Friedrich-Schiller-Universität Jena
Physikalisch-Astronomische Fakultät

# Design and Implementation of Vectorized Pseudorandom Number Generators and their Application to Simulations of Photon Propagation

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 19, 2019

**Abstract**

*The topic of this thesis is the development of a software library, called* pxart*, in the C++ programming language. It 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 processors. For this, a new interface was created to simplify their initialization and usage. Through the implementation of appropriate benchmarks and the simulation of photon propagation it was shown that the vectorized generators could indeed accelerate the execution of randomized algorithms.*

**Zusammenfassung**

*Das Thema dieser Arbeit ist die Entwicklung einer Softwarebibliothek, genannt* pxart*, in der Programmiersprache C++. Diese ermöglicht es, die Leistung auf Zufallszahlen basierender physikalischer Simulationen zu verbessern. Die Bibliothek besteht aus Pseudozufallszahlengeneratoren, die die SSE- und AVX-Befehlssätze moderner Intel Prozessoren nutzen. Es wurde eine neue Schnittstelle geschaffen, um deren Initialisierung und Nutzung zu vereinfachen. Durch die Implementierung geeigneter Benchmarks und der Simulation der Photonenausbreitung konnte gezeigt werden, dass die vektorisierten Generatoren die Ausführung randomisierter Algorithmen tatsächlich beschleunigen.*

## Acknowledgments

I am grateful to 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 and Johanna Vielemeyer for assisting in writing the thesis and proofreading.

# Contents

# List of Figures

# List of Tables

# List of Definitions and Theorems

# List of Code

# List of Abbreviations and Acronyms

| Abbreviation | Definition |
|---|---|
| iid | Independently and Identically Distributed |
| CDF | Cumulative Distribution Function |
| SLLN | Strong Law of Large Numbers |
| API | Application Programming Interface |
| RAII | Resource Acquisition is Initialization |
| SFINAE | Specialization Failure is not an Error |
| STL | Standard Template Library |
| ALU | Arithmetic Logic Unit |
| CPU | Central Processing Unit |
| IO | Input/Output |
| SISD | Single Instruction Single Data |
| SIMD | Single Instruction Multiple Data |
| MIMD | Multiple Instruction Multiple Data |
| AVX | Advanced Vector Extensions |
| SSE | Streaming SIMD Extensions |
| IPC | Instructions per Cycle |
| CPI | Cycles per Instruction |
| BSDF | Bidirectional Scattering Distribution Function |
| LTE | Light Transport Equation |
| RNG | Random Number Generator |
| TRNG | True Random Number Generator |
| PRNG | Pseudorandom Number Generator |
| LCG | Linear Congruential Generator |
| MSWS | Middle Square Weyl Sequence RNG |
| PCG | Permuted Congruential Generator |

# Symbol Table

| Symbol | Definition |
|---|---|
| **Logic** | |
| $\exists \ldots : \ldots$ | There exists . . ., such that . . . |
| $a := b$ | $a$ is defined by $b$. |
| **Set Theory** | |
| $\{\ldots\}$ | Set Definition |
| $\{\ldots \mid \ldots\}$ | Set Definition with Condition |
| $x \in A$ | $x$ is an element of the set $A$. |
| $A \subset B$ | The set $A$ is a subset of the set $B$. |
| $A \cap B$ | Intersection — $\{x \mid x \in A \text{ and } x \in B\}$ for sets $A$, $B$ |
| $A \cup B$ | Union — $\{x \mid x \in A \text{ or } x \in B\}$ for sets $A$, $B$ |
| $A \setminus B$ | Relative Complement — $\{x \in A \mid x \notin B\}$ for sets $A$, $B$ |
| $A \times B$ | Cartesian Product — $\{(x, y) \mid x \in A, y \in B\}$ for sets $A$ and $B$ |
| $A^n$ | $n$-fold Cartesian Product of Set $A$ |
| $\emptyset$ | Empty set — $\{\}$. |
| $\#A$ | Number of Elements in the Set $A$ |
| $\mathcal{P}(A)$ | Power Set of Set $A$ |
| **Special Sets** | |
| $\mathbb{N}$ | Set of Natural Numbers |
| $\mathbb{N}_0$ | $\mathbb{N} \cup \{0\}$ |
| $\mathbb{P}$ | Set of Prime Numbers |
| $\mathbb{Z}$ | Set of Integers |
| $\mathbb{Z}_n$ | Set of Integers Modulo $n$ |
| $\mathbb{F}_m$ | Finite Field with $m \in \mathbb{P}$ Elements |
| $\mathbb{F}_m^{p \times q}$ | Set of $p \times q$-Matrices over Finite Field $\mathbb{F}_m$ |
| $\mathbb{F}_2$ | Finite Field of Bits |
| $\mathbb{F}_2^n$ | Set of $n$-bit Words |
| $\mathbb{R}$ | Set of Real Numbers |
| $\mathbb{R}^n$ | Set of $n$-dimensional Real Vectors |
| $\mathbb{S}^2$ | Set of Directions — $\left\{x \in \mathbb{R}^3 \mid \|x\| = 1\right\}$ |
| **Functions** | |
| $f \colon X \to Y$ | $f$ is a function with domain $X$ and range $Y$. |
| $\mathrm{id}_X$ | Identity Function over the Set $X$ |
| $f \circ g$ | Composition of Functions $f$ and $g$ |
| $f^{-1}$ | Inverse Image of Function $f$ |
| $f^n$ | $n$-fold Composition of Function $f$ |
| **Bit Arithmetic** | |
| $x_{n-1} \ldots x_1 x_0$ | $n$-bit Word $x$ of Set $\mathbb{F}_2^n$ |
| $x \leftarrow a$ | Left Shift of all Bits in $x$ by $a$ |
| $x \rightarrow a$ | Right Shift of all Bits in $x$ by $a$ |
| $x \circlearrowleft a$ | Circular Left Shift of all Bits in $x$ by $a$ |
| $x \oplus y$ | Bit-Wise Addition of $x$ and $y$ |
| $x \odot y$ | Bit-Wise Multiplication of $x$ and $y$ |
| $x \mid y$ | Bit-Wise Or of $x$ and $y$ |

| Symbol | Definition |
|---|---|
| **Probability Theory** | |
| $\mathcal{B}(\mathbb{R})$ | Borel $\sigma$-Algebra over $\mathbb{R}$ |
| $(\Sigma, \mathcal{A})$ | Measurable Space over $\Sigma$ with $\sigma$-Algebra $\mathcal{A}$ |
| $\lambda$ | Lebesgue Measure |
| $\int_U f \,\mathrm{d}\lambda$ | Lebesgue Integral of $f$ over $U$ |
| $\mathrm{L}^2(U, \lambda)$ | Set of Square-Integrable Functions over the Set $U$ with Respect to the Lebesgue Measure $\lambda$ |
| $(\Omega, \mathcal{F}, P)$ | Probability Space over $\Omega$ with $\sigma$-Algebra $\mathcal{A}$ and Probability Measure $P$ |
| $\int_\Omega X \,\mathrm{d}P$ | Integral of Random Variable $X$ with respect to Probability Space $(\Omega, \mathcal{A}, P)$ |
| $\int_\Omega X(\omega) \,\mathrm{d}P(\omega)$ | $\int_\Omega X \,\mathrm{d}P$ |
| $P_X$ | Distribution of Random Variable $X$ |
| $\mathbb{E}\, X$ | Expectation Value of Random Variable $X$ |
| $\mathrm{var}\, X$ | Variance of Random Variable $X$ |
| $\sigma(X)$ | Standard Deviation of Random Variable $X$ |
| $\mathbb{1}_A$ | Characteristic Function of Set $A$ |
| $\delta_\omega$ | Dirac Delta Distribution over $\mathbb{S}^2$ with respect to $\omega \in \mathbb{S}^2$ |
| $\bigotimes_{n \in I} P_n$ | Product Measure of Measures $P_n$ Indexed by the Set $I$ |
| **Miscellaneous** | |
| $(x_n)_{n \in I}$ | Sequence of Values $x_n$ with Index Set $I$ |
| $\lvert x \rvert$ | Absolute Value of $x$ |
| $\lVert x \rVert$ | Norm of Vector $x$ |
| $x \mod y$ | $x$ Modulo $y$ |
| $\gcd(\rho, k)$ | Greatest Common Divisor of $\rho$ and k |
| $\max(x, y)$ | Maximum of $x$ and $y$ |
| $\lim_{n \to \infty} x_n$ | Limit of Sequence $(x_n)_{n \in \mathbb{N}}$ |
| $\sum_{k=1}^{n} x_k$ | Sum over Values $x_k$ for $k \in \mathbb{N}$ with $k \leq n$ |
| $\dim X$ | Dimension of $X$ |
| $\lceil x \rceil$ | Ceiling Function |
| $\langle x, y \rangle$ | Scalar Product |
| $[a, b]$ | $\{x \in \mathbb{R} \mid a \leq x \leq b\}$ |
| $(a, b)$ | $\{x \in \mathbb{R} \mid a < x < b\}$ |
| $[a, b)$ | $\{x \in \mathbb{R} \mid a \leq x < b\}$ |
| **Constants** | |
| $\infty$ | Infinity |
| $\pi$ | $3.1415926535\ldots$ — Pi |
| **Units** | |
| $1\,\mathrm{B}$ | $1\,\mathrm{Byte} = 8\,\mathrm{bit}$ |
| $1\,\mathrm{GiB}$ | $2^{30}\,\mathrm{B}$ |
| $1\,\mathrm{s}$ | $1\,\mathrm{Seconds}$ |
| $1\,\mathrm{min}$ | $1\,\mathrm{Minutes} = 60\,\mathrm{s}$ |
| $1\,\mathrm{GHz}$ | $1\,\mathrm{Gigahertz} = 10^9\,\mathrm{Hertz}$ |

# 1 Introduction

For various mathematical and physical problems, there exists no feasible, deterministic algorithm to solve them. Especially, the simulations of physical systems with many coupled degrees of freedom, such as fluids and global illumination, 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 quickly evaluate complex equations. (Bauke and Mertens 2007; Landau and Binder 2014; Müller-Gronbach, Novak, and Ritter 2012; Pharr, Jakob, and Humphreys 2016)[1]

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). 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. (Bauke and Mertens 2007; Intel 2018b; L'Ecuyer 1994, 2015; L'Ecuyer et al. 2017; Volchan 2002)

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 single instruction multiple data (SIMD) and multiple instruction multiple data (MIMD). Hence, to efficiently use the computing power of a processor for Monte Carlo algorithms not bound by memory, 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 constrains 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. Often, there are already manually vectorized algorithms that need to use random numbers, such as in Blacher (2018). As a consequence, vectorizing PRNGs is even essential to provide the utilities for these implementations. (Barash, Guskova, and Shchur 2017; Bauke and Mertens 2007; Dolbeau 2016; Fog 2019a,b,c,d,e; Hennessy and Patterson 2019; Intel 2019b; L'Ecuyer et al. 2017; Patterson and Hennessy 2014)

The C++ programming language is an adequate candidate for the development of vectorized PRNGs. It is one of the most used programming languages in the world and can be applied to small research projects as well as large enterprise programs. C++ 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 Intel's Streaming SIMD Extensions (SSE), Intel's Advanced Vector Extensions (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 latest language specification called C++17. The

[1] In this thesis, citations concerning a whole paragraph will be given after the last sentence.

language is evolving by its communities improvements and therefore it keeps to be modern. 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. (*cppreference.com*; Intel 2019a; Meyers 2014; Reddy 2011; *Standard C++ Foundation* 2019; Stroustrup 2014; Vandevoorde, Josuttis, and Gregor 2018)

Lots of PRNGs have been implemented by different libraries with different Application Programming Interfaces (APIs), such as the STL of C++, *Boost*, *Intel® Math Kernel Library*, and *RNGAVXLIB*. For example, the STL and *Boost* provide a large set of robust PRNGs which are not vectorized but well documented. Their API makes them likely to be used but shows some flaws. It does not allow to explicitly use the vectorization capabilities of a PRNG and gives you a bad default seeding. The use of standard distributions is difficult and not adjustable. *RNGAVXLIB* provides highly optimized, open-source, vectorized implementations with bad documentation and difficult-to-use code. The *Intel® Math Kernel Library*, too, provides vectorized PRNGs, but is not available as open-source and uses difficult-to-use interfaces. Besides complete libraries, there are also standalone implementations of vectorized PRNGs in Lemire (2018) and Lemire (2019). So far, there has been no easily-accessible, portable, and open-source software library written in C++ which gives a coherent, easy-to-use and consistent interface for vectorized PRNGs. (Boost 2019; GCC 2019a; Guskova, Barash, and Shchur 2016; Intel 2019d; O'Neill 2015c)

In this thesis, precisely in sections 6 and 7, we develop a new library, called *pxart*, in the C++ programming language[2]. *pxart* vectorizes a few 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 random facilities 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. The necessary theoretical background to understand the design- and the implementation-specific aspects is given in the sections 2, 4, and 5. Here, we will give a brief introduction to computer architecture, Monte Carlo methods and random sequences. Especially, section 5 will give a mathematically rigorous introduction to PRNGs to make the reader familiar with the topic. Section 3 refers to the previous work concerning the vectorization of PRNGs. To show that our vectorized implementations result in a performance improvement without reducing the statistical quality of generated random numbers, we have used standard test suites, namely *dieharder* and *TestU01*, and created benchmarks to measure the actual speed-up in section 8. At the end, we have designed an application based on Monte Carlo algorithms to simulate essential parts of photon propagation and global illumination to show the usage of our PRNGs in the context of physics. In the sections 9 and 10, the evaluation is shown followed by a discussion dealing with further improvements. (Brown 2019; L'Ecuyer and Simard 2007, 2009)

[2]Markus Pawellek (2019). *pxart. Packed Extensions for Advanced Random Techniques: Library and Applications for Random Number Generators*. URL: https://github.com/lyrahgames/pxart (visited on 12/11/2019).

# 2 Preliminaries

To systematically approach the implementation of PRNGs, basic knowledge in the topics of stochastics and statistics 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. Afterwards, we will give a brief overview of template mechanisms in the C++ programming language and the fundamentals of modern computer architecture.

## 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 section 5.

According to Schmidt (2009), Kolmogorov embedded probability theory in the theory of measurement and integration. Although it heavily relies on these theoretical structures, probability theory is one of the most important applications of measurement 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, pp. 193–195)

> **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 $\Omega$ the set of all possible outcomes of a random experiment.*

Due to the complex definition[3] 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 $X : \Omega \to \Sigma$ 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*

[3] Notation and symbols not directly defined are explained in the symbol table.

*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 uncountably many elements. Therefore the following definition of independence takes only a countable 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* $I \subset \mathbb{N}$ *and* $X_i$ *be a random variable for all* $i \in I$*. Then these random variables are independent if the following equation holds for all finite subsets* $J \subset I$ *whereby we denote the respective random vector with* $X_J := (X_i)_{i \in J}$.

$$P_{X_J} = \bigotimes_{i \in J} P_{X_i}$$

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). 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, pp. 274–276) 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. It uses a sequence of identically and independently distributed (iid) random variables. Please note, there exist many more variations of this theorem. We will 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* $\mu$*. 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 an adequate 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 which is 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. 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 keeps evolving 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 which will provides even more advanced features concerning template metaprogramming and concurrency. At the time of writing this thesis, C++17 is the 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 will 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. If 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 be simply ignored.

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

**`std::enable_if_t`** 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.

**`std::declval`** 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`.

**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.

## 2.3 Fundamentals of Computer Architecture

To be able to design and implement vectorized algorithms for SIMD architectures, 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.

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)

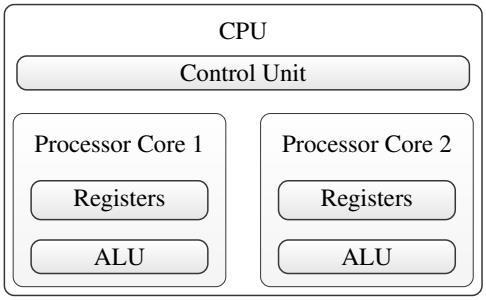



Figure 2.1: Basic components of a typical CPU with multiple processing cores in an hierarchical order. There is only one control unit which is handling communication between different cores by coordinating the execution of program instructions. Every processor core employs its own registers and ALUs to provide an MIMD architecture.

and the Advanced Vector Extensions (AVX) (Fog 2019a,b,c,d,e; Intel 2019a). 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. 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).

