U are finite-dimensional vector spaces over the finite field $\mathbb{F}_m$ and T and G are linear transformations. In this case, we identify T and G with their respective matrices such that $T \in \mathbb{F}_m^{p \times p}$ and $G \in \mathbb{F}_m^{q \times p}$, whereas $p := \dim S$ and $q := \dim U$. If G cannot be represented by a linear transformation we say $\mathcal{G}$ is a scrambled linear PRNG modulo m.

The state and output space have to be vector spaces over a finite field such that linear transformations are well-defined. The definition makes sure that for linear PRNGs both, the transition and the generator function, are linear transformations. This property is strong and tends to reduce the statistical performance of the PRNG. Therefore in practice, often at least the generator function is chosen to be non-linear (Blackman and Vigna 2019). Examples for scrambled linear PRNGs are given by some generators of the PCG family (O'Neill 2014) and the Xoroshiro128+ (Blackman and Vigna 2019). Due to their non-linear generator function, these PRNGs are more difficult to analyze. As a consequence, the following lemma about

the equidistribution and periodicity applies only to linear PRNGs and may under certain circumstances serve as a foundation for a further theoretical analysis of scrambled linear PRNGs (Bauke and Mertens 2007; L'Ecuyer 2015).

LEMMA 5.7: (Period and Equidistribution of a Linear PRNG)

For $m \in \mathbb{P}$, let $\mathcal{G} := (S, T, U, G)$ be a linear PRNG modulo m with $p := \dim S$. Furthermore, let the characteristic polynomial of T be a primitive polynomial over $\mathbb{F}_m$ and let G be a full rank matrix with $q := \operatorname{rank} G$. Then for all seeds $s_0 \in S \setminus \{0\}$ the respective pseudorandom sequences $(u_n)_{n\in\mathbb{N}}$ are periodic with period $m^p - 1$ and for all elements $u \in U$ the following holds.

$$n_u := \#\{n \in \mathbb{N} \mid n \leq m^p - 1, u_n = u\} = \begin{cases} m^{p-q} - 1 & : u = 0 \\ m^{p-q} & : \text{else} \end{cases}$$

In particular, the sequence (u_n) is equidistributed with respect to the following probability measure μ.

$$\mu \colon \mathcal{P}(U) \to [0, 1] \ , \qquad \mu(\{u\}) := \frac{n_u}{m^p - 1}$$

We will give no proof for this lemma. For linear PRNGs, it is not possible to get an equidistribution on the complete output space U. Instead the zero element is always reached once less often, introducing bias in the respective probability distribution μ. But for big values of p, this bias is neglectable as the following limit states.

$$\lim_{p\to\infty} \frac{1}{m^p - 1} = 0$$

Apart from this, by reaching the maximal period, a linear PRNG gives us the equidistribution of its output values for free. Hence, both properties, equidistribution and maximal periodicity, can be mathematically proven by showing that the characteristic polynomial of T is a primitive polynomial over the underlying field. This gives us a general tool for the analyzation of linear PRNGs and explains their widespread use.

Let us again consider an example to further highlight the usage of linear PRNGs. For this, let $\mathcal{G} := (S, T, U, G)$ be a linear PRNG modulo 2, $s_0 \in S$ its initial state, $(u_n)_{n\in\mathbb{N}}$ the respective pseudorandom sequence and $(s_n)_{n\in\mathbb{N}}$ the respective sequence of states. Sequences will again be denoted by writing their elements consecutively with their periodic part marked by an overline. In this example, column vectors will be written as row vectors without spaces or commas.

$$S := \mathbb{F}_2^3 \ , \qquad U := \mathbb{F}_2^2 \ , \qquad s_0 := (001)$$

$$T := \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 1 \end{pmatrix} \ , \qquad G := \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

In this example, G has full rank and the characteristic polynomial of T is given by the following expression which is a primitive polynomial over $\mathbb{F}_2$.

$$p_T(x) = \det(T - xI) = x^3 + x^2 + 1$$

As a result, we will get the maximal period of 7 elements for the state and output sequence.

$$(s_n) = \overline{(001)(011)(111)(110)(101)(010)(100)}$$

$$(u_n) = \overline{(11)(11)(10)(01)(10)(00)(01)}$$

In one period, (u_n) reaches the elements (01), (10) and (11) exactly two times whereas (00) only appears one time.

5.6.5 Predictability and Security

There are a lot applications where PRNGs are used to encrypt crucial data. As a matter of fact, such usability infers certain restrictions on a PRNG. In general, PRNGs implemented for this way of use have to be secure and unpredictable in some sense. Making sure a PRNG is not predictable, typically goes at the cost of speed and reproducibility. For the simulation of mathematical and physical problems based on partial differential equations, we do not need to use secure PRNGs. Instead, we will focus on performance and statistical quality. Therefore no further discussion or rigorous concept of security will be given.

