Tina's Random Number Generator Library

Version 4.24

Heiko Bauke

March 27, 2021

"The state of the art for generating uniform deviates has advanced considerably in the last decade and now begins to resemble a mature field."

Press et al. [63]

Contents

1 TRNG in a nutshell

1.1 Introduction

The Monte Carlo method is a widely used and commonly accepted simulation technique in physics, operations research, artificial intelligence, and other fields, and pseudo-random numbers (PRNs) are its key resource. All Monte Carlo simulations include some sort of averaging of independent samples, a calculation that is embarrassingly parallel. Hence it is no surprise that more and more large scale simulations are run on parallel systems like networked workstations, clusters, multicore systems or high-performance graphics cards. For each Monte Carlo simulation the quality of the PRN generator (PRNG) is a crucial factor. In a parallel environment the quality of a PRNG is even more important than in a non-parallel environment to some extent because feasible sample sizes are easily $10 \ldots 10^4$ times as large as on a sequential machine. The main problem, however, is the parallelization of the PRNG itself.

Application programmers and scientists need not to grapple with all the technical details of pseudo-random number generation if a PRNG library is used. The following requirements are frequently demanded from a library for (parallel) pseudo-random number generation:

- The library should provide a set of different interchangeable algorithms for pseudo-random number generation.
- For each algorithm different well tested parameter sets should be provided that guarantee a long period and good statistical properties.
- The internal state of a PRNG can be saved for later use and restored. This makes it possible to stop a simulation and to carry on later.
- PRNGs have to support block splitting and leapfrog, see section 2.1.
- The library should provide methods for generating random variables with various distributions, uniform and non-uniform.
- The library should be implemented in a portable, speed-optimized fashion.

If these are also your requirements for a PRNG library, you should go with Tina's Random Number Generator Library.

Tina's Random Number Generator Library (TRNG) is a state of the art C++ pseudo-random number generator library for sequential and parallel Monte Carlo simulations. Its design principles are based on the extensible random number generator facility that was introduced in the C++11 standard [27, 28]. The TRNG library features an object oriented design, is easy to use and has been speed optimized. Its implementation does not depend on any communication library or hardware architecture. TRNG is suited for shared memory as well as for distributed memory computers and may be used in any parallel programming environment, e. g., Message Passing Interface Standard or OpenMP. All generators that are implemented by TRNG have been subjected to thorough statistical tests in sequential and parallel setups, see also section 8.

This reference is organized as follows. In chapter 2 we present some basic techniques for parallel random number generation, chapter 3 introduces the basic concepts of TRNG, whereas chapter 4 describes all classes of TRNG in detail. In chapter 5 we give installation instructions, and chapter 6 presents some example programs that demonstrate the usage of TRNG in sequential as well as in parallel Monte Carlo applications. Chapter 7 deals with some implementation details and performance issues. We complete the TRNG reference with the presentation of some statistical tests of the PRNGs of TRNG in chapter 8 and answer some FAQs in chapter 9.

This manual can be read in several ways. You might read this manual chapter by chapter from the beginning to its end. Impatient readers should read at least chapter 2 to familiarize themselves with some basic terms that are used in this text before they jump to chapter 5 and chapter 6. These chapters deal with the installation and give some example code. Chapters 3 and 4 are mainly for reference and the reader will come back to them again and again.

The TRNG manual is not written as an introduction to the Monte Carlo method. It is assumed that the reader already knows the basic concepts of Monte Carlo. Novices in the Monte Carlo business find further information in various textbooks on this topic [21, 66, 57, 35, 34, 52].

1.2 History

TRNG started in 2000 as a student research project. Its implementation as well as its technical design has changed several times. Starting with version 4.0 we adopted the interface proposed by [11] and finally adopted by the C++11 standard [27, 28].

Version 4.0 Initial release of TRNG that implements the interface proposed by [11].

Version 4.1 Additive and exclusive-or lagged Fibonacci generators with two and four feedback taps have been added to the set of PRNGs. Lagged Fibonacci generators do not provide any splitting facilities. TRNG implements the template function `generate_canonical` introduced by [11].

Version 4.2 Documentation has been revised. Minor bug-fixes to lagged Fibonacci generators.

Version 4.3 Rayleigh distribution and class for correlated normal distributed random numbers added. Changed default parameter sets for generators `mrg3s`, `mrg5s`, `yarn3s`, and `yarn5s`. The new parameter sets perform better in the spectral test.

Version 4.4 Class for discrete distributions rewritten to allow efficient change of relative probabilities after initialization. New random number engine `lcg64_shift` introduced.

Version 4.5 Minor improvements and bug fixes. Utility functions `uniformcc`, `uniformco`, `uniformoc`, and `uniformoo` had been reimplemented as suggested by Bruce Carneal. The new implementation of these functions is slightly faster and generates random numbers that are distributed more evenly in the intervals $[0,1]$, $[0,1)$, $(0,1]$, and $(0,1)$ respectively. Added support for Snedecor-F- and Student-t-distribution and the class `fast_discrete_dist` for faster generation of discrete random numbers withe arbitrary distribution.

Version 4.6 Reimplementation of `generate_canonical`, added sequential random number engines `mt19937` and `mt19937_64` (Mersenne twister generators). All classes for continuous random number distributions had been reimplemented as template classes. The template parameter determines the `result_type` and may be `float`, `double` or `long double`,

`double` is the default template parameter. Bugfixes for several continuous random number distributions.

Version 4.7 In order to prevent name clashes macros in header file `trng/config.hpp` have been put into its own namespace `TRNG`. Section 6 has been extended to demonstrate how to write parallel Monte Carlo applications using TRNG and Intel Threading Building Blocks.

Version 4.8 Performance improvements for `split` methods of the classes `mrg`*n*, `mrg`*n*`s`, `yarn`*n*, and `yarn`*n*`s`. The computational complexity has been reduced from linear (in the number of sub-streams) to logarithmic scaling.

Version 4.9 A new random number distribution class `hypergeometric_dist` and a new random number engine class `mlcg2_64` have been implemented. Performance improvements for `split` methods of the classes `lcg64` and `lcg64_shift`. The computational complexity has been reduced from linear (in the number of sub-streams) to logarithmic scaling. Applied various corrections[1] and clarifications to the TRNG documentation. TRNG compiles now with Sun Studio compiler. Starting from version 4.9, the TRNG library is distributed under the terms of a BSD style license (3-clause license).

Version 4.10 Two additional random number distribution classes `twosided_exponential_dist` and `truncated_normal_dist` have been implemented.

Version 4.11 TRNG starts to support parallel processing on graphics cards via the CUDA architecture. Various minor improvements.

Version 4.12 Bug fixes and various minor improvements.

Version 4.13 Bug-fix and service release.

Version 4.14 Some minor changes of the class interfaces, bugfix for class `binomial_dist`. Starting with version 4.14 we move from the class interface as proposed by [11] to the class interface of the C++11 standard [27, 28]. These interfaces differ in some details only. Adopting the C++11 interface for TRNG allows to mix TRNG classes and classes from the C++11 random number library, see section 6.4 for details.

Version 4.15 Bug-fix and service release. Improvements mainly related to the build system. The additional random number distribution classes `maxwell_dist` and `beta_dist` have been implemented. New e-mail address trng@mail.de.

Version 4.16 Bug-fix and service release. Some bug fixes for classes `discrete_distribution` and `beta_dist` have been applied. (One of the corresponding bugs appeared in the class `discrete_distribution` if the number of weights was a power of 2. The other bugs were syntactical errors preventing TRNG to compile.) TRNG 4.16 features the new random number distribution class `negative_binomial_dist`.

Version 4.17 Bug-fix and service release.

Version 4.18 The additional random number distribution class `zero_truncated_poisson_dist` has been implemented.

[1] Many thanks to Rodney Sparapani.

Version 4.19 Random number engines use internally integer types of exactly 32 bits or 64 bits, respectively, instead of `(unsigned) long int` and `(unsigned) long long int`. New typedefs for lagged Fibonacci generators have been introduced. The old ones (ending with `_ul` or `_ull`) are architecture dependent and should be considered as depreciated. This and later versions will not compile on exotic platforms where none of the integer types `int`, `long int`, and `long long int` has exactly 32 or 64 bits. This version beaks ABI compatibility to earlier versions but retains source code compatibility.

Version 4.20 Bug-fix and service release.

Version 4.21 Bug-fix and service release. Fixes numerical convergence problems in the inverse of the incomplete Beta function.

Version 4.22 This maintenance release removes old code for supporting C++ language standards older than C++11. Many minor code enhancements and bug fixes have been applied. The autotools-based build system has been replaced by CMake to modernize the build process and enhance portability, see installation instructions. The negative binomial distribution has been generalized to real-valued parameters.

Version 4.23 This is primarily a maintenance release focusing on code quality. Starting with this release TRNG employs systematic unit testing on the basis of the Boost unit test frame work. The numerical accuracy of several special mathematical functions (e. g., cumulative probability density of the normal distribution) have been enhanced. The `discard` method of the lagged Fibonacci generators has been re-implemented using an algorithm with logarithmic asymptotic complexity.

Version 4.24 The two new random number engines, called `xoshiro256plus` and `lcg64_count_shift`, have been implemented. New unit tests have been intoduced to extend test coverage. Special-functions unit tests use reference values with improved numerical accuracy now. The numerical accuracy of various special functions has been improved to reach machine precision also in 128-bit floating point number arithmetic, e. g., the inverse cumulative probability distribution of the normal distribution, incomplete gamma functions and the Beta function. An uninitialized memory read access has been fixed. (Many thanks to Mirai Solutions [69] for reporting this issue.) The documentation has been improved and extended. The chapter on quality and statistical tests has been rewritten based on results of the Dieharder test suite.

2 Pseudo-random numbers for parallel Monte Carlo simulations

2.1 Pseudo-random numbers

Monte Carlo methods are a class of computational algorithms for simulating the behavior of various physical and mathematical systems by a stochastic process. While simulating such a stochastic process on a computer, large amounts of random numbers are consumed. Actually, a computer as a deterministic machine is not able to generate random digits. John von Neumann, pioneer in Monte Carlo simulation, summarized this problem in his famous quote:

> "Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin."

For computer simulations we have to content ourselves with something weaker than random numbers, namely *pseudo-random* numbers. We define a stream of PRNs r_i in the following in an informal manner:

- PRNs are generated by a deterministic rule.
- A stream of PRNs r_i cannot be distinguished from a true random sequence by means of practicable methods applying a *finite* set of statistical tests on *finite* samples.

Almost all PRNGs produce a sequence $r_0, r_1, r_2, \ldots$ of PRNs by a recurrence

$$r_i = f(r_{i-1}, r_{i-2}, \ldots, r_{i-k}) , \tag{2.1}$$

and the art of random number generation lies in the design of the function $f(\cdot)$. The objective in PRNG design is to find a transition algorithm $f(\cdot)$ that yields a PRNG with a long period and good statistical properties within the stream of PRNs. Statistical properties of a PRNG may be investigated by theoretical or empirical means, see [33]. But experience shows, there is nothing like an ideal PRNG. A PRNG may behave like a perfect source of randomness in one kind of Monte Carlo simulation, whereas it may suffer from significant statistical correlations if it is used in another context, which makes the particular Monte Carlo simulation unreliable.

Numerous recipes for $f(\cdot)$ in (2.1) have been discussed in the literature, see [33, 40] and references therein. We will present some popular schemes and review some of theirs mathematical properties in sections 2.4 and 2.5. Readers how do not want to bother with mathematical details might skip these sections and may come back later if necessary. However, the next two sections on the parallelization of PRN sequences and on playing fair present important concepts of the TRNG library.

2.2 General parallelization techniques for PRNGs

In parallel applications, we need to generate streams $t_{j,i}$ of random numbers [6, 53, 58]. Streams are numbered by $j = 0, 1, \ldots, p-1$, where p is the number of processes. We require statistical

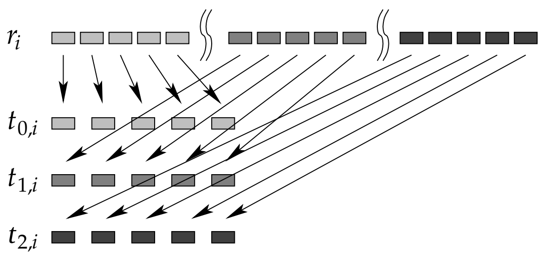


Figure 2.1: Parallelization by block splitting.

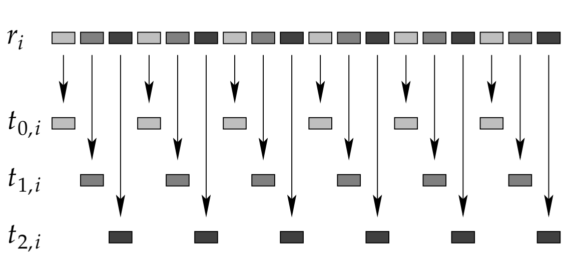


Figure 2.2: Parallelization by leapfrogging.

independence of the $t_{j,i}$ within each stream and between streams as well. Four different parallelization techniques are used in practice:

Random seeding: All processes use the same PRNG but a different "random" seed. The hope is that they will generate non-overlapping and uncorrelated subsequences of the original PRNG. This hope, however, has no theoretical foundation. Random seeding is a violation of Donald Knuth's advice "Random number generators should not be chosen at random" [33].

Parameterization: All processes use the same type of generator but with different parameters for each processor. Example: linear congruential generators with additive constant b_j for the jth stream [62]:

$$t_{j,i} = a \cdot t_{j,i-1} + b_j \bmod 2^e\,, \tag{2.2}$$

where b_j is the $(j+2)$th prime number. Another variant uses different multipliers a for different streams [46]. The theoretical foundation of these methods is weak, and empirical tests have revealed serious correlations between streams [50]. On massive parallel system you may need thousands of parallel streams, and it is not trivial to find a type of PRNG with thousands of "well tested" parameter sets.

Block splitting: Let M be the maximum number of calls to a PRNG by each processor, and let p be the number of processes. Then we can split the sequence r_i of a sequential PRNG into consecutive blocks of length M such that

$$\begin{aligned} t_{0,i} &= r_i \\ t_{1,i} &= r_{i+M} \\ &\dots \\ t_{p-1,i} &= r_{i+M(p-1)}\,. \end{aligned} \tag{2.3}$$

This method works only if we know M in advance or can at least safely estimate an upper bound for M. To apply block splitting it is necessary to jump from the ith random number to the $(i+M)$th number without calculating all the numbers in between, which cannot be done efficiently for many PRNGs. A potential disadvantage of this method is that long range correlations, usually not observed in sequential simulations, may become short range correlations between sub-streams [51, 18]. Block splitting is illustrated in Figure 2.1.