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 execute them, the control unit directs the coordinated operations of the ALU, registers and other components. Today, nearly every processor consists 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 2.1, 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

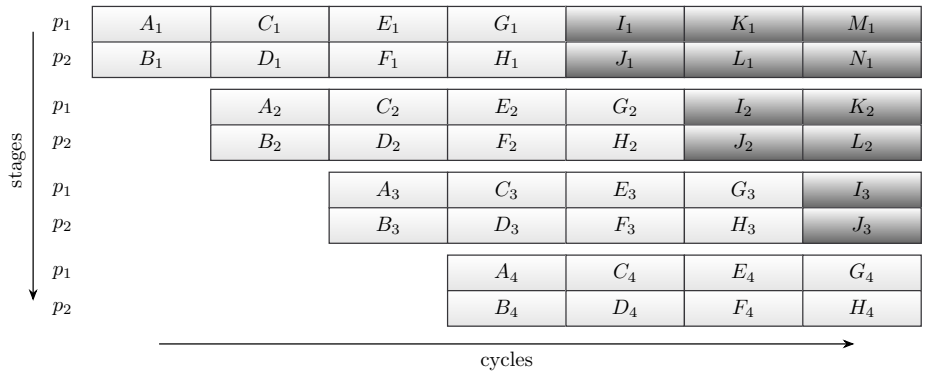



Figure 2.2: Functioning of a processor pipeline with four stages using two execution units $p_1$ and $p_2$. Assume all capital letters represent independent instructions with a latency of four and a throughput of one. For any instruction $X$, we denote its four stages by $X_1$, $X_2$, $X_3$, and $X_4$. After filling the pipeline, every cycle the results of two instructions will be available to the caller.

operation. Hence, with respect to the microarchitecture, instructions suddenly exhibit physical properties, like the time to execute the given instruction. These 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. (Intel 2008)

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 performs the different stages of different instructions concurrently according to their throughput. As a consequence, a pipeline does not reduce the latency of an instruction but reduces its throughput. It is therefore a form of instruction-level parallelism. To further decrease the throughput of a sequence of instructions, the processor core typically uses multiple execution units, also known as ports, to run independent instructions directly in parallel. A pipeline with two execution units executing a sequence of independent instructions with a latency of four and a throughput of one is shown in figure 2.2. In real-world processors, different instructions typically exhibit different latencies and throughputs. The varying capabilities of different execution units concerning their instruction set do not allow instruction-level parallelization for arbitrary instructions. (Dolbeau 2016; Fog 2019b,c; Intel 2018a, 2019a)

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 efficiently handle control instructions, especially branches, the CPU uses branch prediction. The processor tries to guess the outcome of the

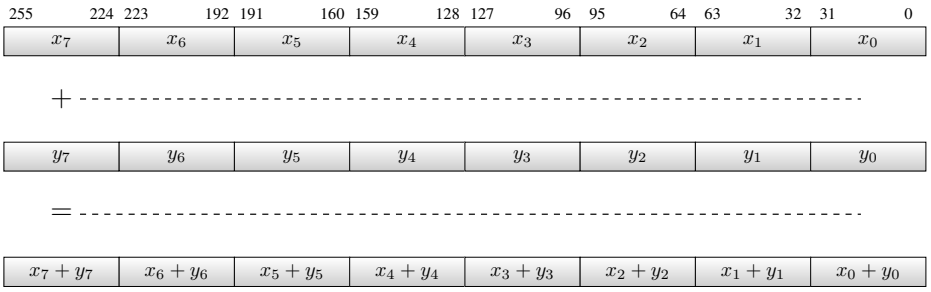



Figure 2.3: Functioning of SIMD vector registers and their operations by means of an addition applied to two 256-bit register each containing eight packed 32-bit values. The result is again a 256-bit vector consisting of eight 32-bit values.

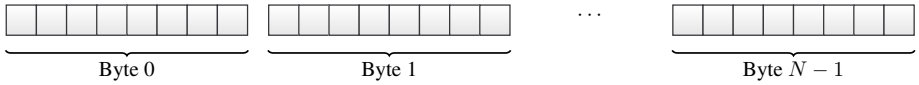



Figure 2.4: Visualization of memory with $N \in \mathbb{N}$ bytes as a sequence of bits where each byte can be referenced by its memory address.

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. (Fog 2019c; Hennessy and Patterson 2019; Patterson and Hennessy 2014)

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-bit register of a processor featuring the AVX instruction set could contain four 64-bit double-precision or eight 32-bit single-precision floating-point numbers. One operation, like addition or multiplication, is then performed on all contained elements simultaneously. Advanced computations demand the use permutation and shuffle operations which are changing the order of elements inside the register to be able to use SIMD operations. Figure 2.3 shows the addition operation using two 256-bit registers each with eight 32-bit values. The choice of 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-bit double precision values reducing the throughput instead of eight single precision values. Implementations of algorithms which are not using any SIMD utilities will be denoted as scalar implementations and as vectorized implementations otherwise. The process of transforming a scalar implementation into a vectorized implementation is called vectorization. (Dolbeau 2016; Fog 2019b,c; Intel 2019b)

### 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 2.4. Fetching instructions from memory or transferring data between the CPU and memory,

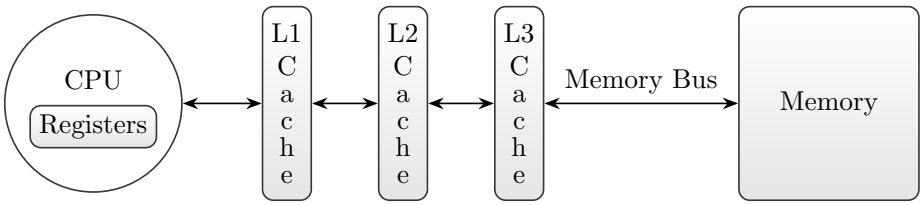



Figure 2.5: Non-persistent part of the memory hierarchy for a modern laptop or desktop computer. Between the main memory and the CPU registers different levels of caches are used to reduce the latency for fetching instructions and data. Modeled after Hennessy and Patterson (2019, p. 79).

therefore requires the usage of those memory addresses to be able to reference data in the sequence of bytes. Each byte can be altered by program execution through storing instructions. (Patterson and Hennessy 2014)

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. Further reduction of the memory access latency is achieved by always fetching a fixed-length group of bytes which are called a cache line at once and by prefetching consecutive cache lines for content that is expected to be needed soon possibly wasting time if the data will not be used. Modern architectures commonly featuring a cache line size of $64\,\mathrm{byte}$. Figure 2.5 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 L1 cache which is further split into an instruction cache and a data cache (Intel 2018a). This reduces the overall complexity of L1 caches and as a result decreases the cache access time. For example, the L1 cache of the *Intel® Core™ i7-7700K Processor* has a size of $256\,\mathrm{KiB}$ divided into two 128-KiB caches for instructions and data. The L2 cache has a size of $1\,\mathrm{MiB}$ and the L3 cache a size of $8\,\mathrm{MiB}$. (Hennessy and Patterson 2019, pp. 78–83; Intel 2017b, 2018a)

Let $n := 2^k$ for some $k \in \mathbb{N}_0$ be a power of two. We say that a variable in memory is $n$-byte aligned if its starting address is divisible by $n$ without remainder. Due to the processor microarchitecture and the memory hierarchy variables either have to exhibit a certain alignment or enable faster access times when properly aligned. Modern systems typically provide a 16-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-byte alignment. The AVX instruction set of Intel CPUs is working with 256-bit vector registers which do not have to be 32-byte aligned but should allow for a slightly improved performance otherwise. Since the common sizes of cache lines today is 64 byte, arrays of data should be even 64-byte aligned to get the highest performance in a computation iterating over the data. (Fog 2019c)

Comparing the time of executing instructions in the CPU and fetching data from main memory, the memory access will in general be a hundred times higher. The construction of processors and memory hierarchies therefore separates algorithms into two groups — memory-bound and CPU-bound algorithms. The performance of memory-bound algorithms is mainly determined by the access time and latency when communicating with memory or caches. For example, this would be the case if a computation needs to access a lot of different data elements which are not fitting into the caches from memory and to use them only once. The performance of an algorithm operating on data completely fitting into the caches or even CPU registers is typically bound by the latency and throughput of its instructions and the frequency of the processor. Hence, we call it a CPU-bound algorithm. Optimizing an algorithm for instruction-level and data-level parallelism by using SIMD vectors and tweaking latencies and throughput only makes sense if the processor does not have to wait for data from memory — the algorithm has to be bound by the CPU. (Hennessy and Patterson 2019; Patterson and Hennessy 2014)

### SIMD Instruction Set Features in C++

Exploiting the SIMD capabilities on modern Intel processors is done by using the SSE and AVX instruction set features of the underlying microarchitecture in the assembler code. 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 to vectorize the code and we as programmers have to fall back to a manual alternative. The behavior of automatic vectorization can vary on different machine architectures. As a consequence, the improvement in performance based on automatic vectorization is typically not portable. (Fog 2019a,b,c,d,e)

To get the best of both techniques, Intel provides so-called SIMD intrinsics for the SSE and AVX instruction sets. These 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. Furthermore, the code will be compiled to the same instructions on different machine architectures which makes it performance-portable. 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

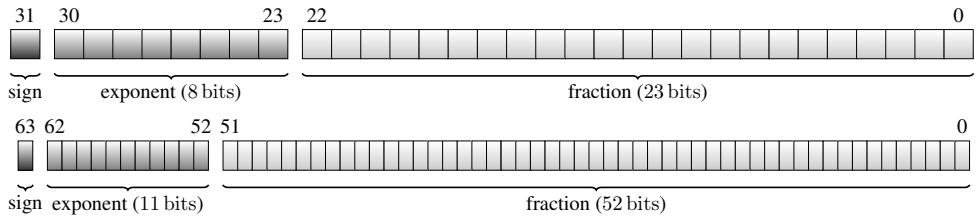



Figure 2.6: Single- (top) and double-precision (bottom) floating-point layout according to the IEEE 754 standard. The boxes represent the bits of the underlying registers and the numbers above them mark the bit index. Both encodings consist of a sign, a fractional part and an exponent. The figure was modeled after Patterson and Hennessy (2014, pp. 197–198).

different microarchitectures is given by Intel (2019a) and Fog (2019b). As a consequence, we rely on this variant to develop the vectorized implementations of PRNGs and algorithms. (Fog 2019a,b,c,d,e; Intel 2019b)

### Floating-Point Representation

Generating a sequence of random numbers uniformly distributed in the unit interval requires the result to be a floating-point value. Thus, we have to be able to transform a sequence of random bits representing an unsigned integer to real numbers. Today, types representing real numbers in a computer are typically encoded in the floating-point format that does not fix the binary point (Patterson and Hennessy 2014). Hence, real values can be approximated with an adaptive precision. For our purposes, the format can be seen to be superior to other fixed-point formats. The IEEE standard for floating-point arithmetic, also called IEEE 754, is a technical standard describing the single- and double-precision floating-point encodings used in modern processors. We will give a brief definition of the representation of normalized numbers and refer to Patterson and Hennessy (2014, pp. 196–220) for more information. Figure 2.6 shows the bit layouts for both encodings schematically.

**DEFINITION 2.6:** Floating-Point Format for Normalized Numbers

*Let $p, q \in \mathbb{N}$ be the numbers of bits used for the exponent and the fraction respectively. Furthermore, let $b \coloneqq (b_i)_{i=0}^{p+q}$ be a finite sequence of bits in $\mathbb{F}_2$. We define the sign $s$, the exponent $e$, the offset $o$, and the significand $m$ as follows.*

$$s \coloneqq b_{p+q} \qquad o \coloneqq 2^{p-1} - 1 \qquad e \coloneqq \sum_{k=0}^{p-1} b_{k+q} 2^k \qquad m \coloneqq 1 + \sum_{i=1}^{q} b_{q-i} 2^{-i}$$

*Assume $b$ represents a normalized number, that is $0 < e < 2^p - 1$. Then we call the value $b$ the $(p, q)$-precision floating-point encoding of $x$.*

$$x = (-1)^s \cdot m \cdot 2^{e-o}$$

*For $p = 8$ and $q = 23$, we call $b$ a single-precision floating-point encoding. For $p = 11$ and $q = 52$, $b$ will be called a double-precision floating-point encoding.*

# 3 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. The respective mathematical concepts and properties concerning PRNGs are given in section 5.

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 are 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 standard Mersenne Twister, called 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 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-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 supporting AVX or even AVX512 will not make full use of the complete vector registers because the algorithm is specialized for the older SSE registers.

With the *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 to change. (Intel 2019d)

Barash, Guskova, and Shchur (2017) describe how to implement a lot of PRNGs by using the AVX instruction set facilities. They, too, provide a library, called *RNGAVXLIB*, for academic use with an implementation of the MT19937 even surpassing Intel's implementation (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 makes it unreadable, difficult to understand and not generalizable to other SIMD architectures.

In Lemire (2018) and Lemire (2019), Lemire supplies two open-source code snippets for the vectorization of a permuted congruential generator (PCG) and a Xorshift generator with Intel's AVX and AVX512 instruction set features by using the respective intrinsics in C++. 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 which is also known as Xoroshiro128+. 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. Blacher (2018) already uses a vectorized implementation of the Xoroshiro128+ to calculate the pivot of randomized and vectorized quicksort implementation without further testing its performance or quality.

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 Middle Square Weyl Sequence RNG (MSWS). The idea is to raise the performance of the Xoroshiro128+ in comparison to an automatic vectorization process and to measure its statistical quality and speed-up. The MSWS has not been vectorized, yet. It is a 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.

# 4 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 to 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, section 1, the simulation of physical and mathematical systems can be quite time intensive. Many degrees of freedom in a resulting partial differential equation make 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 4.1:** 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 $d \in \mathbb{N}$, 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 4.1:** 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.*

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

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".

## 4.1 Monte Carlo Integration and the Computation of $\pi$

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. (Müller-Gronbach, Novak, and Ritter 2012)

**DEFINITION 4.2:** 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.*

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

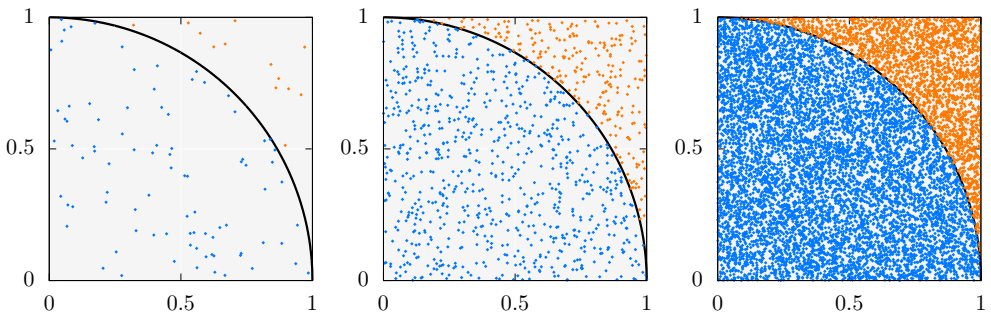
The domain of definition has to be restricted so that the method has a chance of reducing the overall estimation error. Additionally, the function $f$ should be square-integrable such that we are able to get an upper bound on the standard deviation. The following lemma will show that Monte Carlo integration is indeed a Monte Carlo method with the properties of a direct simulation. (Müller-Gronbach, Novak, and Ritter 2012)

**LEMMA 4.2:** Monte Carlo Integration Estimates Value of Integral

*Choose the same setting as in the above definition 4.2. 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.*

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

In the proof, seen in appendix A, 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




(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 4.7: Sample points for different realizations of the Monte Carlo integration $\mathrm{MCI}_n(f)$ for the computation of $\pi$. Points that lie in the quarter of the unit circle are shown in 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 $\mu$ and the relative error with respect to $\pi$ is expressed by $\varepsilon$. The variable $s$ names the count of samples that lie in the unit circle.

deviation. To get a deeper understanding of this method, consider the estimation of $\pi$ by Monte Carlo integration. For this, we would like to compute the area of a quarter of a circle which is strongly related to $\pi$. Figure 4.7 shows the execution of the following process for different realizations.

