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, December 9, 2019

Abstract

The topic of this thesis is the development of a software library, called "pxart", in the C++ programming language that makes it possible to improve the performance of physical simulations that need to use random numbers. The library consists of pseudorandom number generators exploiting the SSE and AVX instruction set facilities of modern Intel CPUs. Therefore, we created a new interface to simplify their initialization and usage. Through the implementation of appropriate benchmarks and physically-based applications, like photon propagation and the Ising model, we were able to show that our generators could indeed accelerate the execution of randomized algorithms.

Acknowledgments

I am grateful to Joachim Gießen and Bernd Brügmann for making it possible to write a thesis about an interdisciplinary topic in mathematics, physics and informatics. I thank Mark Blacher for his supervising, his helpful suggestions, and for our interesting discussions. Also great thanks to Clemens Anschütz for programming assistance and Kerstin Pawellek for assisting in writing the thesis and proofreading.

Contents

List of Figures

List of Tables

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 (Pharr, Jakob, and Humphreys 2016). 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 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.

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

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 The C++ Programming Language

As already told in the introductory section 1, the C++ programming language is the perfect candidate for developing high-performance low-level structures while keeping the high degree of abstraction that makes the use of libraries easier and more consistent. C++ features multiple programming styles, like procedural programming, data abstraction, object-oriented

programming, as well as generic programming, also known as template metaprogramming (Stroustrup 2014; Vandevoorde, Josuttis, and Gregor 2018). The type mechanism makes C++ a strongly typed language. To exploit this, we will always try to map problems to an abstract data structure. Furthermore, the built-in facilities of C++, such as templates, function overloading, type deduction, and lambda expressions, simplify the type handling and the generalization of algorithms. Additionally, C++ provides a standard library, called the standard template library (STL), consisting of header files providing default templates to use for a wide variety of problems. In this thesis, we will rely on the random utilities the STL exhibits in its header `random`. We will also assume basic knowledge in C++ and refer to Stroustrup (2014) and Meyers (2014) for a detailed introduction to the general usage of the language. A complete online reference of the language is given by *cppreference.com*.

The C++ programming language evolves over time by defining different language standards every three years which are published by the ISO C++ standardization committee. Newer standards typically introduce modern language features, fix old behavior and add new algorithms and templates to the STL. Hence, modern C++ separates into the standards C++11, C++14, and C++17 each published in the year 2011, 2014, and 2017, respectively. Currently, we are waiting for the C++20 standard specification to be able to use even more advanced features concerning template metaprogramming and concurrency. At the time of writing this thesis, C++17 is most modern specification and as a consequence it will be used throughout the code. (Meyers 2014; Stroustrup 2014; Vandevoorde, Josuttis, and Gregor 2018)

To design the API of a library supplying vectorized RNGs and some advanced utilities, we will heavily rely on different template metaprogramming techniques. For getting deeper into the topic, we refer to Vandevoorde, Josuttis, and Gregor (2018) and again to Meyers (2014). Here, we will only be able to list the most important terms, techniques and rules that will be used throughout the code.

Template Argument Deduction "In order to instantiate a function template, every template argument must be known, but not every template argument has to be specified. When possible, the compiler will deduce the missing template arguments from the function arguments. " (*cppreference.com*)

Overloading Function Templates "Like ordinary functions, function templates can be overloaded. That is, you can have different function definitions with the same function name so that when that name is used in a function call, a C++ compiler must decide which one of the various candidates to call. " (Vandevoorde, Josuttis, and Gregor 2018)

Variadic Templates "Since C++11, templates can have parameters that accept a variable number of template arguments. This feature allows the use of templates in places where you have to pass an arbitrary number of arguments of arbitrary types. " (Vandevoorde, Josuttis, and Gregor 2018)

Perfect Forwarding C++11 introduced so-called move semantics to optimize specific copy operations by moving internal resources instead of creating a deep copy of them. Perfect forwarding is a template-based pattern that forwards the basic properties of a type concerning its reference and modification type.

SFINAE Substituting template arguments to resolve function template overloading could lead to errors by creating code that makes no sense. The principle "substitution failure

is not an error" (SFINAE) states that in these circumstances the overload candidates with such substitution problems will simply ignored.

This is a specifier introducing an unevaluated context from which it is deducing the type of the given expression without actually evaluating the expression.

This is a helper template from the STL to ignore function templates by using SFINAE under certain compile conditions. If the template argument evaluates to true then `std::enable_if_t` will evaluate to an actual type. Otherwise, it will have no meaning triggering the SFINAE principle for overloads.

The given STL function template is only declared, not defined and therefore cannot be called in evaluated contexts. It can be used as a placeholder for an object reference of a specific type. Typically, this routine will be inserted instead of a default constructor in the unevaluated context argument of `decltype`.

`decltype` `std::declval<T>()`

Type Traits: Type traits are general functions defined over types to modify or evaluate them. In the STL a typical examples is given by `std::is_same_v` which is evaluating if two types are the same.

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.

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

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

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

□

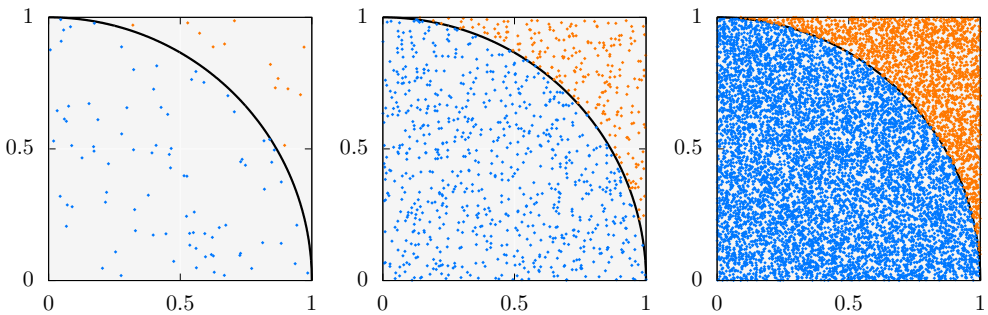
In the proof, we basically applied the lemma about the direct simulation. So we get the same convergence rate for the expectation value with respect to the standard deviation. To get a deeper understanding of this method, consider the estimation of π by Monte Carlo integration. For this, we would like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

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

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



(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.

using the above lemma 3.2.

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

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

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

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. Furthermore, figure 2 shows that the error of the estimation can also be estimated and will be more precise for a larger sample count.

Later, we will use the computation of π as benchmark routine to measure the performance of PRNGs with respect to different aspects of their implementation. In these cases, the error of π should be in some given range Assume we want to use 10^8 samples to estimate the value of π. According to the formula above for the standard deviation, we get an error of approximately $0,00016$ which means we should at least get a precision of four digits, such that π should approach the value 3.1415 with a varying last digit.

To use the described Monte Carlo integration for estimating π, an actual implementation in C++ is needed. The following code snippet provides the basic algorithm relying on the random utilities given by the standard template library (STL) of the C++ programming language. Because C++ is a strongly typed language which is working with templates for type abstraction, the given code uses templates to generalize the usage of different RNG types, as well as different real and integer number types. Typically, we will use the `float` type as the real number type and the `int` type as the integer type.

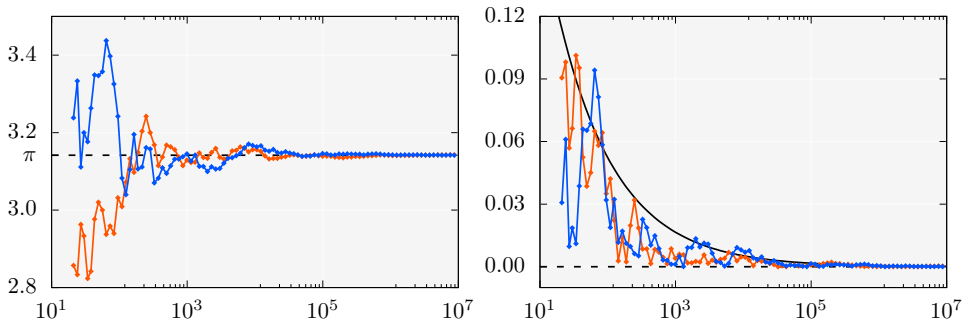


Figure 2: Each of the diagrams shows two different versions, colored in orange and blue, of realizations of the Monte Carlo integration $\mathrm{MCI}_n(f)$ for the computation of π for different values of n. The left one displays the estimated value of π and the right one displays its relative error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

Code: monte_carlo_pi.hpp

```
#pragma once
#include <random>

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);
    samples_in_circle += (x * x + y * y <= 1);
  }
  return static_cast<Real>(samples_in_circle) / samples * 4;
}
```

The implementation of the algorithm does not introduce any irregularities and can directly be deduced from the mathematical formulation. First, we define the standard uniform distribution for the RNG and an integer number `samples_in_circle` as zero which will be used to count the number of samples that lie inside the unit circle. In the following `for` loop every run will construct two uniformly distributed random numbers which together will define the position of random two-dimensional point in the unit square. We then evaluate the circle condition adding it to `samples_in_circle` resulting in an incrementation by one if it is true and zero otherwise. At the end, the code computes the correct result by calculating the division and scaling the output.

The computation of π is only an academic example that should not be used in reality because there are much more efficient ways to estimate it. But as we will see, it is perfectly suited as a benchmark to test different kinds of RNGs because the main part of the algorithm consists of generating a lot of random numbers while using their values to actually compute a result. Evaluating the circle condition for generated random numbers is fast and will not introduce a lot of bias in the actual measurement. Besides, it is the most simplest example of a non-trivial Monte Carlo integration. A lot of physical problems rely on Monte Carlo integration. In the end, all these problems can be broken down into similar form as the computation of π.

3.2 Metropolis-Hastings Algorithm and the Ising Model

3.3 Photon Propagation and Physically Based Rendering

In computer graphics, the rendering of realistic images based on physical principles requires the simulation of global illumination effects. Global illumination is formally described by the so-called rendering equation, also known as light transport equation (LTE). There exist several different formulations of the LTE, such as the surface form and the path integral formulation, which are all equivalent and are used to derive advanced methods to actually estimate the light distribution in space consisting of differing objects. The LTE can be derived by applying principles of geometric optics to the law of the conservation of energy for electromagnetic radiation. A detailed explanation is given in Pharr, Jakob, and Humphreys (2016) and a mathematically rigorous discussion can be read in Pawellek (2017).

Typically, the LTE cannot be evaluated analytically for complex geometries. As a consequence, there are multiple famous simulation strategies, like path tracing, bidirectional path tracing, metropolis light transport, and photon mapping, to approximate its solutions. All of these strategies have in common that they are somehow estimating the light distribution in space through Monte Carlo methods. Because of the integral appearing inside the equation, all algorithms make great use of Monte Carlo integration in different ways. Path tracing, for example, is using the path integral formulation of the LTE by importance sampling different paths of light through the scene starting from the observer and ending at a light source. At every vertex of the path, Russian roulette decides if the ray has to be reflected, transmitted or absorbed. In general, the algorithm has to compute several hundreds of sample paths for every pixel on the screen to reduce the noise and get an acceptable estimation of the actual light distribution. As a consequence, the tracing of rays and the propagation of photons through a scene needs to use a really large amount of random numbers.

The tracing of light rays is strongly connected to photon propagation in space. From an abstract point of view, we can think of light rays as single photons or packets of photons. Light rays are used to gather the radiance arriving at the observer whereas photons propagate flux from a light source. Both of them adhere to the same laws even if they are interpreted differently. Hence, photon tracing works exactly the same way as ray tracing. A usual application of this phenomenon is the photon mapping algorithm. The movement and transmission of differing photons or rays does not depend on each other. Due to the high amount of random numbers needed and the large degree of independence the simulation of global illumination by using Monte Carlo methods is a perfect candidate to measure the performance improvement by applying vectorized PRNGs. This claim is even supported by an already existing strong usage of SIMD instructions for the current most efficient ray and path tracing engines (Intel 2019b).

In this thesis, it is not possible to provide a complete ray tracing framework to test the application of vectorized PRNGs. But we are able to construct a simplified version which is using the developed PRNGs to simulate the propagation of photons. This smaller simulation focuses on the most important physical effects and ignores subtleties that would introduce bias in the performance measurements. We will provide the physical background concerning photon propagation.

Surface Interaction

The interaction of photons with surfaces of objects is strongly associated with the properties of the materials the objects consist of. Typically, the behavior can be modeled by bidirectional scattering distribution functions (BSDFs) which give the portion of light scattered in the outgoing direction with respect to the incoming direction. Concerning photons, for a fixed incoming direction BSDFs can be thought of as probability densities over the outgoing direction of the photon after the surface scattering has taken place. We refer again to Pharr, Jakob, and Humphreys (2016) and Pawellek (2017).

To simulate complex optical properties of object materials BSDFs have to sampled with the usage of appropriate random variables.

We will only assume ideal reflection and refraction.

Volume Scattering

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 2017a,b).

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).

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

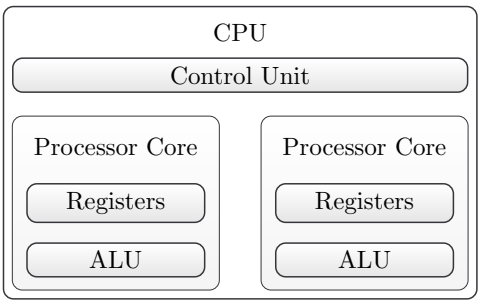


Figure 3: The figure shows the basic components of a typical CPU with multiple processing cores in an hierarchical order. There is only one control unit which is handling communication between different cores by coordinating the execution of program instructions. Every processor core employs its own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

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 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

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.