Leapfrog: The leapfrog method distributes a sequence r_i of random numbers over p processes by decimating this base sequence such that

$$\begin{aligned} t_{0,i} &= r_{pi} \\ t_{1,i} &= r_{pi+1} \\ &\dots \\ t_{p-1,i} &= r_{pi+(p-1)} \,. \end{aligned} \tag{2.4}$$

Leapfrogging is illustrated in Figure 2.2. It is the most versatile and robust method for parallelization and it does not require an a priori estimate of how many random numbers will be consumed by each processor. An efficient implementation requires a PRNG that can be modified to generate directly only every pth element of the original sequence. Again this excludes many popular PRNGs.

At first glance block splitting and leapfrog seem to be quite different approaches. But in fact, these are closely related to each other. Because if leapfrog is applied to any *finite* base sequence the leapfrog sequences are cyclic shifts of each other. Consider an arbitrary sequence r_i with period T. If $\gcd(T, p) = 1$, all leapfrog sequences $t_{1,i}, t_{2,i}, \dots, t_{p,i})$ are cyclic shifts of each other, i. e., for every pair of leapfrog sequences $t_{j_1,i}$ and $t_{j_2,i}$ of a common base sequence r_i with period T there is a constant s, such that $t_{j_1,i} = t_{j_2,i+s}$ for all i, and s is at least $\lfloor T/p \rfloor$. Furthermore, if $\gcd(T, p) = d > 1$, the period of each leapfrog sequence equals T/d and there are d classes of leapfrog sequences. Within a class of leapfrog sequences there are p/d sequences, each sequence is just a cyclic shift of another and the size of the shift is at least $\lfloor T/p \rfloor$.

The first two methods, random seeding and parameterization, have little or no theoretical backup, but their weakest point is yet another. The results of a simulation should not depend on the number of processors it runs on. Leapfrog and block splitting do allow to organize simulations such that the same random numbers are used independently of the number of processors. With parameterization or random seeding the results will always depend on the parallelization, see section 6.2 for details. PRNGs that do not support leapfrog and block splitting should not be used in parallel simulations.

2.3 Playing fair

We say that a parallel Monte Carlo simulation *plays fair*, if its outcome is strictly independent of the underlying hardware. Fair play implies the use of the same PRNs in the same context, independently of the number of parallel processes. It is mandatory for debugging, especially in parallel environments where the number of parallel processes varies from run to run, but another benefit of playing fair is even more important: Fair play guarantees that the quality of a PRNG with respect to an application does not depend on the degree of parallelization.

Obviously the use of parameterization or random seeding prevent a simulation from playing fair. Leapfrog and block splitting, on the other hand, do allow the use of the same PRNs within the same context independently of the number of parallel streams.

Consider the site percolation problem. A site in a lattice of size N is occupied with some probability, and the occupancy is determined by a PRN. M random configurations are generated. A naive parallel simulation on p processes could split a base sequence into p leapfrog streams and having each process generate $\approx M/p$ lattice configurations, independently of the other processes. Obviously this parallel simulation is not equivalent to its sequential version that consumes PRNs from the base sequence to generate one lattice configuration after another. The effective shape of the resulting lattice configurations depends on the number of processes. This parallel algorithm does not play fair.

We can turn the site percolation simulation into a fair playing algorithm by leapfrogging on the level of lattice configurations. Here each process consumes distinct contiguous blocks of PRNs form the sequence r_i, and the workload is spread over p processors in such a way that each process analyzes each pth lattice. If we number the processes by their rank i from 0 to $p-1$ and the lattices form 0 to $M-1$, each process starts with a lattice whose number equals its own rank. That means process i has to skip $i \cdot N$ PRNs from the sequence r_i before the first lattice configuration is generated. Thereafter each process can skip $p-1$ lattices, i. e., $(p-1) \cdot N$ PRNs and continue with the next lattice. In section 6.2 we investigate this approach in more detail and will give further examples of fair playing Monte Carlo algorithms and their implementation.

Organizing simulation algorithms such that they play fair is not always as easy as in the above example, but with a little effort one can achieve fair play in more complicated situations, too. This may require the combination of block splitting and the leapfrog method, or iterated leapfrogging. Sometimes it is also necessary to use more than one stream of PRNs per process, e. g. in the Swendsen Wang cluster algorithm [71, 57] one may use one PRNG to construct the bond percolation clusters and another PRNG to decide if a cluster has to be flipped.

2.4 Linear recurrences

The majority of the PRNG algorithms that are implemented by TRNG are based on linear recurrences in prime fields. Thus, we review some of theirs mathematical properties in this section.

2.4.1 Linear congruential generators

Linear recurrences where introduced as PRNGs by Lehmer [42], who proposed the linear congruential generator (LCG) with the recurrence

$$r_i = a \cdot r_{i-1} + b \bmod m\,, \tag{2.5}$$

with $a = 23$, $b = 0$, and $m = 10^8 + 1$. Obviously, the period of such a generator cannot exceed m. If $b = 0$ then period will be at most $m - 1$, because $r_i = 0$ is a fixed point. In fact, the original Lehmer generator has a period of only 5 882 352.

The period of a LCG depends on the choice of its parameter. There are two important kinds of moduli m that allow for a maximal period, namely moduli that are a power of 2 and prime moduli. For prime moduli, a has to be a generating element of the multiplicative group modulo m and $b = 0$. While for power of 2 moduli, a and b must be odd and $a - 1$ has to be a multiple of four. These and more theoretical properties of LCGs are presented in [33]

Parallelization

One may show by complete induction that the M-fold successive iteration of (2.5) is given by

$$r_i = a^M r_{i-M} + b \sum_{j=0}^{M-1} a^j \bmod m\,. \tag{2.6}$$

Note that $\sum_{j=0}^{M-1} a^j$ may be computed efficiently if M is a power of 2, say $M = 2^e$, by employing

$$\sum_{j=0}^{2^e-1} a^j \bmod m = \prod_{j=0}^{e-1} \left(1 + a^{2^j}\right) \bmod m\,. \tag{2.7}$$

If M is not a power of two, we can use the more general relation

$$\sum_{j=0}^{M-1} a^j \bmod m = \prod_{j=0}^{e-1} \left(1 + a^{2^j}\right) + a^{2^e} \sum_{j=0}^{M-2^e-1} a^j \bmod m \tag{2.8}$$

instead, where e denotes the largest integer such that $M \leq 2^e$. The left side as well as the right side of (2.8) include terms of the form $\sum_j a^j \bmod m$, but on the rigth hand side the number of terms in the sum is much smaller. Applying of (2.8) recursively allows an efficient computation of $\sum_{j=0}^{M-1} a^j \bmod m$ and, therefore, an efficient implementation of block splitting and leapfrogging.

2.4.2 Linear feedback shift register sequences

The majority of the PRNG algorithms that are implemented by TRNG are based on so-called linear feedback shift register sequences. Therefore, we review some of theirs mathematical properties in this section. Readers how do not want to bother with mathematical details might skip this as well as the next section on YARN generators and may come back later if necessary.

Knuth [32] proposed a generalization of Lehmer's method known as multiple recurrence generator (MRG) that obeys the recurrence

$$r_i = a_1 r_{i-1} + a_2 r_{i-2} + \ldots + a_n r_{i-n} \bmod m \tag{2.9}$$

with prime modulus m. In the theory of finite fields, a sequence of type (2.9) is called *linear feedback shift register sequence*, or LFSR sequence for short. Note that a LFSR sequence is fully determined by specifying n coefficients $(a_1, a_2, \ldots, a_n)$ plus n initial values $(r_1, r_2, \ldots, r_n)$. There is a wealth of rigorous results on LFSR sequences that can (and should) be used to construct a good PRNG. Here we only discuss a few but important facts without proofs. A detailed presentation of LFSR sequences including theorems and proofs can be found in [22, 29, 43, 44, 20, 74].

Since the all zero tuple $(0, 0, \ldots, 0)$ is a fixed-point of (2.9), the maximum period of a LFSR sequence cannot exceed $m^n - 1$. The following theorem tells us precisely how to choose the coefficients $(a_1, a_2, \ldots, a_n)$ to achieve this period [33]:

Theorem 1 The LFSR sequence (2.9) over $\mathbb{F}_m$ has period $m^n - 1$, if and only if the characteristic polynomial

$$f(x) = x^n - a_1 x^{n-1} - a_2 x^{n-2} - \ldots - a_n \tag{2.10}$$

is *primitive* modulo m.

A monic polynomial $f(x)$ of degree n over $\mathbb{F}_m$ is primitive modulo m, if and only if it is irreducible (i. e., cannot be factorized over $\mathbb{F}_m$), and if it has a primitive element of the extension field $\mathbb{F}_{m^n}$ as one of its roots. The number of primitive polynomials of degree n modulo m is equal to $\phi(m^n - 1)/n = \mathcal{O}\left(m^n/(n\ln(n\ln m))\right)$ [73], where $\phi(x)$ denotes Euler's totient function. As a consequence a random polynomial of degree n is primitive modulo m with probability $\simeq 1/(n\ln(n\ln m))$, and finding primitive polynomials reduces to testing whether a given polynomial is primitive. The latter can be done efficiently, if the factorization of $m^n - 1$ is known [29], and most computer algebra systems offer a procedure for this test.

Theorem 2 Let r_i be an LFSR sequence (2.9) with a primitive characteristic polynomial. Then each k-tuple $(r_{i+1}, \ldots, r_{i+k})$ occurs m^{n-k} times per period for $k \leq n$ (except the all zero tuple for $k = n$).

From this theorem it follows that, if a k-tuple of consecutive numbers with $k \leq n$ is chosen randomly from a LFSR sequence, the outcome is uniformly distributed over all possible k-tuples in $\mathbb{F}_m$. This is exactly what one would expect from a truly random sequence. In terms of Compagner's ensemble theory tuples of size less than or equal to n drawn from a LFSR sequence with primitive characteristic polynomial are indistinguishable from truly random tuples [14, 15].

Theorem 3 Let r_i be an LFSR sequence (2.9) with period $T = m^n - 1$ and let α be a complex mth root of unity and $\overline{\alpha}$ its complex conjugated. Then

$$C(h) := \sum_{i=1}^{T} \alpha^{r_i} \cdot \overline{\alpha}^{r_{i+h}} = \begin{cases} T & \text{if } h = 0 \bmod T \\ -1 & \text{if } h \neq 0 \bmod T \end{cases}. \tag{2.11}$$

$C(h)$ can be interpreted as autocorrelation function of the sequence, and Theorem 3 tells us that LFSR sequences with maximum period have autocorrelations that are very similar to the autocorrelations of a random sequence with period T. Together with the nice equidistribution properties (Theorem 2) this qualifies LFSR sequences with maximum period as *pseudo-noise sequences*, a term originally coined by Golomb for binary sequences [22, 29].

Parallelization

As a matter of fact, LFSR sequences do support leapfrog and block splitting very well. Block splitting means basically jumping ahead in a PRN sequence. In the case of LFSR sequences this can be done quite efficiently. Note, that by introducing a companion matrix $\boldsymbol{A}$, the linear recurrence (2.9) can be written as a vector matrix product.

$$\begin{pmatrix} r_{i-(n-1)} \\ \vdots \\ r_{i-1} \\ r_i \end{pmatrix} = \underbrace{\begin{pmatrix} 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 1 \\ a_n & a_{n-1} & \ldots & a_1 \end{pmatrix}}_{\boldsymbol{A}} \begin{pmatrix} r_{i-n} \\ \vdots \\ r_{i-2} \\ r_{i-1} \end{pmatrix} \bmod m \tag{2.12}$$

From this formula it follows immediately that the M-fold successive iteration of (2.9) may be written as

$$\begin{pmatrix} r_{i-(n-1)} \\ \vdots \\ r_{i-1} \\ r_i \end{pmatrix} = \boldsymbol{A}^M \begin{pmatrix} r_{i-M-(n-1)} \\ \vdots \\ r_{i-M-1} \\ r_{i-M} \end{pmatrix} \bmod m\,. \tag{2.13}$$

Matrix exponentiation can be accomplished in $\mathcal{O}\left(n^3 \ln M\right)$ steps via binary exponentiation (also known as exponentiation by squaring).

Implementing leapfrogging efficiently is less straightforward. Calculating $t_{j,i} = r_{pi+j}$ via

$$\begin{pmatrix} r_{pi+j-(n-1)} \\ \vdots \\ r_{pi+j-1} \\ r_{pi+j} \end{pmatrix} = \boldsymbol{A}^p \begin{pmatrix} r_{p(i-1)+j-(n-1)} \\ \vdots \\ r_{p(i-1)+j-1} \\ r_{p(i-1)+j} \end{pmatrix} \bmod m \tag{2.14}$$

is no option, because $\boldsymbol{A}^p$ is usually a dense matrix, in which case calculating a new element from the leapfrog sequence requires $\mathcal{O}\left(n^2\right)$ operations instead of $\mathcal{O}\left(n\right)$ operations in the base sequence.

The following theorem assures that the leapfrog subsequences of LFSR sequences are again LFSR sequences [29]. This will provide us with a very efficient way to generate leapfrog sequences.

Theorem 4 Let r_i be a LFSR sequence based on a primitive polynomial of degree n with period $m^n - 1$ (pseudo-noise sequence) over $\mathbb{F}_m$, and let (t) be the decimated sequence with lag $p > 0$ and offset j, e. g.

$$t_{j,i} = r_{pi+j}\,. \tag{2.15}$$

Then $t_{j,i}$ is a LFSR sequence based on a primitive polynomial of degree n, too, if and only if p and $m^n - 1$ are coprime, e. g. $\gcd(m^n - 1, p) = 1$. In addition, r_i and $t_{j,i}$ are not just cyclic shifts of each other, except when

$$p = m^h \bmod (m^n - 1) \tag{2.16}$$

for some $0 \le h < n$. If $\gcd(m^n - 1, p) > 1$ the sequence $t_{j,i}$ is still a LFSR sequence, but not a pseudo-noise sequence.

It is not hard to find prime numbers m such that $m^n - 1$ has very few (and large) prime factors. For such numbers, the leapfrog method yields pseudo-noise sequences for any reasonable number of parallel streams [6]. While Theorem 4 ensures that leapfrog sequences are not just cyclic shifts of the base sequence (unless (2.16) holds), the leapfrog sequences are cyclic shifts of each other, see section 2.2.

Theorem 4 tells us that all leapfrog sequences of a LFSR sequence of degree n can be generated by another LFSR of degree n or less. The following theorem gives us a recipe to calculate the coefficients $(b_1, b_2, \ldots, b_n)$ of the corresponding leapfrog feedback polynomial.

