Friedrich-Schiller-Universität Jena
Physikalisch-Astronomische Fakultät

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

MASTER'S THESIS

for obtaining the academic degree

Master of Science (M.Sc.) in Physics

submitted by Markus Pawellek

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

Primary Reviewer:	Bernd Brügmann
Primary Supervisor:	Joachim Gießen

Jena, August 14, 2019

Abstract

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

Acknowledgements

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

Contents

List of Figures

List of Abbreviations

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

Symbol Table

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

1 Introduction

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

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

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

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

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

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

2 Background

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

2.1 Mathematical Preliminaries

Stochastics

The observation of random processes resulted in the construction of probability theory. But an introduction to it can be given without a further formalized concept of randomness. Randomness itself plays a minor role and is used in form of realizations of random variables. As a consequence probability theory allows us to observe randomness without defining it. Actually, typical formalizations rely on probability theory. This connection makes the development of RNGs possible. Hence, in the following we will give only the formal definition of relevant structures without further discussions and will postpone an examination of randomness to the next subsection.

DEFINITION 2.1: (Probability Space)

A tuple $(\Omega, \mathcal{A}, P)$ with a non-empty set Ω, a σ-Algebra $\mathcal{A}$, and probability measure P is called probability space if the following conditions hold.

1. $\emptyset, \Omega \in \mathcal{A}$
2. $A \in \mathcal{A} \implies \Omega \setminus A \in \mathcal{A}$
3. $\{A_n \mid n \in \mathbb{N}\} \subset \mathcal{A} \implies \bigcup_{n\in\mathbb{N}} A_n \in \mathcal{A}$

1. $P(\Omega) = 1$
2. $\{A_n \mid n \in \mathbb{N}\} \subset \mathcal{A}, A_n \cap A_m = \emptyset \implies P(\bigcup_{n\in\mathbb{N}} A_n) = \sum_{n\in\mathbb{N}} P(A_n)$

DEFINITION 2.2: (Random Variable)

A random variable is a measurable function $X\colon \Omega \to \mathbb{R}$ on a probability space.

We call $X(\omega)$ for $\omega \in \Omega$ a realization of X.

DEFINITION 2.3: (Independence)

DEFINITION 2.4: (Probability Measure of a Random Variable)

$$P_X = P \circ X^{-1}$$

DEFINITION 2.5: (Expectation Value)

$$\mathbb{E}X = \int_\Omega X \,\mathrm{d}P$$

DEFINITION 2.6: (Probability Density and Cumulative Distribution Function)

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

PROPOSITION 2.1:

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

Finite Fields

2.2 Pseudorandom Number Generators

PRNGs were first introduced by Neumann. A further discussion about randomness will make clear why the design of PRNGs can be seen as art.

Random and Pseudorandom Sequences

Originating in gambling and physical processes, randomness is a difficult concept and drives many philosophical discussions. Typically humans have a bad idea of randomness. Randomness makes only sense when it is applied to a sequence of values. Because we want to generate random numbers we only need a formal mathematical structure to work with. However, a widely accepted unique formal concept has not been found. But as stated in Volchan 2002 the question if a sequence is random decides at infinity. A random sequence, in general, is not computable or compressible by an algorithm. Even the methods to test this kind of randomness cannot be computed. For the development of RNGs on a computer, we cannot use such concepts. A computer in our sense is only capable of using finite sequences and has to compute its randomness to check it. Therefore, again as stated in Volchan 2002, we will stick to 'if it looks random, it is random'. In Kneusel 2018 the concept of randomness was simplified.

Random and Pseudorandom Number Generators

DEFINITION 2.7: (Pseudorandom Number Generator)

A tuple (S, s_0, T, U, G) is called a PRNG. S is a non-empty, finite set of states. $s_0 \in S$ is the initial state. $T\colon S \to S$ is the transition function. U is a non-empty, finite set of output symbols. $G\colon S \to U$ is the output function which generates an output symbol for every state.

DEFINITION 2.8: (Pseudorandom Sequence of PRNG)