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

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

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

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

Simulating the integral of $f$ will therefore give us an estimation of $\pi$. For a more detailed analysis, we will also compute the exact standard deviation of this Monte Carlo integration by using the above lemma 4.2.

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

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

$$\sigma\left(\mathrm{MCI}_n(f)\right) = \sqrt{\frac{\pi(4-\pi)}{n}} \leq 2\sqrt{\frac{\pi}{n}} = \sqrt{\frac{\lambda(U)}{n} \int_U f^2 \,\mathrm{d}\lambda}$$

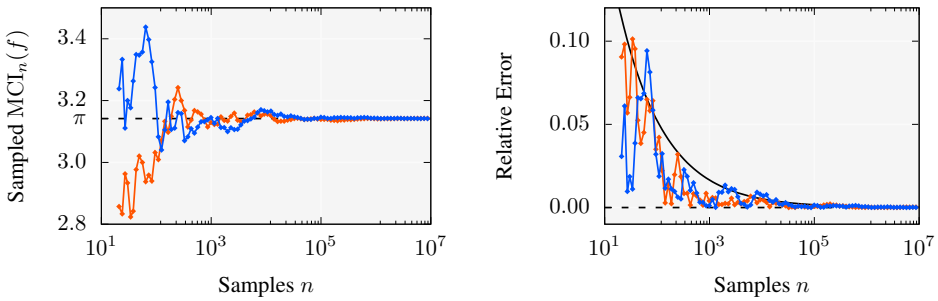



Figure 4.8: Each of the diagram shows two different versions, colored in orange and blue, of realizations of the Monte Carlo integration $\mathrm{MCI}_n(f)$ for the computation of $\pi$ for different values of $n$. The left one displays the estimated value of $\pi$ and the right one displays its relative error with respect to $\pi$. The black line hereby shows the exact relative standard deviation with respect to $\pi$ of the Monte Carlo integration.

In figure 4.8, 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 4.7. As a consequence, the estimation of $\pi$ will be more precise. Furthermore, figure 4.8 shows that the error of the estimation can also be estimated and will be more precise for a larger sample count. (Landau and Binder 2014; Müller-Gronbach, Novak, and Ritter 2012; Pharr, Jakob, and Humphreys 2016)

Later, we will use the computation of $\pi$ as benchmark routine to measure the performance of PRNGs with respect to different aspects of their implementation. In these cases, the error of $\pi$ should be in some given range. Assume we want to use $10^8$ samples to estimate the value of $\pi$. 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 $\pi$ should approach the value $3.1415$ with a varying last digit.

To use the described Monte Carlo integration for estimating $\pi$, an actual implementation in C++ is needed. The following code snippet provides the basic algorithm relying on the random utilities given by the 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.