Theorem 5 Let t_i be a (periodic) LFSR sequence over the field $\mathbb{F}_m$ and $f(x)$ its characteristic polynomial of degree n. Then the coefficients $(b_1, b_2, \ldots, b_n)$ of $f(x)$ can be computed from $2n$

successive elements of t_i by solving the linear system

$$\begin{pmatrix} t_{i+n} \\ t_{i+n+1} \\ \vdots \\ t_{i+2n-1} \end{pmatrix} = \begin{pmatrix} t_{i+n-1} & \dots & t_{i+1} & t_i \\ t_{i+n} & \dots & t_{i+2} & t_{i+1} \\ \vdots & \ddots & \vdots & \vdots \\ t_{i+2n-2} & \dots & t_{i+n} & t_{i+n-1} \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} \bmod m \tag{2.17}$$

over $\mathbb{F}_m$.

Starting from the base sequence we determine $2n$ values of the sequence t_i by applying the leapfrog rule. Then we solve (2.17) by Gaussian elimination to get the characteristic polynomial for a new LFSR generator that yields the elements of the leapfrog sequence directly with each call. If the matrix in (2.17) is singular, the linear system has more than one solution, and it is sufficient to pick one of them. In this case it is always possible to generate the leapfrog sequence by a LFSR of degree less than the degree of the original sequence.

Choice of modulus

LFSR sequences can be defined over any prime field. In particular LFSR sequences over $\mathbb{F}_2$ with sparse feedback polynomials are popular sources of PRNs [31, 75, 33] and generators of this type can be found in various software libraries. This is due to the fact that multiplication in $\mathbb{F}_2$ is trivial, addition reduces to exclusive-or and the modulo operation comes for free. As a result, generators that operate in $\mathbb{F}_2$ are extremely fast. Unfortunately, these generators suffer from serious statistical defects [19, 23, 68, 75] that can be blamed to the small size of the underlying field [4]. In parallel applications we have the additional drawback that, if the leapfrog method is applied to a LFSR sequence with sparse characteristic polynomial, the new sequence will have a dense polynomial. The computational complexity of generating values of the LFSR sequence grows from $\mathcal{O}(1)$ to $\mathcal{O}(n)$. Remember that for generators in $\mathbb{F}_2$, n is typically of order 1000 or even larger to get a long period $2^n - 1$ and reasonable statistical properties.

The theorems and parallelization techniques we have presented so far do apply to LFSR sequences over any finite field $\mathbb{F}_m$. Therefore we are free to choose the prime modulus m. In order to get maximum entropy on the macrostate level [54] m should be as large as possible. A good choice is to set m to a value that is of the order of the largest representable integer of the computer. If the computer deals with e-bit registers, we may write the modulus as $m = 2^e - k$, with k reasonably small. In fact if $k(k+2) \leq m$ modular reduction can be done reasonably fast by a few bit-shifts, additions and multiplications, see chapter 7. Furthermore a large modulus allows us to restrict the degree of the LFSR to rather small values, e. g. $n \approx 4$, while the PRNG has a large period and good statistical properties.

In accordance with Theorem 4, a leapfrog sequence of a pseudo-noise sequence is a pseudo-noise sequence, too, if and only if its period $m^n - 1$ and the lag p are coprime. For that reason $m^n - 1$ should have a small number of prime factors. It can be shown that $m^n - 1$ has at least three prime factors and if the number of prime factors does not exceed three, then m is necessarily a Sophie-Germain Prime and n a prime larger than two [6].

To sum up, the modulus m of a LFSR sequence should be a Sophie-Germain Prime, such that $m^n - 1$ has not more than three prime factors and such that $m = 2^e - k$ and $k(k+2) \leq m$ for some integers e and k.

2.4.3 Matrix linear congruential generators

It has been shown before that multiple recurrence generators can be written as a matrix equation with a companion matrix. Matrix linear congruential generators are based on generalized recurrence of the form [24, 16]

$$\boldsymbol{r}_i = \boldsymbol{A}\boldsymbol{r}_{i-1} \mod m\,, \tag{2.18}$$

where m is a prime number, $\boldsymbol{r}_i$ denotes vector of n elements and $\boldsymbol{A}$ represents an $n \times n$ invertible matrix over the filed $\mathbb{F}_m$. The elements of $\boldsymbol{r}_i$ and $\boldsymbol{A}$ are integers $\in 0, 1, \ldots, m-1$, i. e., the elements of the field $\mathbb{F}_m$. The state $\boldsymbol{r}_i$ of such a generator can take m^n different values. The state $\boldsymbol{r}_i = (0, 0, \ldots, 0)$ is a fixed point of the recurrence (2.18). Therefore, the period of a matrix linear congruential generator cannot exceed $m^n - 1$. This maximal period is attained if the matrix $\boldsymbol{A}$ is chosen appropriately, i. e., the matrix $\boldsymbol{A}$ is such that its rank equals $m^n - 1$.

Typical parameters that are employed for matrix linear congruential generators are $m = 2$ or m equal to a large prime that is close to the largest integer that can be represented by a machine register. The parameter n must be relatively large, e.g., $n \geq 64$, in the former case to reach a sufficent period, whereas in the latter case $n = 2$ or $n = 3$ may be sufficient depending on the size of m. The matrix $\boldsymbol{A}$ is often designed to allow an efficinet implementation of the matrix-vector multiplication $\boldsymbol{A}\boldsymbol{r}_{i-1} \mod m$ while ensuring that the generator reaches the maximal period.

The parallelization of matrix linear congruential generators via block splitting and leapfrogging is straight forward. The M-fold successive iteration of (2.18) is given by

$$\boldsymbol{r}_i = \boldsymbol{A}^M \boldsymbol{r}_{i-M} \bmod m\,. \tag{2.19}$$

Block splitting can be directly implemented by the application of (2.19). Leapfrogging can be realized by replacing the matrix $\boldsymbol{A}$ by $\boldsymbol{A}^p$, where p denotes the number of independent streams. It should be noted, however, that if $\boldsymbol{A}$ has been chosen to be sparse to allow an efficient implementation of the matrix-vector product $\boldsymbol{A}\boldsymbol{r}_{i-1}$ then $\boldsymbol{A}^p$ is no longer sparse, which may render leapfrogging impractical.

2.5 Non-linear transformations and YARN sequences

LFSR sequences over prime fields with a large prime modulus seem to be ideally suited as PRNGs. They have, however, a well known weakness. When used to sample coordinates in d-dimensional space, pseudo-noise sequences cover every point for $d < n$, and every point except $(0, 0, \ldots, 0)$ for $d = n$. For $d > n$ the set of positions generated is obviously sparse, and the linearity of the production rule (2.9) leads to the concentration of the sampling points on n-dimensional hyper-planes [25, 37], see also Figure 2.3. This phenomenon, first noticed by Marsaglia in 1968 [45], constitutes one of the well known tests of randomness applied to PRNGs, the so-called spectral test [33]. The spectral test checks the behavior of a generator when its outputs are used to form d-tuples. Closely related to this mechanism are the observed correlations in other empirical tests like the birthday spacings test and the collision test [39, 41]. Non-linear generators do quite well in all these tests, but compared to LFSR sequences they have much less nice and *provable* properties and they are not suited for fair playing parallelization.

To get the best of both worlds we propose a delinearization that preserves all the nice properties of linear pseudo-noise sequences. That means each element of a linear pseudo-noise

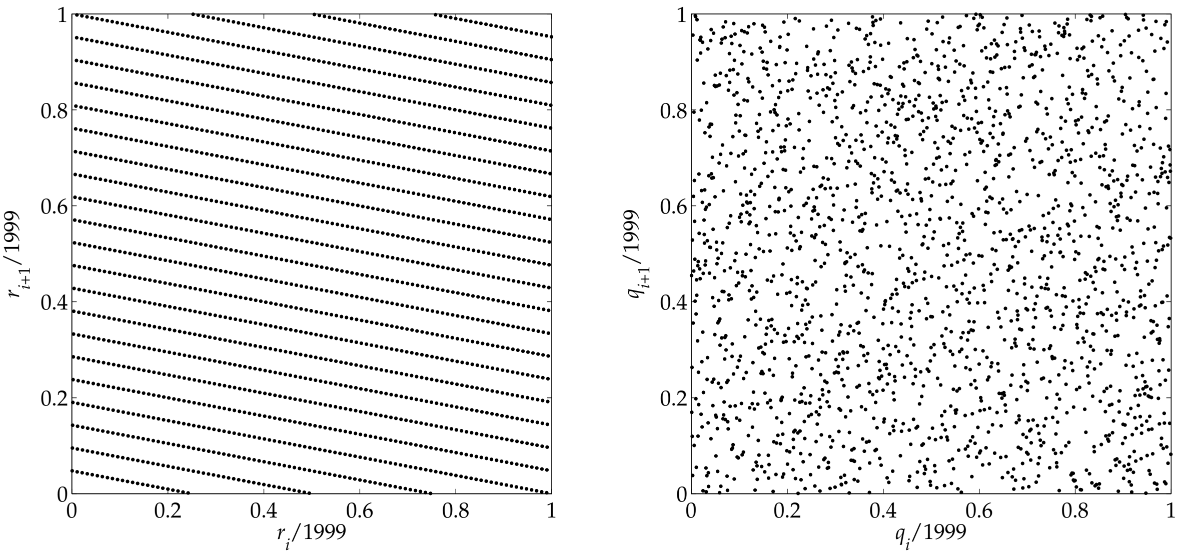


Figure 2.3: Exponentiation of a generating element in a prime field is an effective way to destroy the linear structures of LFSR sequences. Both pictures show the full period of the generator. Left: $r_i = 95 \cdot r_{i-i} \bmod 1999$. Right: $q_i = 1099^{r_i} \bmod 1999$ with $r_i = 95 \cdot r_{i-i} \bmod 1999$.

sequence $q_i \in \mathbb{F}_m$ is transformed to another element in $\mathbb{F}_m$ by a non-linear bijective mapping. If m is prime, such a bijective mapping is given by an exponentiation.

Theorem 6 Let r_i be a pseudo-noise sequence in $\mathbb{F}_m$, and let g be a generating element of the multiplicative group $\mathbb{F}_m^*$. Then the sequence q_i with

$$q_i = \begin{cases} g^{r_i} \bmod m & \text{if } r_i > 0 \\ 0 & \text{if } r_i = 0 \end{cases} \tag{2.20}$$

is a pseudo-noise sequence, too.

The proof of this theorem is trivial: since g is a generator of $\mathbb{F}_m^*$, the map (2.20) is bijective. We call delinearized generators based on Theorem 6 YARN generators (yet another random number).

The linearity is completely destroyed by the map (2.20), see Figure 2.3. Let $L_{(r)}(l)$ denote the linear complexity of the subsequence $(r_1, r_2, \ldots, r_l)$. This function is known as the linear complexity profile of r_i. For a truly random sequence it grows on average like $l/2$. Figure 2.4 shows the linear complexity profile $L_{(r)}(l)$ of a typical YARN sequence. It shows the same growth rate as a truly random sequence up to the point where more than 99 % of the period have been considered. Sharing the linear complexity profile with a truly random sequence, we may say that the YARN generator is as non-linear as it can get.

The non-linear transform by exponentiation in Theorem 6 has to be carried out in a prime field $\mathbb{F}_m$. If the underlying generator produces integers in some range $[0, m)$, where m is not prime (i. e. a power of two), another kind of non-linear transformation has to be applied to improve the underlying generator. For $m = 2^e$ Press et al. [63] suggest to transform the output

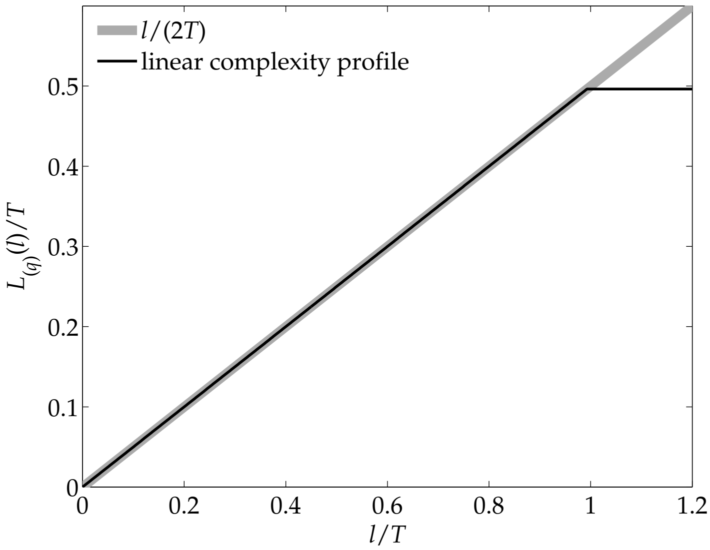
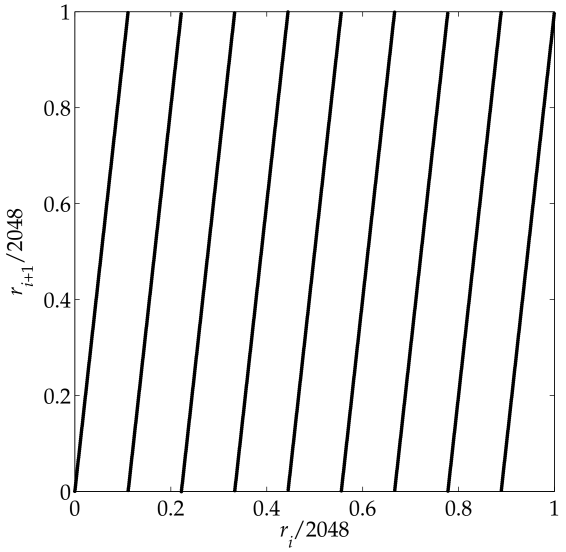


Figure 2.4: Linear complexity profile $L_{(q)}(l)$ of a YARN sequence, produced by the recurrence $r_i = 173 \cdot r_{i-1} + 219 \cdot r_{i-2} \bmod 317$ and $q_i = 151^{r_i} \bmod 317$. The period of this sequence equals $T = 317^2 - 1$.

r_i of a base generator by

$$\begin{aligned} t_{i,0} &= r_i \\ t_{i,1} &= t_{i,0} \oplus (t_{i,0} \gg a_0) \\ t_{i,2} &= t_{i,1} \oplus (t_{i,1} \ll a_1) \\ t_{i,3} &= t_{i,2} \oplus (t_{i,2} \gg a_2) \\ q_i &= t_{i,3} \end{aligned} \tag{2.21}$$

where $\oplus$ denotes binary addition (exclusive-or), $x \gg n$ bit-shift of x to the right of size n and $x \ll n$ bit-shift of x to the left of size n, respectively. The parameters a_0, a_1 and a_2 have to be chosen suitable to make (2.21) a bijective mapping from r_i to q_i, see [63]. Figure 2.5 shows how the mapping (2.21) efficiently destroys the lattice structures of linear congruential generators modulo a power of two.