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. (Dolbeau 2016; Fog 2019c)

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

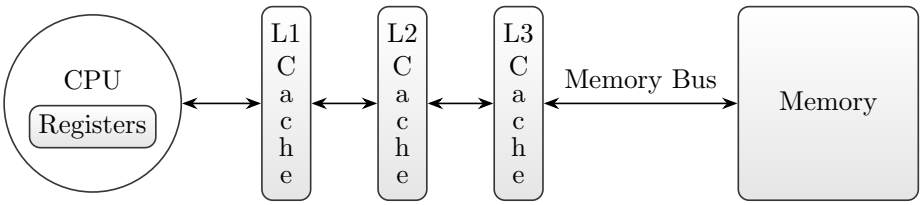


Figure 7: The figure shows a scheme of the non-persistent part of the memory hierarchy for a modern laptop or desktop computer. Modeled after Hennessy and Patterson (2019, p. 79).

sequence of bytes. Each byte can be altered by program execution through storing instructions. (Patterson and Hennessy 2014)

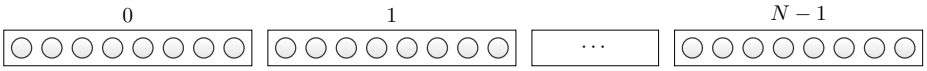


Figure 6: This figure visualizes memory with $N \in \mathbb{N}$ bytes as a sequences of bits where each byte can be referenced by its memory address.

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 (Intel 2018a). 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)

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`. (Intel 2019a)

5 Pseudorandom Number Generators

5.1 Random Sequences

In the above section **??** 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 2018b).

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 2018b). 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 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

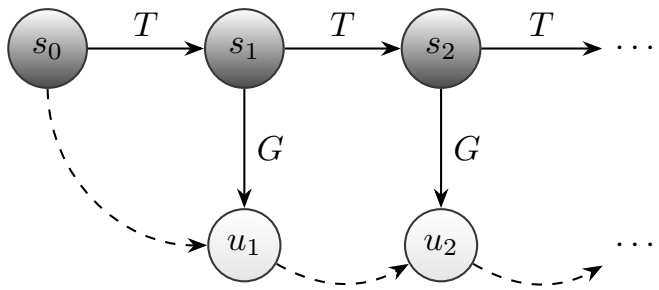


Figure 8: The figure shows a scheme about the generation of a pseudorandom sequence for a given PRNG $\mathcal{G} := (S, T, U, G)$ and seed value $s_0 \in S$. The internal state is advanced by the transition function T through a recurrence relation. To get an output value for the pseudorandom sequence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

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 2018b).

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. □

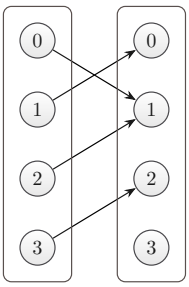
Thus, every pseudorandom sequence will repeat itself after it reached a certain point. The period and the transient are greatly affected by the number of states and the transition function of the PRNG. To get a better insight, we will examine the following idealized examples with different transition 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.

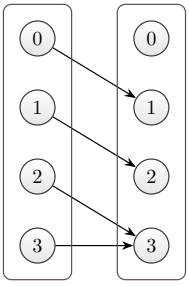
$$(u_n) = u_1 \dots u_{\tau-1}\overline{u_\tau \dots u_{\tau+\rho-1}}$$

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.



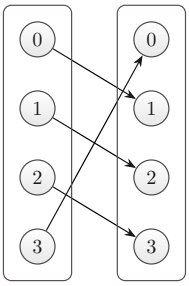
$$T(x) := \begin{cases} 1 & : x \in \{0,2\} \\ 0 & : x = 1 \\ 2 & : x = 3 \end{cases}, \qquad (u_n) = \begin{cases} \overline{10} & : s_0 \in \{0,2\} \\ \overline{01} & : s_0 = 1 \\ 2\overline{10} & : s_0 = 3 \end{cases}$$

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.



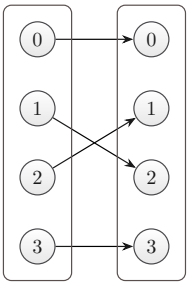
$$T(x) := \begin{cases} x+1 & : x < 3 \\ 3 & : x = 3 \end{cases}, \qquad (u_n) = \begin{cases} 12\overline{3} & : s_0 = 0 \\ 2\overline{3} & : s_0 = 1 \\ \overline{3} & : s_0 \geq 2 \end{cases}$$

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.

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

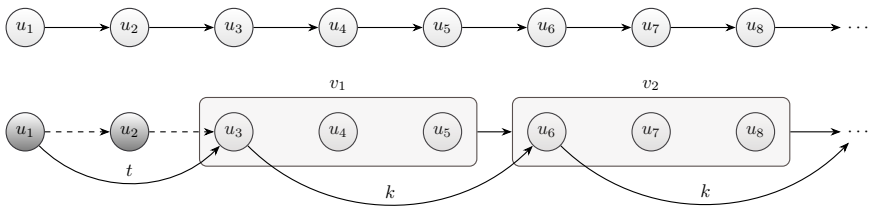


Figure 9: The upper part of the figure shows a schematic view of an arbitrary sequence of values $(u_n)_{n\in\mathbb{N}}$ in an arbitrary non-empty set U. The lower part visualizes the corresponding k-dimensional vector sequence $(v_n)_{n\in\mathbb{N}}$ with translation t, whereby $k = 3$ and $t = 2$. 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.

> *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.

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

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

In the last few years, the vectorization of PRNGs has not received great attention. We will give a brief overview of the major contributions concerning the implementation of vectorized PRNGs.

Because the SIMD routines are a form of data-level parallelism, their application to PRNGs has to exploit the facility of multiple streams for small generators to take full advantage of the whole size of vector registers. According to Fog (2015), there is a number of techniques to initialize multiple, independent pseudorandom streams for vector registers. One of it uses multiple instances of the same generator with different seeds. This method can lead to overlapping subsequences. Fog calculates the probability of this to happen and gives an example of when to ignore a possible overlap. Some PRNGs offer a jump-ahead feature which can be used to initialize all instances with only one seed. Each stream will then be a non-overlapping subsequence of the same generator with certain length. There are families of PRNGs that are described by the same algorithm only differing in their underlying parameters. Using different sets of parameters for every instance will generate independent streams. If the PRNG has a larger state size, like the MT19937, we do not have to deal with multiple instances of the same generator. Instead, multiple streams can be generated by computation of consecutive elements. Fog compares these methods and gives a general advice. Furthermore, he explains that it will be better to combine different generators for superior statistical properties. But this approach has no direct impact on vectorization techniques as we first have to discuss the implementation of the stand-alone generators. Combining those will then be a trivial operation that will be not be discussed here.

Saito and Matsumoto (2008) presented an alternative algorithm, called SFMT, to the widely used Mersenne Twister optimized for the SSE instruction set which is working with $128\,\mathrm{bit}$ data types. The authors have been one of the first people to officially vectorize PRNGs. They provide a library for academic usage and could successfully show that their generation scheme is much more efficient when using SSE based instructions (Saito and Matsumoto 2017). On the other hand, processor architectures are evolving. Using the SFMT on modern hardware with AVX or even AVX512 support will not make full use of the complete vector registers because the algorithm is specialized for the older SSE registers.

With *Intel® Math Kernel Library*, Intel provides a high-performance library for C and C++ exploiting the features and microarchitectures of Intel processors. Besides optimized mathematical functions, they also provide several different implementations of vectorized PRNGs ready to be used after linking the library. Especially we get variants of the standard Mersenne Twister MT19937 and the SFMT to name the most common generators. Because they develop their own CPUs, Intel is able to create fine-tuned code for nearly every given microarchitecture resulting in a high-quality of their implementations. However, the source code of the library is not open source and cannot be easily adjusted for differing architectures that are not built by Intel. Furthermore, the interface to use the routines of the library tend to be complicated and due to their backwards compatibility it is unlikely that this will change.

Barash, Guskova, and Shchur (2017) describes how to implement a lot of PRNGs by using the AVX instruction set. They as well provide a library for academic use with an implementation of the MT19937 even surpassing Intel's implementation (*RNGAVXLIB*; Guskova, Barash, and Shchur 2016). But the interface of the library is again C-compatible and introduces an overhead in development time while using C++. Barash, Guskova, and Shchur manually

vectorize their code through inline assembly statements which make their code unreadable, difficult to understand and not generalizable to other SIMD architectures.

In Lemire (*simdpcg*) and Lemire (*SIMDxorshift*), Lemire supplies two open source code snippets for the vectorization of a PCG and a Xorshift generator with the AVX and AVX512 instruction sets by using Intel intrinsics. Therefore the code is well suited for analyzing the application of SIMD intrinsics to PRNGs and comparing different vectorization schemes. His implementations are not tested and not well documented.

Vigna (2018) presents a modern PRNG with good statistical properties. In his explanation, Vigna talks about triggering automatic vectorization by using four independent streams of the named generator. He mentions, that it sometimes seems to be difficult to successfully force the compiler to generate vector intrinsics. As a consequence, a manual vectorization technique should be much more appropriate as we will show in the next sections.

According to the explanations and descriptions given, we have chosen to vectorize another variant of the MT19937 with the SSE and AVX instruction set because, even if it shows some statistical flaws and has a big state, it is the de facto standard for current applications. Additionally, the MT19937 is reliable, has an extremely long period, can be used for multidimensional simulations and will perfectly serve as an academic example on how to vectorize a PRNG. Furthermore, we will show how to implement two PRNGs with the SSE and AVX instruction set, namely the Xoroshiro128+ and the MSWS, which were not vectorized yet. The idea is to raise the performance of the Xoroshiro128+ in comparison to an automatic vectorization process. The MSWS is non-linear but fast and modern generator for which we have to use a slightly different vectorization technique. All this will give us an insight into the vectorization of PRNGs.

7 Design of the API

The design of an API for a library making the use of vectorized PRNGs possible consists of multiple parts. To exploit the statistical properties of all the implemented generators proper seeding routines have to be provided. Seeding of PRNGs should be easy but not magically hiding important information, like the RNG that is used for the initialization. The implementation of the MT19937 given by `std::mt19937` in the STL of the C++ programming language for example, internally uses an easier PRNG to initialize its whole state vector by only one truly random number (GCC 2019a). The interface of `std::mt19937` does not show this information to the programmer or the reader and even forces the user of the library to employ this process. Furthermore, a generator `RNG` only used for seeding will typically constructed directly as an rvalue in the constructor of the `std::mt19937`.

```
std::mt19937 rng{RNG{}()};
```

Because we cannot use the complete RNG as an argument, we first have to make sure that we call the advancing routine of the seeding generator. This behavior reduces the statistical performances of the initialization and complicates the interface even more by adding an extra pair of parentheses.

Another part we have to deal with are fast and easy-to-use distributions for RNGs to at least generate uniform random numbers. The current C++ standard supplies different distributions in form of functor templates that use the same algorithm for all generators by templatizing their function operators.

```
std::uniform_real_distribution<float> dist{}; auto x = dist(rng);
```

This API design consists of two major drawbacks. The user always has to first construct the object of the according distribution type and afterwards using its function operator to actually get a random number that fulfills the conditions of the distribution. Hence, even for simple use cases an API overhead emerges by utilizing random numbers via the STL of C++. The second problem is that every RNG has to use the same distribution routine which may not be optimized to work well with a given generator. Especially for uniformly distributed random numbers, some generators can exploit the implementation of their transition and generator function to achieve better performance. But in C++, we are not able to specialize the function operator template in a distribution functor from the outside. The approach does not scale well with the number of different RNGs.

In the last few years, there has been some development in the area of API design for RNGs. Klammler (2016), O'Neill (2015b,c), and Song and O'Neill (2016), to name a few, provide perfect basic ideas to improve the usage of random utilities in the C++ language. We will rely on this work and want to go a step further. We strive for a transparent API that can be used easily with different seeding alternatives. Nearly every RNG should be available as a source for seeding another RNG. In addition, uniform distribution functions will be introduced that do not need to be called as functor objects. Through template meta programming, we give generators the possibility of specializing such helper functions through member functions of the same name. This idea makes it even possible to specialize these functions outside of the RNG structure for a non-intrusive adjustment. Furthermore, by providing those functions we keep compatibility to the standard distributions.

7.1 Utilities

The ideas for the API given above require us to rely on the template meta programming facilities C++ provides. In general, we would like to use helper functions, as well as member functions for specific tasks, like generating a vector of random numbers. If a generator provides a member function the helper function should use the given specialization whereas otherwise a default algorithm should be used. Hence, at compile-time we need to evaluate if calling a specific member is a valid expression. In Vandevoorde, Josuttis, and Gregor (2018), there is given a technique that allows us to evaluate the validity of expressions.