5.7 Implementation-Specific Properties and Features

5.7.1 Seekability

5.7.2 Seeding and Consistency

5.7.3 Ease of Implementation

5.7.4 Portability

5.7.5 Speed

5.7.6 Alignment, Caching, Code Size, Memory Size and Complexity

5.7.7 Scalability, Parallelism, Vectorization and Multiple Streams

5.8 Analyzation

visualization, proof, experiments, benchmarks (runtime), test suites, code analyzation

statistical quality and performance vs implementation quality and performance

Visualizations: randograms 2d and 3d, histograms, simulation plots and images

Period and Uniformity, Empirical Testing, predictability and Security, Speed, Memory Size, Code Size, Output Range, Seekability, multiple streams, k-dimensional equidistribution, theoretical support, repeatability, portability, ease of implementation

5.9 Examples

DEFINITION 5.10: (Linear Congruential Generator)

Let $m \in \mathbb{N}$ with $m \geq 2$ and $a, c \in \mathbb{Z}_m$. We define the PRNG $\mathrm{LCG}(m, a, c) \coloneqq (S, T, U, G)$

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

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

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.*

DEFINITION 5.11: (Linear Feedback Shift Registers)

DEFINITION 5.12: (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.*

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

$$T \colon S \to S$$

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

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

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

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

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

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

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

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

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

DEFINITION 5.13: (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* $c \in \mathbb{Z}_{2^t}$*.*

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

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

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

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

DEFINITION 5.14: (Xoroshiro128+)

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

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

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

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

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

DEFINITION 5.15: (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.*

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

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

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

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

6 Previous Work

7 Design of the API

Code: Is-Valid Utility

```
namespace detail {

template <typename F, typename... Args,
          typename = decltype(std::declval<F>()(std::declval<Args&&>()...)
               )>
std::true_type is_valid(void*);

template <typename F, typename... Args>
std::false_type is_valid(...);

}  // namespace detail

inline constexpr auto is_valid = [](auto f) constexpr {
  return [](auto&&... args) constexpr {
    return decltype(
        detail::is_valid<decltype(f), decltype(args)&&...>(nullptr)){};
  };
};
```

Code: Uniform Template

```
constexpr auto has_uniform_01 =
    is_valid([](auto&& x) -> decltype(x.uniform()) {});
constexpr auto has_uniform =
    is_valid([](auto&& x, auto&& y, auto&& z) -> decltype(x.uniform(y, z))
          {});

template <typename Real, typename RNG>
constexpr inline auto uniform(RNG&& rng) noexcept
    -> std::enable_if_t<!decltype(has_uniform_01(rng))::value, Real> {
  return detail::uniform<Real>(std::forward<RNG>(rng)());
}

template <typename Real, typename RNG>
constexpr inline auto uniform(RNG&& rng) noexcept
    -> std::enable_if_t<decltype(has_uniform_01(rng))::value, Real> {
  return std::forward<RNG>(rng).uniform();
}

template <typename Real, typename RNG>
constexpr inline auto uniform(RNG&& rng, Real a, Real b) noexcept
    -> std::enable_if_t<!decltype(has_uniform(rng, a, b))::value, Real> {
  return detail::uniform(std::forward<RNG>(rng)(), a, b);
}

template <typename Real, typename RNG>
constexpr inline auto uniform(RNG&& rng, Real a, Real b) noexcept
    -> std::enable_if_t<decltype(has_uniform(rng, a, b))::value, Real> {
  return std::forward<RNG>(rng).uniform(a, b);
```

```
}

template <typename Real>
constexpr inline Real uniform(std::mt19937& rng, Real a = 0,
                              Real b = 1) noexcept {
  return detail::uniform(static_cast<uint32_t>(rng()), a, b);
}

template <typename Real>
constexpr inline Real uniform(std::mt19937&& rng, Real a = 0,
                              Real b = 1) noexcept {
  return detail::uniform(static_cast<uint32_t>(std::move(rng)()), a, b);
}
```

7.1 C++ Concepts

7.2 Uniform Random Bit Generator

7.3 Random Number Engine

7.4 Seeding and Seed Sequences

7.5 Distributions

7.6 Algorithms

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.

8 Testing Framework

Code: Scalar Monte Carlo π

```
template <typename Real, typename Integer, typename RNG>
inline Real monte_carlo_pi(RNG& rng, Integer samples) noexcept {
  // std::uniform_real_distribution<Real> dist{0, 1};
  Integer samples_in_circle{};
  for (auto i = samples; i > 0; --i) {
    // const auto x = dist(rng);
    // const auto y = dist(rng);
    const auto x = pxart::uniform<Real>(rng);
    const auto y = pxart::uniform<Real>(rng);
    samples_in_circle += (x * x + y * y <= 1);
  }
  return static_cast<Real>(samples_in_circle) / samples * 4;
}
```

Code: AVX Monte Carlo π

```
template <typename Integer, typename RNG>
inline float monte_carlo_pi(RNG& rng, Integer samples) noexcept {
  const auto uniform = [](__m256i x) {
    const auto tmp = _mm256_srli_epi32(x, 9);
    const auto tmp2 = _mm256_or_si256(tmp, _mm256_set1_epi32(0x3f800000));
    return _mm256_sub_ps(_mm256_castsi256_ps(tmp2), _mm256_set1_ps(1.0f));
  };

  auto samples_in_circle = _mm256_setzero_si256();

  for (auto i = samples; i > 0; i -= 8) {
    const auto x = uniform(rng());
    const auto y = uniform(rng());
    const auto radius = _mm256_add_ps(_mm256_mul_ps(x, x), _mm256_mul_ps(y
        , y));
    const auto mask = _mm256_castps_si256(
        _mm256_cmp_ps(radius, _mm256_set1_ps(1.0f), _CMP_LE_OQ));
    samples_in_circle = _mm256_add_epi32(
        samples_in_circle, _mm256_and_si256(_mm256_set1_epi32(1), mask));
  }
  samples_in_circle = _mm256_hadd_epi32(samples_in_circle,
      samples_in_circle);
  samples_in_circle = _mm256_hadd_epi32(samples_in_circle,
      samples_in_circle);
  return 4.0f *
         (reinterpret_cast<uint32_t*>(&samples_in_circle)[0] +
          reinterpret_cast<uint32_t*>(&samples_in_circle)[4]) /
         samples;
}
```