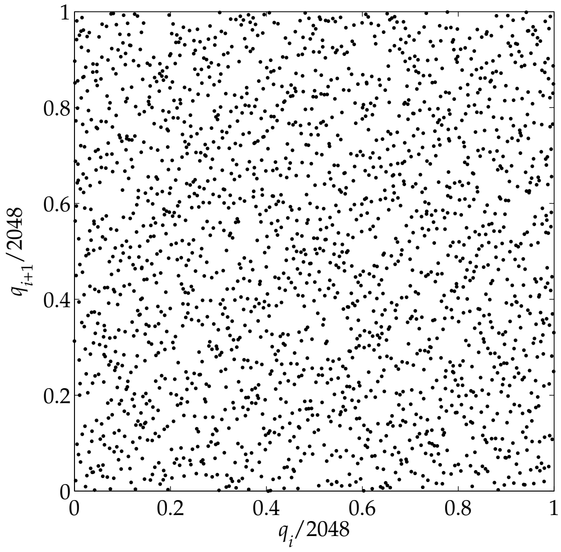


Figure 2.5: The non-linear mapping (2.21) destroys the lattice structures of linear congruential generators. Both pictures show the full period of the generator. Left: $r_i = 9 \cdot r_{i-i} + 1 \bmod 2048$. Right: q_i given by (2.21) with $a_0 = 5$, $a_1 = 9$, $a_2 = 2$ and $r_i = 9 \cdot r_{i-i} + 1 \bmod 2048$.

3 Basic concepts

The TRNG library consists of a loose bunch of classes. These classes can be divided into two kinds of classes, *random number engines* and *random number distributions*.

Random number engines are the workhorses of TRNG. Each random number engine implements some algorithm that is used to produce pseudo-random numbers. The notion of a random number engine as it is used by TRNG was introduced by [11] and it is a very general concept. For example, the random number engine concept does not specify what kind of pseudo-random numbers (integers, floating point numbers or just bits) are generated. All random number engine classes of TRNG implement the concept of a random number engine that has been introduced in [11] and that was later included in the C++11 language standard [27]. As a library of *parallel* random number generators, however, TRNG extends the notion of a random number engine to a *parallel random number engine*. To fulfill the requirements of a parallel random number engine, a class has to fulfill all the requirements of a random number engine and in addition some further requirements that make them applicable for parallel Monte Carlo simulations. The random number engine concept and the parallel random number engine concept will be discussed in detail in section 3.1.

A random number engine is not very useful by itself. To write some real world Monte Carlo applications we need random number distribution classes, too. A random number distribution class converts the output of an arbitrary random number engine into a pseudo-random number drawn from some specific distribution. The general concept of a random number distribution is discussed in section 3.2.

Note that the design of TRNG was initially based on a proposal for the 2011 revision of the C++ language standard [11]. This proposal has eventually become part of the C++ 11 language standard with some minor modifications. This language standard (as well its successors C++ 13 and C++ 17) is supported by all mayor C++ compilers now. TRNG version 4.22 and later versions follow the conventions of the random number generator facility of the C++ standard library, no longer supporting the original proposal [11]. This means, TRNG requires a compiler that supports C++ 11 (or any later language standard) and TRNG classes can be used in combination with classes of the random number generator facility of the C++ standard library.

3.1 Random number engines

To be a random number engine, a class has to fulfill a set of requirements that we will summarize as follows, see [11] for details. A class `X` satisfies the requirements of a random number engine, if the expressions as shown in Table 3.1 are valid and have the indicated semantics. In that table and throughout this section,

- `T` is the type named by `X`'s associated `result_type`;
- `t` is a value of `T`;
- `u` is a value of `X`, `v` is an lvalue of `X`, `x` and `y` are (possibly const) values of `X`;

- s is a value of integral type;
- g is an lvalue, of a type other than X, that defines a zero-argument function object returning values of type `unsigned long`;
- os is an lvalue of the type of some class template specialization `std::basic_ostream <charT, traits>`; and
- is is an lvalue of the type of some class template specialization `std::basic_istream <charT, traits>`.

Table 3.1: Random number engine requirements.

expression	return type	pre/post-condition	complexity
`X::result_type`	`T`	`T` is an arithmetic type other than `bool`.	compile-time
`u()`	`T`	Sets the state to $u_{i+1} = \mathsf{TA}(u_i)$ and returns $\mathsf{GA}(u_i)$. If `X` is integral, returns a value in the closed interval [`X::min()`,`X::max()`]; otherwise, returns a value in the open interval $(0, 1)$.	amortized constant
`u.discard(s)`	`void`	pre: `s` is of type `unsigned long long`. post: Internal state of the random number engine is changed in such a way that the engine jumps s steps ahead.	$\mathcal{O}$ (d) or less
`X::min()`	`T`, if `X` is integral; otherwise `int`.	If `X` is integral, denotes the least value potentially returned by `operator()`; otherwise denotes 0.	compile-time
`X::max()`	`T`, if `X` is integral; otherwise `int`.	If `X` is integral, denotes the greatest value potentially returned by `operator()`; otherwise denotes 1.	compile-time
`X()`		Creates an engine with the same initial state as all other default-constructed engines of type `X`.	$\mathcal{O}$ (size of state)
`X(s)`		Creates an engine with initial state determined by `static_cast<unsigned long>(s)`.	$\mathcal{O}$ (size of state)
`X(g)`		Creates an engine with initial state determined by the results of successive invocations of g. Throws what and when g throws.	$\mathcal{O}$ (size of state)
`u.seed()`	`void`	post: `u==X()`	same as `X()`
`u.seed(s)`	`void`	post: `u==X(s)`	same as `X(s)`
`u.seed(g)`	`void`	post: If g does not throw, u==v, where the state of v is as if constructed by `X(g)`. Otherwise, the exception is re-thrown and the engine s state is deemed invalid. Thereafter, further use of u is undefined except for destruction or invoking a function that establishes a valid state.	same as `X(g)`
`x==y`	`bool`	With S_x and S_y as the infinite sequences of values that would be generated by repeated calls to `x()` and `y()`, respectively, returns `true` if $S_x = S_y$; returns `false` otherwise.	$\mathcal{O}$ (size of state)
`x!=y`	`bool`	`!(x==y)`	$\mathcal{O}$ (size of state)

Table 3.1: Random number engine requirements continued.

expression	return type	pre/post-condition	complexity
`os << x`	reference to the type of `os`	With `os`.*fmtflags* set to `std::ios_base::dec|std::ios_base::fixed|std::ios_base::left` and the fill character set to the space character, writes to `os` the textual representation of `x`'s current state. In the output, adjacent numbers are separated by one or more space characters. post: The `os`.*fmtflags* and fill character are unchanged.	$\mathcal{O}$ (size of state)
`is >> v`	reference to the type of `is`	Sets `v`'s state as determined by reading its textual representation from `is`. If bad input is encountered, ensures that `v`'s state is unchanged by the operation and calls `is.setstate(std::ios::failbit)` (which may throw `std::ios::failure`). pre: The textual representation was previously written using an `os` whose imbued locale and whose type's template specialization arguments `charT` and `traits` were the same as those of `is`. post: The `is`.*fmtflags* are unchanged.	$\mathcal{O}$ (size of state)

Table 3.2: Parallel random number engine requirements.

expression	return type	pre/post-condition	complexity
`split(p, s)`	`void`	pre: `s` and `p` are of type `unsigned int` with $\mathtt{s} < \mathtt{p}$. If $\mathtt{s} \geq \mathtt{p}$ an exception `std::invalid_argument` is thrown. post: Internal parameters of the random number engine are changed in such a way that future calls to `operator()` will generate the `s`th sub-stream of `p` sub-streams. Sub-streams are numbered from 0 to $\mathtt{p} - 1$. The complexity of `operator()` will not change.	polynomial in size of state, (at most) linear in `p` and `s`
`jump2(s)`	`void`	pre: `s` is of type `unsigned int`. post: Internal state of the random number engine is changed in such a way that the engine jumps $2^{\mathtt{s}}$ steps ahead.	polynomial in size of state and `s`
`jump(s)`	`void`	pre: `s` is of type `unsigned long long`. post: Internal state of the random number engine is changed in such a way that the engine jumps `s` steps ahead.	polynomial in size of state and the logarithm of `s`

A random number engine object x has at any given time a state x_i for some integer $i \geq 0$. Upon construction, a random number engine x has an initial state x_0. The state of an engine may be established by invoking its constructor, `seed` member function, `operator=`, or a suitable `operator>>`.

The specification of each random number engine defines the size of its state in multiples of the size of its `result_type`, given as an integral constant expression. The specification of each random number engine also defines

- the *transition algorithm* TA by which the state x_i of an engine is advanced to its *successor state* x_{i+1}, and
- the *generation algorithm* GA by which the state of an engine is mapped to a value of type `result_type`.

Furthermore, a random number engine shall fulfill the requirements of the concepts "Copy-Constructible" and of "Assignable". That means roughly, random number engines support copy and assignment operations with the same semantic like build-in types as `int` or `double`. Copy construction and assignment shall each be of complexity $\mathcal{O}$ (size of state).

Random number engine requirements had been adopted from [11]. For parallel Monte Carlo applications we extend the concept of a random number engine to a parallel random number engine. Such an engine has to meet all the requirements of a parallel random number engine and additionally the requirements shown in Table 3.2. A parallel random number engine provides block splitting and leapfrog. It is demanded that leapfrog is implemented in such a way that the complexity of `operator()` will not depend on how many sub-streams a stream has been split into. That means, a valid implementation of leapfrog will not just calculate all random numbers of a stream and then throw away bunches of numbers to derive the random numbers of a leapfrog sub-stream. This rather strong requirement restricts the number of pseudo-random number generator algorithms that are proper for parallel random number engines. But LFSR sequences and YARN generators, which had been discussed in sections 2.4.2 and 6, meet these conditions easily. Note that the methods `discard` and `jump` have the same effect but `jump` has tighter time-complexity requirements.

3.2 Random number distributions

To model the concept of a random number distribution a class has to fulfill a set of requirements that we will summarize as follows, refer to [11] for details. A class `X` satisfies the requirements of a random number distribution if the expressions shown in Table 3.3 are valid and have the indicated semantics, and if `X` and its associated types also satisfies all other requirements of this section. In that table and throughout this section,

- `T` is the type named by `X`'s associated `result_type`;
- `P` is the type named by `X`'s associated `param_type`;
- `u` is a value of `X` and `x` is a (possibly const) value of `X`;
- `glb` and `lub` are values of `T` respectively corresponding to the greatest lower bound and the least upper bound on the values potentially returned by `u`'s `operator()`, as determined by the current values of `u`'s parameters;
- `p` is a value of `P`;
- `e` is an lvalue of an arbitrary type that satisfies the requirements of a uniform random number generator;

- `os` is an lvalue of the type of some class template specialization `basic_ostream<charT, traits>`; and
- `is` is an lvalue of the type of some class template specialization `basic_istream<charT, traits>`.

The specification of each random number distribution identifies an associated mathematical *probability density function* $p(z)$ or an associated discrete *probability function* $P(z_i)$. Such functions are typically expressed using certain externally supplied quantities known as the *parameters of the distribution*. Such distribution parameters are identified in this context by writing, for example, $p(z|a,b)$ or $P(z_i|a,b)$, to name specific parameters, or by writing, for example, $p(z|\{p\})$ or $P(z_i|\{p\})$, to denote the parameters `p` of a distribution taken as a whole.

Furthermore a random number distribution shall fulfill the requirements of the concepts "CopyConstructible" and of "Assignable". That means roughly, random number distributions support copy and assignment operations with the same semantic like build-in types like `int` or `double`. Copy construction and assignment shall each be of complexity $\mathcal{O}$ (size of state).

For each of the constructors of `X` taking arguments corresponding to parameters of the distribution, `P` shall have a corresponding constructor subject to the same requirements and taking arguments identical in number, type, and default values. Moreover, for each of the member functions of `X` that return values corresponding to parameters of the distribution, `P` shall have a corresponding member function with the identical name, type, and semantics.

Table 3.3: Random number distribution requirements.

expression	return type	pre/post-condition	complexity
`X::result_type`	`T`	`T` is an arithmetic type.	compile-time
`X::param_type`	`P`		compile-time
`X(p)`		Creates a distribution whose behavior is indistinguishable from that of a distribution newly constructed directly from the values used to construct `p`.	same as `p`'s construction
`u.reset()`	`void`	Subsequent uses of `u` do not depend on values produced by `e` prior to invoking `reset`.	constant
`x.param()`	`P`	Returns a value `p` such that `X(p).param()==p`.	no worse than the complexity of `X(p)`
`u.param(p)`	`void`	post: `u.param() == p`.	no worse than the complexity of `X(p)`
`u(e)`	`T`	With `p=u.param()`, the sequence of numbers returned by successive invocations with the same object `e` is randomly distributed according to the associated $p(z\|\{p\})$ or $P(z_i\|\{p\})$ function.	amortized constant number of invocations of `e`
`u(e,p)`	`T`	The sequence of numbers returned by successive invocations with the same objects `e` and `p` is randomly distributed according to the associated $p(z\|\{p\})$ or $P(z_i\|\{p\})$ function	
`x.min()`	`T`	Returns `glb`.	constant
`x.max()`	`T`	Returns `lub`.	constant
`os << x`	reference to the type of `os`	Writes to os a textual representation for the parameters and the additional internal data of x. post: The `os`.*fmtflags* and fill character are unchanged.	
`is >> u`	reference to the type of `is`	Restores from is the parameters and additional internal data of u. If bad input is encountered, ensures that u's state is unchanged by the operation and calls `is.setstate(ios::failbit)` (which may throw `std::ios::failure`). pre: `is` provides a textual representation that was previously written using an os whose imbued locale and whose type's template specialization arguments `charT` and `traits` were the same as those of `is`. post: The `is`.*fmtflags* are unchanged.	

4 TRNG classes

In chapter 3 the abstract concepts of (parallel) random number engines and random number distributions had been introduced. Now we look at some actual realizations of these concepts. TRNG provides several (parallel) random number engines and random number distributions. Each engine and each distribution is implemented by its own class that resides in the name space `trng`.