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)){};
  };
};
```

This code makes highly use of template meta programming artifacts, like SFINAE and type traits. First in the details, it uses an overloaded variadic helper function template `detail::is_valid` to decide if the call of a specific function type `F` with the varying amount of argument types `Args` would be a valid call. The call is put into an unevaluated context by using the `std::declval` template utility from C++ which is able to return an object of a class without specific constructors in such an unevaluated context. To decide the validity of this statement it relies on a default template parameter and the `decltype` utility of the C++ language. If the call to the given function object would be possible, `decltype` will indeed deduce the return type of the function call and correctly set the default template argument. In this case, the first overload would be chosen so that the return type of `detail::is_valid` would be `std::true_type`. In all other cases, the second overload with the return type `std::false_type` would be activated because due to SFINAE the first overload would not be taken into account. We have to make sure to decide on the validity with the generalized boolean types `std::true_type` and `std::false_type` from the STL of C++ because we will use the given templates in unevaluated contexts where no evaluation of an actual `bool` variable would be possible.

The routine `is_valid` is actually a lambda expression that returns another lambda expression which is then evaluating the validity of its arguments by the helper template `detail::is_valid`. The function object `f` will be a lambda expression that lists conditions for abstract types which have to be valid to fulfill the concept we want to define. The arguments `args` will then be the actual variables to be inserted in the pattern test of `f`. The two interleaved lambda expressions

separate the function object `f` from its varying amount of arguments `args` for convenience. Additionally, they allow us to use variables in unevaluated contexts.

To better understand the utility, we will directly apply it to construct a helper function template `generate` which is calling `std::generate` from the STL of C++ for a given RNG as a default. But as a specialization we would like to call the member function with the same name of the generator if it exists. We do this again through the use of SFINAE.

Code: Generate Utility

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

template <typename RNG, typename ForwardIt>
constexpr auto generate(RNG&& rng, ForwardIt first, ForwardIt last)
    -> std::enable_if_t<!decltype(has_generate(rng, first, last)){}()> {
  std::generate(first, last, std::ref(rng));
}

template <typename RNG, typename ForwardIt>
constexpr auto generate(RNG&& rng, ForwardIt first, ForwardIt last)
    -> std::enable_if_t<decltype(has_generate(rng, first, last))::value> {
  std::forward<RNG>(rng).generate(first, last);
}
```

First, we are defining a general pattern `has_generate` based on `is_valid` to be able to decide the validity of the expression `x.generate(y,z)` for arbitrary types by using a lambda expression. Again, we have to make sure the lambda expression is deducing its return in an unevaluated context. Hence, `decltype` is used as a trailing return type with the pattern to evaluate. Next, we declare an overloaded helper function template `generate`. SFINAE is applied for overload resolution by using `std::enable_if_t` in the return type of the functions. In the template argument, we use the general pattern `has_generate(rng,first,last)` and deduce its type by `decltype`. The type will either be `std::true_type` or `std::false_type`. In the first case, `std::enable_if_t` will be evaluated to be the type `void`. In the other case, `std::enable_if_t` will be no valid type and the function will be not be taken into account in the overload resolution. The overloads of `generate` use complementary conditions in the template argument of `std::enable_if_t`, so that at all times only one of them will be available in the overload resolution. As a result, `generate(rng,first,last)` will call `rng.generate(first,last)` if the type of `rng` provides the named member function and `std::generate(first,last,std::ref(rng))` otherwise.

7.2 Seeding

Seeding of RNGs through the use of other RNGs will be accomplished by the use of perfect forwarding. In contrast to the ideas in Klammler (2016), we allow for rvalue initialization making the API simpler. Consider the following code snippet that introduces an arbitrary RNG type `rng_type`.

Code: Seeding Strategy

```
struct rng_type {
  // ...
  template <typename RNG>
  explicit rng_type(RNG&& rng);
  // ...
};
```

`rng_type` is a structure with an explicit forwarding constructor available for seeding which is able match any other RNG type. Please note the typical usage of a universal reference to deduce forwarding types. The implementation of the constructor should call either the `generate` method or the function operator of the given RNG type to initialize its own state. For a few exceptional cases, the given code can lead to certain artifacts in the template deduction process. This behavior is discussed in Meyers (2014, pp. 188–197). For RNGs, there is typically no need to provide an extra degree of type security. But we can easily add it through the use of `std::enable_if_t` to turn on the SFINAE process.

Code: Advanced Seeding Strategy

```
struct rng_type {
  // ...
  template <  //
      typename RNG,
      typename = std::enable_if_t<!std::is_same_v<rng_type, std::decay_t<RNG>>>>
  explicit rng_type(RNG&& rng);

  rng_type(const rng_type&) = default;
  rng_type& operator=(const rng_type&) = default;
  rng_type(rng_type&&) = default;
  rng_type& operator=(rng_type&&) = default;
  // ...
};
```

This implementation removes the explicit forwarding constructor for seeding when we want to copy or move the state of the structure itself. For the implementations of PRNGs, we have tested both variants and could not spot any differences. Hence, further we will use the simpler variant. According to Vandevoorde, Josuttis, and Gregor (2018), in the future of C++ concepts will be available which will make such an initialization artifact even simpler and more efficient.

7.3 Distributions

The design of a uniform distribution `uniform` via helper functions and member specializations will be handled the same way as for the function template `generate`. This time, we only have to take care of additional overloads. By calling `uniform` without function arguments, we have to provide the return type as a template argument. For floating-point types, the result should be a number in the interval $[0, 1)$. Specifying integer number types the output will use the complete range of the template argument. If we call `uniform` with additional arguments specifying the range of the output, we are able to use template argument deduction for the return type based

on the range arguments. We separate those two cases to further optimize the frequently used first function call without arguments. The following code shows the implementation.

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);
}
```

We use two `is_valid` patterns to describe the existence of `uniform` member functions with and without additional arguments. The actual `uniform` functions then use the same overload resolution via SFINAE as the `generate` function.

The result type of the Mersenne Twister `std::mt19937` from the STL is given by `uint_fast32_t` which, on modern machines, will evaluate to `uint64_t` with 64 bit size instead of the used 32 bit. Because the uniform distribution utilities that we will provide in the implementation use their argument types to overload themselves, the Mersenne Twister would call the wrong function. We can prohibit this behavior without changing the inner structure of the `std::mt19937` type by creating another helper function overload especially for the Mersenne Twister. In this implementation, we are making sure its return type will be casted to the `uint32_t` type.

The defined API gives us the ability of a powerful default distribution together with the possibility of specializing it for some generators through member functions. If we cannot

access the inner structure of an RNG then we may even insert slightly more complicated helper specializations to adjust the output of RNGs not created by the programmer itself.

7.4 Algorithms

7.5 PRNG Concept

We do not want to introduce a lot of overhead to the design concept of PRNGs and vectorized PRNGs. The compiler should automatically find out which algorithm to use. This can be done by specialization according to the result type of the function operator. If this should be not enough, tag dispatching can be used.

The output could have arbitrary size but it is always interpreted as a finite length bit stream. For AVX this means a $256\,\mathrm{bit}$ stream. This explains how to interleave SIMD streams and how to test them.

Interfaces for SIMD intrinsics will always be specialized templates even for wrapper classes because SIMD is behaving differently to SISD architectures. If we are working with SIMD vector the least surprising interface is to use the SIMD vector as result type directly. Through inheritance we or someone from the outside could add another interface by using wrapper classes without changing the implementation. Therefore seeding will be a specialized routine.

8 Implementation of Vectorized PRNGs

In the section 5, we have already discussed the mathematical foundations of PRNGs, as well as some examples, like the MT19937. In this section, we will first implement a scalar variant of the chosen generators to better understand their structure and design. Moreover, they serve as a testing facility to make sure the output of the vectorized PRNGs is correct. After explaining the critical points in the implementation, we will show how to vectorize the given generators. For this, it is helpful to first discuss which technique to use for the vectorization. Afterwards a theoretical analysis of the used vector intrinsics concerning their latency and throughput will follow to pinpoint bottlenecks of the implementation possibly resulting in future work.

8.1 Mersenne Twister

The MT19937 is de facto standard of applications using random numbers to compute their results. In the original paper, Matsumoto and Nishimura have given a standard implementation in the C programming language. This implementation is easily adjustable to work in C++. To map the C-style functional programming to modern C++ paradigms, we introduce a functor that stores the complete state of the Mersenne Twister. The functional operator is then used to represent the application of the transition and generator function to receive a new random number. The introduction of the functor grants us the complete power of the C++ template, type, and class system while preserving an easy-to-use interface.

In Kneusel (2018), a special seeding routine was used, such that the actual initialization process only needs one truly random number. In the STL of the C++ programming language the MT19937 is already implemented. The corresponding class is called `std::mt19937` and offers a constructor with only one integral number of 32 bit length as seeding value. This constructor uses the initialization routine described by Kneusel (2018). We want to make sure, that our implementation gives exactly the same output as the standard variant while keeping the advantages of our own API. As a consequence, we have to introduce a structure which can be interpreted as a PRNG and should serve as a default seeder for the MT19937.

Additionally, the types and parameters of the MT19937 will be defined as compile-time constants inside the functor to make sure the code stays maintainable by not inserting magic numbers in the implementation. To guarantee that there is no runtime overhead, we defined those variables as `static constexpr`.

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;
};
```

Implementing the default seeder is done the same way as the MT19937 itself. First, we introduce a functor with the appropriate state while using the function operator as advancing routine. The initialization from Kneusel (2018) was slightly changed to adapt to the new functor definitions. By designing a fast-forward constructor, it is possible to seed our own MT19937 with any other RNG including the default seeder.

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{}} {}
```

Advancing the state of the MT199937 by asking for a random number is a more complicated algorithm. The recurrence of the Mersenne Twister is linear and gives us an equation on how to get to the next element based on the 624 current elements. According to the mathematics, after the computation we would have to move every element of the state one step further. But accessing each element in the state vector for every random number would greatly reduce the performance due to cache misses and the time to move every element. Instead, Matsumoto and Nishimura cache the index of the current element in the state. While the cached index is small enough, advancing the state only results in incrementing the index and returning the state value at this position by applying the generator function. Only if the index reaches the end of the state vector the complete state vector will be regenerated with respect to the recursive relation. In our code, the regeneration of the state vector uses a lambda expression to reduce code duplication and improve code transparency. The lambda expression keeps the code local in contrast to helper or member function. Hence, the compiler will probably inline all calls to the lambda expression optimizing the scalar code in the best possible way.

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;
}
```

The design of our Mersenne Twister ensures that the output will be the same as the output of the MT19937 from the STL of the C++ programming language. Additionally, the interface provides a more general seeding variant which improves statistical performance for seeding generators other than the default seeder. The new initialization facility is easier to use and gives a better insight into the used seeding values to the reader of the code. The function to advance the state is easier to understand, maintainable and generalizable. Last but not least, as we will show, our MT19937 implementation is able to provide a better performance in an actual application.

Because the state of the MT19937 is very large in comparison to the size of the SSE and AVX vector registers, we rely on a typical vectorization technique described by Fog (2015) that does not need to initialize multiple instances of the same generator. The scalar

implementation offers a large degree of data independence. After the regeneration of the complete state vector, the generator function has to be applied on each individual element to provide random numbers. This problem describes the perfect use case for SIMD intrinsics. In the AVX case, we can read eight $32\,\mathrm{bit}$ values at the same time by the usage of a vector register and apply the generator function on each element in parallel by using the respective intrinsics. In the SSE case, we would only read four elements. The number of elements in the state vector is divisible by the eight and four without remainder. Hence, we do not have to take care of the boundaries of the state vector.

The moment every package of eight or four elements in the state vector was read, the regeneration has to be triggered. The computation of the new state vector exhibits some data dependencies we have to take care of. In the scalar variant of the regeneration process the loop of all elements was split into three parts — one for the first 227 elements, one for the following 396 elements and one for the last element. The splitting followed from the dependencies given by the recurrence defining the Mersenne Twister algorithm. All elements in the first loop have to access the elements at the same position, one position ahead and 397 steps ahead. Reaching the 227. element will then forbid us to go 397 steps further, because the position would not lie inside the state vector. But the requested element has already been produced at position zero. Therefore all elements in the second loop need to access the elements at the same position, one position ahead and 227 steps backwards. For the last element the successive element would again lie outside the state vector. Hence, the last elements uses the elements at the last position and positions zero and 396.

Thus all newly generated elements in the first loop are independent of each other and can therefore be vectorized. This is also true for the second loop after the first loop has been executed. The problem lies in the fact that the number 397, as well as 227, is not divisible by eight or four without remainder. This leads to the appearance of a boundary vector term for the last element at the end of the state vector and one between the first and the second loop which needs to access successive and previous elements at the same time. Handling these boundary terms explicitly through vector intrinsics will result in complex loading, storing and blend operations. We have chosen another path by slightly changing the layout of the underlying data structure. Instead of saving only 624 elements for the state vector, we append values at the end of the state vector to get a buffer that should reduce complexity. The number of values we append is given by the amount of elements that fit into a vector register. After the generation of the first vector unit in the first loop, we will copy these values to the end into the buffer. In this case, the last value in the state vector will no longer be a boundary term because it can access the consecutive element through the buffer. The boundary term between the first and the second loop can now be completely handled by the vectorized first loop. This method makes the code implementation easier by making the state bigger. But the number of elements we put into the buffer is small in comparison to the MT19937 state size and therefore an acceptable price to pay.

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{}} {}
```

As explained, the structure of the vectorized version does not really change. We should only make sure that the state vector will be at least 32 Byte aligned. To further optimize the structure, we could force 64 Byte Alignment to provide improved caching possibilities by starting with a cache line.

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;
}
```

Because the state vector is aligned, we can exploit this by using aligned load operations for elements that will be used without a shift. For all other elements, we still have to use unaligned load operations. All parts in the vectorized advancing routine were then directly translated into their corresponding vector intrinsics.

8.2 Xoroshiro

The scalar implementation of the Xoroshiro128+ follows the same rules as the scalar implementation of the MT19937. Vigna provides the C code for the generator on his website (Vigna 2018). We again use a functor which saves the state and advances it by calling the function operator. Compile-time constants will again be declared as `static constexpr` to reduce the

runtime overhead. In comparison to the MT19937, the state of the Xoroshiro128+ can be easily described by two 64 bit unsigned integer values. For the advancing routine, the generator needs a utility which is rotating the bits of a 64 bit unsigned integer.