9 Implementation of Vectorized PRNGs

9.1 Linear Congruential Generators

9.2 Mersenne Twister

Code: Scalar MT19937 Structure

```
struct mt19937 {
  using uint_type = uint32_t;
  using result_type = uint_type;
  static constexpr size_t word_size = 32;
  static constexpr size_t state_size = 624;
  static constexpr size_t shift_size = 397;
  static constexpr size_t mask_bits = 31;
  static constexpr uint_type xor_mask = 0x9908b0dfu;
  static constexpr uint_type tempering_b_mask = 0x9d2c5680u;
  static constexpr uint_type tempering_c_mask = 0xefc60000u;
  static constexpr size_t tempering_u_shift = 11;
  static constexpr size_t tempering_s_shift = 7;
  static constexpr size_t tempering_t_shift = 15;
  static constexpr size_t tempering_l_shift = 18;
  static constexpr uint_type default_seed = 5489u;
  static constexpr uint_type init_multiplier = 1812433253u;

  static constexpr uint_type mask =  //
      (~uint_type{}) >> (sizeof(uint_type) * 8 - word_size);
  static constexpr uint_type upper_mask =  //
      ((~uint_type{}) << mask_bits) & mask;
  static constexpr uint_type lower_mask =  //
      (~upper_mask) & mask;

  struct default_seeder;

  template <typename RNG>
  constexpr explicit mt19937(RNG&& rng);

  constexpr mt19937();

  mt19937(const mt19937&) = default;
  mt19937& operator=(const mt19937&) = default;
  mt19937(mt19937&&) = default;
  mt19937& operator=(mt19937&&) = default;

  constexpr result_type operator()() noexcept;
  constexpr result_type min() noexcept { return uint_type{}; }
  constexpr result_type max() noexcept { return (~uint_type{}) & mask; }

  uint_type state[state_size];
  int state_index = state_size;
};
```

Code: Scalar MT19937 Seeding

```
struct mt19937::default_seeder {
  constexpr explicit default_seeder(uint_type s = default_seed) : x{s &
      mask} {}
  constexpr uint_type operator()() noexcept;
  constexpr uint_type min() noexcept { return uint_type{}; }
  constexpr uint_type max() noexcept { return (~uint_type{}) & mask; }
  uint_type x;
  uint_type c{0};
};

constexpr auto mt19937::default_seeder::operator()() noexcept -> uint_type
    {
  const auto result = x;
  x = (x ^ (x >> (word_size - 2)));
  x = (init_multiplier * x + (++c)) & mask;
  return result;
}

template <typename RNG>
constexpr mt19937::mt19937(RNG&& rng) {
  pxart::generate(std::forward<RNG>(rng), state, state + state_size);
}

constexpr mt19937::mt19937() : mt19937{default_seeder{}} {}
```

Code: Scalar MT19937 Advancing

```
constexpr auto mt19937::operator()() noexcept -> result_type {
  if (state_index >= state_size) {
    const auto transition = [this](int k, int k1, int k2) constexpr {
      const auto x = (state[k] & upper_mask) | (state[k1] & lower_mask);
      state[k] = state[k2] ^ (x >> 1) ^ ((x & 0x01) ? xor_mask : 0);
    };

    for (int k = 0; k < state_size - shift_size; ++k)
      transition(k, k + 1, k + shift_size);
    for (int k = state_size - shift_size; k < state_size - 1; ++k)
      transition(k, k + 1, k + shift_size - state_size);
    transition(state_size - 1, 0, shift_size - 1);

    state_index = 0;
  }

  auto y = state[state_index++];
  y ^= (y >> tempering_u_shift);
  y ^= (y << tempering_s_shift) & tempering_b_mask;
  y ^= (y << tempering_t_shift) & tempering_c_mask;
  y ^= (y >> tempering_l_shift);
  return y;
}
```

Code: AVX MT19937 Structure

```
struct mt19937 {
  using uint_type = uint32_t;
  using result_type = uint_type;
  using simd_type = __m256i;
  static constexpr size_t simd_size = sizeof(simd_type) / sizeof(
      result_type);

  static constexpr size_t word_size = 32;
  static constexpr size_t state_size = 624;
  static constexpr size_t shift_size = 397;
  static constexpr size_t mask_bits = 31;
  static constexpr uint_type xor_mask = 0x9908b0dfu;
  static constexpr uint_type tempering_b_mask = 0x9d2c5680u;
  static constexpr uint_type tempering_c_mask = 0xefc60000u;
  static constexpr size_t tempering_u_shift = 11;
  static constexpr size_t tempering_s_shift = 7;
  static constexpr size_t tempering_t_shift = 15;
  static constexpr size_t tempering_l_shift = 18;
  static constexpr uint_type default_seed = 5489u;
  static constexpr uint_type init_multiplier = 1812433253u;

  static constexpr uint_type mask = (~uint_type{}) >>
                                    (sizeof(uint_type) * 8 - word_size);
  static constexpr uint_type upper_mask = ((~uint_type{}) << mask_bits) &
      mask;
  static constexpr uint_type lower_mask = (~upper_mask) & mask;

  template <typename RNG>
  explicit mt19937(RNG&& rng);

  mt19937();

  mt19937(const mt19937&) = default;
  mt19937& operator=(const mt19937&) = default;
  mt19937(mt19937&&) = default;
  mt19937& operator=(mt19937&&) = default;

  simd_type operator()() noexcept;
  constexpr result_type min() noexcept { return uint_type{}; }
  constexpr result_type max() noexcept { return (~uint_type{}) & mask; }

  uint_type state[state_size + simd_size] __attribute__((aligned(32)));
  int state_index = state_size;
};

template <typename RNG>
inline mt19937::mt19937(RNG&& rng) {
  generate(std::forward<RNG>(rng), state, state + state_size);
}

inline mt19937::mt19937() : mt19937{pxart::mt19937::default_seeder{}} {}
```