4.1 Random number engines

In this section we give a detailed documentation of all random number engines. Each subsection describes the public interface of one random number engine and focuses on aspects that are specific for a particular random number engine. This includes extensions to the random number engine interface as well as algorithmic details. The part of the public interface, that is mandatory for each (parallel) random number engine, will not be discussed in detail. Read section 3.1 instead. Table 4.1 gives an overview over all random number engines of TRNG.

All classes that will be describe in this section model either a random number engine or a parallel random number engine and therefore fulfill the requirements introduced in section 3.1. But for convenience their interface provides even more. For example all random number engines model a *random number generator* as well. The notion of a random number generator had been introduced by the C++ Standard Template Library. A random number generator is a class that provides an `operator()(long)` that returns a uniformly distributed random integer larger than or equal to zero but less than its argument. That makes TRNG (parallel) random number engines applicable to the STL algorithm `std::random_shuffle`. Additionally TRNG (parallel) random number engines provide a function `name()` that returns a string with the name of the random number engine.

4.1.1 Linear congruential generators and variants

The classes `trng::lcg64` and `trng::lcg64_shift` implement linear congruential generators. Both generators are based on the transition algorithm [42, 33]

$$r_{i+1} = a \cdot r_i + b \bmod 2^{64} \,.$$

The state of this generator at time i is given by r_i. Its period equals 2^{64} if and only if b is odd and $a \bmod 4 = 1$ [33]. The statistical properties of linear congruential generators depend crucial on the choice of the multiplier a, which has to be chosen carefully.

This linear congruential generator `trng::lcg64` is the quick and dirty generator of TRNG. It's dammed fast, see section 7, but even for proper chosen parameters a and b the lower bits of r_i are less random than the higher order bits. The class `trng::lcg64` should be avoided whenever the randomness of lower bits have a significant impact to the simulation. In [36] L'Ecuyer warns about multiplicative linear congruential generators (in the following quotation denoted as MLCG) with $r_{i+1} = a \cdot r_i \bmod m$:

Table 4.1: Random number engines of TRNG.

random number engine	description	concept
`trng::lcg64`	linear congruential generator with modulus 2^{64}	parallel random number engine
`trng::lcg64_shift`	linear congruential generator with modulus 2^{64} with additional bit-shift transformation	parallel random number engine
`trng::mrg`*n*	multiple recurrence generator based on a linear feedback shift register sequence over $\mathbb{F}_{2^{31}-1}$ of depth n	parallel random number engine
`trng::mrg`*n*`s`	multiple recurrence generator based on a linear feedback shift register sequence over $\mathbb{F}_m$ of depth n, with m being a Sophie-Germain Prime	parallel random number engine
`trng::yarn`*n*	YARN sequence based on a linear feedback shift register sequence over $\mathbb{F}_{2^{31}-1}$ of depth n	parallel random number engine
`trng::yarn`*n*`s`	YARN sequence based on a linear feedback shift register sequence over $\mathbb{F}_m$ of depth n, with m being a Sophie-Germain Prime	parallel random number engine
`trng::lagfib`*n*`xor`	lagged Fibonacci generator with n feedback taps and exclusive-or operation	random number engine
`trng::lagfib`*n*`plus`	lagged Fibonacci generator with n feedback taps and addition	random number engine
`trng::xoshiro256plus`	xoshiro (xor/shift/rotate)	random number engine
`trng::mt19937`	Mersenne twister generating 32 random bits	random number engine
`trng::mt19937_64`	Mersenne twister generating 64 random bits	random number engine

> "If $m = 2^e$ where e is the number of bits on the computer word, and if one can use unsigned integers without overflow checking, the products modulo m are easy to compute: just discard the overflow. This is quick and simple. For that reason, MLCGs with moduli of this form are used abundantly in practice, despite their serious drawbacks. Some nuclear physicists, for instance, perform simulations that use billions of random numbers on supercomputers and are quite reluctant to give up using them [...]. Usually, they also generate many substreams in parallel. In view of the above remarks, all this appears dangerous. Perhaps some people like playing with fire."

The same warning applies if $b \neq 0$. In spite of its weakness this generator is well suited for a large classes of generic Monte Carlo schemes, e. g. simulating a (biased) coin or cluster Monte Carlo [19].

But in some kinds of simulations linear congruential generators reveal their weakness, i. e. their lattice structure, see left part of Figure 2.5. There are two general approaches to improve linear congruential generators: output transformation and combination with other generators. Both approaches are employed in the classes `trng::lcg64_shift` and `trng::lcg64_count_shift`. Both classes are based on the linear recursion

$$r_{i+1} = a \cdot r_i + b \bmod 2^{64}\,.$$

The class `trng::lcg64_shift` destroys the lattice structure of r_i by the non-linear output transformation

$$\begin{aligned} t_{i,0} &= r_i \\ t_{i,1} &= t_{i,0} \oplus (t_{i,0} \gg 17) \\ t_{i,2} &= t_{i,1} \oplus (t_{i,1} \ll 31) \\ t_{i,3} &= t_{i,2} \oplus (t_{i,2} \gg 8) \\ q_i &= t_{i,3} \end{aligned}$$

that yields the pseudo-random number q_i from r_i. Here, $\oplus$ denotes binary addition (exclusive-or), $x \gg n$ bit-shift of x to the right of size n and $x \ll n$ bit-shift of x to the left of size n, respectively. Class `trng::lcg64_shift` is only slightly slower than `trng::lcg64` but the statistical quality is considerably increased by the non-linear transformation.

The class `trng::lcg64_count_shift` combines two linear congruences to construct a combined generator with a period that is larger than the periods of the two underlying generators. More precisely, it is based on the two recurrences

$$\begin{aligned} r_{i+1} &= a \cdot r_i + b \bmod 2^{64}\,, \\ r'_{i+1} &= r'_i + c \bmod 2^{61} - 1\,, \end{aligned}$$

with $c = 1\,425\,089\,352\,415\,399\,810$. The output transform for this generator is defined as

$$\begin{aligned} t_{i,0} &= r_i + r'_i \bmod 2^{64} \\ t_{i,1} &= t_{i,0} \oplus (t_{i,0} \gg 17) \\ t_{i,2} &= t_{i,1} \oplus (t_{i,1} \ll 31) \\ t_{i,3} &= t_{i,2} \oplus (t_{i,2} \gg 8) \\ q_i &= t_{i,3}\,. \end{aligned}$$

The modulus of the second recurrence $2^{61} - 1$ is a Mersenne prime. Thus, both moduli are coprime and the period of the combined generator is the product $2^{64}(2^{61} - 1) \approx 2^{125}$. The sequence r'_i is a counting sequence with non-unit increment, which is trivial to parallelize via block splitting and leap frogging. It is, however, a rather poor pseudo-random number sequence. In combination with the other linear congruence for r_i it merely serves to yield a large period of the combined generator and due to the output transform the statistical properties of the combined generator are much better than those of the base sequences r_i and r'_i.

The class `trng::lcg64` is declared in the header file `trng/lcg64.hpp` and its public interface is given as follows:

```
namespace trng {

  class lcg64 {
  public:
```

First the necessary type, static class constants, and the call operator are declared.

```
    typedef unsigned long long result_type;
    result_type operator()();
    static constexpr result_type min();
    static constexpr result_type max();
```

We also define some parameter and status classes that will be used internally and by the constructor.

```
    class parameter_type;
    class status_type;
```

TRNG provides four parameter sets for a and b, which are chosen to give good statistical properties. Three of these are taken from [38], the default parameter set had been found by the author of the TRNG library.

$$a = 18\,145\,460\,002\,477\,866\,997\,, \quad b = 1$$

```
    static const parameter_type Default;
```

$$a = 2\,862\,933\,555\,777\,941\,757\,, \quad b = 1$$

```
    static const parameter_type LEcuyer1;
```

$$a = 3\,202\,034\,522\,624\,059\,733\,, \quad b = 1$$

```
    static const parameter_type LEcuyer2;
```

$$a = 3\,935\,559\,000\,370\,003\,845\,, \quad b = 1$$

```
    static const parameter_type LEcuyer3;
```

An instance of class `trng::lcg64` can be instantiated by various constructors as specified for a random number engine. Additionally a non-default parameter set may be given.

```
    explicit lcg64(parameter_type=Default);
    explicit lcg64(unsigned long, parameter_type=Default);
    template<typename gen>
    explicit lcg64(gen &, parameter_type P=Default);
```

The class `trng::lcg64` provides all necessary seeding functions (see Table 3.1) and an additional function that sets r_i.

```
    void seed();
    void seed(unsigned long);
    template<typename gen>
    void seed(gen &);
    void seed(unsigned long long);
```

The following three methods are necessary for a parallel random number engine.

```
    void split(unsigned int, unsigned int);
    void jump2(unsigned int);
    void jump(unsigned long long);
    void discard(unsigned long long);
```

Furthermore, the class `trng::lcg64` provides a function that returns the string `lcg64` and an operator `operator()`.

```
    static const char * name();
    long operator()(long);
  };
```

Random number engines are comparable and can be written to or read from a stream.

```
  bool operator==(const lcg64 &, const lcg64 &);
  bool operator!=(const lcg64 &, const lcg64 &);
  template<typename char_t, typename traits_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const lcg64 &);
  template<typename char_t, typename traits_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, lcg64 &);
}
```

The class `trng::lcg64_shift` provides the same public interface as `trng::lcg64`.

```
namespace trng {

  class lcg64_shift {
  public:
    using result_type = unsigned long long;
    result_type operator()();
    static constexpr result_type min();
    static constexpr result_type max();
    class parameter_type;
    class status_type;
    static const parameter_type Default;
    static const parameter_type LEcuyer1;
    static const parameter_type LEcuyer2;
    static const parameter_type LEcuyer3;
    explicit lcg64_shift(parameter_type=Default);
    explicit lcg64_shift(unsigned long, parameter_type=Default);
    template<typename gen>
    explicit lcg64_shift(gen &, parameter_type P=Default);
    void seed();
    void seed(unsigned long);
    template<typename gen>
    void seed(gen &);
    void seed(unsigned long long);
    void split(unsigned int, unsigned int);
    void jump2(unsigned int);
    void jump(unsigned long long);
    void discard(unsigned long long);
    static const char * name();
    long operator()(long);
  };
```

```
  bool operator==(const lcg64_shift &, const lcg64_shift &);
  bool operator!=(const lcg64_shift &, const lcg64_shift &);
  template<typename char_t, typename traits_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const lcg64_shift &);
  template<typename char_t, typename traits_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, lcg64_shift &);
}
```

The class `trng::lcg64_count_shift` provides the same public interface as `trng::lcg64` and `trng::lcg64_shift`.

```
namespace trng {

  class lcg64_count_shift {
  public:
    using result_type = unsigned long long;
    result_type operator()();
    static constexpr result_type min();
    static constexpr result_type max();
    class parameter_type;
    class status_type;
    static const parameter_type Default;
    static const parameter_type LEcuyer1;
    static const parameter_type LEcuyer2;
    static const parameter_type LEcuyer3;
    explicit lcg64_count_shift(parameter_type=Default);
    explicit lcg64_count_shift(unsigned long, parameter_type=Default);
    template<typename gen>
    explicit lcg64_shift(gen &, parameter_type P=Default);
    void seed();
    void seed(unsigned long);
    template<typename gen>
    void seed(gen &);
    void seed(unsigned long long);
    void split(unsigned int, unsigned int);
    void jump2(unsigned int);
    void jump(unsigned long long);
    void discard(unsigned long long);
    static const char * name();
    long operator()(long);
  };

  bool operator==(const lcg64_count_shift &, const lcg64_count_shift &);
  bool operator!=(const lcg64_count_shift &, const lcg64_count_shift &);
  template<typename char_t, typename traits_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const lcg64_count_shift &);
  template<typename char_t, typename traits_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, lcg64_count_shift &);
}
```

4.1.2 Multiple recursive generators

TRNG offers several multiple recursive generators based on LFSR sequences over prime fields $\mathbb{F}_m$ with different numbers of feedback taps. These are implemented by the classes `trng::mrg2`, `trng::mrg3`, `trng::mrg3s`, `trng::mrg4`, `trng::mrg5`, and `trng::mrg5s`. Table 4.2 summarizes the key features of these classes. The transition algorithm of a multiple recursive generator with n feedback taps reads

$$r_i = a_1 \cdot r_{i-1} + a_2 \cdot r_{i-2} + \ldots + a_n \cdot r_{n-2} \bmod m\,.$$

The state of this generator at time i is given by $(r_{i-1}, r_{i-2}, \ldots, r_{i-n})$. See section 2.4.2 for details on LFSR sequences.

The prime modulus m that characterizes the prime field $\mathbb{F}_m$ was either chosen as the Mersenne Prime (classes `trng::mrg`n) or a Sophie-Germain Prime such that $m^n - 1$ has as few prime factors as possible (classes `trng::mrg`n`s`). The former choice gives us some performance benefits, see section 7.1, whereas the second has some theoretical advantages, see section 2.4.2.

The classes `trng::mrg`n and `trng::mrg`n`s` implement the interface described in section 3.1. Each class defines some parameter and status classes that will be used internally and by the constructor. Furthermore for each generator several parameter sets are given, see Table 4.3. Most of the parameter sets are taken from [37] and chosen to give generators with good statistical properties.

An instance of a class `trng::mrg`n or `trng::mrg`n`s` can be instantiated by various constructors as specified for a random number engine. Additionally a non-default parameter set may be chosen. The classes `trng::mrg`n and `trng::mrg`n`s` provide all necessary seeding functions (see Table 3.1) and additionally a function that sets the internal state $(r_{i-1}, r_{i-2}, \ldots, r_{i-n})$. This function should never be called with all arguments set to zero. The classes `trng::mrg`n and `trng::mrg`n`s` model the concept of a parallel random number engine and therefore the methods

```
void split(unsigned int, unsigned int);
void jump2(unsigned int);
void jump(unsigned long long);
void discard(unsigned long long);
```

are implemented. Furthermore the classes `trng::mrg`n or `trng::mrg`n`s` provide a function that returns a string with its name and an operator `operator()`. Random number engines are comparable and can be written to or read from a stream.