Code 4.1: Basic Monte Carlo $\pi$ Computation

```
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 a random two-dimensional point in the unit square. We then evaluate the circle condition and add its result to `samples_in_circle` incrementing it by one if the condition is true and by zero otherwise. At the end, the code computes the correct result by calculating the division and scaling the output.

The computation of $\pi$ 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 common 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 forms as the computation of $\pi$. (Landau and Binder 2014; Müller-Gronbach, Novak, and Ritter 2012; Pharr, Jakob, and Humphreys 2016)

## 4.2 Photon Propagation and Physically Based Rendering

A photon is an elementary particle and the quantum of the electromagnetic field. It exhibits properties of both waves, like wavelength and polarization, and particles, such as position and momentum. This is also known as wave-particle duality. In vacuum, photons always move at the speed of light. Their invariant mass is zero and therefore they are thought of as massless particles. Due to Heisenberg's uncertainty principle, we are in general not able to measure their position and momentum at the same time with infinite precision. Because we are concerned about the photon propagation in space at a large scale, we will use an abstract concept of a photon to approximate physical laws and making simulations feasible. We think of an abstract photon as a packet of photons with a definite position and velocity representing their respective expectation values of all the photons inside the packet. Since the positions and velocities are then based on statistical calculations, this will allow us to ignore the uncertainty principle. Additionally, we assume no particular polarization or wavelength in the packet can be determined. Thus, the wavelength will be ignored and the abstract photon will be seen as unpolarized.

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 conse-


Figure 4.9: Example scene showing an Audi R8 rendered by the path tracing algorithm. The simulation included ideal reflections and refractions, as well as the diffuse scattering of light at the surfaces. The simulation used 1024 samples per pixel to reduce the noise of the image. (Pawellek 2017)

quence, 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. Figure 4.9 shows an example scene rendered by the path tracing algorithm for which more than a thousand samples per pixel had to be generated. 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. (Pharr, Jakob, and Humphreys 2016)

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 packets of photons. Light rays are used to gather the radiance arriving at the observer whereas photons propagate flux from a light source. At large scale, 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 seems to be convenient 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 2019c).

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

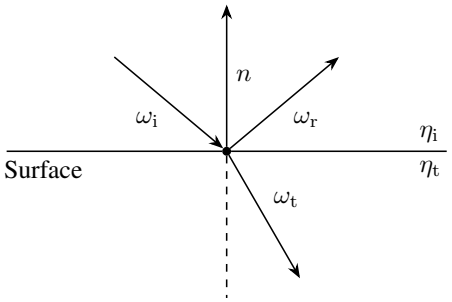



Figure 4.10: Ideal reflection and refraction scheme and notation. The surface with surface normal $n$ separates the outer medium with refractive index $\eta_\mathrm{i}$ from the inner medium with refractive index $\eta_\mathrm{t}$. $\omega_\mathrm{i}$, $\omega_\mathrm{r}$, and $\omega_\mathrm{t}$ denote the incoming, reflection, and refraction direction, respectively.

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 be sampled with the usage of appropriate random variables. In this context, we will only assume ideal reflection and refraction of photons to not introduce unnecessary complexness to the application.

Ideal reflection of a photon means that the angle between its incoming direction and the surface normal is the same as the angle between the outgoing direction and the surface normal. In general, we can define it as follows.

**DEFINITION 4.3:** Ideal Reflection

*For any incoming direction $\omega_\mathrm{i} \in \mathcal{S}^2$ its ideally reflected direction $\omega_\mathrm{r}$ with respect to a surface with surface normal $n \in \mathcal{S}^2$ is given by the following equation.*

$$\omega_\mathrm{r} = \omega_\mathrm{i} - 2 \langle n, \omega_\mathrm{i} \rangle n$$

A refractive index is a dimensionless number and represents the slow-down in speed when light is traveling through the according medium. The ideal refraction of photons traveling through a surface which is separating two media with varying refractive indices can be described by the following definition.

**DEFINITION 4.4:** Ideal Refraction

*For any incoming direction $\omega_\mathrm{i} \in \mathcal{S}^2$ its ideally refracted direction $\omega_\mathrm{t}$ with respect to a surface with surface normal $n \in \mathcal{S}^2$ is given by the following equation.*

$$\omega_\mathrm{t} = \begin{cases} \frac{\eta_\mathrm{i}}{\eta_\mathrm{t}} \left(\omega_\mathrm{i} - \langle n, \omega_\mathrm{i} \rangle\, n\right) - \sqrt{\xi} n & : \xi > 0 \\ \omega_\mathrm{r} & : \xi \leq 0 \end{cases}$$

$$\xi \coloneqq \frac{\eta_\mathrm{i}^2}{\eta_\mathrm{t}^2} \left(1 - \langle n, \omega_\mathrm{i} \rangle^2\right)$$

*Hereby, we assume that $\omega_\mathrm{r}$ is the ideally reflected direction and that the surface is separating the outer medium with refractive index $\eta_\mathrm{i} \in [1, \infty)$ from the inner medium with refractive index $\eta_\mathrm{t} \in [1, \infty)$.*

Figure 4.10 shows ideal reflection and refraction schematically. The connection between the angle of incidence and the angle of emergence is given by Snell's law. If the outer medium exhibits a larger refractive index than the inner medium then there exists a critical incident angle at which total reflection of the incoming photon occurs. This explains the distinction of cases in the above definition. Here, the second case represents total reflection.

A photon can either be reflected or transmitted but not both at the same time. Hence, the surface coefficients $\alpha, \beta \in [0, 1]$ for reflection and refraction with $\alpha + \beta = 1$ of photons should be interpreted as probabilities. The interaction process can then be described by a probability density $p_{\omega_\mathrm{i}}$ over the outgoing directions $\omega_\mathrm{o} \in \mathcal{S}^2$ with a fixed incoming direction $\omega_\mathrm{i} \in \mathcal{S}^2$. In the case of ideal reflection $\omega_\mathrm{r}$ and refraction $\omega_\mathrm{t}$, we have

$$p_{\omega_\mathrm{i}}(\omega_\mathrm{o}) \coloneqq \alpha \delta_{\omega_\mathrm{o}}(\omega_\mathrm{r}) + \beta \delta_{\omega_\mathrm{o}}(\omega_\mathrm{t})$$

where $\delta_{\omega_\mathrm{o}}$ denotes the Dirac delta distribution over $\mathcal{S}^2$. To sample the photon paths from such a probability density, Russian roulette is used to choose between reflection and refraction.

**LEMMA 4.3:** Russian Roulette

*Choose the same settings as in definition 4.2. Furthermore, assume $\alpha, \beta \in [0, 1]$ with $\alpha + \beta = 1$ and $f = \alpha f_1 + \beta f_2$ for functions $f_1, f_2 \in \mathrm{L}^2(U, \lambda)$. We define the roulette function $R$ as follows.*

$$R \colon [0, 1] \to \{1, 2\} \qquad R(x) \coloneqq \begin{cases} 1 & : x \leq \alpha \\ 2 & : \text{else} \end{cases}$$

*In this case, for all $x \in U$ and $k \in \mathbb{N}$ the following equation holds.*

$$f(x) = \mathbb{E}\, f_{R \circ X_k}(x)$$

We will give no proof of this lemma. Russian roulette does not introduce any bias and allows us to only choose one path for the photon. This comes at the cost of a higher variance of the estimation. If we would evaluate both, reflection and refraction, at once and scale the results according to $\alpha$ and $\beta$ then the estimation would have a variance of zero. But always tracing the possible paths soon becomes infeasible due to the exponential growth of different paths for every surface interaction. Russian roulette reduces this computational amount.

### Volume Scattering

Photons propagating through participating media are affected by absorption and scattering. Both processes are in general modeled by statistical processes. For our purpose, absorption will be ignored to reduce the complexity of the physical simulation. Scattering changes the direction of the photon. This process is described by rotation matrices and therefore scattering is much more appropriate for vectorization than absorption. (Pharr, Jakob, and Humphreys 2016)

As a photon passes through a medium, it may collide with other particles and be scattered in different directions. The probability of a scattering event is modeled by the scattering coefficient which in our case equals to the extinction. We will only assume homogeneous media with a constant extinction. Hence, the actual probability of a scattering event after traveling a short distance is given by Beer's law. Because we want to show the application of RNGs to physical simulations, it is sufficient to directly define this probability without using the extinction. (Pharr, Jakob, and Humphreys 2016)

Scattering itself is described by phase functions which are probability densities. Based on an incoming direction, they give the probability that the photon will be scattered in a certain outgoing direction. In this thesis, isotropic phase functions are used. They are invariant under rotations and therefore only depend on the angle between the incoming and outgoing direction. According to Pharr, Jakob, and Humphreys (2016), a widely used family of phase functions was developed by Henyey and Greenstein in the year 1941. This family cannot be deduced by physical laws but instead was specifically designed to be easy to fit to measured scattering data by only using a single parameter. (Pharr, Jakob, and Humphreys 2016; Wang and Jacques 1995)

**DEFINITION 4.5:** Henyey-Greenstein Family of Phase Functions

*Let $g \in [-1,1]$ be the asymmetry parameter, $\omega_\mathrm{i} \in \mathcal{S}^2$ be the incoming direction of a photon and $\omega_\mathrm{o} \in \mathcal{S}^2$ be a possible outgoing direction. The probability density $p\colon [-1,1] \to [0,\infty)$ describing the distribution of $\cos\vartheta := \langle \omega_\mathrm{i}, \omega_\mathrm{o} \rangle$ after a scattering process has taken place is then given by the following equation.*

$$p_g(\cos\vartheta) := \frac{1}{2} \frac{1-g^2}{(1+g^2-2g\cos\vartheta)^{\frac{3}{2}}}$$

To actually use the probability density $p_g$, we have to sample it. That means we have to construct a sequence of iid random variables with a distribution given by $p_g$. For this we will rely on a uniformly distributed sequence of iid random variables by applying a transformation to change their distribution. The following lemma can be shown by using the inversion method. A detailed derivation is given in Wang and Jacques (1995).

**LEMMA 4.4:** Sampling Henyey-Greenstein Phase Functions

*Let $g \in [-1,1]$ be the asymmetry parameter, $(X_n)_{n\in\mathbb{N}}$ be a uniformly distributed sequence of iid random variables in the unit interval. Furthermore, let $(\cos\vartheta_n)_{n\in\mathbb{N}}$*

*be a sequence of random variables given by the following transformation.*

$$\cos\vartheta_n = \begin{cases} \frac{1}{2g}\left[1+g^2-\left(\frac{1-g^2}{1-g+2gX_n}\right)^2\right] & : g \neq 0 \\ 2X_n - 1 & : g = 0 \end{cases}$$

*Then,* $(\cos\vartheta_n)$ *is a sequence of iid random variables distributed according to* $p_g$*.*

The given lemma makes it possible to simulate scattering processes adhering to the probability density $p_g$ only by using uniformly distributed random numbers in the unit interval.

# 5 Pseudorandom Number Generators

## 5.1 Random Sequences

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

The first problem about randomness is the word itself. It is unclear and vague because there is no intentional application. To be more specific, we will observe randomness in form of random sequences of real numbers. But as stated in Volchan (2002) the question if a sequence is random decides at infinity. As long as we are only observing finite sequences, we cannot decide if such a sequence is the outcome of a truly random experiment or the result of a non-random algorithm. Following his explanation, Volchan makes it 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 independently 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. A PRNG generates numbers according to a deterministic algorithm. Therefore its output cannot be seen as truly random. But 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 (PRNG)

*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 5.11 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*

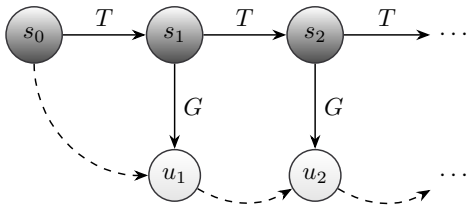



Figure 5.11: Generation of a pseudorandom sequence for a given PRNG $\mathcal{G} \coloneqq (S, T, U, G)$ and seed value $s_0 \in S$. The internal state is advanced by the transition function $T$ through a recurrence relation. To get an output value for the pseudorandom sequence the generator function $G$ is used.

*following equation for all* $n \in \mathbb{N}$.

$$s_{n+1} \coloneqq T(s_n)$$

*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* $s_0$.

$$u_n \coloneqq G(s_n)$$

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* $\mathcal{G} \coloneqq (S, T, U, G)$ *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 mask its deterministic nature very well (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 retain 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 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 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 subsection, 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.

### 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 $\rho$ and $\tau$, such that the equation holds, are called period and transient respectively. In particular, if $\tau$ equals to $1$ we call $(u_n)$ periodic with period $\rho$.*

This means an ultimately periodic sequence will be periodic after it has 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 $\rho$ and $\tau$ are not unique. Let $\rho^*$ be the period and $\tau^*$ be the transient. Then the equation given in the above definition holds for all the values of $\rho$ and $\tau$ 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. The proof can be found in appendix A.

**LEMMA 5.2:** Pseudorandom Sequences are Ultimately Periodic

*Let $\mathcal{G} \coloneqq (S, T, U, G)$ 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 $\rho$ and the transient $\tau$ the following holds.*

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

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

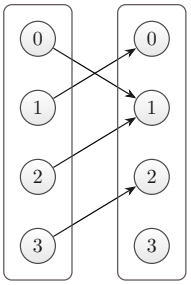
Thus, every pseudorandom sequence will repeat itself after it reached a certain point. The period and the transient are greatly affected by the number of states and the transition function of the PRNG. To get a better insight, we will examine the following idealized examples with different transition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

$$S \coloneqq U \coloneqq \mathbb{Z}_4 \qquad G \coloneqq \mathrm{id}$$

For a seed $s_0 \in S$ the respective pseudorandom sequence $(u_n)_{n\in\mathbb{N}}$ with period $\rho$ and transient $\tau$ 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.

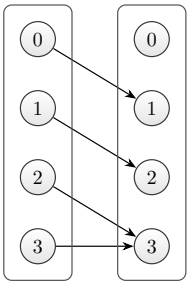
$$(u_n) = u_1 \dots u_{\tau-1}\overline{u_\tau \dots u_{\tau+\rho-1}}$$

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 $T$.



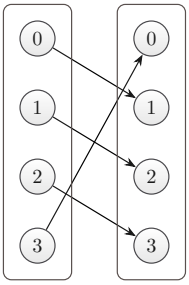

$$T(x) \coloneqq \begin{cases} 1 & : x \in \{0, 2\} \\ 0 & : x = 1 \\ 2 & : x = 3 \end{cases} \qquad (u_n) = \begin{cases} \overline{10} & : s_0 \in \{0, 2\} \\ \overline{01} & : s_0 = 1 \\ 2\overline{10} & : s_0 = 3 \end{cases}$$

The first example shows a transition function which is not bijective and 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.



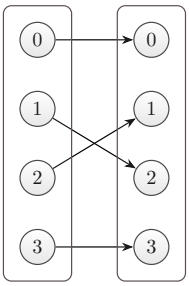

$$T(x) \coloneqq \begin{cases} x + 1 & : x < 3 \\ 3 & : x = 3 \end{cases} \qquad (u_n) = \begin{cases} 12\overline{3} & : s_0 = 0 \\ 2\overline{3} & : s_0 = 1 \\ \overline{3} & : s_0 \geq 2 \end{cases}$$

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.




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

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.




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

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.

### 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 $\mu$ 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 $\mu$ 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 $\mu$ 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 $\mu$ 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 in appendix A 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 $\tau$ and period $\rho$. Furthermore, let $\mu$ be a probability measure on $(U, \mathcal{P}(U))$. Then the following statements are equivalent.*

*(i) $(u_n)$ is equidistributed with respect to $\mu$.*

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

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

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 $\mathcal{G} \coloneqq (S, T, U, G)$ 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.*

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

### 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. 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 $n \in \mathbb{N}$.*

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

*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 5.12 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 $\rho$ and transient $\tau$. In this case, every corresponding $k$-dimensional*

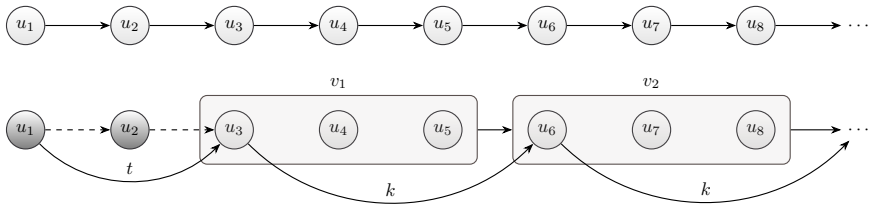



Figure 5.12: The upper part of the figure shows a schematic view of an arbitrary sequence of values $(u_n)_{n\in\mathbb{N}}$ in an arbitrary non-empty set $U$. The lower part visualizes the corresponding $k$-dimensional vector sequence $(v_n)_{n\in\mathbb{N}}$ with translation $t$, whereby $k = 3$ and $t = 2$. 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.

> *vector sequence $(v_n)_{n\in\mathbb{N}}$ with translation $t$ is ultimately periodic with period $\rho'$ and transient $\tau'$ defined as follows.*
>
> $$\rho' := \frac{\rho}{\gcd(\rho, k)} \qquad \tau' := \left\lceil \frac{\max(0, \tau - 1 - t)}{k} \right\rceil + 1$$

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 $\mu$ 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 $\mu$ if for all $t \in \mathbb{N}_0$ the corresponding $k$-dimensional vector sequence with translation $t$ is equidistributed with respect to $\mu$. If $\mu$ 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 $\rho$. 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 $\mathcal{G} := (S, T, U, G)$ be a PRNG, $s_0 \in S$ its initial state and $(u_n)_{n\in\mathbb{N}}$ the respective pseudorandom sequence with period $\rho$. The corresponding $k$-dimensional vector sequence with translation $t$ will be called $(v_n)_{n\in\mathbb{N}}$. Sequences will 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 $k = 3$.

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

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

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 $k = 3$.

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

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

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 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.

### 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 $\mathcal{G} \coloneqq (S, T, U, G)$ be a linear PRNG modulo $m$ with $p \coloneqq \dim S$. Furthermore, let the characteristic polynomial of $T$ be a primitive polynomial over $\mathbb{F}_m$ and let $G$ be a full rank matrix with $q \coloneqq \operatorname{rank} G$. 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.*

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

*In particular, the sequence $(u_n)$ is equidistributed with respect to the following probability measure $\mu$.*

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

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 barely introducing bias in the respective probability distribution $\mu$. 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 $\mathcal{G} \coloneqq (S, T, U, G)$ 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.

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

$$T \coloneqq \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 1 \end{pmatrix} \qquad G \coloneqq \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

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 $\mathbb{F}_2$.

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

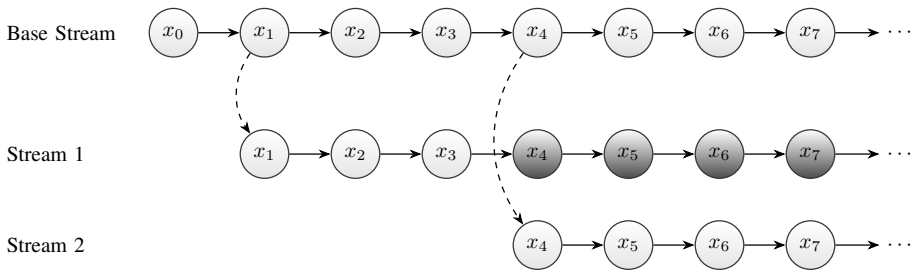



Figure 5.13: Generation of multiple streams by randomly seeding multiple instances of the same PRNG. Multiple instances of a full-period generator will cause overlapping subsequences.

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.

### Predictability and Security

There are a lot of 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.

## 5.6 Implementation-Specific Properties and Features

The implementation of a PRNG in a computer exhibits properties, such as code size, memory size, and speed, that cannot be described by its abstract definition. In general, we strive for simple and portable code but typically we have to find a trade-off between usability and runtime performance. Both, code size and memory size, should be as small as possible so that the data structure and the advancing routine of a generator fit into the caches of the computer. In this case, their execution is only bound by the CPU. To raise the speed of random number generation, optimizing code complexity concerning flow and data dependencies is necessary.

To use PRNGs for vectorized or parallelized algorithms, we have to provide multiple independent streams of random numbers. There are several methods to create and initialize such random streams. Exploiting vectorization facilities, usually multiple instances of the same generator which are seeded differently are used. For a full-period PRNG different seeds generate the same periodic part of a pseudorandom sequence starting at varying positions. Therefore the random initialization of all instances may cause the overlap of subsequences which is reducing statistical quality. Figure 5.13 shows this overlap by means of an example. (Fog 2015; L'Ecuyer et al. 2017)

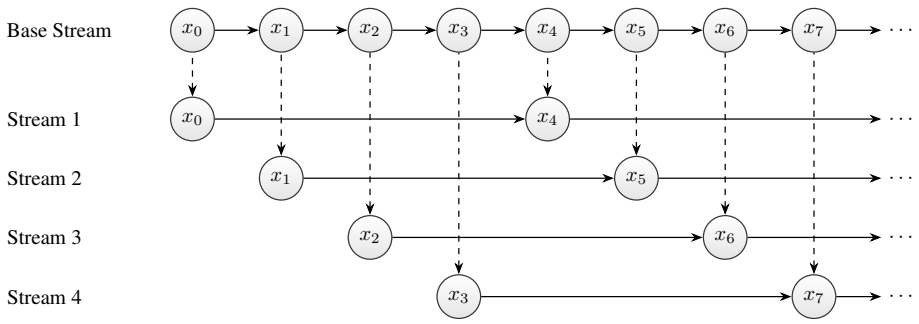



Figure 5.14: Generation of multiple streams by leapfrogging the output of a single PRNG instance. Consecutive values of a pseudorandom sequence will belong to different streams.

Some generators provide a so-called jumping routine that is able to advance the state of the PRNG by a large amount of steps. The Xoroshiro128+ for example makes it possible to discard $2^{64}$ or $2^{96}$ random numbers at once. Initializing multiple instances by using a jumping routine would guarantee non-overlapping subsequences with a length of the amount of random numbers discarded. (Fog 2015; L'Ecuyer et al. 2017)

If the state of the generator is sufficiently large, only one instance would be used. Multiple streams are then constructed by the leafrogging method which can be seen in figure 5.14. In this case, consecutive pseudorandom values will belong to different streams. (Fog 2015; L'Ecuyer et al. 2017)

## 5.7 Examples

### Linear Congruential Generator

A linear congruential Generator (LCG) is the most simplest example of an actual PRNG. The LCG uses modular arithmetic together with a simple recurrence relation. By carefully tweaking the parameters of the generator, enough randomness tests can be passed. (Kneusel 2018)

**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* $\mathrm{LCG}(m, a, c) := (S, T, U, G)$

$$S := U := \mathbb{Z}_m \qquad G := \mathrm{id}_{\mathbb{Z}_m}$$

$$T \colon S \to S \qquad T(x) := ax + c \mod m$$

*We call* $\mathrm{LCG}(m, a, c)$ *the linear congruential generator with modulus* $m$, *multiplier* $a$ *and increment* $c$.

If $c \neq 0$ then the LCG is non-linear and is able to provide equidistributed pseudorandom sequences with a maximum period of $m$. The generator function is the identity and therefore the LCG should not be used in more than one dimension. (Kneusel 2018)

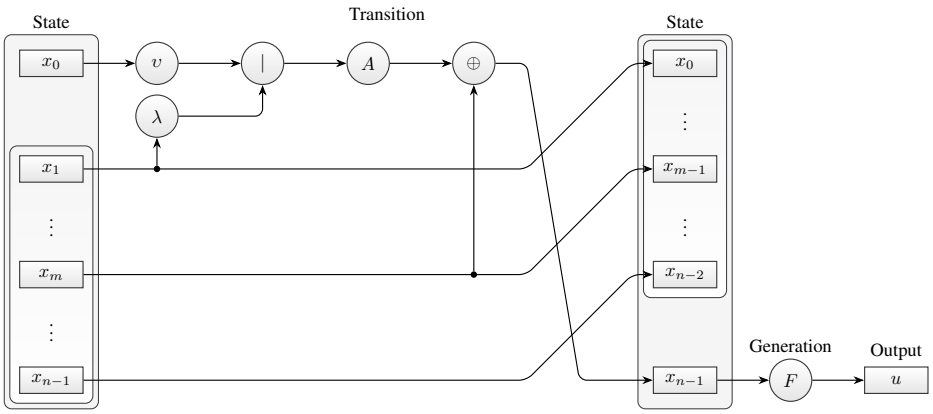



Figure 5.15: Transition and generator function of the Mersenne Twister. The notation is taken from definition 5.11.

### Mersenne Twister

The Mersenne twister is a linear PRNG with a maximum period of $2^{19937} - 1$. Currently, it is the de facto standard when generating pseudorandom sequences. The mathematics behind the Mersenne twister are quite complex and we are not able to provide a rigorous introduction. But we will give an exact definition in the strict sense of a PRNG which was omitted in the original paper (Matsumoto and Nishimura 1998). Additionally, figure 5.15 gives a schematic view of its transition and generator function. The generator was the first fast generator to produce 32-bit random numbers which exhibits 623-dimensional equidistribution and a high statistical quality. (Kneusel 2018)

**DEFINITION 5.11:** Mersenne Twister

*Let* $w, n, m \in \mathbb{N}$ *and* $r \in \mathbb{N}_0$ *with* $m \leq n$ *and* $r < w$. *Further, let* $a, b, c \in \mathbb{F}_2^w$ *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.*

$$S \coloneqq (\mathbb{F}_2^w)^n \qquad U \coloneqq \mathbb{F}_2^w$$

$$T \colon S \to S \qquad G \colon S \to U \qquad G(x) \coloneqq F(x_{n-1})$$

*For all* $k \in \mathbb{N}_0$ *with* $k < n - 1$, *we define* $T$ *as follows.*

$$T_k(x) = x_{k+1} \qquad T_{n-1}(x) = x_m \oplus A\left(\upsilon(x_0) \,|\, \lambda(x_1)\right)$$

*Hereby, the following helper functions are used.*

$$\upsilon, \lambda \colon \mathbb{F}_2^w \to \mathbb{F}_2^w$$

$$\upsilon(x) = \underbrace{x_{w-1} \cdots x_r}_{w-r} \underbrace{0 \cdots 0}_{r} \qquad \lambda(x) = \underbrace{0 \cdots 0}_{w-r} \underbrace{x_{r-1} \cdots x_0}_{r}$$

$$A \colon \mathbb{F}_2^w \to \mathbb{F}_2^w \qquad A(x) \coloneqq \begin{cases} x \to 1 & : x_0 = 0 \\ (x \to 1) \oplus a & : x_0 = 1 \end{cases}$$

$$f_1, f_2, f_3, f_4 \colon \mathbb{F}_2^w \to \mathbb{F}_2^w$$

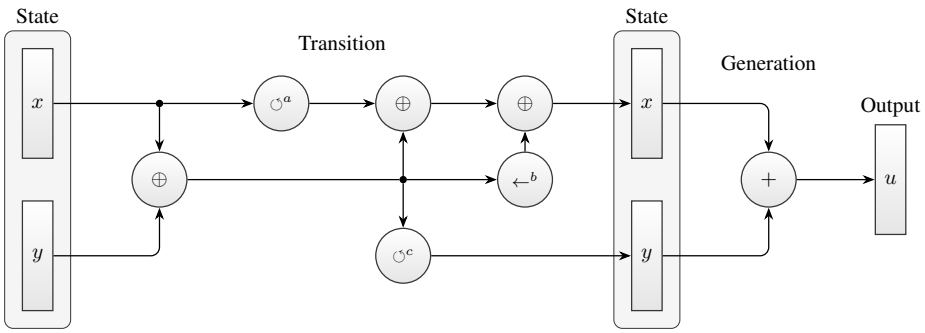



Figure 5.16: Transition and generation function of the Xoroshiro128+. The notation is taken from definition 5.12.

$$f_1(x) \coloneqq x \oplus (x \to u) \qquad f_2(x) \coloneqq x \oplus ((x \leftarrow s) \odot b)$$
$$f_3(x) \coloneqq x \oplus ((x \leftarrow t) \odot c) \qquad f_4(x) \coloneqq x \oplus (x \to l)$$

$$F \colon \mathbb{F}_2^w \to \mathbb{F}_2^w \qquad F(x) \coloneqq f_4 \circ f_3 \circ f_2 \circ f_1(x)$$

*The Mersenne twister with the following parameters is called the MT19937.*

| | | |
|---|---|---|
| $w = 32$ | $a = \texttt{0x9908b0df}$ | $u = 11$ |
| $n = 624$ | $b = \texttt{0x9d2c5680}$ | $s = 7$ |
| $m = 397$ | $c = \texttt{0xefc60000}$ | $t = 15$ |
| $r = 31$ | | $l = 18$ |

### Xoroshiro128+

The Xoroshiro128+ is a scrambled linear PRNG with a maximum period of $2^{128} - 1$. Figure 5.16 visualizes its transition and generator function which are introduced in the following definition. Its state consists of two 64-bit integers that are interleaved by the use of different linear transformations when calling the transition function. The generator function scrambles the output through the use of a non-linear addition operation and returns a 64-bit integer. (Kneusel 2018; Vigna 2018)

**DEFINITION 5.12:** Xoroshiro128+

*Let* $a, b, c \in \mathbb{Z}_{64}$. *Then* Xoroshiro128+$(a, b, c) \coloneqq (S, T, U, G)$ *is defined as a PRNG in the following way.*

$$S \coloneqq \left(\mathbb{F}_2^{64}\right)^2 \qquad U \coloneqq \mathbb{F}_2^{64} \qquad T \colon S \to S \qquad G \colon S \to U$$

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

$$f \colon \mathbb{F}_2^{64} \to \mathbb{F}_2^{64} \qquad f(x, y) \coloneqq x \oplus y$$

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

*If not otherwise stated, we assume* $a = 24$, $b = 16$, *and* $c = 37$.

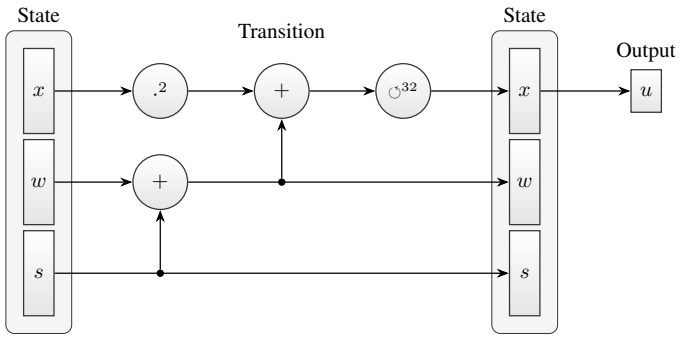



Figure 5.17: Transition and generation function of the middle square Weyl sequence RNG (MSWS). The notation is taken from definition 5.13.

### Middle Square Weyl Sequence RNG

The middle square Weyl sequence RNG (MSWS) is another modern generator which is based on a non-linear squaring routine in its transition function. It provides pseudorandom sequences of 32-bit integers with a period of $2^{64}$. In figure 5.17, again a schematic view of the transition and generator function is given. (Widynski 2019)

**DEFINITION 5.13:** Middle Square Weyl Sequence RNG (MSWS)

*We define* MSWS $:= (S, T, U, G)$ *as a PRNG in the following way.*

$$S := \mathbb{Z}_{2^{64}}^3 \qquad U := \mathbb{Z}_{2^{32}}$$

$$T\colon S \to S \qquad T(x, w, s) = ((x^2 + w + s) \circlearrowleft 32, w + s, s)$$

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

# 6 Design of the Library Interface

The design of a library API using various vectorized PRNGs consists of several parts, as can be seen in figure 6.18. 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 be 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.

Other parts 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 the 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.

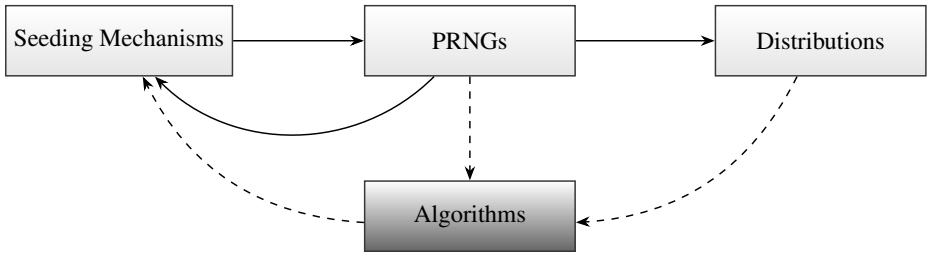



Figure 6.18: Main parts to be considered for a software library that is providing access to scalar and vectorized PRNGs. The arrows show possible interactions between different parts for which convenient interfaces should be created. In this thesis, algorithms will not be considered.

In the last few years, there has been some development in the area of API design for RNGs. Klammler (2016), O'Neill (2015c,d), and Song and O'Neill (2016), to name a few. They provide perfect basic ideas to improve the usage of random utilities in the C++ language. We will rely on this work and 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 these functions we keep compatibility to the standard distributions.

## 6.1 Concepts

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.

## 6.2 Template 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 a given technique that allows us to evaluate the validity of expressions.

Code 6.1: 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 highly makes use of template meta programming artifacts, like SFINAE and type traits. To be precise, 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 6.2: 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 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.

## 6.3 Seeding Mechanism

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 6.3: 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 to 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 6.4: 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.

## 6.4 Uniform Distribution Functions

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 is a value of 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.

Code 6.5: Uniform Template Is-Valid Patterns

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

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.

Code 6.6: Uniform Template