Code: AVX MT19937 Advancing

```
inline auto mt19937::operator()() noexcept -> simd_type {
  if (state_index >= state_size) {
    const auto transition = [this](int k, int k_shift) constexpr {
      const auto simd_upper_mask = _mm256_set1_epi32(upper_mask);
      const auto simd_lower_mask = _mm256_set1_epi32(lower_mask);
      const auto simd_zero = _mm256_setzero_si256();
      const auto simd_one = _mm256_set1_epi32(1);
      const auto simd_xor_mask = _mm256_set1_epi32(xor_mask);

      const auto s0 =
          _mm256_load_si256(reinterpret_cast<const simd_type*>(&state[k]))
              ;
      const auto s1 =
          _mm256_loadu_si256(reinterpret_cast<const simd_type*>(&state[k +
               1]));
      const auto ss = _mm256_loadu_si256(
          reinterpret_cast<const simd_type*>(&state[k_shift]));

      const auto y = _mm256_or_si256(_mm256_and_si256(s0, simd_upper_mask)
           ,
                                     _mm256_and_si256(s1, simd_lower_mask)
                                          );
      const auto mag01 = _mm256_and_si256(
          simd_xor_mask,
          _mm256_cmpgt_epi32(_mm256_and_si256(y, simd_one), simd_zero));
      const auto tmp2 = _mm256_xor_si256(_mm256_srli_epi32(y, 1), mag01);
      const auto result = _mm256_xor_si256(ss, tmp2);
      return result;
    };

    const auto first = transition(0, shift_size);
    _mm256_store_si256(reinterpret_cast<simd_type*>(&state[0]), first);
    _mm256_store_si256(reinterpret_cast<simd_type*>(&state[state_size]),
         first);

    int k = simd_size;
    for (; k < state_size - shift_size; k += simd_size) {
      const auto result = transition(k, k + shift_size);
      _mm256_store_si256(reinterpret_cast<simd_type*>(&state[k]), result);
    }
    for (; k < state_size; k += simd_size) {
      const auto result = transition(k, k + shift_size - state_size);
      _mm256_store_si256(reinterpret_cast<simd_type*>(&state[k]), result);
    }

    state_index = 0;
  }

  auto x = _mm256_load_si256(
      reinterpret_cast<const simd_type*>(&state[state_index]));
  state_index += simd_size;
  x = _mm256_xor_si256(x, _mm256_srli_epi32(x, tempering_u_shift));
  x = _mm256_xor_si256(x,
                       _mm256_and_si256(_mm256_slli_epi32(x,
                            tempering_s_shift),
                                        _mm256_set1_epi32(tempering_b_mask
                                             )));
```

```
  x = _mm256_xor_si256(x,
                       _mm256_and_si256(_mm256_slli_epi32(x,
                            tempering_t_shift),
                                        _mm256_set1_epi32(tempering_c_mask
                                             )));
  x = _mm256_xor_si256(x, _mm256_srli_epi32(x, tempering_l_shift));
  return x;
}
```

9.3 Permuted Congruential Generators

9.4 Xoroshiro

Code: Scalar Xoroshiro128+ Structure

```
struct xoroshiro128plus {
  using uint_type = uint64_t;
  using result_type = uint_type;
  static constexpr size_t word_size = 64;
  static constexpr size_t rotation_a = 24;
  static constexpr size_t shift_b = 16;
  static constexpr size_t rotation_c = 37;

  static constexpr uint_type rotate_left(uint_type x, size_t k) noexcept {
    return (x << k) | (x >> (64 - k));
  }

  xoroshiro128plus() = default;
  xoroshiro128plus(const xoroshiro128plus&) = default;
  xoroshiro128plus& operator=(const xoroshiro128plus&) = default;
  xoroshiro128plus(xoroshiro128plus&&) = default;
  xoroshiro128plus& operator=(xoroshiro128plus&&) = default;

  xoroshiro128plus(uint_type x, uint_type y) : s0{x}, s1{y} {}
  template <typename RNG>
  constexpr explicit xoroshiro128plus(RNG&& rng)
      : s0{(static_cast<uint_type>(rng()) << 32) |
           static_cast<uint_type>(rng())},
        s1{(static_cast<uint_type>(rng()) << 32) |
           static_cast<uint_type>(rng())} {}

  constexpr auto operator()() noexcept;
  constexpr void jump() noexcept;
  constexpr void long_jump() noexcept;
  static constexpr auto min() noexcept { return uint_type{}; }
  static constexpr auto max() noexcept { return ~uint_type{}; }

  uint_type s0{1314472907419283471ul};
  uint_type s1{7870872464127966567ul};
};
```