The detailed interface of the classes `trng::mrg`n or `trng::mrg`n`s` is given as follows:

```
namespace trng {

  class mrg2 {
  public:
    using result_type = long;
    result_type operator()();
    static constexpr result_type min();
    static constexpr result_type max();

    class parameter_type;
    class status_type;

    static const parameter_type LEcuyer1;
    static const parameter_type LEcuyer2;
```

Table 4.2: Key features of multiple recursive generator classes.

class	header file	feedback taps n	prime field $\mathbb{F}_m$	period	return value of `name()`
`trng::mrg2`	`trng/mrg2.hpp`	2	$\mathbb{F}_{2^{31}-1}$	$m^2-1 \approx 2^{62} \approx 4.61 \cdot 10^{18}$	`mrg2`
`trng::mrg3`	`trng/mrg3.hpp`	3	$\mathbb{F}_{2^{31}-1}$	$m^3-1 \approx 2^{93} \approx 9.90 \cdot 10^{27}$	`mrg3`
`trng::mrg3s`	`trng/mrg3s.hpp`	3	$\mathbb{F}_{2^{31}-21\,069}$	$m^3-1 \approx 2^{93} \approx 9.90 \cdot 10^{27}$	`mrg3s`
`trng::mrg4`	`trng/mrg4.hpp`	4	$\mathbb{F}_{2^{31}-1}$	$m^4-1 \approx 2^{124} \approx 2.13 \cdot 10^{37}$	`mrg4`
`trng::mrg5`	`trng/mrg5.hpp`	5	$\mathbb{F}_{2^{31}-1}$	$m^5-1 \approx 2^{155} \approx 4.57 \cdot 10^{46}$	`mrg5`
`trng::mrg5s`	`trng/mrg5s.hpp`	5	$\mathbb{F}_{2^{31}-22\,641}$	$m^5-1 \approx 2^{155} \approx 4.57 \cdot 10^{46}$	`mrg5s`

Table 4.3: Parameter sets for multiple recursive generators.

parameter set	a_1	a_2	a_3	a_4	a_5
`trng::mrg2::LEcuyer1`	1 498 809 829	1 160 990 996			
`trng::mrg2::LEcuyer2`	46 325	1 084 587			
`trng::mrg3::LEcuyer1`	2 021 422 057	1 826 992 351	1 977 753 457		
`trng::mrg3::LEcuyer2`	1 476 728 729	0	1 155 643 113		
`trng::mrg3::LEcuyer3`	65 338	0	64 636		
`trng::mrg3s::trng0`	2 025 213 985	1 112 953 677	2 038 969 601		
`trng::mrg3s::trng1`	1 287 767 370	1 045 931 779	58 150 106		
`trng::mrg4::LEcuyer1`	2 001 982 722	1 412 284 257	1 155 380 217	1 668 339 922	
`trng::mrg4::LEcuyer2`	64 886	0	0	64 322	
`trng::mrg5::LEcuyer1`	107 374 182	0	0	0	104 480
`trng::mrg5s::trng0`	1 053 223 373	1 530 818 118	1 612 122 482	133 497 989	573 245 311
`trng::mrg5s::trng1`	2 068 619 238	2 138 332 912	671 754 166	1 442 240 992	1 526 958 817

```
    explicit mrg2(parameter_type=LEcuyer1);
    explicit mrg2(unsigned long, parameter_type=LEcuyer1);
    template<typename gen>
    explicit mrg2(gen &, parameter_type P=LEcuyer1);

    void seed();
    void seed(unsigned long);
    template<typename gen>
    void seed(gen &);
    void seed(result_type, result_type);

    void split(unsigned int, unsigned int);
    void jump2(unsigned int);
    void jump(unsigned long long);
    void discard(unsigned long long);

    static const char * name();
    long operator()(long);
  };

  bool operator==(const mrg2 &, const mrg2 &);
  bool operator!=(const mrg2 &, const mrg2 &);
  template<typename char_t, typename traits_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const mrg2 &);
  template<typename char_t, typename traits_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, mrg2 &);
}
```

```
namespace trng {

  class mrg3 {
  public:
    using result_type = long;
    result_type operator()();
    static constexpr result_type min();
    static constexpr result_type max();

    class parameter_type;
    class status_type;

    static const parameter_type LEcuyer1;
    static const parameter_type LEcuyer2;
    static const parameter_type LEcuyer3;

    explicit mrg3(parameter_type=LEcuyer1);
    explicit mrg3(unsigned long, parameter_type=LEcuyer1);
    template<typename gen>
    explicit mrg3(gen &, parameter_type P=LEcuyer1);

    void seed();
    void seed(unsigned long);
    template<typename gen>
    void seed(gen &);
    void seed(result_type, result_type, result_type);
```

```
    void split(unsigned int, unsigned int);
    void jump2(unsigned int);
    void jump(unsigned long long);
    void discard(unsigned long long);

    static const char * name();
    long operator()(long);
  };

  bool operator==(const mrg3 &, const mrg3 &);
  bool operator!=(const mrg3 &, const mrg3 &);
  template<typename char_t, typename traits_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const mrg3 &);
  template<typename char_t, typename traits_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, mrg3 &);
}
```

```
namespace trng {

  class mrg3s {
  public:
    using result_type = long;
    result_type operator()();
    static constexpr result_type min();
    static constexpr result_type max();

    class parameter_type;
    class status_type;

    static const parameter_type trng0;
    static const parameter_type trng1;

    explicit mrg3s(parameter_type=trng0);
    explicit mrg3s(unsigned long, parameter_type=trng0);
    template<typename gen>
    explicit mrg3s(gen &, parameter_type P=trng0);

    void seed();
    void seed(unsigned long);
    template<typename gen>
    void seed(gen &);
    void seed(result_type, result_type, result_type);

    void split(unsigned int, unsigned int);
    void jump2(unsigned int);
    void jump(unsigned long long);
    void discard(unsigned long long);

    static const char * name();
    long operator()(long);
  };

  bool operator==(const mrg3s &, const mrg3s &);
  bool operator!=(const mrg3s &, const mrg3s &);
```

```
  template<typename char_t, typename traits_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const mrg3s &);
  template<typename char_t, typename traits_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, mrg3s &);
}
```

```
namespace trng {

  class mrg4 {
  public:
    using result_type = long;
    result_type operator()();
    static constexpr result_type min();
    static constexpr result_type max();

    class parameter_type;
    class status_type;

    static const parameter_type LEcuyer1;
    static const parameter_type LEcuyer2;

    explicit mrg4(parameter_type=LEcuyer1);
    explicit mrg4(unsigned long, parameter_type=LEcuyer1);
    template<typename gen>
    explicit mrg4(gen &, parameter_type P=LEcuyer1);

    void seed();
    void seed(unsigned long);
    template<typename gen>
    void seed(gen &);
    void seed(result_type, result_type, result_type, result_type);

    void split(unsigned int, unsigned int);
    void jump2(unsigned int);
    void jump(unsigned long long);
    void discard(unsigned long long);

    static const char * name();
    long operator()(long);
  };

  bool operator==(const mrg4 &, const mrg4 &);
  bool operator!=(const mrg4 &, const mrg4 &);
  template<typename char_t, typename traits_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const mrg4 &);
  template<typename char_t, typename traits_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, mrg4 &);
}
```

```
namespace trng {

  class mrg5 {
  public:
```

```
    using result_type = long;
    result_type operator()();
    static constexpr result_type min();
    static constexpr result_type max();

    class parameter_type;
    class status_type;

    static const parameter_type LEcuyer1;

    explicit mrg5(parameter_type=LEcuyer1);
    explicit mrg5(unsigned long, parameter_type=LEcuyer1);
    template<typename gen>
    explicit mrg5(gen &, parameter_type P=LEcuyer1);

    void seed();
    void seed(unsigned long);
    template<typename gen>
    void seed(gen &);
    void seed(result_type, result_type, result_type, result_type, result_type);

    void split(unsigned int, unsigned int);
    void jump2(unsigned int);
    void jump(unsigned long long);
    void discard(unsigned long long);

    static const char * name();
    long operator()(long);
  };

  bool operator==(const mrg5 &, const mrg5 &);
  bool operator!=(const mrg5 &, const mrg5 &);
  template<typename char_t, typename traits_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const mrg5 &);
  template<typename char_t, typename traits_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, mrg5 &);
}
```

```
namespace trng {

  class mrg5s {
  public:
    using result_type = long;
    result_type operator()();
    static constexpr result_type min();
    static constexpr result_type max();

    class parameter_type;
    class status_type;

    static const parameter_type trng0;
    static const parameter_type trng1;

    explicit mrg5s(parameter_type=trng0);
    explicit mrg5s(unsigned long, parameter_type=trng0);
```

```
    template<typename gen>
    explicit mrg5s(gen &, parameter_type P=trng0);

    void seed();
    void seed(unsigned long);
    template<typename gen>
    void seed(gen &);
    void seed(result_type, result_type, result_type, result_type, result_type);

    void split(unsigned int, unsigned int);
    void jump2(unsigned int);
    void jump(unsigned long long);
    void discard(unsigned long long);

    static const char * name();
    long operator()(long);
  };

  bool operator==(const mrg5s &, const mrg5s &);
  bool operator!=(const mrg5s &, const mrg5s &);
  template<typename char_t, typename traits_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const mrg5s &);
  template<typename char_t, typename traits_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, mrg5s &);
}
```

4.1.3 YARN generators

The classes `trng::yarn`n and `trng::yarn`n`s` implement so-called YARN generators (yet another random number generator). Table 4.4 summarizes the key features of these classes. Each of them is based on a multiple recursive generator with n feedback taps, for which the transition algorithm reads

$$r_i = a_1 \cdot r_{i-1} + a_2 \cdot r_{i-2} + \ldots + a_n \cdot r_{i-n} \bmod m\,.$$

The state of this generator at time i is given by $(r_{i-1}, r_{i-2}, \ldots, r_{i-n})$. See section 2.4.2 for details on LFSR sequences.

The prime modulus m that characterizes the prime field $\mathbb{F}_m$ was either chosen as the Mersenne Prime (classes `trng::mrg`n) or a Sophie-Germain Prime such that $m^n - 1$ has as few prime factors as possible (classes `trng::mrg`n`s`). The former choice gives us some performance benefits, see section 7.1, whereas the second has some theoretical advantages, see section 2.4.2.

While pure multiple recursive generators return the r_i as pseudo-random numbers directly, a YARN generator "shuffles" the output of the underlying multiple recursive generator by a bijective mapping. In the case of a YARN generator with modulus m this mapping reads

$$q_i = \begin{cases} b^{r_i} \bmod m & \text{if } r_i > 0 \\ 0 & \text{if } r_i = 0 \end{cases},$$

where b is a generating element of the multiplicative group modulo m. This bijective mapping destroys the linear structures of the linear feedback shift register sequence. But on the other hand the new sequence q_i inherits all the nice features of the linear feedback shift register sequence r_i, e. g. its period. Block splitting and leapfrog methods can be implemented as easily as for multiple recursive generators, see section 2.4.2 and 2.5 for details.

The classes `trng::yarn`*n* and `trng::yarn`*n*`s` implement the interface described in section 3.1. Each class defines some parameter and status classes that will be used internally and by the constructor. Furthermore for each generator several parameter sets are given, see Table 4.3. Most of the parameter sets are taken from [37] and chosen to give generators with good statistical properties.

An instance of a class `trng::yarn`*n* or `trng::yarn`*n*`s` can be instantiated by various constructors as specified for a random number engine. Additionally a non-default parameter set may be chosen. The classes `trng::yarn`*n* and `trng::yarn`*n*`s` provide all necessary seeding functions (see Table 3.1) and additionally a function that sets the internal state $(r_{i-1}, r_{i-2}, \ldots, r_{i-n})$. This function should never be called with all arguments set to zero. The classes `trng::yarn`*n* and `trng::yarn`*n*`s` model the concept of a parallel random number engine and therefore the methods

```
void split(unsigned int, unsigned int);
void jump2(unsigned int);
void jump(unsigned long long);
void discard(unsigned long long);
```

are implemented. Furthermore, the classes `trng::yarn`*n* or `trng::yarn`*n*`s` provide a function that returns a string with its name and an operator `operator()`. Random number engines are comparable and can be written to or read from a stream.

The detailed interface of the classes `trng::mrg`*n* or `trng::mrg`*n*`s` is given as follows:

```
namespace trng {

  class yarn2 {
  public:
    using result_type = long;
    result_type operator()();
    static constexpr result_type min();
    static constexpr result_type max();

    class parameter_type;
    class status_type;

    static const parameter_type LEcuyer1;
    static const parameter_type LEcuyer2;

    explicit yarn2(parameter_type=LEcuyer1);
    explicit yarn2(unsigned long, parameter_type=LEcuyer1);
    template<typename gen>
    explicit yarn2(gen &, parameter_type P=LEcuyer1);

    void seed();
    void seed(unsigned long);
    template<typename gen>
    void seed(gen &);
    void seed(result_type, result_type);

    void split(unsigned int, unsigned int);
    void jump2(unsigned int);
```

Table 4.4: Key features of YARN generator classes.

class	header file	feedback taps n	prime field $\mathbb{F}_m$	period	return value of `name()`
`trng::yarn2`	`trng/yarn2.hpp`	2	$\mathbb{F}_{2^{31}-1}$	$m^2-1 \approx 2^{62} \approx 4.61 \cdot 10^{18}$	`yarn2`
`trng::yarn3`	`trng/yarn3.hpp`	3	$\mathbb{F}_{2^{31}-1}$	$m^3-1 \approx 2^{93} \approx 9.90 \cdot 10^{27}$	`yarn3`
`trng::yarn3s`	`trng/yarn3s.hpp`	3	$\mathbb{F}_{2^{31}-21\,069}$	$m^3-1 \approx 2^{93} \approx 9.90 \cdot 10^{27}$	`yarn3s`
`trng::yarn4`	`trng/yarn4.hpp`	4	$\mathbb{F}_{2^{31}-1}$	$m^4-1 \approx 2^{124} \approx 2.13 \cdot 10^{37}$	`yarn4`
`trng::yarn5`	`trng/yarn5.hpp`	5	$\mathbb{F}_{2^{31}-1}$	$m^5-1 \approx 2^{155} \approx 4.57 \cdot 10^{46}$	`yarn5`
`trng::yarn5s`	`trng/yarn5s.hpp`	5	$\mathbb{F}_{2^{31}-22\,641}$	$m^5-1 \approx 2^{155} \approx 4.57 \cdot 10^{46}$	`yarn5s`

Table 4.5: Parameter sets for YARN generators.