```
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);
}
```

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 $64\,\text{bit}$ size instead of the used $32\,\text{bit}$. 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.

Code 6.7: Uniform MT19937 Overload

```
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);
}
```

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 which are not created by the programmer itself.

# 7 Implementation

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.

## 7.1 MT19937

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.

Code 7.1: Scalar MT19937 Structure Types and Constants

```
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;

    // ...
};
```

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 7.2: Scalar MT19937 Structure

```
struct mt19937 {
  // ...

  struct default_seeder;

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

  // Advancing Routine
  constexpr result_type operator()() noexcept;

  // State
  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 7.3: Scalar MT19937 Seeding

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

// Advancing Routine of the Default Seeder
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;
}
```

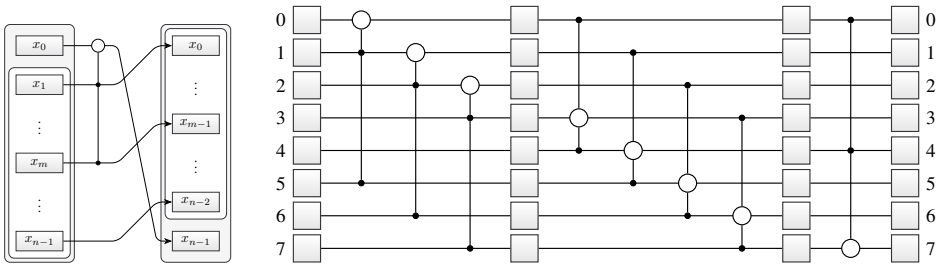



Figure 7.19: Implementation of the Mersenne twister by precomputing its transition through a loop over the elements in the state. The left image shows a shorthand notation for a single call to the transition function. The right image visualizes the loop over all elements by using the simpler notation for a simplified version of the Mersenne Twister with a state size of eight and a shift size of five.

```
// Seeding Constructor of the MT19937
template <typename RNG>
constexpr mt19937::mt19937(RNG&& rng) {
  pxart::generate(std::forward<RNG>(rng), state, state + state_size);
}
```

Advancing the state of the MT19937 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, as can be seen in figure 5.15. 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 and are precomputing 624 steps of the transition function. This guarantees that newly generated elements do not have to be moved to the end of the state but instead can reside at their current positions. Because we are transforming every state element by applying the transition function 624 times, the precomputation step has to loop over all elements. To make sure the transformation is only accessing elements that lie inside the state, the loop has to be split into three distinct parts. The transition of a single element depends on two other elements in the state. Every dependency introduces a split because it could lie outside the state by shifting forward. In figure 7.19, this process is shown schematically.

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 7.4: Scalar MT19937 Advancing

```
constexpr auto mt19937::operator()() noexcept -> result_type {
  // Precompute 624 transitions when all state elements have been read.
  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);
    };

    // Transition Loop
    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;
  }

  // Generator Function
  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-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 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 is 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 227th 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 elements through the buffer. The boundary term between the first and the second loop can now be completely handled by the vectorized first loop. The vectorization of the loop is shown in figure 7.20. 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 7.5: AVX MT19937 Structure

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

  // ...

  // Advancing Routine
  simd_type operator()() noexcept;

  // ...

  // State with Buffer
  uint_type state[state_size + simd_size] __attribute__((aligned(32)));
  int state_index = state_size;
};
```

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.

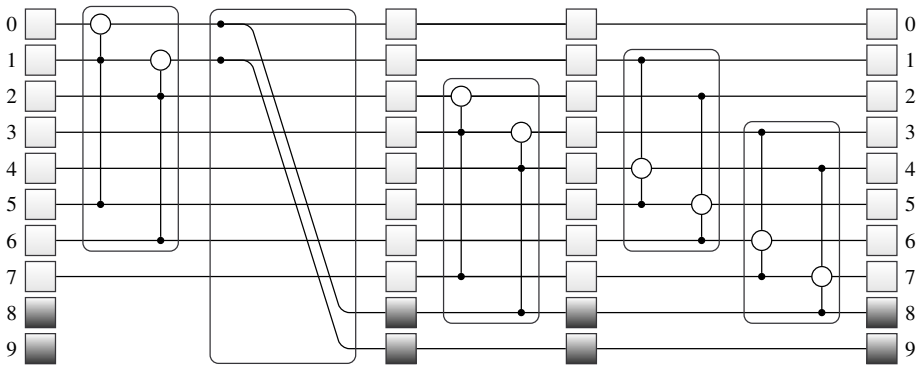



Figure 7.20: Vectorization of the loop of Mersenne twister implementation by adding a buffer (gray) with the size of an SIMD vector at the end of the state. Here, the same simplified version as in figure 7.19 is used. It is assumed SIMD vectors are able to contain two elements of the state at once. Vectorizable operations are shown in boxes. By copying the first vector containing the new state elements into the buffer, no SIMD vector will overlap the first and the second loop.

Code 7.6: 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;
    };

    // AVX Transition Loop
    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;
  }

  // Generator Function
  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 are then directly translated into their corresponding vector intrinsics.

Except `_mm256_cmpgt_epi32`, all AVX intrinsics which are used in the code exhibit a latency of 1 on the typical microarchitectures. `_mm256_cmpgt_epi32` has a latency of 3 and a throughput of 1 and therefore represents some kind of bottleneck that cannot be avoided. The throughput for loading and storing vector values is 0.25. Hence, the loading of the three dependent values in the transition function should run in parallel. To even tweak the code further, we could always use two AVX vector registers at the same time by interleaving their operations. This would exploit the lower throughput of other intrinsics. In this case, only the shift operations in the generation function, like `_mm256_slli_epi32`, would bound the overall performance due to their throughput of 1. (Fog 2019b; Intel 2019a)

In listing B.1, the assembly code of the vectorized advancing routine is shown. We have compiled the source code through the use of *Compiler Explorer* with the highest degree of optimization enabled (Godbolt 2019). It can be seen that the AVX intrinsics that were used for the implementation could directly be translated to their according assembler instructions. Besides, the automatic optimization of the compiler is still active and helps to further decrease latencies or throughput. Hence, the use of SIMD intrinsics in C++ code can be seen as superior to the implementation by writing inline assembly code.

## 7.2 Xoroshiro128+

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.

The Xoroshiro128+ offers a jump-ahead feature for discarding $2^{64}$ or $2^{96}$ elements of the

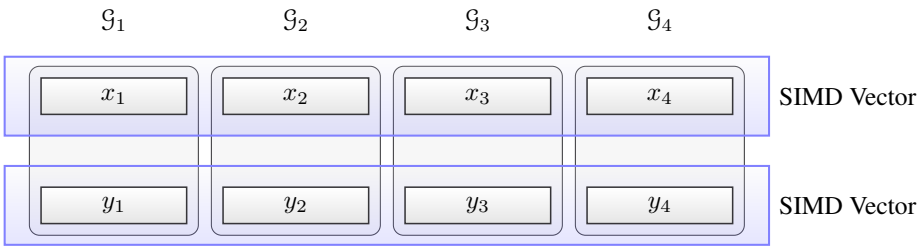



Figure 7.21: Memory layout concerning SIMD vectors of four instances of the Xoroshiro128+. SIMD vectors are shown as blue box.

output. In listing B.2, we are only providing the implementation of the smaller jump. Both jumping routines differ only in their masking numbers.

Code 7.7: Scalar Xoroshiro128+ Advancing

```
constexpr auto xoroshiro128plus::operator()() noexcept {
  constexpr auto rotate_left = [](uint_type x, size_t k) -> uint_type {
    return (x << k) | (x >> (64 - k));
  };

  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;
}
```

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 structure therefore does not really change and uses the SIMD types instead of the 64-bit unsigned integer values. The layout of the data structure can be seen in figure 7.21.

Code 7.8: AVX Xoroshiro128+ Structure

```
struct xoroshiro128plus {
  // ...

  // ...

  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, given in listing B.3, 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 7.9: AVX Xoroshiro128+ Advancing

```
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));
}

inline auto xoroshiro128plus::operator()() noexcept {
  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;
}
```

Due to the latencies and throughputs of the AVX intrinsics used in this implementation, the advancing routine does not introduce a major bottleneck in the generation of random numbers. But to exploit the throughput of every operation, we would need to use multiple independent AVX vector registers at once.

In the code B.4, B.5, and B.6, we provide the assembly code generated by *Compiler Explorer* for calling the advancing routine once, twice, and four times, respectively (Godbolt 2019). As before, all AVX intrinsics were correctly translated into their respective assembler instruction. Looking at the listings B.5 and B.6, we even see that the compiler completely inlined multiple calls to the advancing routine automatically. In contrast to the MT19937, the AVX implementation of the Xoroshiro128+ enables us to keep using the vector registers without cache or memory communication while generating new random numbers. This property will provide a higher performance.

## 7.3 MSWS

The MSWS is a simple, modern, and non-linear PRNG that uses a state which is 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.

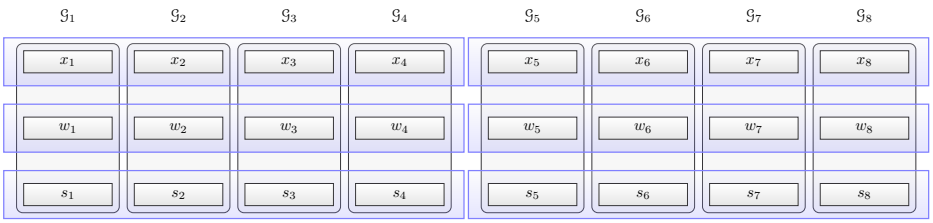


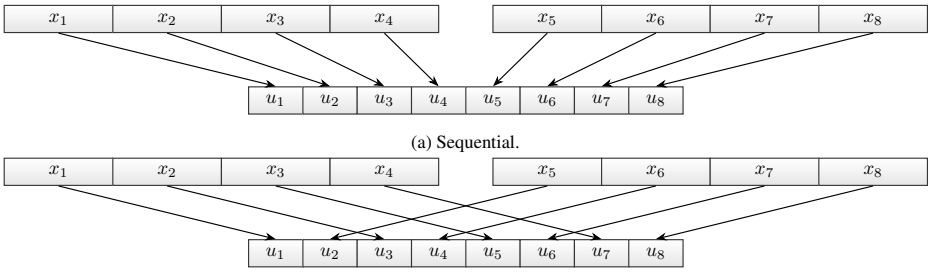

Figure 7.22: Memory layout concerning SIMD vectors of eight instances of the MSWS. SIMD vectors are shown as blue box.

Code 7.10: Scalar MSWS

```
struct msws {
  using uint_type = uint64_t;
  using result_type = uint32_t;

  // ...

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

  constexpr result_type operator()() noexcept;

  // ...

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

// Advancing Routine
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-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. A visualization of the data structure layout is given in figure 7.22.

Advancing the state of the vectorized MSWS introduced difficulties because we have to compute the square of the first 64-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




(a) Sequential.

(b) Interleaved.

Figure 7.23: Two variants of merging the two output vectors of the vectorized MSWS containing 64-bit integers into one SIMD vector containing 32-bit integers by means of AVX vector registers.

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 $x = x_1 2^{32} + x_0$.

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

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 or permutation operations. Two variants are shown in figure 7.23.

Code 7.11: 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],
```

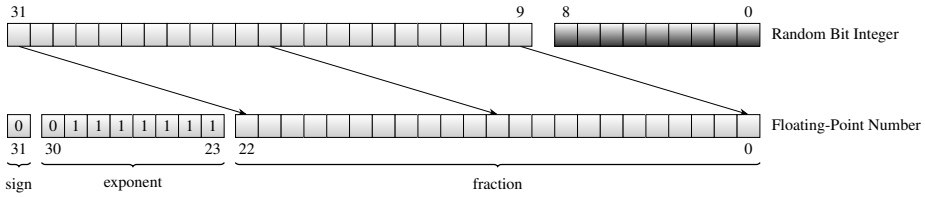



Figure 7.24: Generating a uniformly distributed floating-point number by shifting bits of a given random integer of the same size. According to the IEEE 754 floating-point standard, sign and exponent bits have to be set, such that the number will be in the interval $[1, 2)$.

```
                                    _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));
}
```

In the AVX implementation, we use `_mm256_mul_epu32` and `_mm256_mullo_epi32` to compute the square of the first element in the state. They exhibit a latency of $5$ and $10$ and a throughput of $0.5$ and $1$ on the typical microarchitectures, respectively. Thus, in comparison to the other intrinsics, they tremendously reduce the performance of the advancing routine. To exploit their throughput, we would have to run ten of these instructions for independent AVX vectors consecutively. One 256-bit SIMD vector contains four 64-bit MSWS generators. Therefore at least 40 instances of the PRNG have to be called in parallel to remove the latency bottleneck of the multiplication operations. Due to the risk of lowering the statistical quality, we have only used eight generators in parallel.