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};
};
```

The Xoroshiro128+ offers a jump-ahead feature for discarding 2^{64} or 2^{96} elements of the output. Here, we are only providing the implementation of the smaller jump. Both jumping routines differ only in their masking numbers.

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;
}
```

Due to the small state size of the Xoroshiro128+, the vectorization has to use multiple instances of the same generator to exploit the full capabilities of the SIMD intrinsics. Vigna only gives one parameter set for the PRNG. As a consequence, we have to use different seeds for all instances possibly causing overlapping subsequences or the jump-ahead feature to make sure the instances do not overlap for at least 2^{64} or 2^{96} values. For the AVX implementation, we will use four instances of the Xoroshiro128+. The underlying therefore does not really change and uses the SIMD types instead of the $64\,\mathrm{bit}$ unsigned integer values.

Code: AVX Xoroshiro128+ Structure

```
struct xoroshiro128plus {
  using uint_type = uint64_t;
  using simd_type = __m256i;
  using result_type = uint_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;
};
```

With this approach, every instance of the generator is completely independent of the other instances. This is again perfect for the application of SIMD intrinsics. Thus, the function for advancing the state is implemented by its corresponding vector intrinsics. The jump function in the inner loop uses a branch to decide which code path to execute. We do not want this branch to slow down the code by switching to scalar execution. Hence, we will map the branch to vector intrinsics by executing the inner computation in every case and masking the result based on the vectorized evaluation of the branch condition.

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 $\times 4$ 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
```

8.3 Middle Square Weyl Generator

The MSWS is a simple, modern, and non-linear PRNG that uses a state based on 64 bit unsigned integer values but is returning only a 32 bit unsigned integer. The scalar implementation will again use the same approach as the MT19937 and the Xoroshiro128+ as it is given as a C implementation in Widynski (2019). The seeding routine for the MSWS is more complicated and requires a more sophisticated algorithm.

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));
}
```

The MSWS does not provide different parameters or a jump-ahead feature. Its small state size requires us to use multiple instances of independent generators to exploit the size of the vector registers. Thus, all instances have to be initialized with different seeds possibly generating overlapping subsequences. But because its output is only given by a $32\,\mathrm{bit}$ value, we have to generate eight random numbers at once. We can decide to call four instances of the same generator twice or instead use eight instances. While benchmarking the generators, the variant for calling each instance twice reduced the actual throughput of the vectorized PRNG and was therefore discarded from the implementation and measurement. In our vectorized implementation, we use eight instances of the MSWS resulting in the following data structure.

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];
};
```

Advancing the state of the vectorized MSWS introduced difficulties because we have to compute the square of the first $64\,\mathrm{bit}$ value. Either the AVX nor the SSE instruction set is

providing a multiplication for 64 bit unsigned integer numbers. Accordingly, we have to implement our own function for squaring the 64 bit values in a vector register based on given intrinsics. Let $x \in \mathbb{N}_0$ with $x < 2^{64}$ be a 64 bit unsigned integer and let $x_0, x_1 \in \mathbb{N}_0$ with $x_0, x_1 < 2^{32}$ be its 32 bit representation, such that $x = x_1 2^{32} + x_0$.

$$x^2 = x_1^2 2^{64} + 2x_1x_0 2^{32} + x_0^2 \equiv 2x_1x_0 2^{32} + x_0^2 \mod 2^{64}$$

This equation uses a 64 bit multiplication for 32 bit numbers to deal with the carry bits of the computation. The SSE and AVX instruction sets provide this operation, such that the added complexity of the computation can be used to find an optimal implementation.

Afterwards, the first part of the vectorization is again straightforward because all the instances are independent so that every statement can directly be interchanged with its corresponding vector intrinsic. At the end of the first part, two variables of SIMD type will provide 64 bit values as random numbers. The MSWS algorithm forces us to use only the lower 32 bit of the given values resulting in only one vector type with the doubled amount of values as result. To do that, both variables have to be convoluted by shuffle and permutation operations. The squaring function and the permutation operations at the end will probably exhibit a high latency and therefore reduce the speed-up of the vectorization.

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));
}
```

8.4 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) | 0x3ff0000000000000ULL);
  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);
}
```

8.5 Summary

9 Testing Framework

9.1 Unit Tests

While working on modules of a library, we always have to make sure to test their functionality. Typically, this is done via unit and integration tests. We do not want to go into details of testing but instead want to show a few unit tests that were created and used during the development process. Besides consistency and correctness, this will additionally show how to apply the library in other code. To append and create new tests easily, a unit testing framework called "doctest" was used (Kirilov 2019). Because all PRNGs are in general tested the same way, we will only describe the testing facilities by means of the MT19937 and AVX intrinsics if not otherwise stated.

Code: Scalar MT19937 Advance Test

```
TEST_CASE("mt19937 Random Initialization with Default Seeder") {
  const auto seed = std::random_device{}();
  std::mt19937 std_rng{seed};
  pxart::mt19937 my_rng{pxart::mt19937::default_seeder{seed}};

  const int n = 10000000;
  for (auto i = n; i > 0; --i) REQUIRE(std_rng() == my_rng());
}
```

In our scalar implementation of the MT19937, we wanted to guarantee that its output does not differ in comparison to the output of the `std::mt19937` of the STL of C++. Using only the seeding process and the advancing methods of the generators, we came up with the most simplest solution. We use a truly random seed given from `std::random_device` to initialize the standard MT from the STL and our own implementation the same way. Afterwards, we are requiring the equality of their output for several million calls to the function operator. It does not necessarily fulfill all good design principles for unit tests but asks for enough information, such that multiple successful runs of this test imply the correctness of the implementation. Therefore, we avoided an infeasible brute-force method of testing all possible outputs. As we have designed another API, our own MT has to be seeded with its default seeder, such that it is able to use only one 32 bit random number in the construction process. Please note that `std::random_device` does not have to be truly random (O'Neill 2015a,c) but in this case can be replaced by other seeding alternatives (O'Neill 2015d).

Testing the vectorized implementation of an advancing method can be done the same way by comparing each element of an SIMD vector with the output of the scalar version. The use of appropriate `using` declarations or even a wrapper class would makes the code more readable.

Code: AVX MT19937 Advance Test

```
TEST_CASE("simd256::mt19937 Random Initialization with Default Seeder") {
  using result_type = pxart::simd256::mt19937::result_type;
  constexpr auto simd_size = pxart::simd256::mt19937::simd_size;

  const auto seed = std::random_device{}();
  pxart::mt19937 rng{pxart::mt19937::default_seeder{seed}};
```

```
  pxart::simd256::mt19937 vrng{pxart::mt19937::default_seeder{seed}};

  const int n = 10000000;
  for (int i = 0; i < n; i += simd_size) {
    const auto v = vrng();
    for (int j = 0; j < simd_size; ++j) {
      const auto srnd = rng();
      const auto vrnd = reinterpret_cast<const result_type*>(&v)[j];
      REQUIRE(srnd == vrnd);
    }
  }
}
```

As a special case, the vectorization of the jump function of the Xoroshiro128+ can be tested by checking the equality of the internal state variables. Here, we have to take care when initializing the four scalar instances to test against. We first set their state directly to be able to run the comparison the same way after doing the jump.

Code: AVX Xoroshiro128+ Jump Test

```
TEST_CASE("simd256::xrsr128p Jump Vectorization") {
  constexpr auto simd_size = pxart::simd256::xrsr128p::simd_size;

  pxart::simd256::xrsr128p rng1{std::random_device{}};
  pxart::xrsr128p rng2[simd_size];

  for (int i = 0; i < simd_size; ++i) {
    rng2[i].s0 = reinterpret_cast<const uint64_t*>(&rng1.s0)[i];
    rng2[i].s1 = reinterpret_cast<const uint64_t*>(&rng1.s1)[i];
  }

  rng1.jump();
  for (int it = 0; it < simd_size; ++it) rng2[it].jump();

  for (int i = 0; i < simd_size; ++i) {
    const auto tmp0 = reinterpret_cast<const uint64_t*>(&rng1.s0)[i];
    const auto tmp1 = reinterpret_cast<const uint64_t*>(&rng1.s1)[i];
    CHECK(rng2[i].s0 == tmp0);
    CHECK(rng2[i].s1 == tmp1);
  }
}
```

9.2 Statistical Testing

The topic of this thesis is not to develop new PRNGs. As a consequence, we will not test the statistical performance of the already known scalar PRNGs. Instead, we have to examine the randomness of the created multiple streams embodied by the SIMD vectors. As stated in Kneusel (2018, pp. 160–162), we will combine all streams of random numbers into a single one by interleaving the samples as shown in figure 10. The combined stream will then be used as an input to the common test suites “TestU01” and *dieharder* (Brown 2019; L’Ecuyer and Simard 2007).

By using the described method, testing vectorized PRNGs without multiple instances will not give us further insights to their randomness properties because interleaving the samples

Figure 10: The figure shows how the samples of multiple streams can be interleaved to produce a combined stream which can be used as input for common statistical test suites.

would produce the same output as the scalar version. This especially true for the MT19937. Hence, we only test the output of the vectorized Xoroshiro128+ and the MSWS to guarantee the vectorization process is not reducing the statistical performance of the generators. The process of testing was geared to Kneusel (2018, pp. 141–155) and O'Neill (2017a).

Dieharder Test Suite

To test external RNGs with the *dieharder* test suite, the typical approach is to generate a stream of random bits which will be used as input for the test suite by using a pipeline. The following code snippet provides a bit stream for the AVX implementation of the Xoroshiro128+ by writing its binary output to standard output. The given implementation can be easily adjusted to work with the other generators by changing the type of the generator.

Code: AVX Xoroshiro128+ Bit Stream

```
#include <cstdio>
#include <iostream>
#include <pxart/simd256/xoroshiro128plus.hpp>
#include <random>

using namespace std;

int main() {
  // Speed up the output of bits by not syncing
  ios_base::sync_with_stdio(false);
  cin.tie(nullptr);

  // Initialize RNG.
  pxart::simd256::xrsr128p rng{random_device{}};
  // Write the binary representation of generated random numbers into the
  // bitstream.
  while (true) {
    const auto sample = rng();
    cout.write(reinterpret_cast<const char*>(&sample), sizeof(sample));
  }
}
```

The `main` procedure starts with two statements that speed up the output process by first disabling the synchronization between the C and C++ IO streams and then untying the standard output stream `cout` from the standard input stream `cin`. We do not want to mix C and C++ IO operations and we do not need a thread-safe output stream so that disabling syncing should make the code faster (*cppreference.com*). Because our bit stream will only use the output operation independently from any input routine we make sure the usage of `cin` will not flush the buffer of `cout` to further raise the speed of the output (*cppreference.com*). Afterwards, we initialize the generator by the TRNG `std::random_device` of the C++ STL and use an infinite loop to write its output to `cout` in binary form. For this, we first generate a new random number by advancing the state of `rng` and the calling `cout.write`. Using the compiled program as input for the *dieharder* test suite can be done on the command line in the following way.

```
bench/xrsr128p_simd256_bitstream | dieharder -a -g 200
```

TestU01 Test Suite

The "TestU01" is a C software library and can be directly used inside a C++ program. Because we have designed a C++ API for our vectorized generators and TestU01 is only able to read a continuous sequence of 32 bit random numbers, we have to wrap their output in a C-compatible way by using functions and global variables. The following code snippet shows again the easiest possible implementation for the vectorized Xoroshiro128+. By the changing the type of the RNG, other generators can be tested as well. Please note, there are much more advanced methods to use TestU01 in C++ code but we only wanted to show the essential parts.

Code: AVX Xoroshiro128+ TestU01

```
extern "C" {
#include <TestU01.h>
}
#include <cstdint>
#include <iostream>
#include <pxart/simd256/xoroshiro128plus.hpp>
#include <random>
#include <sstream>

pxart::simd256::xrsr128p rng{std::random_device{}};
decltype(rng()) cache;
constexpr int cache_size = sizeof(decltype(rng())) / sizeof(uint32_t);
int cache_index = 0;

inline uint32_t serializer() noexcept {
  if (!cache_index) cache = rng();
  const auto result = reinterpret_cast<const uint32_t*>(&cache)[cache_index];
  cache_index = (cache_index + 1) % cache_size;
  return result;
}

int main(int argc, char** argv) {
  if (argc != 2) {
    std::cout << "usage: " << argv[0] << " <n:{0,1,2}>\n"
              << "0\tsmall crush\n"
              << "1\tcrush\n"
              << "2\tbig crush\n";
    return -1;
  }