parameter set	a_1	a_2	a_3	a_4	a_5	b
`trng::yarn2::LEcuyer1`	1 498 809 829	1 160 990 996				123 567 893
`trng::yarn2::LEcuyer2`	46 325	1 084 587				123 567 893
`trng::yarn3::LEcuyer1`	2 021 422 057	1 826 992 351	1 977 753 457			123 567 893
`trng::yarn3::LEcuyer2`	1 476 728 729	0	1 155 643 113			123 567 893
`trng::yarn3::LEcuyer3`	65 338	0	64 636			123 567 893
`trng::yarn3s::trng0`	2 025 213 985	1 112 953 677	2 038 969 601			1 616 076 847
`trng::yarn3s::trng1`	1 287 767 370	1 045 931 779	58 150 106			1 616 076 847
`trng::yarn4::LEcuyer1`	2 001 982 722	1 412 284 257	1 155 380 217	1 668 339 922		123 567 893
`trng::yarn4::LEcuyer2`	64 886	0	0	64 322		123 567 893
`trng::yarn5::LEcuyer1`	107 374 182	0	0	0	104 480	123 567 893
`trng::yarn5s::trng0`	1 053 223 373	1 530 818 118	1 612 122 482	133 497 989	573 245 311	889 744 251
`trng::yarn5s::trng1`	2 068 619 238	2 138 332 912	671 754 166	1 442 240 992	1 526 958 817	889 744 251

```
    void jump(unsigned long long);
    void discard(unsigned long long);

    static const char * name();
    long operator()(long);
  };

  bool operator==(const yarn2 &, const yarn2 &);
  bool operator!=(const yarn2 &, const yarn2 &);
  template<typename char_t, typename traits_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &t, const yarn2 &);
  template<typename char_t, typename traits_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, yarn2 &);
}
```

```
namespace trng {

  class yarn3 {
  public:
    using result_type = long;
    result_type operator()();
    static constexpr result_type min();
    static constexpr result_type max();

    class parameter_type;
    class status_type;

    static const parameter_type LEcuyer1;
    static const parameter_type LEcuyer2;

    static const parameter_type LEcuyer3;

    explicit yarn3(parameter_type=LEcuyer1);
    explicit yarn3(unsigned long, parameter_type=LEcuyer1);
    template<typename gen>
    explicit yarn3(gen &, parameter_type P=LEcuyer1);

    void seed();
    void seed(unsigned long);
    template<typename gen>
    void seed(gen &);
    void seed(result_type, result_type, result_type);

    void split(unsigned int, unsigned int);
    void jump2(unsigned int);
    void jump(unsigned long long);
    void discard(unsigned long long);

    static const char * name();
    long operator()(long);
  };

  bool operator==(const yarn3 &, const yarn3 &);
  bool operator!=(const yarn3 &, const yarn3 &);
  template<typename char_t, typename traits_t>
```

```
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const yarn3 &);
  template<typename char_t, typename traits_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, yarn3 &);
}
```

```
namespace trng {

  class yarn3s {
  public:
    using result_type = long;
    result_type operator()();
    static constexpr result_type min();
    static constexpr result_type max();

    class parameter_type;
    class status_type;

    static const parameter_type trng0;
    static const parameter_type trng1;

    explicit yarn3s(parameter_type=trng0);
    explicit yarn3s(unsigned long, parameter_type=trng0);
    template<typename gen>
    explicit yarn3s(gen &, parameter_type P=trng0);

    void seed();
    void seed(unsigned long);
    template<typename gen>
    void seed(gen &);
    void seed(result_type, result_type, result_type);

    void split(unsigned int, unsigned int);
    void jump2(unsigned int);
    void jump(unsigned long long);
    void discard(unsigned long long);

    static const char * name();
    long operator()(long);
  };

  bool operator==(const yarn3s &, const yarn3s &);
  bool operator!=(const yarn3s &, const yarn3s &);
  template<typename char_t, typename traits_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const yarn3s &);
  template<typename char_t, typename traits_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, yarn3s &);
}
```

```
namespace trng {

  class yarn4 {
  public:
    using result_type = long;
```

```
    result_type operator()();
    static constexpr result_type min();
    static constexpr result_type max();

    class parameter_type;
    class status_type;

    static const parameter_type LEcuyer1;
    static const parameter_type LEcuyer2;

    explicit yarn4(parameter_type=LEcuyer1);
    explicit yarn4(unsigned long, parameter_type=LEcuyer1);
    template<typename gen>
    explicit yarn4(gen &, parameter_type P=LEcuyer1);

    void seed();
    void seed(unsigned long);
    template<typename gen>
    void seed(gen &);
    void seed(result_type, result_type, result_type, result_type);

    void split(unsigned int, unsigned int);
    void jump2(unsigned int);
    void jump(unsigned long long);
    void discard(unsigned long long);

    static const char * name();
    long operator()(long);
  };

  bool operator==(const yarn4 &, const yarn4 &);
  bool operator!=(const yarn4 &, const yarn4 &);
  template<typename char_t, typename traits_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const yarn4 &);
  template<typename char_t, typename traits_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, yarn4 &);
}
```

```
namespace trng {

  class yarn5 {
  public:
    using result_type = long;
    result_type operator()();
    static constexpr result_type min();
    static constexpr result_type max();

    class parameter_type;
    class status_type;

    static const parameter_type LEcuyer1;

    explicit yarn5(parameter_type=LEcuyer1);
    explicit yarn5(unsigned long, parameter_type=LEcuyer1);
    template<typename gen>
```

```
    explicit yarn5(gen &, parameter_type P=LEcuyer1);

    void seed();
    void seed(unsigned long);
    template<typename gen>
    void seed(gen &);
    void seed(result_type, result_type, result_type, result_type, result_type);

    void split(unsigned int, unsigned int);
    void jump2(unsigned int);
    void jump(unsigned long long);
    void discard(unsigned long long);

    static const char * name();
    long operator()(long);
  };

  bool operator==(const yarn5 &, const yarn5 &);
  bool operator!=(const yarn5 &, const yarn5 &);
  template<typename char_t, typename traits_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const yarn5 &);
  template<typename char_t, typename traits_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, yarn5 &);
}
```

```
namespace trng {

  class yarn5s {
  public:
    using result_type = long;
    result_type operator()();
    static constexpr result_type min();
    static constexpr result_type max();

    class parameter_type;
    class status_type;

    static const parameter_type trng0;
    static const parameter_type trng1;

    explicit yarn5s(parameter_type=trng0);
    explicit yarn5s(unsigned long, parameter_type=trng0);
    template<typename gen>
    explicit yarn5s(gen &, parameter_type P=trng0);

    void seed();
    void seed(unsigned long);
    template<typename gen>
    void seed(gen &);
    void seed(result_type, result_type, result_type, result_type, result_type);

    void split(unsigned int, unsigned int);
    void jump2(unsigned int);
    void jump(unsigned long long);
    void discard(unsigned long long);
```

```
    static const char * name();
    long operator()(long);
  };

  bool operator==(const yarn5s &, const yarn5s &);
  bool operator!=(const yarn5s &, const yarn5s &);
  template<typename char_t, typename traits_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const yarn5s &);
  template<typename char_t, typename traits_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, yarn5s &);
}
```

4.1.4 Lagged Fibonacci generators

The template classes `trng::lagfib2xor`, `trng::lagfib4xor`, `trng::lagfib2plus`, `trng::lagfib4plus` model random number engines (no splitting facilities) and implement lagged Fibonacci generators with two or four feedback taps and exclusive-or or additive operation. The recursion relation of these types of generators read

$$
\begin{aligned}
r_i &= r_{i-A} \oplus r_{i-B} \\
r_i &= r_{i-A} \oplus r_{i-B} \oplus r_{i-C} \oplus r_{i-D} \\
r_i &= r_{i-A} + r_{i-B} \bmod 2^l \\
r_i &= r_{i-A} + r_{i-B} + r_{i-C} + r_{i-D} \bmod 2^l\,.
\end{aligned}
$$

These template classes are parameterized by an unsigned integer type, e. g. `unsigned int` or `unsigned long long`, and the position of the feedback taps with $A < B < C < D$. For properly chosen feedback taps the period of an exclusive-or generator is $2^B - 1$ or $2^D - 1$ respectively, and the period of an plus generator is $(2^B - 1)2^{l-1}$ or $(2^D - 1)2^{l-1}$ respectively, where l denotes the number of significant bits of the integer type given as a template argument. Template classes are declared in the header files `trng/lagfib2xor.hpp`, `trng/lagfib4xor.hpp`, `trng/lagfib2plus.hpp`, and `trng/lagfib4plus.hpp`. For convenience TRNG provides some typedefs for some realizations of lagged Fibonacci generators with two or four feedback taps.