Code: Scalar Xoroshiro128+ Advancing

```
constexpr auto xoroshiro128plus::operator()() noexcept {
  // The order is important. Otherwise jumps will not work properly.
  const auto result = s0 + s1;
  s1 ^= s0;
  s0 = rotate_left(s0, rotation_a) ^ s1 ^ (s1 << shift_b);
  s1 = rotate_left(s1, rotation_c);
  return result;
}

constexpr void xoroshiro128plus::jump() noexcept {
  // Magic numbers depend on rotation and shift arguments.
  constexpr uint_type mask[] = {0xdf900294d8f554a5ul, 0x170865df4b3201fcul
      };
  uint_type result0 = 0;
  uint_type result1 = 0;
  for (int i = 0; i < 2; i++) {
    for (size_t b = 0; b < word_size; b++) {
      // if (mask[i] & (1ul << b)) {
      //   result0 ^= s0;
      //   result1 ^= s1;
      // }
      const auto tmp = (mask[i] & (1ul << b)) ? (~uint_type{}) : (0);
      result0 ^= s0 & tmp;
      result1 ^= s1 & tmp;
      operator()();
    }
  }
  s0 = result0;
  s1 = result1;
}
```

Code: AVX Xoroshiro128+ Structure

```
struct xoroshiro128plus {
  using uint_type = uint64_t;
  using simd_type = __m256i;
  using result_type = simd_type;
  static constexpr size_t simd_size = 4;
  static constexpr size_t word_size = 64;
  static constexpr size_t rotation_a = 24;
  static constexpr size_t shift_b = 16;
  static constexpr size_t rotation_c = 37;

  static inline auto rotate_left(__m256i x, int k) noexcept {
    return _mm256_or_si256(_mm256_slli_epi64(x, k),
                           _mm256_srli_epi64(x, 64 - k));
  }

  xoroshiro128plus() = default;
  xoroshiro128plus(const xoroshiro128plus& rng) = default;
  xoroshiro128plus& operator=(const xoroshiro128plus&) = default;
  xoroshiro128plus(xoroshiro128plus&&) = default;
  xoroshiro128plus& operator=(xoroshiro128plus&&) = default;
```

```
  template <typename RNG>
  explicit xoroshiro128plus(RNG&& rng)
      : s0{_mm256_set_epi32(rng(), rng(), rng(), rng(), rng(), rng(), rng
          (),
                            rng())},
        s1{_mm256_set_epi32(rng(), rng(), rng(), rng(), rng(), rng(), rng
            (),
                            rng())} {}

  auto operator()() noexcept;
  void jump() noexcept;
  void long_jump() noexcept;
  static constexpr auto min() noexcept { return uint_type{}; }
  static constexpr auto max() noexcept { return ~uint_type{}; }

  simd_type s0;
  simd_type s1;
};
```

Code: AVX Xoroshiro128+ Advancing

```
inline auto xoroshiro128plus::operator()() noexcept {
  // The order is important. Otherwise jumps will not work properly.
  const auto result = _mm256_add_epi64(s0, s1);
  s1 = _mm256_xor_si256(s0, s1);
  s0 = _mm256_xor_si256(s1, _mm256_xor_si256(_mm256_slli_epi64(s1, shift_b
      ),
                                              rotate_left(s0, rotation_a)))
                                                  ;
  s1 = rotate_left(s1, rotation_c);
  return result;
}

inline void xoroshiro128plus::jump() noexcept {
  // Magic numbers depend on rotation and shift arguments.
  const simd_type jump_mask[] = {_mm256_set1_epi64x(0xdf900294d8f554a5ul),
                                 _mm256_set1_epi64x(0x170865df4b3201fcul)
                                     };
  const auto zero = _mm256_setzero_si256();
  const auto one = _mm256_set1_epi64x(1ul);
  auto result0 = zero;
  auto result1 = zero;
  for (int i = 0; i < 2; i++) {
    auto bit = one;
    for (size_t b = 0; b < word_size; ++b) {
      // const auto bit = _mm256_slli_epi64(one, b);
      const auto mask =
          _mm256_cmpeq_epi64(_mm256_and_si256(jump_mask[i], bit), zero);
      result0 = _mm256_xor_si256(result0, _mm256_andnot_si256(mask, s0));
      result1 = _mm256_xor_si256(result1, _mm256_andnot_si256(mask, s1));
      s1 = _mm256_xor_si256(s0, s1);
      s0 = _mm256_xor_si256(s1, _mm256_xor_si256(_mm256_slli_epi64(s1,
          shift_b),
                                                  rotate_left(s0,
```

```
                                                        rotation_a)));
      s1 = rotate_left(s1, rotation_c);
      // operator()();
      bit = _mm256_slli_epi64(bit, 1);
    }
  }
  s0 = result0;
  s1 = result1;
}
```

Code: AVX Xoroshiro128+ Advancing Assembler

```
vmovdqa ymm1, YMMWORD PTR [rsp]
vpxor   ymm0, ymm1, YMMWORD PTR [rsp+32]
lea     rdi, [rsp+80]
vpsrlq  ymm3, ymm1, 40
vpsllq  ymm2, ymm0, 16
vpsllq  ymm1, ymm1, 24
vpxor   ymm2, ymm2, ymm0
vpor    ymm1, ymm1, ymm3
vpxor   ymm1, ymm2, ymm1
vmovdqa YMMWORD PTR [rsp], ymm1
vpsrlq  ymm1, ymm0, 27
vpsllq  ymm0, ymm0, 37
vpor    ymm0, ymm0, ymm1
vmovdqa YMMWORD PTR [rsp+32], ymm0
vzeroupper
```