## 7.4 Uniform Distribution Functions for Reals

The uniformly distributed transformation of random 32- and 64-bit integers to single- and double-precision floating-point values is described in Vigna (2018) and Kneusel (2018). Here, the IEEE 754 floating-point standard is used to ensure that bit manipulations will result in the same number on every machine supporting this standard. Because we only want to deal with normalized numbers, a value in the interval $[1, 2)$ should be generated firstly. Such a value is positive and therefore the sign bit of the floating-point value is set to $0$. The actual number representing the exponent of the floating-point encoding should be zero. Due to the offset applied to the exponent value, the content of the exponent has to be equal to the offset. Then the fraction can be filled with random bits to construct a uniformly distributed floating-point value in the interval $[1, 2)$. Low-order bits of a pseudorandom integer tend to provide weak randomness properties. Hence, a shift is used to fill the fraction with the most significant bits of the random integer. In figure 7.24, this can be seen for a 32-bit integer and a single-precision floating-point value. After this computation, the remaining part has to subtract $1$ from the floating-point value to guarantee the uniform distribution in the unit interval $[0, 1)$. The scalar implementation is given in the following code snippet.

Code 7.12: Scalar Uniform Function

```
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 <typename Real>
constexpr Real uniform(uint64_t) noexcept = delete;

template <>
constexpr inline double uniform<double>(uint64_t x) noexcept {
  const auto tmp = ((x >> 12) | 0x3ff0000000000000ULL);
  return (*reinterpret_cast<const double*>(&tmp)) - 1.0;
}
```

The vectorization of the uniform routine is straightforward. Every operation can be replaced by its respective intrinsic. Analyzing the function, `_mm256_sub_ps` and `_mm256_sub_pd` exhibit the highest latency with a value ranging from 3 to 4 on typical microarchitectures (Fog 2019b; Intel 2019a). On the other hand, their throughput ranges from 0.5 to 1. So to further improve the performance of the uniform function, multiple independent values should be computed resulting in better exploitation of the processor pipeline.

Code 7.13: AVX Uniform Function

```
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));
}
```

# 8 Tests, Benchmarks and the Simulation of Photons

## 8.1 Functionality and Correctness

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 8.1: 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 Mersenne twister 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 Mersenne twister 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, `std::random_device` does not have to be truly random (O'Neill 2015a,d) but could be replaced by other seeding alternatives if this should be the case (O'Neill 2015e).

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 code for this is given the appendix in listing B.7. The use of appropriate `using` declarations makes the code more readable. As a special case, the vectorization of the jump function of the Xoroshiro128+, given in listing B.8, can be tested by checking the equality of the internal state variables after successfully testing its advancing routine. 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.

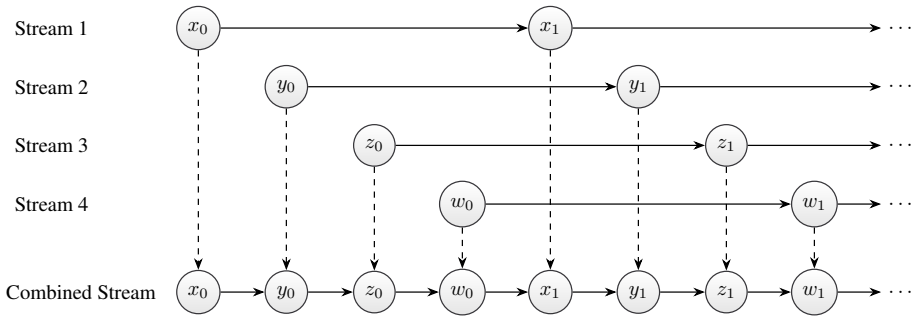



Figure 8.25: Interleaving samples of multiple streams of random numbers to produce a combined stream which can be used as input for common statistical test suites. Modeled after Kneusel (2018, p. 161).

## 8.2 Randomness of Multiple Streams

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 8.25. 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 which use the leapfrogging method to generate multiple streams will not give us further insights to their randomness properties because interleaving the samples would produce the same output as the scalar version. Especially, this is 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. We will not test the jump initialization of the Xoroshiro128+ as it was already tested by Vigna (2018). The process of testing was geared to Kneusel (2018, pp. 141–155) and O'Neill (2017a).

### Test Suite *dieharder*

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 on the command line. The following code snippet provides a bit stream for the AVX implementation of the Xoroshiro128+ by writing its binary output to the standard output. The given implementation can be easily adjusted to work with the other generators by changing the type of the PRNG.

Code 8.2: 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
```

### Test Suite *TestU01*

*TestU01* is a software library written in the C programming language 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 8.3: 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 which is 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.

## 8.3 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 (IPC), average branch misses, average cache misses, and even more (Leis 2019).

To apply the benchmarking routine easily to any possible generator, we use an abstraction class, given in listing B.9 in the appendix, 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 integer type 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 performance of a single PRNG instance to the that of two instances used at once which may give us further insights into latency- and throughput-based issues.

Code 8.4: 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;
  }
```

Above, 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 or L2 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.

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. The code is given in listing B.10. 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.

## 8.4 Monte Carlo $\pi$ Benchmark

The framework used for benchmarking the computation of $\pi$ 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 $\pi$ with vectorized PRNGs.

In section 4.1 about Monte Carlo integration, we have already given a naive approach together with an explanation on how to compute $\pi$ with the random utilities of the C++ STL. The code will be repeated here for convenience.

Code 8.5: Monte Carlo $\pi$ 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 get further insights into latency- and throughput-based issues concerning the implementations of the 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. 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`.

Code 8.6: Monte Carlo $\pi$ 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;
}
```

Removing `std::uniform_real_distribution` given by the STL and instead using our custom uniform distribution function would be a first improvement to the naive version. 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 8.7: Monte Carlo $\pi$ 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 $\pi$ benchmark that is able to use vectorized PRNGs without the need to 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 8.8: Monte Carlo $\pi$ 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-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 $\pi$ would not be realized via Monte Carlo methods in real applications, we have not implemented such a variant.

Implementing the AVX version of the Monte Carlo $\pi$ benchmark is done in the following listing. The nature of the algorithm allows us to interchange nearly every operation with its 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.

Code 8.9: Monte Carlo $\pi$ 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;
}
```

## 8.5 Photon Simulation

To apply our vectorized implementations of PRNGs to a more realistic example, we have implemented a simplified simulation of photon propagation in a homogeneous medium. The photons traveling through the medium may be scattered with respect to the Henyey-Greenstein phase functions defined in section 4.2 about volume scattering. According to Pharr, Jakob, and Humphreys (2016), rendering translucent media, such as fog or opal glass, by simulating subsurface or volume scattering requires a large amount of CPU-intensive work. Therefore a vectorized algorithm using SIMD-aware PRNGs for its sampling seems to be an appropriate candidate to speed up this simulation.

For the photon simulation, we first have to think about the underlying data structure. To make vectorization possible without changing this structure, we have used a structure of arrays for each, the position coordinates and the velocity directions of photons. To simplify the simulation and enable visual output, the system is only computed for two dimensions. The parameters for the discrete time step, the asymmetry parameter of the phase function, and the probability that scattering occurs in the time step are stored in the system, too.

Code 8.10: Photon Simulation Structure

```
struct system {
  // System of Photons
  std::vector<float> p_x{};  // Positions x-coordinate
  std::vector<float> p_y{};  // Positions y-coordinate
  std::vector<float> v_x{};  // Velocities x-direction
  std::vector<float> v_y{};  // Velocities y-direction

  // Medium
  float scatter{0.1f};  // Scattering Probability
  float g{0.8f};        // Asymmetry Parameter

  // Discretization
  float time_step{1e-3f};
};
```

The basic algorithm has to execute the following parts when advancing the system of photons one time step. First, the photons are moved according to the direction of their velocity. Afterwards, we have to check if a scattering has occured by using a uniformly distributed random variable. If a photon will be scattered, then the Henyey-Greenstein phase function will be sampled to get a value for the cosine of the angle used for the rotation. Another random number will then give us the sign of the sine of the angle. At the end, the matrix-vector product for the rotation will be applied to set the new direction of the velocity. This process has to be repeated several thousand times to simulate the time evolution of the system. The following code shows the scalar implementation. Figures 8.26 and 8.27 show the time evolution of the photon simulation for positive and negative asymmetry parameters of the Henyey-Greenstein phase function by means of an example.

Code 8.11: Photon Simulation Scalar Advancing

```
template <typename RNG>
inline void advance(system& sys, RNG&& rng) noexcept {
  for (int i = 0; i < sys.p_x.size(); ++i) {
    // Move photon according to its velocity.
    sys.p_x[i] += sys.time_step * sys.v_x[i];
    sys.p_y[i] += sys.time_step * sys.v_y[i];

    // Sample if scattering occurs.
    if (pxart::uniform<float>(rng) > sys.scatter) continue;

    // Sample Henyey-Greenstein phase function to scatter.
    const auto g = sys.g;
    const auto tmp = (1 - g * g) / (1 - g + 2 * g * pxart::uniform<float>(rng));
    const auto cos_angle = (1 + g * g - tmp * tmp) / (2 * g);
    // Sample the sign of the sine of the angle.
    const auto sign = (pxart::uniform<float>(rng) > 0.5) ? 1 : -1;
    const auto sin_angle = sign * std::sqrt(1 - cos_angle * cos_angle);

    // Rotate velocity of photon by matrix-vector product with rotation matrix.
    const auto new_x = cos_angle * sys.v_x[i] - sin_angle * sys.v_y[i];
    const auto new_y = sin_angle * sys.v_x[i] + cos_angle * sys.v_y[i];
    sys.v_x[i] = new_x;
    sys.v_y[i] = new_y;
  }
}
```

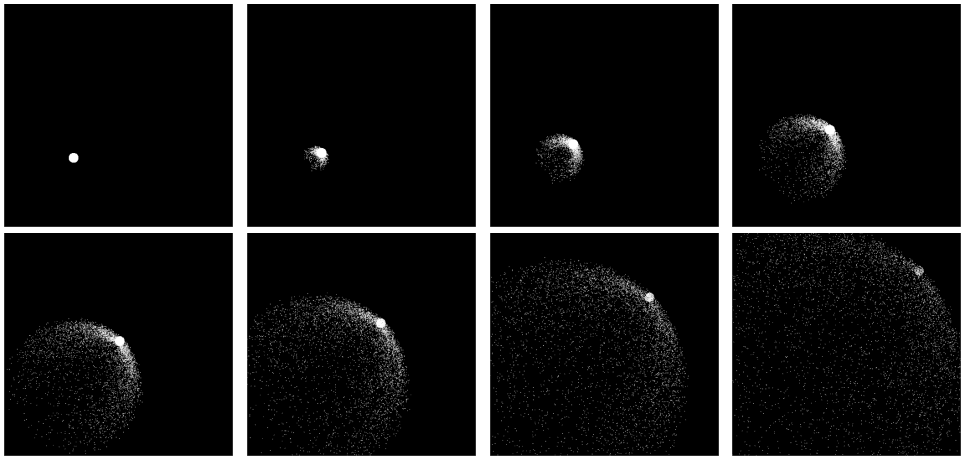


Figure 8.26: Time evolution of the photon simulation for a positive asymmetry parameter. The initial state of the system is given by a ball of photons moving to the upper-right corner.

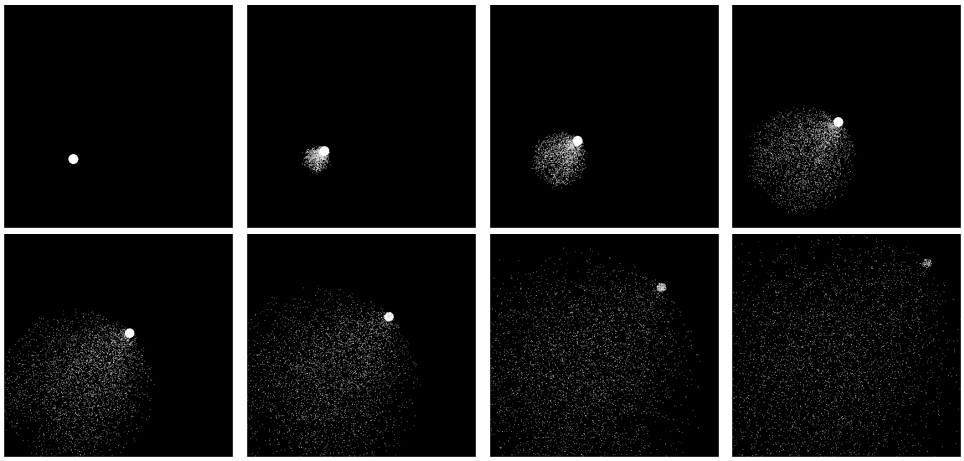


Figure 8.27: Time evolution of the photon simulation for a negative asymmetry parameter. The initial state of the system is given by a ball of photons moving to the upper-right corner.

The vectorization of this algorithm mostly consists of the interchanging of standard instructions with their according SIMD intrinsics. We have chosen to pack the positions and velocities of eight photons in AVX registers. Care has to be taken when translating the control branches. Using `_mm256_cmp_ps`, we can evaluate a condition for eight photons at the same time. The results are given as a mask. Thus, we execute the complete scattering transformation for all photons. At the end, we use `_mm256_blendv_ps` together with the mask to decide which part of the vector indeed has to be changed due to scattering. To show the need of vectorized PRNGs for applications that were already vectorized, we have created two different AVX versions. One version uses a scalar PRNG to generate its random numbers. The other version takes again a vectorized PRNG. Both variants are shown below. The complete implementation of the second version is given in the appendix in listing B.11.

Code 8.12: Scalar PRNG Call for AVX Implementation

```
const auto rnd =
    _mm256_set_ps(pxart::uniform<float>(rng), pxart::uniform<float>(rng),
                  pxart::uniform<float>(rng), pxart::uniform<float>(rng),
                  pxart::uniform<float>(rng), pxart::uniform<float>(rng),
                  pxart::uniform<float>(rng), pxart::uniform<float>(rng));
```

Code 8.13: AVX PRNG Call for AVX Implementation

```
const auto rnd = pxart::simd256::uniform<float>(rng);
```

# 9 Evaluation and Results

In section 7, 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 8 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 varying by their raw computing powers. The machines are modern Linux-based 64-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.

## 9.1 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. Such behavior possibly introduces bias and reduces the statistical quality because different streams of random numbers may be correlated.

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.

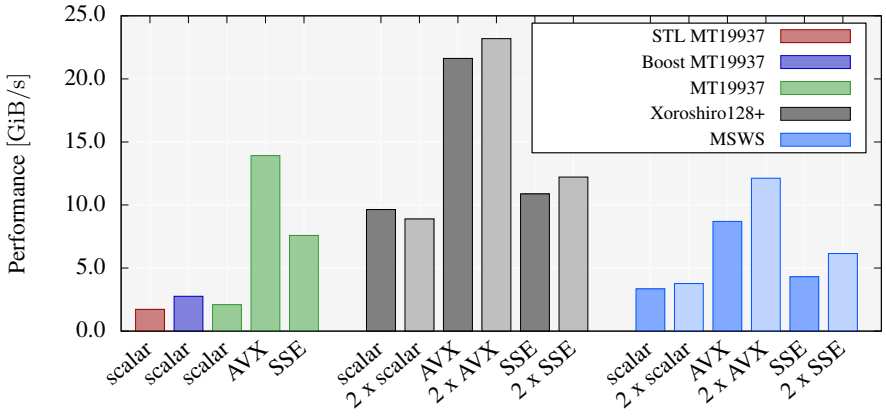



Figure 9.28: Performance resulting from the generation benchmark running on the *Intel® Core™ i7-7700K Processor* measured in GiB of random numbers per second for the different variants of the implemented PRNGs. For convenience, the performances of the STL and Boost implementation of the MT19937 are shown as well. The data can be found in table 9.1.

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 were 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.

## 9.2 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 9.1 and 9.2, the results for running the generation benchmark and the Monte Carlo $\pi$ 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 appendix C. Figures 9.28 and 9.29 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 $\pi$ benchmark, we only show the speed-up in figure 9.30. The prefix "2 x" appearing in all figures states that two independent instances of the given generator were used.