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

  swrite_Basic = false;
  auto gen = unif01_CreateExternGenBits("xrsr128p simd256", serializer);
  switch (n) {
    case 0:
      bbattery_SmallCrush(gen);
      break;
    case 1:
      bbattery_Crush(gen);
      break;
    case 2:
      bbattery_BigCrush(gen);
      break;
  }
  unif01_DeleteExternGenBits(gen);
}
```

To make sure that we supply a sequence of 32 bit values without changing the actual output of the RNG, a wrapper function is used to serialize larger random values. We are using a small cache which stores the last generated random value. By calling `serializer`, we go to the next 32 bit chunk of data inside the cache and return those values. When we reach the end of the cache, we advance the actual generator and again store its value inside the cache and start anew. Both, generator and cache, are stored as global variables to reduce their access time in comparison to `static` variables inside the function `serializer`.

In the `main` function, arguments given on the command line decide which test battery to run. We allocate a TestU01 generator given by an external function through the call of `unif01_CreateExternGenBits`. Afterwards, we run one of the chosen test batteries, Small Crush, Crush, or BigCrush. At the end, `unif01_DeleteExternGenBits` deallocates the external generator.

The given procedure to serialize random output larger than 32 bit is not unique and will not be able to uncover all possible statistical flaws (O'Neill 2017a). But an exhaustive variant of testing is quite time-intensive and will make the results much more complicated to interpret. In the end, there will always be statistical and empirical tests which will detect the deterministic behavior of PRNGs. As a consequence, the required statistical quality heavily depends on the application a PRNG is used for. For our purposes, the given testing process is completely sufficient.

9.3 Generation Benchmark

Benchmarking the generation of random numbers is tricky. We have encountered several inconsistencies when measuring and comparing the speed of different PRNG implementations. In general, small adjustments in the code of the benchmark completely changed the behavior and performance of the scalar PRNG versions. Furthermore, even the order of PRNGs affected the performance of several generators. Due to the compiler optimization process, some implementations of scalar PRNGs will be optimized with a varying effectiveness while used in different benchmarking scenarios. Sometimes the compiler will notice that computed values are not needed by other parts of the program. As a result, it will optimize the code such that the PRNG to be measured will never be called. Of course, this completely falsifies the performance measurement and makes comparisons impossible. Hence, we recommend to measure the performance of random number generation in an actual application. The following benchmark which is split into three smaller parts was designed to be independent of the order of PRNGs. Advancing the state of a generator is forced by using an underlying cache with a capacity of several thousands of bytes in which random numbers will be stored. The compiler is not allowed to remove the cache by means of optimization. With the benchmark, it is possible to measure the speed-up introduced by vectorization. We used the *PerfEvent* benchmarking wrapper to simplify the implementation of the measurement process. Additionally, it gives us further insights, such as the average instructions per cycle, average branch misses, average cache misses, and even more (Leis 2019).

Code: Generation Benchmark Structure

```
struct benchmark {
  explicit benchmark(size_t samples);
```

```
  template <typename result_type, typename RNG>
  benchmark& run(const char* name, RNG&& rng) noexcept;
  template <typename RNG>
  benchmark& run(const char* name, RNG&& rng) noexcept {
    return run<decltype(rng())>(name, forward<RNG>(rng));
  }
  template <typename result_type, typename RNG>
  benchmark& run(const char* name, RNG&& rng1, RNG&& rng2) noexcept;
  template <typename RNG>
  benchmark& run(const char* name, RNG&& rng1, RNG&& rng2) noexcept {
    return run<decltype(rng1())>(name, forward<RNG>(rng1), forward<RNG>(rng2));
  }
  benchmark& separate() noexcept;

  // Member Variables
  static constexpr int cache_size = 1 << 14;
  size_t n{};
  BenchmarkParameters params{};
  bool header = true;
};
```

To apply the benchmarking routine easily to any possible generator, we use an abstraction class which saves the context variables, such as the number of samples and the *PerfEvent* parameters. The member function templates `run` are used to actually execute the benchmarking routine and write the results to the standard output. We overload the template to automatically deduce the type of a given RNG and the algorithm to use. Because on modern systems the `std::mt19937` of the C++ STL uses a 64 bit variable to output a 32 bit random number, we have to introduce the possibility of manually setting the type the return type will be casted to. To not always specify it explicitly, the second overload makes sure to automatically deduce the return type by the use of `decltype`. The third and the fourth variant implement the same benchmarking routine for two generators of the same type by alternately advancing their states and writing their output into the cache. We use this facility to compare the latency of a single PRNG instance to the latency of two instances used at once.

Code: Generation Benchmark Implementation

```
template <typename result_type, typename RNG>
benchmark& run(const char* name, RNG&& rng) noexcept {
  params.setParam("npc", to_string(sizeof(result_type) / sizeof(uint32_t)));
  params.setParam("name", name);

  // Initialize cache to store temporary data.
  constexpr auto cache_count = cache_size / sizeof(result_type);
  array<result_type, cache_count> cache;
  // Warm up the cache.
  for (int i = 0; i < 100; ++i)
    for (auto& x : cache) x = rng();
  // Perform actual benchmark in new scope.
  {
    PerfEventBlock e(n * cache_count, params, header);
    for (int i = 0; i < n; ++i)
      for (auto& x : cache) x = rng();
  }
  // At end of scope, destroy PerfEventBlock and output measurements.

  header = false;
  return *this;
```

```
}
```

Here, we only show the benchmark implementation for a single instance of a PRNG. The version using two instances follows directly from the given code. At first, we add *PerfEvent* parameters, like the name of the PRNG, to get a more detailed output on the command line for every measurement. Afterwards, the cache which is preventing the removal of the call to the PRNG by compiler optimization is initialized. To warm up the cache, several thousand random numbers will be generated and stored in the cache without measuring performance. With this, we make sure the state of the PRNG and the cache already lie inside the L1 cache of the computer hardware if possible. The insertion of a warm-up process greatly reduced the noise of measurements between multiple benchmark runs. Then we start a new scope that is used for the actual benchmark. We construct a `PerfEventBlock` that immediately starts with the measuring process. Due to the RAII principle of C++, the `PerfEventBlock` will be destroyed at the end of the scope finishing the measurements and outputting the results. Between its construction and destruction, we run over the cache filling it with random numbers generated by the PRNG again and yet again. Typically, the time for only one call to the advancing routine of a generator is unmeasurable because it is too fast. Hence, we accumulate the time needed for several thousand calls and calculate the average time needed for the generation of one number.

Code: Generation Benchmark Application

```
int main(int argc, char** argv) {
  if (2 != argc) {
    cout << "usage:\n" << argv[0] << " <number of elements>\n";
    return -1;
  }
  stringstream input{argv[1]};
  uint64_t sample_count;
  input >> sample_count;

  random_device rd{};
  benchmark{sample_count}  //
      .run<uint32_t>("std mt  ", std::mt19937{rd()})
      .run("boost mt", boost::random::mt19937{rd()})
      .separate()
      .run("mt19937 ", pxart::mt19937{rd})
      .run(".simd256", pxart::simd256::mt19937{rd})
      .run(".simd128", pxart::simd128::mt19937{rd})
      .separate()
      .run("xrsr128p", pxart::xrsr128p{rd})
      .run("     .x2", pxart::xrsr128p{rd}, pxart::xrsr128p{rd})
      .run(".simd256", pxart::simd256::xrsr128p{rd})
      .run("    ..x2", pxart::simd256::xrsr128p{rd},
           pxart::simd256::xrsr128p{rd})
      .run(".simd128", pxart::simd128::xrsr128p{rd})
      .run("    ..x2", pxart::simd128::xrsr128p{rd},
           pxart::simd128::xrsr128p{rd})
      .separate()
      .run("msws    ", pxart::msws{rd})
      .run("     .x2", pxart::msws{rd}, pxart::msws{rd})
      .run(".simd256", pxart::simd256::msws{rd})
      .run("    ..x2", pxart::simd256::msws{rd}, pxart::simd256::msws{rd})
      .run(".simd128", pxart::simd128::msws{rd})
```

```
        .run("    ..x2", pxart::simd128::msws{rd}, pxart::simd128::msws{rd});
}
```

To easily use the benchmarking structure for multiple runs with different RNGs, we always return a reference to itself. This allows us to chain the calls in the `main` function reducing the usage overhead. Inside the `main` function, the sample count is read from the command line and then put into the constructor of `benchmark`. Afterwards, a list of runs for different generators exploiting the single and double instance facilities is given with the help of the chaining mechanism. In addition, chaining allows those functions to be directly appended to an rvalue of type `benchmark` to further reduce code complexity.

9.4 Monte Carlo π Benchmark

The framework used for benchmarking the computation of π is the same as the one we have constructed for the generation benchmark. As a consequence, in this section we will only discuss different versions on how to compute π with vectorized PRNGs.

In section 3.1 about Monte Carlo integration, we have already given a naive approach together with an explanation on how to compute π with the random utilities of the C++ STL. The code will be repeated here for convenience.

Code: Monte Carlo π Benchmark Naive Version

```
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);
    samples_in_circle += (x * x + y * y <= 1);
  }
  return static_cast<Real>(samples_in_circle) / samples * 4;
}
```

For every version of the benchmark using only one instance of an RNG, we have another version executing the same algorithm with two instances of the same RNG type. Again, we do this to compare the latency for multiple instances of PRNGs. We give the example only for the naive version and will omit the other multiple instance versions to keep the focus on the algorithms.

Code: Monte Carlo π Benchmark Naive Version with Two Instances

```
template <typename Real, typename Integer, typename RNG>
inline Real monte_carlo_pi(RNG&& rng1, RNG&& rng2, 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(rng1);  // Use first instance.
```

```
    const auto y = dist(rng2);  // Use second instance.
    samples_in_circle += (x * x + y * y <= 1);
  }
  return static_cast<Real>(samples_in_circle) / samples * 4;
}
```

Adding the second instance to the naive algorithm is simple. We use the first instance of the RNG to generate the first random number `x` and the second instance to the generate the second random number `y`.

A first improvement to the naive version can be introduced by removing `std::uniform_real_distribution` given by the STL and instead using our custom uniform distribution function. This is done to be able to fairly evaluate the efficiency of the vectorization process for PRNGs. Again, the algorithm is only applicable to scalar generators.

Code: Monte Carlo π Benchmark with Custom Distribution

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

To show the superiority of vectorized PRNGs, we need a variant of the Monte Carlo π benchmark that is able to use vectorized PRNGs without the need the vectorize the computation itself. We should see an improvement in performance. A developer therefore can use the generator without changing the complete implementation of his or her algorithm.

Code: Monte Carlo π Benchmark with Cache

```
template <typename Real, typename Integer, typename RNG>
inline Real monte_carlo_pi(RNG&& rng, Integer samples) noexcept {
  constexpr int cache_size = sizeof(decltype(rng())) / sizeof(Real);
  Integer samples_in_circle{};
  for (Integer i = 0; i < samples; i += cache_size) {
    const auto cache_x = pxart::uniform<Real>(rng);
    const auto cache_y = pxart::uniform<Real>(rng);
    for (int j = 0; j < cache_size; ++j) {
      const auto x = reinterpret_cast<const Real*>(&cache_x)[j];
      const auto y = reinterpret_cast<const Real*>(&cache_y)[j];
      samples_in_circle += (x * x + y * y <= 1);
    }
  }
  return static_cast<Real>(samples_in_circle) / samples * 4;
}
```

The code uses the same idea as the generation benchmark before. It introduces a cache to store the vectorized output in. A loop over random numbers then has to be adjusted to get the random numbers from the cache. If every random number of the cache was read, we regenerate the content of the cache by calling the advancing routine of the PRNG. The implementation accomplishes this by adding another `for` loop inside the already existing one. For the benchmark, we always assume the number of samples to take is a multiple of the cache size. Due to the small size of the cache, this does not impose any serious restrictions. In a real application, we would need to introduce code that is dealing with the remaining part of the samples. Such a loop over a few elements will not change the outcome of the measurements taken by the benchmark. We wanted to keep the code simple and therefore omitted this part here and in the following.

After introducing a cache for vectorized PRNGs, it is only logical to provide a completely vectorized benchmark for the SSE and the AVX instruction sets to further improve the performance. Again, we will show the code only for the single-precision AVX implementation to keep the focus on the algorithm. Using vector intrinsics, we somehow restrict our algorithm to use $32\,\mathrm{bit}$ integers as sample count. For the measurements taken in this thesis, this is not of importance. Adding support for the single-precision case with more than 2^{32} samples is possible but would introduce a lot more overhead to the code by not giving further insights to the performance of vectorized PRNGs. Because the computation of π would not be realized via Monte Carlo methods in real applications, we have not implemented such a variant.

Code: Monte Carlo π Benchmark Single-Precision AVX Version