$(s_n)_{n\in\mathbb{N}}$ *is the respective sequence of states*

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

Pseudorandom sequence $(u_n)_{n\in\mathbb{N}}$

$$u_n = G(s_n)$$

$$s_0 \xrightarrow{T} s_1 \xrightarrow{T} s_2 \xrightarrow{T} \dots \tag{1}$$

$$G \downarrow \qquad \downarrow \tag{2}$$

$$u_1 \qquad u_2 \qquad \dots \tag{3}$$

2.3 SIMD-Capable Processors

Architecture of Modern Central Processing Units

SIMD Instruction Sets and Efficiency

SSE, AVX, AVX512

2.4 Simulation in Physics and Mathematics

Mathematical and Physical Preliminaries

Baseline Model Problems

2.5 Summary

3 Previous Work

3.1 The C++ API and Further Progressions

3.2 Techniques for Vectorization and Parallelization

3.3 Summary

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

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

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

4 Design of the API

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

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

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

5 Testing Framework

6 Implementation of Vectorized PRNGs

6.1 Linear Congruential Generatiors

6.2 Mersenne Twister

6.3 Permuted Congruential Generators

6.4 Xoroshiro

6.5 Middle Square Weyl Generator

6.6 Summary

7 Application to Simulations

8 Evaluation and Results

godbolt google benchmark intel vtune amplifier testu01 dieharder

9 Conclusions

References

General

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

Barash, L. Yu. and Lev. N. Shchur: *PRAND: GPU Accelerated Parallel Random Number Generation Library: Using Most Reliable Algorithms and Applying Parallelism of Modern GPUs and CPUs*. In: *Computer physics communications* 185.4 (2014), pp. 1343–1353.

— *RNGSSELIB: Program Library for Random Number Generation. More Generators, Parallel Streams of Random Numbers and Fortran Compatibility*. In: *Computer Physics Communications* 184.10 (2013), pp. 2367–2369.

— *RNGSSELIB: Program Library for Random Number Generation, SSE2 Realization*. In: *Computer Physics Communications* 182.7 (2011), pp. 1518–1527.

Bauke, Heiko and Stephan Mertens: *Random numbers for large-scale distributed Monte Carlo simulations*. In: *Physical Review E* 75.6 (2007), p. 066701.

Demirag, Yigit: *Vectorization Studies of Random Number Generators on Intel's Haswell Architecture*. CERN openlab, 2014.

Fog, Agner: *Calling Conventions for Different C++ Compilers and Operating Systems*. Agner Fog, 2018. URL: https://www.agner.org/optimize/calling_conventions.pdf.

— *Optimizing Software in C++: An Optimization Guide for Windows, Linux and Mac Platforms*. Agner Fog, 2018. URL: https://www.agner.org/optimize/optimizing_cpp.pdf.

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

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

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

Kneusel, Ronald T.: *Random Numbers and Computers*. Springer, 2018. ISBN: 978-3-319-776696-5.

Landau, David P. and Kurt Binder: *A Guide to Monte Carlo Simulations in Statistical Physics*. Fourth Edition. Cambridge University Press – University of Cambridge, 2015. ISBN: 978-1-107-07402-6.

L'Ecuyer, Pierre: "Random number generation with multiple streams for sequential and parallel computing". In: *2015 Winter Simulation Conference (WSC)*. IEEE. 2015, pp. 31–44.

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

Leobacher, Gunther and Friedrich Pillichshammer: *Introduction to Quasi-Monte Carlo Integration and Applications*. Birkhäuser – Springer, 2010. ISBN: 978-3-319-03424-9.

Marsaglia, George et al.: *Xorshift RNGs*. In: *Journal of Statistical Software* 8.14 (2003), pp. 1–6.

Matsumoto, Makoto and Takuji Nishimura: *Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator*. In: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 8.1 (1998), pp. 3–30.

McCool, Michael, Arch D. Robison, and James Reinders: *Structured Parallel Programming: Patterns of Efficient Computation*. Morgan Kaufmann – Elsevier, 2012. ISBN: 978-0-12-415993-8.

Müller-Gronbach, Thomas, Erich Novak, and Klaus Ritter: *Monte Carlo-Algorithmen*. Springer, 2012. ISBN: 978-3-540-89140-6.

O'Neill, Melissa E.: *PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation*. In: *ACM Transactions on Mathematical Software* (2014).

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

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

Widynski, Bernard: *Middle Square Weyl Sequence RNG*. In: *arXiv preprint arXiv:1704.00358v3* (2019).

Programming

cppreference.com. July 15, 2019. URL: https://en.cppreference.com/w/.

Galowicz, Jacek: *C++17 STL Cookbook*. Packt, 2017. ISBN: 978-1-78712-049-5.

Josuttis, Nicolai M.: *The C++ Standard Library: A Tutorial and Reference*. Second Edition. Addison-Wesley – Pearson Education, 2012. ISBN: 978-0-321-62321-8.

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

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

Scott, Craig: *Professional CMake: A Practical Guide*. Version: 1.0.3. Craig Scott, 2018. URL: https://crascit.com.

Standard C++ Foundation. July 15, 2019. URL: https://isocpp.org/.

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

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

Williams, Anthony: *C++ Concurrency in Action*. Second Edition. Manning Publications, 2019. ISBN: 9781617294693.

A Build System

Fog 2018b

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, August 14, 2019

Markus Pawellek