Code: AVX Xoroshiro128+ Advancing $\times 2$ Assembler

```
vmovdqa ymm0, YMMWORD PTR [rsp]
vpxor   ymm1, ymm0, YMMWORD PTR [rsp+32]
lea     rdi, [rsp+80]
vpsrlq  ymm3, ymm0, 40
vpsllq  ymm2, ymm1, 16
vpsllq  ymm0, ymm0, 24
vpxor   ymm2, ymm2, ymm1
vpor    ymm0, ymm0, ymm3
vpsrlq  ymm3, ymm1, 27
vpxor   ymm2, ymm2, ymm0
vpsllq  ymm0, ymm1, 37
vpor    ymm0, ymm0, ymm3
vpsrlq  ymm3, ymm2, 40
vpxor   ymm0, ymm0, ymm2
vpsllq  ymm2, ymm2, 24
vpsllq  ymm1, ymm0, 16
vpor    ymm2, ymm2, ymm3
vpxor   ymm2, ymm2, ymm1
vpsrlq  ymm1, ymm0, 27
vpxor   ymm2, ymm2, ymm0
vpsllq  ymm0, ymm0, 37
```

```
vpor    ymm0, ymm0, ymm1
vmovdqa YMMWORD PTR [rsp], ymm2
vmovdqa YMMWORD PTR [rsp+32], ymm0
vzeroupper
```

Code: AVX Xoroshiro128+ Advancing ×4 Assembler

```
vmovdqa ymm0, YMMWORD PTR [rsp]
vpxor   ymm1, ymm0, YMMWORD PTR [rsp+32]
lea     rdi, [rsp+80]
vpsrlq  ymm3, ymm0, 40
vpsllq  ymm2, ymm1, 16
vpsllq  ymm0, ymm0, 24
vpxor   ymm2, ymm2, ymm1
vpor    ymm0, ymm0, ymm3
vpsrlq  ymm3, ymm1, 27
vpxor   ymm2, ymm2, ymm0
vpsllq  ymm0, ymm1, 37
vpor    ymm0, ymm0, ymm3
vpsrlq  ymm1, ymm2, 40
vpxor   ymm0, ymm0, ymm2
vpsllq  ymm3, ymm2, 24
vpor    ymm3, ymm3, ymm1
vpsllq  ymm2, ymm0, 16
vpxor   ymm3, ymm3, ymm2
vpsllq  ymm1, ymm0, 37
vpsrlq  ymm2, ymm0, 27
vpxor   ymm3, ymm3, ymm0
vpor    ymm1, ymm1, ymm2
vpsrlq  ymm4, ymm3, 40
vpxor   ymm1, ymm1, ymm3
vpsllq  ymm2, ymm3, 24
vpsllq  ymm0, ymm1, 16
vpsrlq  ymm3, ymm1, 27
vpor    ymm2, ymm2, ymm4
vpxor   ymm2, ymm2, ymm0
vpsllq  ymm0, ymm1, 37
vpxor   ymm2, ymm2, ymm1
vpor    ymm0, ymm0, ymm3
vpxor   ymm0, ymm0, ymm2
vpsrlq  ymm3, ymm2, 40
vpsllq  ymm2, ymm2, 24
vpsllq  ymm1, ymm0, 16
vpor    ymm2, ymm2, ymm3
vpxor   ymm2, ymm2, ymm1
vpsrlq  ymm1, ymm0, 27
vpxor   ymm2, ymm2, ymm0
vpsllq  ymm0, ymm0, 37
vpor    ymm0, ymm0, ymm1
vmovdqa YMMWORD PTR [rsp], ymm2
vmovdqa YMMWORD PTR [rsp+32], ymm0
vzeroupper
```

9.5 Middle Square Weyl Generator

Code: Scalar MSWS

```
struct msws {
  using uint_type = uint64_t;
  using result_type = uint32_t;
  static constexpr size_t word_size = 32;

  constexpr msws() = default;
  msws(const msws&) = default;
  msws& operator=(const msws&) = default;
  msws(msws&&) = default;
  msws& operator=(msws&&) = default;

  template <typename RNG>
  explicit msws(RNG&& rng)
      : s{(static_cast<uint64_t>(rng()) << 32) | (rng() << 1) | 0x01} {}

  constexpr result_type operator()() noexcept;
  static constexpr result_type min() noexcept { return result_type{}; }
  static constexpr result_type max() noexcept { return ~result_type{}; }

  uint_type s = 0xb5ad4eceda1ce2a9;
  uint_type x = 0;
  uint_type w = 0;
};

constexpr auto msws::operator()() noexcept -> result_type {
  x *= x;
  x += (w += s);
  return x = ((x >> 32) | (x << 32));
}
```

Code: AVX MSWS Structure

```
struct msws {
  using uint_type = uint64_t;
  using result_type = uint32_t;
  using simd_type = __m256i;
  static constexpr size_t simd_size = 8;

  static simd_type _mm256_square_epi64(simd_type x) noexcept;
  simd_type operator()() noexcept;

  template <typename RNG>
  static constexpr uint_type seed(RNG&& rng) {
    return (static_cast<uint_type>(rng()) << 32) | (rng() << 1) | 0x01;
  }
  template <typename RNG>
  explicit msws(RNG&& rng)
      : step{_mm256_set_epi64x(seed(rng), seed(rng), seed(rng), seed(rng))
          ,
            _mm256_set_epi64x(seed(rng), seed(rng), seed(rng), seed(rng))
                },
```

```
        root{_mm256_setzero_si256(), _mm256_setzero_si256()},
        weyl{_mm256_setzero_si256(), _mm256_setzero_si256()} {}

  simd_type step[2];
  simd_type root[2];
  simd_type weyl[2];
};
```