```
template <typename Integer, typename RNG>
inline float monte_carlo_pi(RNG&& rng, Integer samples) noexcept {
  auto samples_in_circle = _mm256_setzero_si256();
  // Use packet of eight values to run over samples.
  for (auto i = samples; i > 0; i -= 8) {
    // Generate random sample in unit square.
    const auto x = pxart::simd256::uniform<float>(rng);
    const auto y = pxart::simd256::uniform<float>(rng);
    // Computer the squared norm of the sample.
    const auto radius = _mm256_add_ps(_mm256_mul_ps(x, x), _mm256_mul_ps(y, y));
    // Evaluate the circle condition and generate a mask.
    const auto mask = _mm256_castps_si256(
        _mm256_cmp_ps(radius, _mm256_set1_ps(1.0f), _CMP_LE_OQ));
    // Increment when sample lies inside circle by using the mask.
    samples_in_circle = _mm256_add_epi32(
        samples_in_circle, _mm256_and_si256(_mm256_set1_epi32(1), mask));
  }
  // Add the values in each half of samples_in_circle.
  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 result by adding the last two values with a scalar operation.
  return 4.0f *
         (reinterpret_cast<uint32_t*>(&samples_in_circle)[0] +
          reinterpret_cast<uint32_t*>(&samples_in_circle)[4]) /
         samples;
}
```

The computation of π allows us to interchange nearly every operation with the corresponding intrinsic operation. Care has to be taken when evaluating the condition. Here, we first compute a mask based on the circle condition and apply the mask to the incrementation operation,

such that samples that are lying outside the circle will not increment `samples_in_circle`. At the end of the `for` loop, `samples_in_circle` consists of eight values which again have to be added. For many samples, this process has no real effect on the performance of the benchmark. Thus, the sum could be completely evaluated through scalar operations. We have used two `_mm256_hadd_ps` calls to add adjacent elements in each half. The last addition is carried out by a scalar operation.

Concerning the precision of the output, the sample count is important and has to be expressed as `uint32_t` or `uint64_t`. To reach the full precision of a single precision floating point value, we have to fill 23 bits of the fraction. Doing this would need approximately 2^{44} samples and therefore a `uint64_t` as a type of the sample variable. With `uint32_t`, we can reach a precision of $2^{-15} \approx 3.05 \cdot 10^{-5}$.

9.5 Photon Benchmark

10 Evaluation and Results

In section 8, we have implemented scalar and vectorized versions of known PRNGs, namely the MT19937, the Xoroshiro128+, and the MSWS. Additionally, unbiased uniform distribution functions for floating-point numbers based on Kneusel (2018, pp. 55–56) and Vigna (2018) were introduced to completely exploit the vector registers as long as possible. The last section 9 described the implementation of tests and benchmarks to measure the performance of PRNGs concerning statistics and execution time. So to prove that the vectorization of PRNGs is indeed improving the performance without reducing statistical quality, we have to apply those benchmarks and analyze their output.

We have run the benchmarks on two different machines with similar architectures with varying raw computing powers. The machines are modern Linux-based $64\,\mathrm{bit}$ platforms providing the GCC C++ compiler version 9.2.0 (GCC 2019b). The given C++ code used for including the library and running the tests and benchmarks was compiled with the optimization flags `-O3` and `-march=native` to guarantee the strongest optimization level. Because the generation of random numbers through PRNGs typically does not have to access main memory and can instead completely be run inside the caches of the CPU, specifying the CPU parameters of a target machine will be sufficient to draw conclusions. The first machine is a laptop and consists of the *Intel® Core™ i5-8250U Processor* (Intel 2017a) whereas the second machine is a custom desktop computer with an installed *Intel® Core™ i7-7700K Processor* (Intel 2017b). Both systems consist of four cores featuring Hyper-Threading and the SSE/AVX instruction set extensions up to SSE4.2 and AVX2. The desktop processor is an high-end performance microprocessor based on the Kaby Lake microarchitecture. It operates at a base frequency of $4.2\,\mathrm{GHz}$ and a Turbo Boost frequency of up to $4.5\,\mathrm{GHz}$ when used in single-core mode. The laptop processor is a mobile microprocessor and is also based on an enhanced version of the Kaby Lake microarchitecture. It operates at a base frequency of $1.6\,\mathrm{GHz}$ and a Turbo Boost frequency of up to $3.4\,\mathrm{GHz}$. We refer to Intel (2017a) and Intel (2017b) for more information.

10.1 Statistical Quality

From a theoretical point of view, interleaving multiple streams of random numbers based on multiple instances of the same generator should not reduce the randomness of the output, such that test suites will be able to measure it. Using multiple instances through SSE/AVX vector registers the state of the vectorized PRNG becomes at least two times as big as the scalar variant. According to O'Neill (2017b), creating a larger state will even make a weak generator stronger concerning its statistical performance. On the other hand, Fog (2015) states that the use of multiple instances of the same generator with different seeds can lead to overlapping subsequences.

For testing the statistical performance, we have used TestU01 version 1.2.3 and version 3.31.1 of dieharder. The computation time to run all the tests in the test batteries of TestU01 approximately ranged from $5\,\mathrm{s}$ for SmallCrush over $20\,\mathrm{min}$ for Crush to $150\,\mathrm{min}$ for BigCrush. While executing the BigCrush battery, 160 statistics were used to estimate the statistical performance of the generators. The execution of all 30 dieharder tests with different parameters always took several minutes.

Running the statistical test suites, we could not find any vulnerabilities. Neither the scalar

nor the vectorized versions of the Xoroshiro128+ and the MSWS systematically failed an empirical test. Even a truly random sequence will fail tests from time to time (Kneusel 2018, p. 142) and so after running the test suites multiple times, we wer able to confirm this for our implementations, too. Every generator rarely produced test failures that did not follow any pattern. We conclude that the usage of multiple instances to vectorize the generators did not reduce their statistical quality.

10.2 Performance Improvement

Applying the designed benchmarks to the implementations of scalar and vectorized PRNGs enables us to evaluate the performance improvement introduced by using vectorization techniques. In tables 1 and 2, the results for running the generation benchmark and the Monte Carlo π benchmark respectively on the *Intel® Core™ i7-7700K Processor* are given for all the implemented PRNG versions and benchmark scenarios. For the MT19937, we have even inserted the data for the STL and Boost implementations to see if the code is indeed outperforming its state-of-art scalar counterparts (Boost 2019; GCC 2019a). All measurements and plots for the *Intel® Core™ i5-8250U Processor* will not be shown in this section to not distract and can be found in the appendix A. Figures 11 and 12 show the respective performance and speedup resulted from running the generation benchmark on the *Intel® Core™ i7-7700K Processor*. For the many variants of the Monte Carlo π benchmark, we only show the speed-up in figure 13. The prefix "2 x" appearing in all figures states that two independent instances of the given generator were used.

Table 1: The table shows the results achieved by running the generation benchmark on the *Intel® Core™ i7-7700K Processor* at a frequency of 4.51 GHz with all implemented variants of given PRNGs. While running the benchmark, 16 GiB of random numbers were generated and temporarily stored in a cache of size 16384 B by iterating 2^{20} times over its content. During the execution, there were no cache or branch misses. The values for cycles, instructions, and IPCs were averaged over the calls to the advancing routine of the respective generator.

PRNG	**Variant**	**Result Size** [B]	**Time** [s]	**Cycles**	**Instructions**	**IPC**
STL MT19937	scalar	4	9.29	9.75	28.34	2.91
Boost MT19937	scalar	4	5.79	6.09	24.86	4.09
MT19937	scalar	4	7.61	7.99	26.71	3.34
	AVX	32	1.15	9.68	34.07	3.52
	SSE	16	2.11	8.86	34.04	3.84
Xoroshiro128+	scalar	8	1.66	3.49	12.00	3.44
	2 x scalar	8	1.80	3.79	12.00	3.17
	AVX	32	0.74	6.26	17.02	2.72
	2 x AVX	32	0.69	5.77	14.01	2.43
	SSE	16	1.47	6.19	17.01	2.75
	2 x SSE	16	1.31	5.49	14.01	2.55
MSWS	scalar	4	4.77	5.01	8.00	1.60
	2 x scalar	4	4.24	4.45	8.50	1.91
	AVX	32	1.84	15.45	31.02	2.01
	2 x AVX	32	1.32	11.13	24.51	2.20
	SSE	16	3.71	15.58	29.01	1.86
	2 x SSE	16	2.60	10.93	24.51	2.24

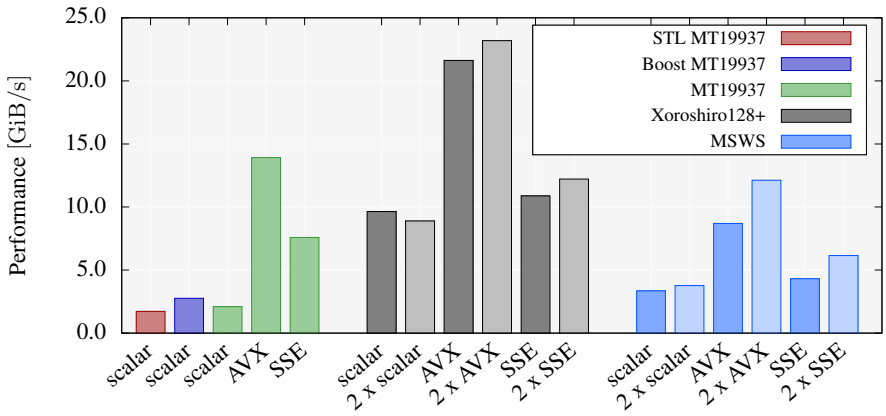


Figure 11: The plot shows the performance resulting from the generation benchmark running on the *Intel® Core™ i7-7700K Processor* measured in GiB of random numbers per second for the different variants of the implemented PRNGs. For convenience, the performances of the STL and Boost implementation of the MT19937 are shown as well. The data can be found in table 1.

Both target machines consist of similar architectures. Comparing the tables 1 and 2 and the figures 11, 12, and 13 to the tables 3 and 4 and the figures 14, 15, and 16 in appendix A respectively, we see that both architectures exhibit nearly the same speed-up, averaged cycles and averaged instructions. Due to differing raw computing power, the execution time and the averaged IPC deviate from each other. Therefore we recommend that quantizing the efficiency of a PRNG implementation should use the averaged cycles and instructions to draw conclusions for a given microarchitecture.

We are mostly interested in the speed-up concerning the execution time of a given benchmark to have a direct comparison between scalar and vectorized generators. According to figure 12, in the generation benchmark we were able to achieve a speed-up greater than one for all vectorized versions of PRNGs.