Both target machines consist of similar architectures. Comparing the tables 9.1 and 9.2 and the figures 9.28, 9.29, and 9.30 to the tables C.4 and C.5 and the figures C.31, C.32, and C.33 in appendix C 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

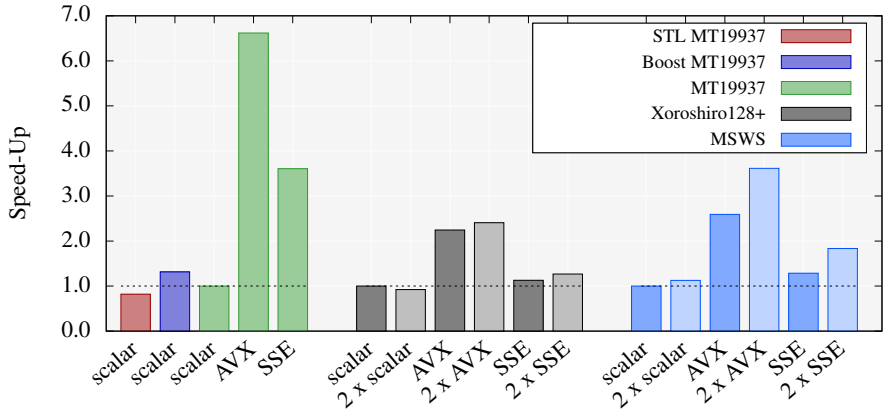



Figure 9.29: Speed-up in execution time with respect to the single-instance scalar version resulting from the generation benchmark running on the *Intel® Core™ i7-7700K Processor* for the different variants of the implemented PRNGs. For convenience, the speed-ups of the STL and Boost implementation of the MT19937 are shown with respect to our implementation. The data can be found in table 9.1.

that quantizing the efficiency of a PRNG implementation should use the averaged cycles and instructions together with the machine-dependent IPC 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 9.29, 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. Additionally, comparing the overall performance of the MT19937 to the other generators by looking at figure 9.28 and table 9.1, we see that it is competitive. For the vectorized versions, the MT19937 is even faster than the MSWS which has a much smaller state size. The caching mechanisms of modern computer architecture make it possible to completely keep the state of the MT19937 in the cache while generating new random numbers. Furthermore, the transition function is linear and does not exhibit difficult flow or data dependencies as can be seen in figure 5.15. The advancing routine can therefore exploit modern processor features much better than the implementations of the MSWS or the Xoroshiro128+. Due to its non-linearity, the MSWS uses complex functions when advancing

Table 9.1: 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 |

its state. Looking at figures 5.16 and 5.17, both generators show a higher data dependency which reduces the efficiency of the processor pipeline. Hence, the modern design of processors makes it difficult to theoretically analyze the implementations of PRNGs. It can even support generators that seem to be slow such that they show a much higher performance than their counterparts.

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 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.

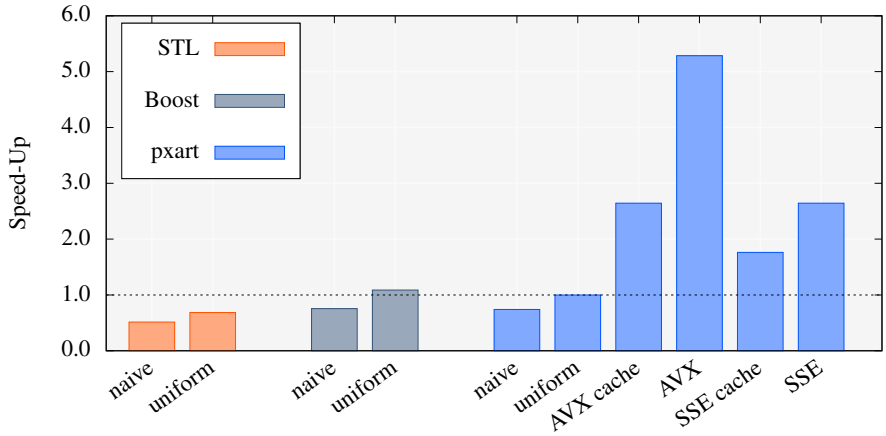
This reduces the possible speed-up reachable by SSE/AVX implementations. But this time, we see a strong improvement in performance when using two instances of the MSWS versions. In the implementation of the MSWS, we have to carry out a $64\,\mathrm{bit}$ multiplication by squaring a number. Especially 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 9.28 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 $\pi$ benchmark in table 9.2 and figure 9.30, we first expected to see similar results. As before, all the vectorized implementations were able to increase the performance of the program. Again, although the MT19937 has a much bigger state, its performance is competitive to the other generators as can be seen in table 9.2. Especially the fast uniform distribution function even outperforms the naive scenario for scalar generators in all cases. The Monte Carlo $\pi$ benchmark is a CPU-bound algorithm and therefore we expected such an outcome. 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 $\pi$ 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 $\pi$ will printed on the screen which forces the compiler to not remove code that is needed for the computation of $\pi$. Therefore the Monte Carlo $\pi$ 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 $\pi$, should always be used to measure performance and speed-up of PRNGs.

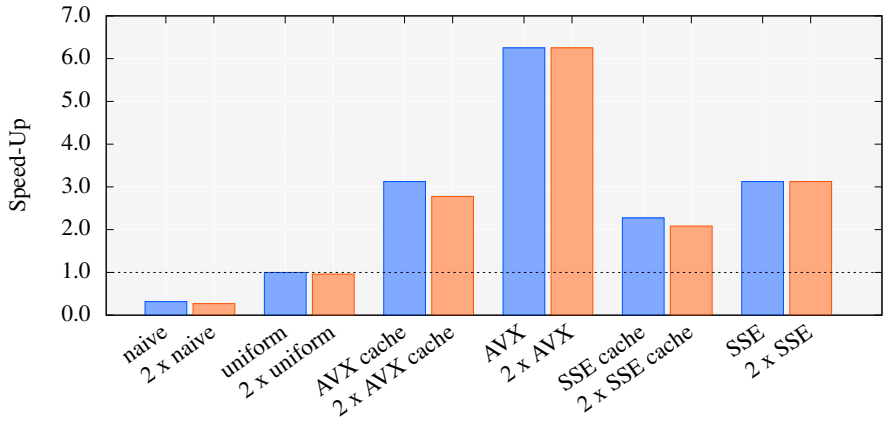
Another important point is that we always could reduce the execution time by a factor ranging from $1.2$ to $2.6$ when using a cache without introducing vector intrinsics to the actual benchmark. By Amdahl's law, the given speed-ups have to be a lot smaller than the theoretical bounds because a great 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.

Table 9.2: Results achieved by running the Monte Carlo $\pi$ 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 $\pi$. It was ensured that the estimation error was small enough according to the calculation at the end of section 4.1. 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.

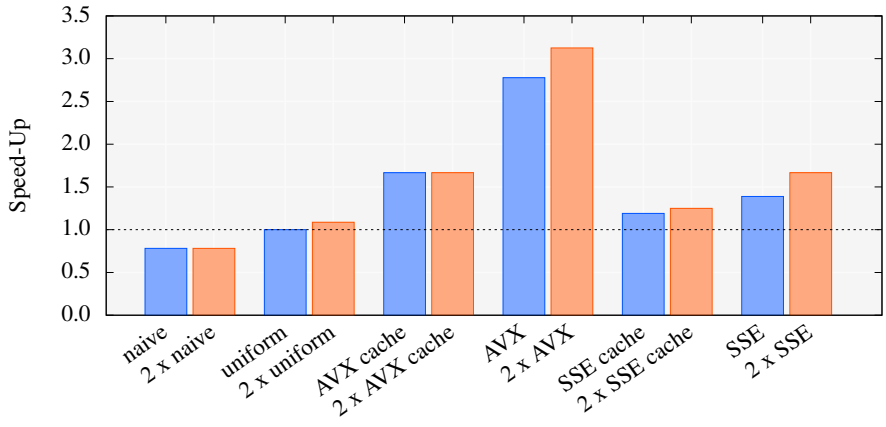
| **PRNG** | **Benchmark** | **Time** [s] | **Cycles** | **Instructions** | **IPC** | **Frequency** [GHz] |
|---|---|---|---|---|---|---|
| STL MT19937 | naive | 0.72 | 31.30 | 85.68 | 2.74 | 4.37 |
| | uniform | 0.53 | 24.03 | 77.67 | 3.23 | 4.51 |
| Boost MT19937 | naive | 0.47 | 21.18 | 66.72 | 3.15 | 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 9.30: Speed-up in execution time with respect to the single-instance uniform version resulting from the Monte Carlo $\pi$ 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 9.2.

Table 9.3: Results of the photon simulation run on the *Intel® Core™ i7-7700K Processor* for the different PRNGs. The system used $10^6$ photons and was advanced $10^3$ times with a time step of $10^{-3}$. The table gives the time taken to run the simulation for the different versions. "Scalar" represents the scalar algorithm using a scalar PRNG. "AVX Algorithm" stands for the vectorized algorithm using a scalar PRNG and "Full AVX" represents the AVX algorithm using a vectorized PRNG. The Speed-Up of the "AVX Algorithm" version is given with respect to the "Scalar" version.

| **PRNG** | **Scalar** [s] | **AVX Algorithm** [s] | **Full AVX** [s] | **Speed-Up** |
|---|---|---|---|---|
| MT19937 | 0.27 | 0.38 | 0.13 | 2.1 |
| Xoroshiro128+ | 0.22 | 0.22 | 0.09 | 2.4 |
| MSWS | 0.20 | 0.24 | 0.13 | 1.5 |

The results of the photon simulation are given in table 9.3. We have used $10^6$ photons and advanced the system $1000$ times by using a time step of $10^{-3}$. We have measured the time taken to compute all steps for the scalar algorithm, the vectorized algorithm using a scalar PRNG, and the fully vectorized version which is using PRNGs implemented by AVX intrinsics. The results show that vectorizing our randomized algorithm which does not use SIMD-aware PRNGs increases the time taken to advance the system. Only the Xoroshiro128+ is capable of maintaining the same performance. Therefore the vectorization technique could not speed up the application. Only by interchanging scalar random number generation by their AVX counterparts, we were able to achieve a higher performance for every implemented PRNG. Please note, due to the complexity of the photon simulation, the possible speed-up is bounded. So even in a more realistic and physically-based application we see an improvement in performance by using vectorized PRNGs.

# 10 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, such as the propagation of photons, 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 $\pi$ 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.

In our implementations, we did not optimize the SIMD code to exploit every single issue concerning the latency and throughput of instructions. Doing so would involve a much larger output size when calling an advancing routine. In general, we have to use multiple independent instances to be able to reorder the instructions and to keep the processor pipeline filled as a consequence. We have only needed one SIMD vector of random numbers per call to not introduce arrays of random numbers that may not fit into the caches. However, users of a software library that is providing random facilities frequently want to generate large arrays of random numbers. So enhancements to the library should involve the latency- and throughput-based optimization of PRNGs. This should be done in such a way that the user is able to choose which vectorized variant best suits his needs.

Applying vectorized PRNGs to applications which were not vectorized, yet, can typically be done by introducing a small cache with the size of the generator output. A further increase in performance is then achieved by vectorizing the application. This task is known to be time-consuming, exhaustive, and error-prone. As a consequence, a future version of the software library should provide an SIMD wrapper class to simplify the usage of intrinsics without reducing their performance. Wrapping SIMD intrinsics was already done by Fog (Fog 2019c) and a lot of other people. All wrappers are used in different contexts and therefore provide a different API that makes their integration into *pxart* difficult. Instead, we would need a new and C++-aware interface which makes the usage of PRNGs easier but on the other hand is able to be implicitly constructed from and converted to low-level SIMD intrinsics. This would not constrain the integration of other wrapper classes or the direct use of intrinsics. Furthermore, it enables us to exploit the template metaprogramming features of the C++ programming language.

Aside from the direct optimization of PRNGs, *pxart* should exhibit a reacher interface for different seeding mechanisms and distributions. For example, to enable good deterministic seeding by using a varying amount of fixed integers, a general default seeder based on O'Neill (2015b) should be implemented. There are applications which need to use different distributions, like Gaussian distributions or uniform integer distributions. Using their scalar versions from the STL of the C++ programming language would reduce the overall performance and therefore making the use of vectorized PRNGs futile. For these special situations, vectorized implementations of varying seeding mechanisms and distributions should be considered in the development.

Finally, further development should take care of thread-based parallelism. Currently, the user of the library has to manually initialize the generators in different threads of a program. This process should be automated such that the use of vectorized PRNGs in many threads guarantees non-overlapping subsequences up to a certain amount of random numbers.

# 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.

Blacher, Mark (2018). "Entwurf und Implementierung vektorisierter Sortieralgorithmen". Masterarbeit. Friedrich-Schiller-Universität Jena. URL: https://ci.inf-i2.uni-jena.de/qa74rag/avx2_sort/blob/master/Masterarbeit_vektorisierte_Sortieralgorithmen.pdf (visited on 12/17/0019).

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).

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

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

Guskova, Maria S., L. Yu. Barash, and Lev. N. Shchur. *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 (2008). *Measuring Instruction Latencies and Throughput*. URL: https://software.intel.com/en-us/articles/measuring-instruction-latency-and-throughput (visited on 12/14/2019).

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

— (2019b). *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).

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

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

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.

— (2009). *TestU01*. URL: http://simul.iro.umontreal.ca/testu01/tu01.html (visited on 12/13/2019).

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

Lemire, Daniel (2018). *simdpcg*. URL: https://github.com/lemire/simdpcg (visited on 11/30/2019).

— (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.

L'Ecuyer, Pierre et al. (2017). "Random Numbers for Parallel Computers. Requirements and Methods, with Emphasis on GPUs". In: *Mathematics and Computers in Simulation* 135. Special Issue: 9th IMACS Seminar on Monte Carlo Methods, pp. 3 –17. ISSN: 0378-4754. DOI: 10.1016/j.matcom.2016.05.005. URL: http://www.sciencedirect.com/science/article/pii/S0378475416300829.

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).

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

— (2015b). *Developing a seed_seq Alternative*. URL: http://www.pcg-random.org/posts/developing-a-seed_seq-alternative.html (visited on 12/19/2019).

— (2015c). *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).

— (2015d). *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).

— (2015e). *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).

— (2019). *pxart. Packed Extensions for Advanced Random Techniques: Library and Applications for Random Number Generators*. URL: https://github.com/lyrahgames/pxart (visited on 12/11/2019).

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.

Reddy, Martin (2011). *API Design for C++*. Morgan Kaufmann – Elsevier. ISBN: 978-0-12-385003-4.

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).

*Standard C++ Foundation* (2019). URL: https://isocpp.org/ (visited on 07/15/2019).

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.

Wang, Lihon and Steven L. Jacques (1995). *Monte Carlo Modeling of Light Transport in Multi-Layered Tissues in Standard C*. University of Texas. URL: https://omlc.org/software/mc/mcml/MCman.pdf (visited on 12/02/2019).

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 Mathematical Proofs

**LEMMA:** Monte Carlo Integration Estimates Value of Integral

*Choose the same setting as in the above definition 4.2. 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:** (Lemma 4.2 on page 18)

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

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

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

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

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

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

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

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

□

**LEMMA:** Pseudorandom Sequences are Ultimately Periodic

*Let $\mathcal{G} \coloneqq (S, T, U, G)$ 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*

> *$\rho$ and the transient $\tau$ the following holds.*
>
> $$1 \leq \rho + \tau - 1 \leq \#S$$
>
> *In particular, if $T$ is bijective $(u_n)$ will be periodic.*

**PROOF:** (Lemma 5.2 on page 32)

Let $(s_n)_{n\in\mathbb{N}}$ be the respective sequence of states and $N \coloneqq \#S$ 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.

$$\rho \coloneqq n - k + 1 \qquad \tau \coloneqq k$$

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. □

> **LEMMA:** Equidistributed Pseudorandom Sequences
>
> *Let $\mathcal{G} \coloneqq (S, T, U, G)$ 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 $\tau$ and period $\rho$. Furthermore, let $\mu$ be a probability measure on $(U, \mathcal{P}(U))$. Then the following statements are equivalent.*
>
> *(i) $(u_n)$ is equidistributed with respect to $\mu$.*
>
> *(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:** (Lemma 5.3 on page 34)

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 $\alpha_u$ 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.