Code: AVX MSWS Advancing

```
inline auto msws::_mm256_square_epi64(simd_type x) noexcept -> simd_type {
  // x = x1 * 2^32 + x_0
  // x^2 = 2 * x_1 * x_2 * 2^32 + x_0^2
  const auto first = _mm256_mul_epu32(x, x);
  const auto second = _mm256_mullo_epi32(x, _mm256_slli_epi64(x, 33));
  return _mm256_add_epi64(first, second);
}

inline auto msws::operator()() noexcept -> simd_type {
  __m256i result[2];

  for (int i = 0; i < 2; ++i) {
    root[i] = _mm256_square_epi64(root[i]);
    weyl[i] = _mm256_add_epi64(weyl[i], step[i]);
    root[i] = _mm256_add_epi64(root[i], weyl[i]);
    root[i] = _mm256_or_si256(_mm256_srli_epi64(root[i], 32),
                              _mm256_slli_epi64(root[i], 32));
    result[i] = root[i];
  }

  return _mm256_blend_epi32(
      _mm256_permutevar8x32_epi32(result[0],
                                  _mm256_set_epi32(7, 5, 3, 1, 6, 4, 2, 0)
                                  ),
      _mm256_permutevar8x32_epi32(result[1],
                                  _mm256_set_epi32(6, 4, 2, 0, 7, 5, 3, 1)
                                  ),
      0b11110000);
  // return _mm256_or_si256(
  //     _mm256_and_si256(result[0], _mm256_set1_epi64x(0xffffffff)),
  //     _mm256_slli_epi64(result[1], 32));
}
```

9.6 Uniform Real Distribution

Code: Scalar Uniform 32bit

```
template <typename Real>
constexpr Real uniform(uint32_t) noexcept = delete;
```

```
template <>
constexpr inline float uniform<float>(uint32_t x) noexcept {
  const auto tmp = ((x >> 9) | 0x3f800000);
  return (*reinterpret_cast<const float*>(&tmp)) - 1.0f;
}

template <>
constexpr inline double uniform<double>(uint32_t x) noexcept {
  const auto tmp = ((static_cast<uint64_t>(x) << 20) | 0
      x3ff0000000000000ULL);
  return (*reinterpret_cast<const double*>(&tmp)) - 1.0;
}
```

Code: AVX Uniform

```
template <typename Real>
inline auto uniform(__m256i) noexcept = delete;

template <>
inline auto uniform<float>(__m256i x) noexcept {
  const auto tmp = _mm256_srli_epi32(x, 9);
  const auto tmp2 = _mm256_or_si256(tmp, _mm256_set1_epi32(0x3f800000));
  return _mm256_sub_ps(_mm256_castsi256_ps(tmp2), _mm256_set1_ps(1.0f));
};

template <>
inline auto uniform<double>(__m256i x) noexcept {
  const auto tmp = _mm256_srli_epi64(x, 12);
  const auto tmp2 =
      _mm256_or_si256(tmp, _mm256_set1_epi64x(0x3ff0000000000000L));
  return _mm256_sub_pd(_mm256_castsi256_pd(tmp2), _mm256_set1_pd(1.0));
}

template <typename Real>
inline auto uniform(__m256i x, Real a, Real b) noexcept = delete;

template <>
inline auto uniform<float>(__m256i x, float a, float b) noexcept {
  const auto scale = _mm256_set1_ps(b - a);
  const auto offset = _mm256_set1_ps(a);
  const auto rnd = pxart::simd256::detail::uniform<float>(x);
  return _mm256_add_ps(_mm256_mul_ps(scale, rnd), offset);
}

template <>
inline auto uniform<double>(__m256i x, double a, double b) noexcept {
  const auto scale = _mm256_set1_pd(b - a);
  const auto offset = _mm256_set1_pd(a);
  const auto rnd = pxart::simd256::detail::uniform<double>(x);
  return _mm256_add_pd(_mm256_mul_pd(scale, rnd), offset);
}
```

9.7 Summary

10 Application to Simulations

11 Evaluation and Results

(*Compiler Explorer*; *Intel Intrinsics Guide*; Leis 2019; Meyers 2014; Vandevoorde, Josuttis, and Gregor 2018)

godbolt google benchmark intel vtune amplifier testu01 dieharder

12 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.

Blackman, David and Sebastiano Vigna (August 1, 2019). "Scrambled Linear Pseudorandom Number Generators". In: *arXiv.org*. URL: https://arxiv.org/abs/1805.01407v2 (visited on 10/26/2019).

Eisner, Tanja and Balint Farkas (January 9, 2019). "Ergodic Theorems". Course Notes to 22. Internetseminar of Virtual Lectures about Classical and Modern Ergodic Theory. URL: https://ergodic.mathematik.uni-leipzig.de/uploads/default/original/1X/74a3d34dfcdce08c3423f3ec62aa43db88abf189.pdf (visited on 10/27/2019).

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

Fog, Agner (2019a). *Calling Conventions for Different C++ Compilers and Operating Systems*. Agner Fog. URL: https://www.agner.org/optimize/calling_conventions.pdf (visited on 11/26/2019).

— (2019b). *Instruction Tables. Lists of Instruction Latencies, Throughputs and Micro-Operation Breakdowns for Intel, AMD and VIA CPUs*. Agner Fog. URL: https://www.agner.org/optimize/instruction_tables.pdf (visited on 11/26/2019).

— (2019c). *Optimizing Software in C++. An Optimization Guide for Windows, Linux and Mac Platforms*. Agner Fog. URL: https://www.agner.org/optimize/optimizing_cpp.pdf (visited on 11/26/2019).