Our scalar implementation of the MT19937 has already beaten the standard version of the STL but was not able to surpass the Boost implementation. However, looking at the speed-up introduced by the usage of SSE and AVX intrinsics, vectorized versions of the MT19937 are more than three times faster by using SSE and more than six times faster by using AVX than their scalar counterparts. Vectorizing the MT19937, the best possible theoretical speed-up should be four when used with SSE registers and eight when used with AVX registers. The compiler already tries to vectorize the scalar versions so that it is not possible to reach this theoretical bound. But this also means that for this scenario the compiler is not able to fully vectorize the scalar implementations automatically, such that manually vectorizing the code is necessary for speeding up the execution. In general, the MT19937 speed-ups introduced by SSE and AVX are close to the theoretical maximum. Thus, taking the extra effort of vectorizing this generator paid off.

The Xoroshiro128+ indeed executes faster by using SSE/AVX intrinsics. The SSE variant realizes a speed-up of approximately 1.1 and the AVX variant a speed-up of 2.2. However, we are much further away from the theoretical bounds than in the case of the MT19937. Using SSE, we should reach a speed-up of two, whereas a speed-up of four would be the bound for the

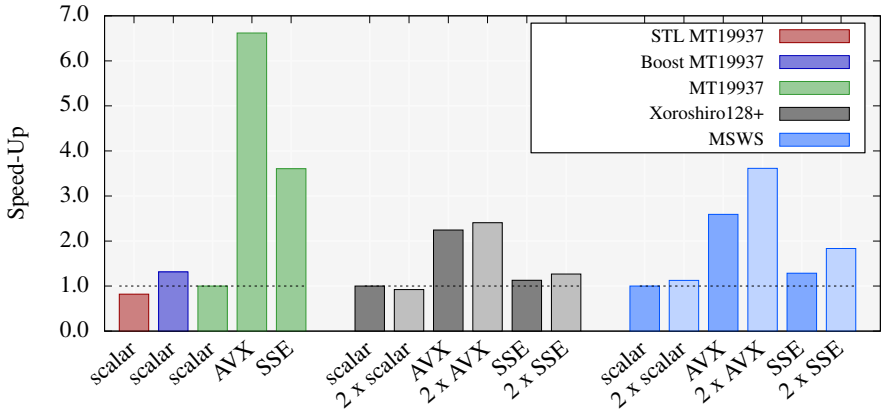


Figure 12: The plot shows the speed-up in execution time with respect to the single-instance scalar version resulting from the generation benchmark running on the *Intel® Core™ i7-7700K Processor* for the different variants of the implemented PRNGs. For convenience, the speed-ups of the STL and Boost implementation of the MT19937 are shown with respect to our implementation. The data can be found in table 1.

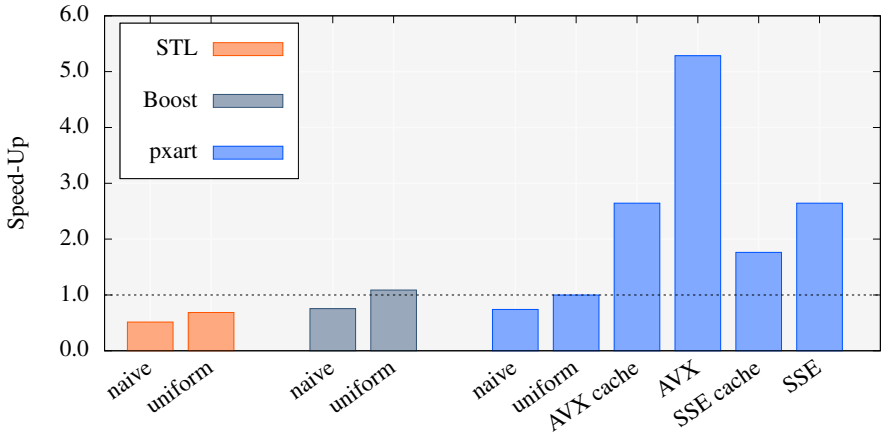
AVX version. We expect that this time the compiler was much more capable of automatically vectorizing large parts of the code in the generation benchmark. If the automatically vectorized code is already exploiting vector intrinsics we cannot hope to reach the theoretical speed-up. But instead the manual vectorization was able to further reduce code and data dependencies in the generation benchmark. Furthermore, by analyzing latency issues through the use of two independent instances of the generator, we see that in some cases SSE/AVX versions can be further optimized by using the doubled state size to keep the CPU pipeline filled. This seems not to be true for the scalar version. The implementation of a vectorized Xoroshiro128+ is not a complex procedure. As a consequence, even for these smaller speed-ups using vectorization techniques turns out to be profitable.

The results for the MSWS in the generation benchmark are similar to the results of the Xoroshiro128+. The compiler is capable of automatically vectorizing large parts of the code. This reduces the possible speed-up reachable by SSE/AVX implementations. But this time, we see a strong improvement in performance when using two instances of the MSWS versions. In the implementation of the MSWS, we have to carry out a $64\,\mathrm{bit}$ multiplication by squaring a number. Especially for vector registers, operations needed to compute this result typically feature an acceptable throughput but also a high latency. Hence, using the doubled amount of independent states would take advantage of these parameters by keeping the multiplication pipeline filled. This also would explain the results. Vectorizing the MSWS has been a little bit more complicated than the procedure for the Xoroshiro128+. On the other hand, we are getting a slightly better speed-up with the potential for further improvements. Figure 11 shows that the performance of the vectorized MSWS versions is smaller in comparison to the other generators. The MSWS seems not to be competitive concerning its execution time, yet. Nevertheless, its non-linearity makes it a perfect candidate for more advanced applications with special statistical requirements on used random numbers.

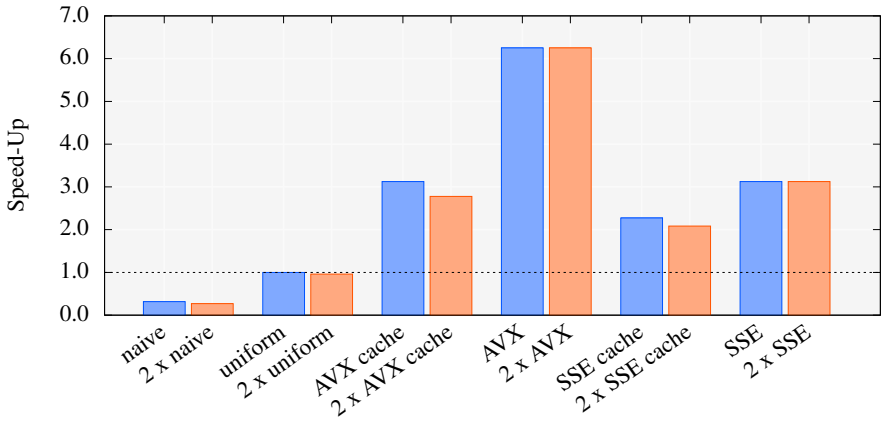
Looking at the data and plots of the Monte Carlo π benchmark in table 2 and figure 13, we first expected to see similar results. As before, all the vectorized implementations were

Table 2: The table shows the results achieved by running the Monte Carlo π Benchmark on the *Intel® Core™ i7-7700K Processor* with all implemented variants of given PRNGs and benchmark scenarios. While running the benchmark, 10^8 samples in the unit square were used to estimate the value of π. It was ensured that the estimation error was small enough according to the calculation at the end of section 3.1. During the execution, there were no cache or branch misses. The values for cycles, instructions, and IPCs were averaged over the number of samples in the unit square.

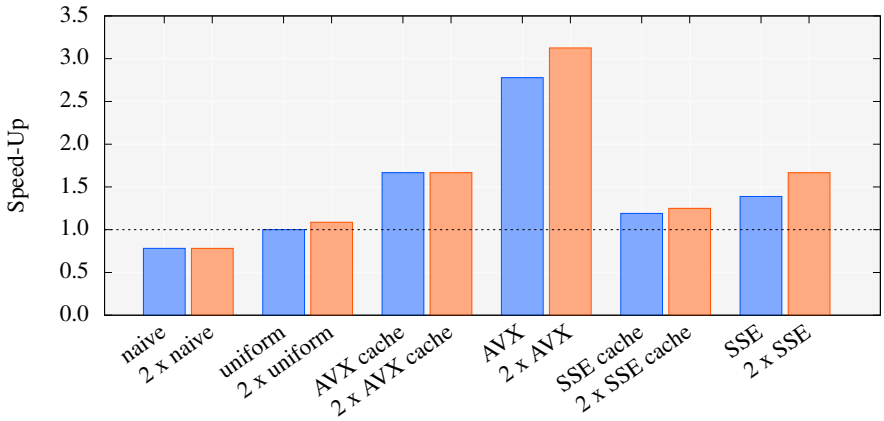
PRNG	**Benchmark**	**Time** [s]	**Cycles**	**Instructions**	**IPC**	**Frequency** [GHz]
STL MT19937	naive	0.72	31.30	85.68	2.74	4.37
	uniform	0.53	24.03	77.67	3.23	4.51
Boost MT19937	naive	0.47	21.18	66.72	3.15	4.51
	uniform	0.33	14.94	58.72	3.93	4.51
MT19937	naive	0.48	21.84	64.43	2.95	4.51
	uniform	0.35	15.81	60.43	3.82	4.51
	AVX cache	0.15	6.75	17.14	2.54	4.51
	AVX	0.07	3.08	9.39	3.05	4.51
	SSE cache	0.21	9.57	25.27	2.64	4.51
	SSE	0.14	6.33	18.77	2.96	4.51
Xoroshiro128+	naive	0.80	36.30	45.01	1.24	4.51
	2 x naive	0.97	43.68	53.02	1.21	4.51
	uniform	0.26	11.55	34.00	2.95	4.51
	2 x uniform	0.27	11.98	42.00	3.51	4.51
	AVX cache	0.08	3.78	11.75	3.11	4.51
	2 x AVX cache	0.09	3.85	12.75	3.31	4.51
	AVX	0.04	1.89	4.50	2.38	4.51
	2 x AVX	0.04	1.87	5.50	2.94	4.51
	SSE cache	0.11	5.03	15.50	3.08	4.51
	2 x SSE cache	0.12	5.29	17.25	3.26	4.51
	SSE	0.08	3.61	9.00	2.49	4.51
	2 x SSE	0.09	3.88	11.00	2.84	4.51
MSWS	naive	0.31	13.99	30.01	2.15	4.51
	2 x naive	0.34	15.34	38.01	2.48	4.51
	uniform	0.25	11.44	26.00	2.27	4.51
	2 x uniform	0.23	10.25	32.00	3.12	4.51
	AVX cache	0.15	6.99	14.91	2.13	4.51
	2 x AVX cache	0.15	6.67	15.90	2.38	4.51
	AVX	0.09	4.09	7.50	1.83	4.51
	2 x AVX	0.08	3.46	8.50	2.46	4.51
	SSE cache	0.21	9.41	21.50	2.28	4.51
	2 x SSE cache	0.20	9.22	23.25	2.52	4.51
	SSE	0.18	8.03	15.00	1.87	4.51
	2 x SSE	0.15	6.79	17.00	2.50	4.51



(a) MT19937



(b) Xoroshiro128+



(c) MSWS

Figure 13: The plot shows the speed-up in execution time with respect to the single-instance uniform version resulting from the Monte Carlo π benchmark running on the *Intel® Core™ i7-7700K Processor* for the different variants of the implemented PRNGs and benchmark scenarios. For convenience, the speed-ups of the STL and Boost implementation of the MT19937 are shown with respect to our implementation. The data can be found in table 2.

able to increase the performance of the program. Especially the fast uniform distribution function even outperforms the naive scenario for scalar generators in all cases. But this time the compiler was rarely capable of automatically vectorizing the code. As a consequence, we could detect much higher speed-ups for the Xoroshiro128+ and MSWS by using vector intrinsics. The speed-ups of the MT19937 are reduced. Using two instances of independent generators only slightly improves the performance of the MSWS and is not really changing the performance of the Xoroshiro128+. We think that this follows from a higher code complexity in the Monte Carlo π benchmark. The compiler optimization of the program flow is not as effective as for the generation benchmark. However, at the end of each benchmark the estimation of π will printed on the screen which forces the compiler to not remove code that is needed for the computation of π. Therefore the Monte Carlo π benchmark did not introduce as much noise to the measurements as the generation benchmark did. We conclude that an actual application, like the computation of π, should always be used to measure performance and speed-up of PRNGs.

Another important point is that we always could reduce the execution time by a factor ranging from 1.2 to 2.6 when using a cache without introducing vector intrinsics to the actual benchmark. By Amdahl's law, the given speed-ups have to be a lot smaller than the theoretical bounds because a great percentage of the code was not vectorized. This means that even in scalar code the usage of vectorized PRNGs improves the overall performance. But to completely exploit the speed-up brought by vectorization of PRNGs, developers should consider the usage of SIMD intrinsics in performance-critical parts of their code which are using random numbers.

11 Conclusions and Future Work

We were able to show that the vectorization of PRNGs and distributions based on SIMD intrinsics can improve the performance of physical simulations which need to access a large amount of random numbers. By applying current statistical and empirical test suites, it was demonstrated that the vectorization of multiple instances does not reduce the statistical performance of the given generators. The time and effort taken to implement the vectorized data structures and advancing routines do not outweigh the enhancement in speed-up but have to be considered when deciding to vectorize an actual application to further increase performance. We made clear that measuring the performance of PRNGs in general should be done inside an actual application that is using random numbers to compute a testable result to not falsify outcomes.

Also, we have developed an API which makes the initialization process and the usage of distributions much simpler while providing possibilities for easily implementing new generators specializing algorithms though the creation of certain member functions. The interface of the library was directly applied to the testing framework and the physical simulations showing its simplicity and robustness. Accelerating the execution of randomized algorithms was achieved through different implementation schemes by using scalar code or exploiting SSE/AVX features. Especially, the estimation of π, the Ising model and the simulation of photon propagation typically build the basis for more advanced physical simulations in optics and quantum physics depending on Monte Carlo integration, Metropolis-Hastings algorithms, importance sampling and Russian roulette. Thus, even in scalar applications not using any SIMD intrinsics, we can recommend the usage of vectorized PRNGs and distributions to speed up the generation of random numbers.