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

This shows the desired equivalence and proofs the lemma. □

**LEMMA:** 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 $\rho$ and transient $\tau$. In this case, every corresponding $k$-dimensional vector sequence $(v_n)_{n\in\mathbb{N}}$ with translation $t$ is ultimately periodic with period $\rho'$ and transient $\tau'$ defined as follows.*

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

**PROOF:** (Lemma 5.5 on page 35)

Choose $n\in\mathbb{N}_0$ and $i\in\mathbb{N}$ with $i\le k$ 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 $\rho$ 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 $\tau$ of $(u_n)$.

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

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, $\rho'$ and $\tau'$ are indeed the smallest possible values such that this equation holds and can therefore be denoted as period and transient of $(v_n)$ respectively. □

# B Further Code

Code B.1: AVX MT19937 Advancing Assembler

```
pxart::simd256::mt19937::operator()():
        movsx   rdx, DWORD PTR [rdi+2528]
        mov     rax, rdx
        cmp     rdx, 623
        ja      .L2
        lea     rdx, [rdi+rdx*4]
        add     eax, 8
.L3:
        vmovdqa ymm1, YMMWORD PTR [rdx]
        mov     DWORD PTR [rdi+2528], eax
        vpsrld  ymm0, ymm1, 11
        vpxor   ymm1, ymm1, ymm0
        vpslld  ymm0, ymm1, 7
        vpand   ymm0, ymm0, YMMWORD PTR .LC4[rip]
        vpxor   ymm1, ymm0, ymm1
        vpslld  ymm0, ymm1, 15
        vpand   ymm0, ymm0, YMMWORD PTR .LC5[rip]
        vpxor   ymm0, ymm0, ymm1
        vpsrld  ymm1, ymm0, 18
        vpxor   ymm0, ymm0, ymm1
        ret
.L2:
        vmovdqa ymm2, YMMWORD PTR .LC0[rip]
        vmovdqa ymm3, YMMWORD PTR .LC1[rip]
        lea     rax, [rdi+32]
        lea     rdx, [rdi+928]
        vpand   ymm0, ymm3, YMMWORD PTR [rdi]
        vpand   ymm1, ymm2, YMMWORD PTR [rdi+4]
        vmovdqa ymm4, YMMWORD PTR .LC2[rip]
        vmovdqa ymm5, YMMWORD PTR .LC3[rip]
        vpor    ymm1, ymm1, ymm0
        vpsrld  ymm6, ymm1, 1
        vpand   ymm0, ymm1, ymm4
        vpxor   xmm1, xmm1, xmm1
        vpcmpgtd        ymm0, ymm0, ymm1
        vpand   ymm0, ymm5, ymm0
        vpxor   ymm0, ymm0, YMMWORD PTR [rdi+1588]
        vpxor   ymm0, ymm0, ymm6
        vmovdqa ymm6, ymm1
        vmovdqa YMMWORD PTR [rdi], ymm0
        vmovdqa YMMWORD PTR [rdi+2496], ymm0
.L4:
        vpand   ymm1, ymm3, YMMWORD PTR [rax]
        vpand   ymm0, ymm2, YMMWORD PTR [rax+4]
        add     rax, 32
        vpor    ymm0, ymm0, ymm1
        vpsrld  ymm1, ymm0, 1
        vpand   ymm0, ymm0, ymm4
        vpxor   ymm1, ymm1, YMMWORD PTR [rax+1556]
        vpcmpgtd        ymm0, ymm0, ymm6
        vpand   ymm0, ymm5, ymm0
        vpxor   ymm0, ymm0, ymm1
        vmovdqa YMMWORD PTR [rax-32], ymm0
        cmp     rdx, rax
        jne     .L4
        lea     rax, [rdi+20]
        lea     rdx, [rdi+1588]
        vpxor   xmm6, xmm6, xmm6
.L5:
        vpand   ymm1, ymm3, YMMWORD PTR [rax+908]
        vpand   ymm0, ymm2, YMMWORD PTR [rax+912]
```

```
        add     rax, 32
        vpor    ymm0, ymm0, ymm1
        vpsrld  ymm1, ymm0, 1
        vpand   ymm0, ymm0, ymm4
        vpxor   ymm1, ymm1, YMMWORD PTR [rax-32]
        vpcmpgtd        ymm0, ymm0, ymm6
        vpand   ymm0, ymm5, ymm0
        vpxor   ymm0, ymm0, ymm1
        vmovdqa YMMWORD PTR [rax+876], ymm0
        cmp     rdx, rax
        jne     .L5
        mov     DWORD PTR [rdi+2528], 0
        mov     rdx, rdi
        mov     eax, 8
        jmp     .L3
```

Code B.2: Xoroshiro128+ Jump

```
constexpr void xoroshiro128plus::jump() noexcept {
  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;
      }
      result0 ^= s0 & tmp;
      result1 ^= s1 & tmp;
      operator()();
    }
  }
  s0 = result0;
  s1 = result1;
}
```

Code B.3: AVX Xoroshiro128+ Jump

```
inline void xoroshiro128plus::jump() noexcept {
  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 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 B.4: 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 B.5: 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 B.6: 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
```

Code B.7: 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);
    }
  }
}
```

Code B.8: 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);
  }
}
```

Code B.9: 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;
```

```
};
```

Code B.10: 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});
}
```

Code B.11: Photon Simulation AVX Advancing

```
template <typename RNG>
inline void advance(photons& sys, RNG& rng) noexcept {
  const auto zero = _mm256_setzero_ps();
  const auto one = _mm256_set1_ps(1);
  const auto half = _mm256_set1_ps(0.5);

  const auto time_step = _mm256_set1_ps(sys.time_step);
  const auto scatter = _mm256_set1_ps(sys.scatter);

  for (int i = 0; i < sys.p_x.size(); i += 8) {
    // Load positions and velocities of eight photons.
    const auto p_x = _mm256_loadu_ps(&sys.p_x[i]);
    const auto p_y = _mm256_loadu_ps(&sys.p_y[i]);
    const auto v_x = _mm256_loadu_ps(&sys.v_x[i]);
    const auto v_y = _mm256_loadu_ps(&sys.v_y[i]);
```

```
    // Move photons according to their velocity.
    const auto new_p_x = _mm256_add_ps(p_x, _mm256_mul_ps(time_step, v_x));
    const auto new_p_y = _mm256_add_ps(p_y, _mm256_mul_ps(time_step, v_y));
    // Store the new positions.
    _mm256_storeu_ps(&sys.p_x[i], new_p_x);
    _mm256_storeu_ps(&sys.p_y[i], new_p_y);

    // Evaluate which photon has to be scattered.
    const auto rnd = pxart::simd256::uniform<float>(rng);
    const auto mask = _mm256_cmp_ps(rnd, scatter, _CMP_LT_OQ);

    // Sample the Henyey-Greenstein phase function.
    const auto g = _mm256_set1_ps(sys.g);
    const auto sq_g = _mm256_mul_ps(g, g);
    const auto g2 = _mm256_add_ps(g, g);
    auto denom = _mm256_sub_ps(one, g);
    denom = _mm256_add_ps(denom, _mm256_mul_ps(g2, rnd));
    const auto numer = _mm256_sub_ps(one, sq_g);
    const auto frac = _mm256_div_ps(numer, denom);
    auto brackets = _mm256_add_ps(one, sq_g);
    brackets = _mm256_sub_ps(brackets, _mm256_mul_ps(frac, frac));
    const auto cos_angle = _mm256_div_ps(brackets, g2);

    // Sample the sign of the sine of the angle.
    const auto rnd2 = pxart::simd256::uniform<float>(rng);
    const auto abs_sin_angle =
        _mm256_sqrt_ps(_mm256_sub_ps(one, _mm256_mul_ps(cos_angle, cos_angle)));
    const auto lr_mask = _mm256_cmp_ps(rnd2, half, _CMP_LT_OQ);
    const auto sin_angle = _mm256_blendv_ps(
        abs_sin_angle, _mm256_sub_ps(zero, abs_sin_angle), lr_mask);

    // Rotate velocity of photons by matrix-vector product with rotation matrix.
    const auto new_v_x = _mm256_sub_ps(_mm256_mul_ps(cos_angle, v_x),
                                       _mm256_mul_ps(sin_angle, v_y));
    const auto new_v_y = _mm256_add_ps(_mm256_mul_ps(sin_angle, v_x),
                                       _mm256_mul_ps(cos_angle, v_y));

    // Store velocities according to scattering evaluation.
    _mm256_storeu_ps(&sys.v_x[i], _mm256_blendv_ps(new_v_x, v_x, mask));
    _mm256_storeu_ps(&sys.v_y[i], _mm256_blendv_ps(new_v_y, v_y, mask));
  }
}
```

# C Further Results

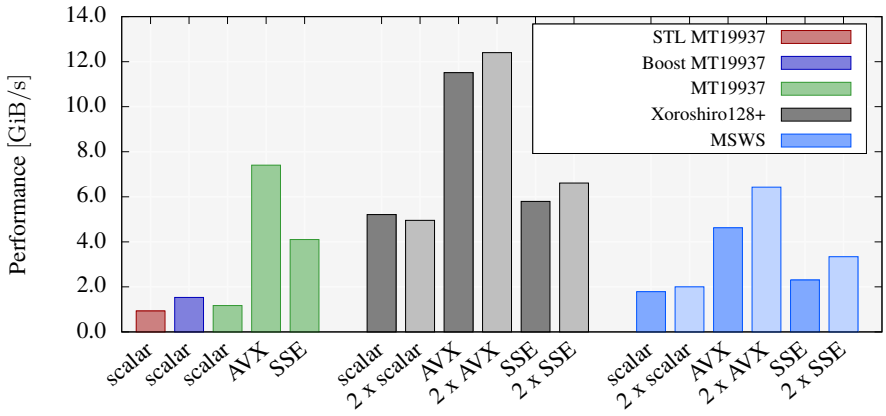



Figure C.31: Performance resulting from the generation benchmark running on the *Intel® Core™ i5-8250U Processor* measured in GiB of random numbers per second for the different variants of the implemented PRNGs. For convenience, the performances of the STL and Boost implementation of the MT19937 are shown as well. The data can be found in table C.4.

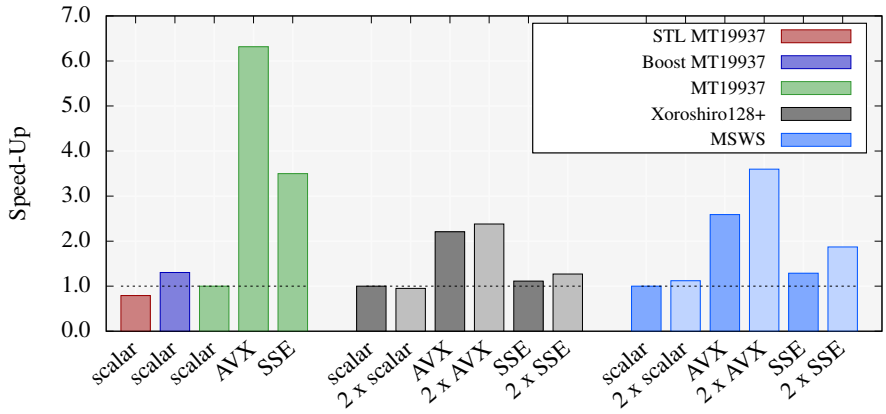
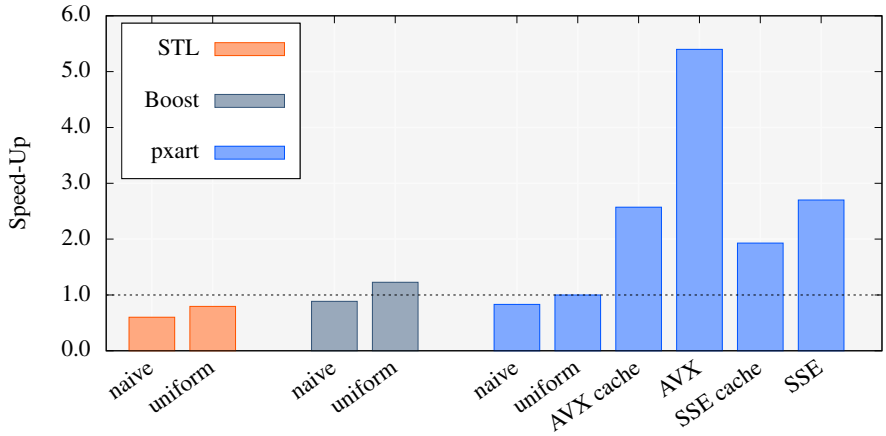


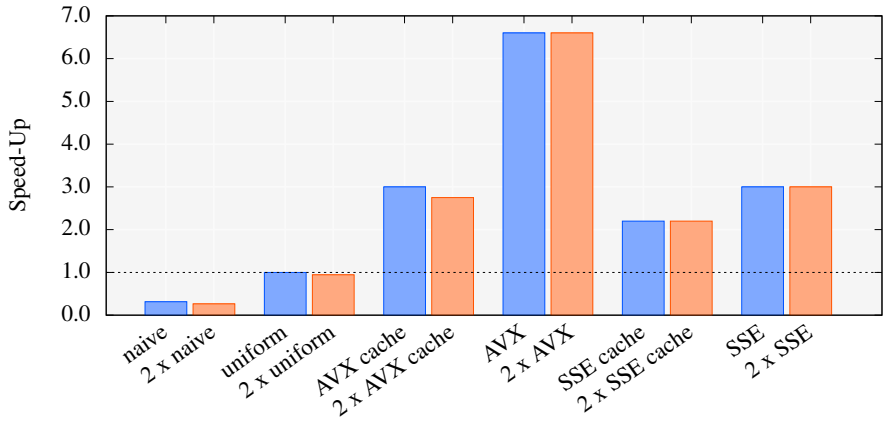

Figure C.32: Speed-up in execution time with respect to the single-instance scalar version resulting from the generation benchmark running on the *Intel® Core™ i5-8250U Processor* for the different variants of the implemented PRNGs. For convenience, the speed-ups of the STL and Boost implementation of the MT19937 are shown with respect to our implementation. The data can be found in table C.4.

Table C.4: 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.

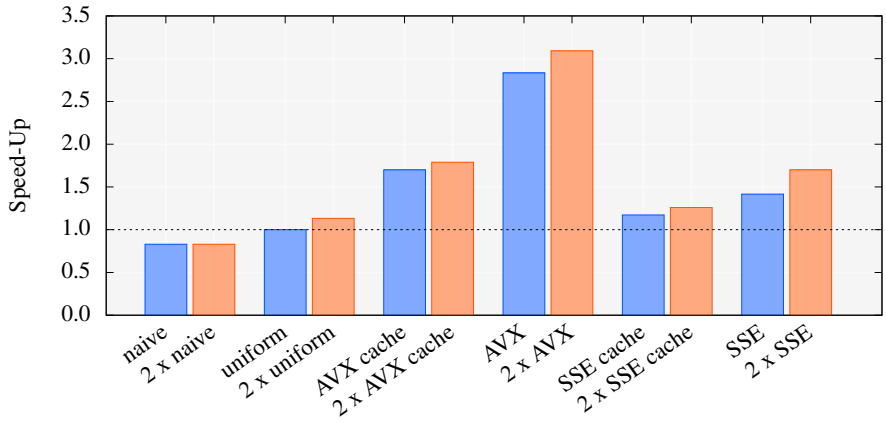
| **PRNG** | **Variant** | **Result Size** [B] | **Time** [s] | **Cycles** | **Instructions** | **IPC** |
|---|---|---|---|---|---|---|
| STL MT19937 | scalar | 4 | 17.23 | 9.24 | 28.34 | 3.07 |
| Boost MT19937 | scalar | 4 | 10.46 | 5.83 | 24.86 | 4.26 |
| MT19937 | scalar | 4 | 13.64 | 7.61 | 26.71 | 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 C.33: Speed-up in execution time with respect to the single-instance uniform version resulting from the Monte Carlo $\pi$ 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 C.5.

Table C.5: Results achieved by running the Monte Carlo $\pi$ 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 $\pi$. It was ensured that the estimation error was small enough according to the calculation at the end of section 4.1. 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 19, 2019

Markus Pawellek