— (2019d). *Optimizing Subroutines in Assembly Language. An Optimization Guide For x86 Platforms*. Agner Fog. URL: https://www.agner.org/optimize/optimizing_assembly.pdf (visited on 11/26/2019).

— (2019e). *The Microarchitecture of Intel, AMD and VIA CPUs. An Optimization Guide for Assembly Programmers and Compiler Makers*. Agner Fog. URL: https://www.agner.org/optimize/microarchitecture.pdf (visited on 11/26/2019).

Godbolt, Matt. *Compiler Explorer*. URL: https://godbolt.org (visited on 11/21/2019).

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.

Guskova, Maria S., L. Yu. Barash, and Lev. N. Shchur (2016). "RNGAVXLIB: Program Library for Random Number Generation, AVX Realization". In: *Computer Physics Communications* 200, pp. 402–405.

Hennessy, John L. and David A. Patterson (2019). *Computer Architecture: A Quantitative Approach*. Sixth Edition. Morgan Kaufmann – Elsevier. ISBN: 978-0-12-811905-1.

Hromkovič, Juraj (2011). *Theoretische Informatik. Formale Sprachen, Berechenbarkeit, Komplexitätstheorie, Algorithmik, Kommunikation und Kryptographie*. 4., aktualisierte Auflage. Vieweg+Teubner — Springer. ISBN: 978-3-8348-0650-5. DOI: 10.1007/978-3-658-06433-4.

Intel. *7th Generation Intel Processor Families for S Platforms and Intel Core X-Series Processor Family. Supporting 7th Generation Intel Core Processor Families, Intel Pentium Processors and Intel Celeron Processors Family for S Platforms Intel Celeron Processors and Intel Core X-Series Processor Platforms, formerly known as Kaby Lake*. URL: https://www.intel.com/content/www/us/en/design/products-and-solutions/processors-and-chipsets/kaby-lake-s/technical-library.html?grouping=rdc%20Content%20Types&sort=title:asc (visited on 11/27/2019).

— *Intel Intrinsics Guide*. URL: https://software.intel.com/sites/landingpage/IntrinsicsGuide/ (visited on 11/21/2019).

— *Intel® Core™ i5-8250U Processor*. URL: https://ark.intel.com/content/www/us/en/ark/products/124967/intel-core-i5-8250u-processor-6m-cache-up-to-3-40-ghz.html (visited on 11/30/2019).

— *Intel® Core™ i7-7700K Processor*. URL: https://ark.intel.com/content/www/us/en/ark/products/97129/intel-core-i7-7700k-processor-8m-cache-up-to-4-50-ghz.html (visited on 11/30/2019).

— (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.

Leis, Viktor (2019). *PerfEvent*. URL: https://github.com/viktorleis/perfevent (visited on 11/21/2019).

Lemire, Daniel and Melissa E. O'Neill (2019). "Xorshift1024*, Xorshift1024+, Xorshift128+ and Xoroshiro128+ Fail Statistical Tests for Linearity". In: *Journal of Computational and Applied Mathematics* 350, 139–142. ISSN: 0377-0427. DOI: 10.1016/j.cam.2018.10.019.

Marsaglia, George et al. (2003). "Xorshift RNGs". In: *Journal of Statistical Software* 8.14, pp. 1–6. DOI: 10.18637/jss.v008.i14.

Matsumoto, Makoto and Takuji Nishimura (1998). "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator". In: *ACM Transactions on*

Modeling and Computer Simulation (TOMACS) 8.1, pp. 3–30. DOI: 10.1145/272991.272995.

Meyers, Scott (2014). *Effective Modern C++*. O'Reilly Media. ISBN: 978-1-491-90399-5.

Müller-Gronbach, Thomas, Erich Novak, and Klaus Ritter (2012). *Monte Carlo-Algorithmen*. Springer. ISBN: 978-3-540-89141-3. DOI: 10.1007/978-3-540-89141-3.

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).

Panneton, François, Pierre L'ecuyer, and Makoto Matsumoto (2006). "Improved Long-Period Generators Based on Linear Recurrences Modulo 2". In: *ACM Transactions on Mathematical Software (TOMS)* 32.1, pp. 1–16. DOI: 10.1145/1132973.1132974.

Patterson, David A. and John L. Hennessy (2014). *Computer Organization and Design. The Hardware/Software Interface*. Fifth Edition. Morgan Kaufmann – Elsevier. ISBN: 978-0-12-407726-3.

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

Vandevoorde, David, Nicolai M. Josuttis, and Douglas Gregor (2018). *C++ Templates: The Complete Guide*. Second Edition. Addison-Wesley – Pearson Education. ISBN: 978-0-321-71412-1.

Vigna, Sebastiano (2016). "An Experimental Exploration of Marsaglia's Xorshift Generators, Scrambled". In: *ACM Transactions on Mathematical Software (TOMS)* 42.4, p. 30. DOI: 10.1145/2845077.

— (2017). "Further Scramblings of Marsaglia's Xorshift Generators". In: *Journal of Computational and Applied Mathematics* 315, pp. 175–181. DOI: 10.1016/j.cam.2016.11.006.

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

Waldmann, Stefan (2017). *Lineare Algebra 1. Die Grundlagen für Studierende der Mathematik und Physik*. Springer Spektrum. ISBN: 978-3-662-49914-6. DOI: 10.1007/978-3-662-49914-6.

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, November 30, 2019

Markus Pawellek