For further development, we recommend to compare our implementations against vectorized state-of-the-art RNGs given by *Intel® Math Kernel Library* and *RNGAVXLIB*. Due to the unique restrictions concerning the statistical performance an application is imposing on an RNG, future work should involve the development of even more vectorized RNGs. This also means that we have to exhaustively test the statistical performance of vectorized PRNGs in different scenarios. These scenarios should be simplified versions of a real-world problem to justify the results and examine the robustness of our current design. By tweaking the testing and benchmarking framework, we are then able to extend the vectorized implementations to other SIMD architectures providing vector registers, like the AVX512 instruction set from Intel. With the release of the C++20 standard specification, we should take advantage on the newly introduced features of C++, like concepts, to reduce the complexity of the API and increase its performance and handling.

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).

Boost (2019). *Boost. C++ Libraries*. URL: https://www.boost.org/ (visited on 12/09/2019).

Brown, Robert G. (2019). *dieharder. A Random Number Test Suite*. URL: https://webhome.phy.duke.edu/~rgb/General/dieharder.php (visited on 12/03/2019).

cppreference.com. URL: https://en.cppreference.com/w/ (visited on 07/15/2019).

Dolbeau, Romain (2016). "Theoretical Peak FLOPS per instruction set on modern Intel CPUs". In: URL: https://pdfs.semanticscholar.org/f9f5/aac6d506825be783e53318853d5eed153e92.pdf (visited on 12/01/2019).

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 (2015). "Pseudo-Random Number Generators for Vector Processors and Multicore Processors". In: *Journal of Modern Applied Statistical Methods* 14.23. URL: http://digitalcommons.wayne.edu/jmasm/vol14/iss1/13.

— (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).

GCC (2019a). *GCC libstdc++*. URL: https://github.com/gcc-mirror/gcc/tree/master/libstdc%2B%2B-v3 (visited on 12/01/2019).

— (2019b). *GCC, the GNU Compiler Collection*. URL: https://gcc.gnu.org/ (visited on 12/08/2019).

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. *RNGAVXLIB*. URL: http://cpc.cs.qub.ac.uk/summaries/AEIT_v3_0.html (visited on 11/30/2019).

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

— *Intel® Math Kernel Library*. URL: https://software.intel.com/en-us/mkl (visited on 11/30/2019).

— (2017a). *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).

— (2017b). *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).

— (2018a). *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*. Revision 003. 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).

— (2018b). *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).

— (2019a). *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Order Number: 248966-042b. URL: https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf?countrylabel=Colombia (visited on 12/02/2019).

— (2019b). *Intel® Embree. High Performance Ray Tracing Kernels*. URL: https://www.embree.org/ (visited on 12/02/2019).

Kirilov, Viktor (2019). *doctest. The Fastest Feature-Rich C++11/14/17/20 Single-Header Testing Framework for Unit Tests and TDD*. URL: https://github.com/onqtam/doctest (visited on 12/03/2019).

Klammler, Moritz (2016). *P0205R0: Allow Seeding Random Number Engines with std::random_device*. URL: http://open-std.org/JTC1/SC22/WG21/docs/papers/2016/p0205r0.html (visited on 12/01/2019).

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.

L'Ecuyer, Pierre and Richard Simard (2007). "TestU01. AC Library for Empirical Testing of Random Number Generators". In: *ACM Transactions on Mathematical Software (TOMS)* 33.4, p. 22.

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

Lemire, Daniel. *simdpcg*. URL: https://github.com/lemire/simdpcg (visited on 11/30/2019).

— *SIMDxorshift*. URL: https://github.com/lemire/SIMDxorshift (visited on 11/30/2019).

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).

— (2015a). *C++ Seeding Surprises*. URL: http://www.pcg-random.org/posts/cpp-seeding-surprises.html (visited on 12/03/2019).

— (2015b). *Ease of Use without Loss of Power*. URL: http://www.pcg-random.org/posts/ease-of-use-without-loss-of-power.html (visited on 12/01/2019).

— (2015c). *Everything You Never Wanted to Know about C++'s random_device*. URL: http://www.pcg-random.org/posts/cpps-random_device.html (visited on 12/01/2019).

— (2015d). *Simple Portable C++ Seed Entropy*. URL: http://www.pcg-random.org/posts/simple-portable-cpp-seed-entropy.html (visited on 12/03/2019).

— (2017a). *How to Test with TestU01*. URL: http://www.pcg-random.org/posts/how-to-test-with-testu01.html (visited on 12/03/2019).

— (2017b). *Too Big to Fail*. URL: http://www.pcg-random.org/posts/too-big-to-fail.html (visited on 12/04/2019).

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.

Pawellek, Markus (2017). "Generierung von Irradiance Maps". Bachelor's Thesis. Friedrich-Schiller-Universität Jena. URL: https://github.com/lyrahgames/irradiance-map-computation/blob/master/main.pdf (visited on 12/02/2019).

Pharr, Matt, Wenzel Jakob, and Greg Humphreys (2016). *Physically Based Rendering. From Theory to Implementation*. Third Edition. Morgan Kaufmann – Elsevier. ISBN: 978-0-12-800645-0. DOI: 10.1016/C2013-0-15557-2.

Saito, Mutsuo and Makoto Matsumoto (2008). "SIMD-Oriented Fast Mersenne Twister: A 128-bit Pseudorandom Number Generator". In: *Monte Carlo and Quasi-Monte Carlo Methods 2006*. Springer, pp. 607–622.

— (2017). *SIMD-oriented Fast Mersenne Twister (SFMT)*. URL: http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/index.html#SFMT (visited on 11/30/2019).

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

Song, R. and Melissa O'Neill (2016). *P0347R0: Simplifying simple uses of <random>*. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0347r0.html (visited on 12/01/2019).

Stroustrup, Bjarne (2014). *The C++ Programming Language*. Fourth Edition. Addison-Wesley – Pearson Education. ISBN: 978-0-321-95832-7.

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.

— (2018). *Xoshiro / Xoroshiro Generators and the PRNG Shootout*. URL: http://prng.di.unimi.it/ (visited on 11/30/2019).

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).

A Further Results

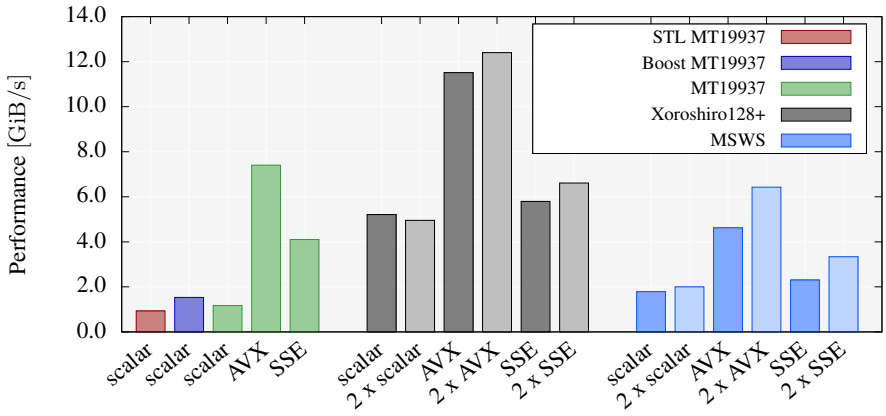


Figure 14: The plot shows the performance resulting from the generation benchmark running on the *Intel® Core™ i5-8250U Processor* measured in GiB of random numbers per second for the different variants of the implemented PRNGs. For convenience, the performances of the STL and Boost implementation of the MT19937 are shown as well. The data can be found in table 3.

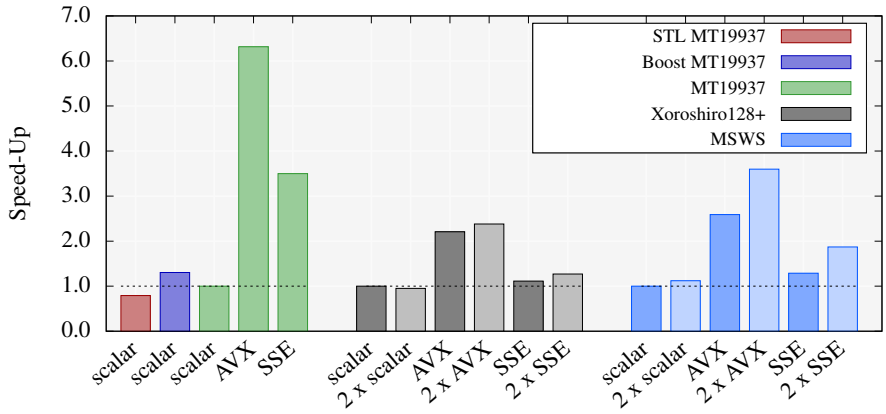
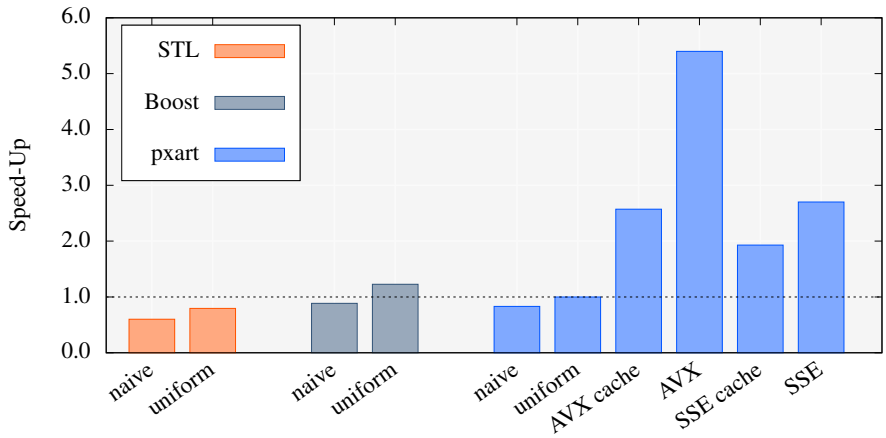


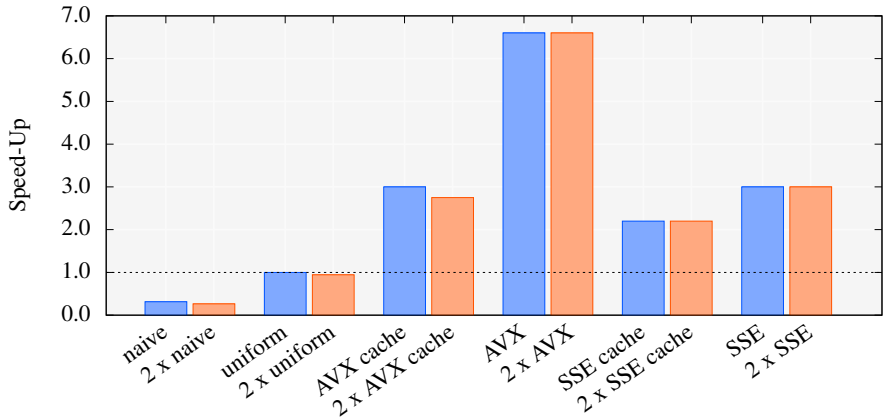
Figure 15: The plot shows the speed-up in execution time with respect to the single-instance scalar version resulting from the generation benchmark running on the *Intel® Core™ i5-8250U Processor* for the different variants of the implemented PRNGs. For convenience, the speed-ups of the STL and Boost implementation of the MT19937 are shown with respect to our implementation. The data can be found in table 3.

Table 3: The table shows the results achieved by running the generation benchmark on the *Intel® Core™ i5-8250U Processor* at a frequency of $4.51\,\mathrm{GHz}$ with all implemented variants of given PRNGs. While running the benchmark, $16\,\mathrm{GiB}$ of random numbers were generated and temporarily stored in a cache of size $16384\,\mathrm{B}$ by iterating 2^{20} times over its content. During the execution, there were no cache or branch misses. The values for cycles, instructions, and IPCs were averaged over the calls to the advancing routine of the respective generator.

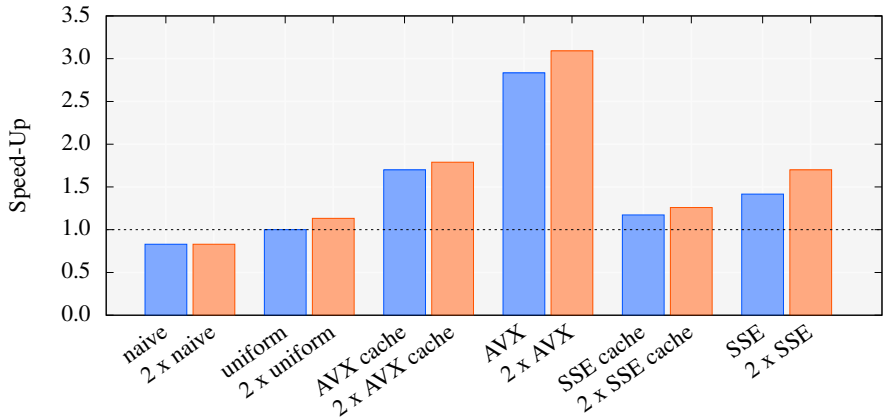
PRNG	**Variant**	**Result Size** [B]	**Time** [s]	**Cycles**	**Instructions**	**IPC**
STL MT19937	scalar	4	17.23	9.24	28.34	3.07
Boost MT19937	scalar	4	10.46	5.83	24.86	4.26
MT19937	scalar	4	13.64	7.61	26.71	3.51
	AVX	32	2.16	9.61	34.07	3.54
	SSE	16	3.90	8.73	34.04	3.90
Xoroshiro128+	scalar	8	3.07	3.45	12.01	3.48
	2 x scalar	8	3.23	3.62	12.01	3.31
	AVX	32	1.39	6.19	17.02	2.75
	2 x AVX	32	1.29	5.75	14.01	2.44
	SSE	16	2.76	6.15	17.01	2.77
	2 x SSE	16	2.42	5.43	14.01	2.58
MSWS	scalar	4	8.96	5.01	8.01	1.60
	2 x scalar	4	8.00	4.47	8.50	1.90
	AVX	32	3.46	15.42	31.02	2.01
	2 x AVX	32	2.49	11.05	24.51	2.22
	SSE	16	6.95	15.52	29.02	1.87
	2 x SSE	16	4.79	10.73	24.51	2.28



(a) MT19937



(b) Xoroshiro128+



(c) MSWS

Figure 16: The plot shows the speed-up in execution time with respect to the single-instance uniform version resulting from the Monte Carlo π benchmark running on the *Intel® Core™ i5-8250U Processor* for the different variants of the implemented PRNGs and benchmark scenarios. For convenience, the speed-ups of the STL and Boost implementation of the MT19937 are shown with respect to our implementation. The data can be found in table 4.

Table 4: The table shows the results achieved by running the Monte Carlo π Benchmark on the *Intel® Core™ i5-8250U Processor* with all implemented variants of given PRNGs and benchmark scenarios. While running the benchmark, 10^8 samples in the unit square were used to estimate the value of π. It was ensured that the estimation error was small enough according to the calculation at the end of section 3.1. During the execution, there were no cache or branch misses. The values for cycles, instructions, and IPCs were averaged over the number of samples in the unit square.

PRNG	**Benchmark**	**Time** [s]	**Cycles**	**Instructions**	**IPC**	**Frequency** [GHz]
STL MT19937	naive	0.90	29.96	85.68	2.86	3.31
	uniform	0.68	23.00	77.68	3.38	3.37
Boost MT19937	naive	0.61	20.69	66.73	3.22	3.37
	uniform	0.44	14.80	58.72	3.97	3.36
MT19937	naive	0.65	21.72	64.45	2.97	3.35
	uniform	0.54	15.84	60.43	3.81	2.95
	AVX cache	0.21	6.39	17.15	2.68	3.00
	AVX	0.10	2.99	9.39	3.14	2.87
	SSE cache	0.28	8.72	25.27	2.90	3.15
	SSE	0.20	5.79	18.77	3.24	2.96
Xoroshiro128+	naive	1.05	33.79	45.03	1.33	3.21
	2 x naive	1.25	41.31	53.02	1.28	3.29
	uniform	0.33	11.09	34.01	3.07	3.33
	2 x uniform	0.35	11.77	42.01	3.57	3.39
	AVX cache	0.11	3.82	11.75	3.07	3.38
	2 x AVX cache	0.12	3.88	12.75	3.28	3.30
	AVX	0.05	1.80	4.50	2.51	3.39
	2 x AVX	0.05	1.84	5.50	2.99	3.39
	SSE cache	0.15	5.01	15.50	3.10	3.37
	2 x SSE cache	0.15	5.26	17.25	3.28	3.39
	SSE	0.11	3.59	9.00	2.51	3.37
	2 x SSE	0.11	3.66	11.00	3.01	3.32
MSWS	naive	0.41	13.73	30.01	2.19	3.38
	2 x naive	0.41	13.91	38.01	2.73	3.37
	uniform	0.34	11.33	26.01	2.30	3.37
	2 x uniform	0.30	10.05	32.01	3.18	3.38
	AVX cache	0.20	6.82	14.88	2.18	3.38
	2 x AVX cache	0.19	6.52	15.88	2.43	3.37
	AVX	0.12	4.09	7.50	1.84	3.38
	2 x AVX	0.11	3.49	8.50	2.44	3.04
	SSE cache	0.29	9.39	21.51	2.29	3.25
	2 x SSE cache	0.27	9.14	23.25	2.54	3.38
	SSE	0.24	7.99	15.00	1.88	3.39
	2 x SSE	0.20	6.67	17.00	2.55	3.39

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, December 9, 2019

Markus Pawellek