The detailed interfaces of the classes `trng::lagfib2xor`, `trng::lagfib4xor`, `trng::lagfib2plus`, `trng::lagfib4plus` are given as follows:

```
namespace trng {

  template<typename integer_type,
  unsigned int A, unsigned int B>
  class lagfib2xor {
  public:
    using result_type = integer_type;
    result_type operator()();
    static constexpr result_type min();
    static constexpr result_type max();

    class status_type;
```

```
    lagfib2xor();
    explicit lagfib2xor(unsigned long);
    template<typename gen>
    explicit lagfib2xor(gen &);

    void seed();
    void seed(unsigned long);
    template<typename gen>
    void seed(gen &);
    void discard(unsigned long long);
  };

  typedef lagfib2xor<unsigned long,       103,   250> r250_ul;
  typedef lagfib2xor<unsigned long long,  103,   250> r250_ull;
  typedef lagfib2xor<unsigned long,       168,   521> lagfib2xor_521_ul;
  typedef lagfib2xor<unsigned long long,  168,   521> lagfib2xor_521_ull;
  typedef lagfib2xor<unsigned long,       273,   607> lagfib2xor_607_ul;
  typedef lagfib2xor<unsigned long long,  273,   607> lagfib2xor_607_ull;
  typedef lagfib2xor<unsigned long,       418,  1279> lagfib2xor_1279_ul;
  typedef lagfib2xor<unsigned long long,  418,  1279> lagfib2xor_1279_ull;
  typedef lagfib2xor<unsigned long,      1029,  2281> lagfib2xor_2281_ul;
  typedef lagfib2xor<unsigned long long, 1029,  2281> lagfib2xor_2281_ull;
  typedef lagfib2xor<unsigned long,       576,  3217> lagfib2xor_3217_ul;
  typedef lagfib2xor<unsigned long long,  576,  3217> lagfib2xor_3217_ull;
  typedef lagfib2xor<unsigned long,      2098,  4423> lagfib2xor_4423_ul;
  typedef lagfib2xor<unsigned long long, 2098,  4423> lagfib2xor_4423_ull;
  typedef lagfib2xor<unsigned long,      4187,  9689> lagfib2xor_9689_ul;
  typedef lagfib2xor<unsigned long long, 4187,  9689> lagfib2xor_9689_ull;
  typedef lagfib2xor<unsigned long,      9842, 19937> lagfib2xor_19937_ul;
  typedef lagfib2xor<unsigned long long, 9842, 19937> lagfib2xor_19937_ull;

  typedef lagfib2xor<uint32_t,  103,   250> r250_32;
  typedef lagfib2xor<uint64_t,  103,   250> r250_64;
  typedef lagfib2xor<uint32_t,  168,   521> lagfib2xor_521_32;
  typedef lagfib2xor<uint64_t,  168,   521> lagfib2xor_521_64;
  typedef lagfib2xor<uint32_t,  273,   607> lagfib2xor_607_32;
  typedef lagfib2xor<uint64_t,  273,   607> lagfib2xor_607_64;
  typedef lagfib2xor<uint32_t,  418,  1279> lagfib2xor_1279_32;
  typedef lagfib2xor<uint64_t,  418,  1279> lagfib2xor_1279_64;
  typedef lagfib2xor<uint32_t, 1029,  2281> lagfib2xor_2281_32;
  typedef lagfib2xor<uint64_t, 1029,  2281> lagfib2xor_2281_64;
  typedef lagfib2xor<uint32_t,  576,  3217> lagfib2xor_3217_32;
  typedef lagfib2xor<uint64_t,  576,  3217> lagfib2xor_3217_64;
  typedef lagfib2xor<uint32_t, 2098,  4423> lagfib2xor_4423_32;
  typedef lagfib2xor<uint64_t, 2098,  4423> lagfib2xor_4423_64;
  typedef lagfib2xor<uint32_t, 4187,  9689> lagfib2xor_9689_32;
  typedef lagfib2xor<uint64_t, 4187,  9689> lagfib2xor_9689_64;
  typedef lagfib2xor<uint32_t, 9842, 19937> lagfib2xor_19937_32;
  typedef lagfib2xor<uint64_t, 9842, 19937> lagfib2xor_19937_64;

}
```

```
namespace trng {

  template<typename integer_type,
  unsigned int A, unsigned int B, unsigned int C, unsigned int D>
  class lagfib4xor {
```

```
public:
  using result_type = integer_type;
  result_type operator()();
  static constexpr result_type min();
  static constexpr result_type max();

  class status_type;

  lagfib4xor();
  explicit lagfib4xor(unsigned long);
  template<typename gen>
  explicit lagfib4xor(gen &);

  void seed();
  void seed(unsigned long);
  template<typename gen>
  void seed(gen &);
  void discard(unsigned long long);
};

typedef lagfib4xor<unsigned long,       471, 1586,  6988,  9689> Ziff_ul;
typedef lagfib4xor<unsigned long long,  471, 1586,  6988,  9689> Ziff_ull;
typedef lagfib4xor<unsigned long,       168,  205,   242,   521> lagfib4xor_521_ul;
typedef lagfib4xor<unsigned long long,  168,  205,   242,   521> lagfib4xor_521_ull;
typedef lagfib4xor<unsigned long,       147,  239,   515,   607> lagfib4xor_607_ul;
typedef lagfib4xor<unsigned long long,  147,  239,   515,   607> lagfib4xor_607_ull;
typedef lagfib4xor<unsigned long,       418,  705,   992,  1279> lagfib4xor_1279_ul;
typedef lagfib4xor<unsigned long long,  418,  705,   992,  1279> lagfib4xor_1279_ull;
typedef lagfib4xor<unsigned long,       305,  610,   915,  2281> lagfib4xor_2281_ul;
typedef lagfib4xor<unsigned long long,  305,  610,   915,  2281> lagfib4xor_2281_ull;
typedef lagfib4xor<unsigned long,       576,  871,  1461,  3217> lagfib4xor_3217_ul;
typedef lagfib4xor<unsigned long long,  576,  871,  1461,  3217> lagfib4xor_3217_ull;
typedef lagfib4xor<unsigned long,      1419, 1736,  2053,  4423> lagfib4xor_4423_ul;
typedef lagfib4xor<unsigned long long, 1419, 1736,  2053,  4423> lagfib4xor_4423_ull;
typedef lagfib4xor<unsigned long,       471, 2032,  4064,  9689> lagfib4xor_9689_ul;
typedef lagfib4xor<unsigned long long,  471, 2032,  4064,  9689> lagfib4xor_9689_ull;
typedef lagfib4xor<unsigned long,      3860, 7083, 11580, 19937> lagfib4xor_19937_ul;
typedef lagfib4xor<unsigned long long, 3860, 7083, 11580, 19937> lagfib4xor_19937_ull;

typedef lagfib4xor<uint32_t,  471, 1586,  6988,  9689> Ziff_32;
typedef lagfib4xor<uint64_t,  471, 1586,  6988,  9689> Ziff_64;
typedef lagfib4xor<uint32_t,  168,  205,   242,   521> lagfib4xor_521_32;
typedef lagfib4xor<uint64_t,  168,  205,   242,   521> lagfib4xor_521_64;
typedef lagfib4xor<uint32_t,  147,  239,   515,   607> lagfib4xor_607_32;
typedef lagfib4xor<uint64_t,  147,  239,   515,   607> lagfib4xor_607_64;
typedef lagfib4xor<uint32_t,  418,  705,   992,  1279> lagfib4xor_1279_32;
typedef lagfib4xor<uint64_t,  418,  705,   992,  1279> lagfib4xor_1279_64;
typedef lagfib4xor<uint32_t,  305,  610,   915,  2281> lagfib4xor_2281_32;
typedef lagfib4xor<uint64_t,  305,  610,   915,  2281> lagfib4xor_2281_64;
typedef lagfib4xor<uint32_t,  576,  871,  1461,  3217> lagfib4xor_3217_32;
typedef lagfib4xor<uint64_t,  576,  871,  1461,  3217> lagfib4xor_3217_64;
typedef lagfib4xor<uint32_t, 1419, 1736,  2053,  4423> lagfib4xor_4423_32;
typedef lagfib4xor<uint64_t, 1419, 1736,  2053,  4423> lagfib4xor_4423_64;
typedef lagfib4xor<uint32_t,  471, 2032,  4064,  9689> lagfib4xor_9689_32;
typedef lagfib4xor<uint64_t,  471, 2032,  4064,  9689> lagfib4xor_9689_64;
typedef lagfib4xor<uint32_t, 3860, 7083, 11580, 19937> lagfib4xor_19937_32;
typedef lagfib4xor<uint64_t, 3860, 7083, 11580, 19937> lagfib4xor_19937_64;
```

```
}
```

```
namespace trng {

  template<typename integer_type,
  unsigned int A, unsigned int B>
  class lagfib2plus {
  public:
    using result_type = integer_type;
    result_type operator()();
    static constexpr result_type min();
    static constexpr result_type max();

    class status_type;

    lagfib2plus();
    explicit lagfib2plus(unsigned long);
    template<typename gen>
    explicit lagfib2plus(gen &);

    void seed();
    void seed(unsigned long);
    template<typename gen>
    void seed(gen &);
    void discard(unsigned long long);
  };

  typedef lagfib2plus<unsigned long,       168,   521> lagfib2plus_521_ul;
  typedef lagfib2plus<unsigned long long,  168,   521> lagfib2plus_521_ull;
  typedef lagfib2plus<unsigned long,       273,   607> lagfib2plus_607_ul;
  typedef lagfib2plus<unsigned long long,  273,   607> lagfib2plus_607_ull;
  typedef lagfib2plus<unsigned long,       418,  1279> lagfib2plus_1279_ul;
  typedef lagfib2plus<unsigned long long,  418,  1279> lagfib2plus_1279_ull;
  typedef lagfib2plus<unsigned long,      1029,  2281> lagfib2plus_2281_ul;
  typedef lagfib2plus<unsigned long long, 1029,  2281> lagfib2plus_2281_ull;
  typedef lagfib2plus<unsigned long,       576,  3217> lagfib2plus_3217_ul;
  typedef lagfib2plus<unsigned long long,  576,  3217> lagfib2plus_3217_ull;
  typedef lagfib2plus<unsigned long,      2098,  4423> lagfib2plus_4423_ul;
  typedef lagfib2plus<unsigned long long, 2098,  4423> lagfib2plus_4423_ull;
  typedef lagfib2plus<unsigned long,      4187,  9689> lagfib2plus_9689_ul;
  typedef lagfib2plus<unsigned long long, 4187,  9689> lagfib2plus_9689_ull;
  typedef lagfib2plus<unsigned long,      9842, 19937> lagfib2plus_19937_ul;
  typedef lagfib2plus<unsigned long long, 9842, 19937> lagfib2plus_19937_ull;

  typedef lagfib2plus<uint32_t,  168,   521> lagfib2plus_521_32;
  typedef lagfib2plus<uint64_t,  168,   521> lagfib2plus_521_64;
  typedef lagfib2plus<uint32_t,  273,   607> lagfib2plus_607_32;
  typedef lagfib2plus<uint64_t,  273,   607> lagfib2plus_607_64;
  typedef lagfib2plus<uint32_t,  418,  1279> lagfib2plus_1279_32;
  typedef lagfib2plus<uint64_t,  418,  1279> lagfib2plus_1279_64;
  typedef lagfib2plus<uint32_t, 1029,  2281> lagfib2plus_2281_32;
  typedef lagfib2plus<uint64_t, 1029,  2281> lagfib2plus_2281_64;
  typedef lagfib2plus<uint32_t,  576,  3217> lagfib2plus_3217_32;
  typedef lagfib2plus<uint64_t,  576,  3217> lagfib2plus_3217_64;
  typedef lagfib2plus<uint32_t, 2098,  4423> lagfib2plus_4423_32;
  typedef lagfib2plus<uint64_t, 2098,  4423> lagfib2plus_4423_64;
```

```
  typedef lagfib2plus<uint32_t, 4187,  9689> lagfib2plus_9689_32;
  typedef lagfib2plus<uint64_t, 4187,  9689> lagfib2plus_9689_64;
  typedef lagfib2plus<uint32_t, 9842, 19937> lagfib2plus_19937_32;
  typedef lagfib2plus<uint64_t, 9842, 19937> lagfib2plus_19937_64;

}
```

```
namespace trng {

  template<typename integer_type,
  unsigned int A, unsigned int B, unsigned int C, unsigned int D>
  class lagfib4plus {
  public:
    using result_type = integer_type;
    result_type operator()();
    static constexpr result_type min();
    static constexpr result_type max();

    class status_type;

    lagfib4plus();
    explicit lagfib2plus(unsigned long);
    template<typename gen>
    explicit lagfib4plus(gen &);

    void seed();
    void seed(unsigned long);
    template<typename gen>
    void seed(gen &);
    void discard(unsigned long long);
  };

  typedef lagfib4plus<unsigned long,       168,  205,   242,   521> lagfib4plus_521_ul;
  typedef lagfib4plus<unsigned long long,  168,  205,   242,   521> lagfib4plus_521_ull;
  typedef lagfib4plus<unsigned long,       147,  239,   515,   607> lagfib4plus_607_ul;
  typedef lagfib4plus<unsigned long long,  147,  239,   515,   607> lagfib4plus_607_ull;
  typedef lagfib4plus<unsigned long,       418,  705,   992,  1279> lagfib4plus_1279_ul;
  typedef lagfib4plus<unsigned long long,  418,  705,   992,  1279> lagfib4plus_1279_ull;
  typedef lagfib4plus<unsigned long,       305,  610,   915,  2281> lagfib4plus_2281_ul;
  typedef lagfib4plus<unsigned long long,  305,  610,   915,  2281> lagfib4plus_2281_ull;
  typedef lagfib4plus<unsigned long,       576,  871,  1461,  3217> lagfib4plus_3217_ul;
  typedef lagfib4plus<unsigned long long,  576,  871,  1461,  3217> lagfib4plus_3217_ull;
  typedef lagfib4plus<unsigned long,      1419, 1736,  2053,  4423> lagfib4plus_4423_ul;
  typedef lagfib4plus<unsigned long long, 1419, 1736,  2053,  4423> lagfib4plus_4423_ull;
  typedef lagfib4plus<unsigned long,       471, 2032,  4064,  9689> lagfib4plus_9689_ul;
  typedef lagfib4plus<unsigned long long,  471, 2032,  4064,  9689> lagfib4plus_9689_ull;
  typedef lagfib4plus<unsigned long,      3860, 7083, 11580, 19937> lagfib4plus_19937_ul;
  typedef lagfib4plus<unsigned long long, 3860, 7083, 11580, 19937> lagfib4plus_19937_ull;

  typedef lagfib4plus<uint32_t,  168,  205,   242,   521> lagfib4plus_521_32;
  typedef lagfib4plus<uint64_t,  168,  205,   242,   521> lagfib4plus_521_64;
  typedef lagfib4plus<uint32_t,  147,  239,   515,   607> lagfib4plus_607_32;
  typedef lagfib4plus<uint64_t,  147,  239,   515,   607> lagfib4plus_607_64;
  typedef lagfib4plus<uint32_t,  418,  705,   992,  1279> lagfib4plus_1279_32;
  typedef lagfib4plus<uint64_t,  418,  705,   992,  1279> lagfib4plus_1279_64;
  typedef lagfib4plus<uint32_t,  305,  610,   915,  2281> lagfib4plus_2281_32;
  typedef lagfib4plus<uint64_t,  305,  610,   915,  2281> lagfib4plus_2281_64;
```

```
  typedef lagfib4plus<uint32_t,  576,  871,  1461,  3217> lagfib4plus_3217_32;
  typedef lagfib4plus<uint64_t,  576,  871,  1461,  3217> lagfib4plus_3217_64;
  typedef lagfib4plus<uint32_t, 1419, 1736,  2053,  4423> lagfib4plus_4423_32;
  typedef lagfib4plus<uint64_t, 1419, 1736,  2053,  4423> lagfib4plus_4423_64;
  typedef lagfib4plus<uint32_t,  471, 2032,  4064,  9689> lagfib4plus_9689_32;
  typedef lagfib4plus<uint64_t,  471, 2032,  4064,  9689> lagfib4plus_9689_64;
  typedef lagfib4plus<uint32_t, 3860, 7083, 11580, 19937> lagfib4plus_19937_32;
  typedef lagfib4plus<uint64_t, 3860, 7083, 11580, 19937> lagfib4plus_19937_64;

}
```

4.1.5 Xoshiro type generator

The xoshiro (xor/shift/rotate) type generators [7] are based on matrix linear congruential generators in $\mathbb{F}_2$. The matrix of the recurrence equation of xoshiro type generators is sparse and has a special form that allows an efficient implementation that uses xor, bit-shift and bit-rotation operations only, for example

$$\boldsymbol{A} = \begin{pmatrix} \boldsymbol{I} & \boldsymbol{I} & \boldsymbol{I} & \boldsymbol{0} \\ \boldsymbol{I} & \boldsymbol{I} & \boldsymbol{S}^a & \boldsymbol{R}^b \\ \boldsymbol{0} & \boldsymbol{I} & \boldsymbol{I} & \boldsymbol{0} \\ \boldsymbol{I} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{R}^b \end{pmatrix} . \tag{4.1}$$

Here $\boldsymbol{I}$ denotes a $w \times w$ identity matrix, $\boldsymbol{S}$ is a $w \times w$ shift matrix and $\boldsymbol{R}$ is a $w \times w$ rotation matrix and a and b denote two integer parameters.

The class `trng::xoshiro256plus` in the header file `trng/xoshiro256plus.hpp` implements an xoshiro type generator with $w = 64$, $a = 17$ and $b = 45$. This means the generator has a 256 bit state vector. Its period equals $2^{256} - 1$. To output a pseudo random number this 256 bit state vector is transformed into a 64 bit integer by adding the lowest 64 bits to the highest 64 bit modulo 2^{64}. The detailed interfaces of the class `trng::xoshiro256plus` is given as follows:

```
namespace trng {

  class xoshiro256plus {
  public:
    using result_type = uint64_t;
    result_type operator()();

    static constexpr result_type min();
    static constexpr result_type max();

    class status_type;

    explicit xoshiro256plus();
    explicit xoshiro256plus(unsigned long);
    explicit xoshiro256plus(result_type s0, result_type s1, result_type s2, result_type s3);
    template<typename gen>
    explicit xoshiro256plus(gen &g);

    void seed();
    void seed(unsigned long);
    template<typename gen>
    void seed(gen &g);
```

```
    void seed(result_type, result_type, result_type, result_type);

    void jump2(unsigned int);
    void jump(unsigned long long);
    void discard(unsigned long long);

    static const char *name();
    long operator()(long);
  };

  bool operator==(const xoshiro256plus &, const xoshiro256plus &);
  bool operator!=(const xoshiro256plus &, const xoshiro256plus &);

  template<typename char_t, typename traits_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const xoshiro256plus &);
  template<typename char_t, typename traits_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, xoshiro256plus &);
}
```

Note that the class `trng::xoshiro256plus` supports block splitting but not leapfrogging.

4.1.6 Mersenne twister generators

The Mersenne twister is a popular random number generator that has been introduced by Makoto Matsumoto and Takuji Nishimura [49]. In TRNG the Mersenne twister comes in two different flavors. The classical Mersenne twister implemented as `trng::mt19937` generates random integers of 32 bits, but there is also a version that generates integers of 64 bits as implemented by `trng::mt19937_64`. These classes are declared in the header files `trng/mt19937.hpp` and `trng/mt19937_64.hpp`. The detailed interfaces of the classes `trng::mt19937` and `trng::mt19937_64` are given as follows:

```
namespace trng {

  class mt19937 {
  public:
    using result_type = unsigned long;
    result_type operator()();
    static constexpr result_type min();
    static constexpr result_type max();

    class parameter_type;
    class status_type;

    mt19937();
    explicit mt19937(unsigned long);
    template<typename gen>
    explicit mt19937(gen &);

    void seed();
    template<typename gen>
    void seed(gen &g);
    void seed(result_type);

    static const char * name();
```

```
    long operator()(long);
  };

  bool operator==(const mt19937 &, const mt19937 &);
  bool operator!=(const mt19937 &, const mt19937 &);
  template<typename char_t, typename traits_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const mt19937 &);
  template<typename char_t, typename traits_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, mt19937 &);
}
```

```
namespace trng {

  class mt19937_64 {
  public:
    using result_type = unsigned long;
    result_type operator()();
    static constexpr result_type min();
    static constexpr result_type max();

    class parameter_type;
    class status_type;

    mt19937_64();
    explicit mt19937_64(unsigned long);
    template<typename gen>
    explicit mt19937_64(gen &);

    void seed();
    void seed(unsigned long);
    template<typename gen>
    void seed(gen &g);
    void seed(result_type);

    static const char * name();
    long operator()(long);
  };

  bool operator==(const mt19937_64 &, const mt19937_64 &);
  bool operator!=(const mt19937_64 &, const mt19937_64 &);
  template<typename char_t, typename traits_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const mt19937_64 &);
  template<typename char_t, typename traits_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, mt19937_64 &);
}
```

4.2 Random number distributions

This section gives a detailed description of all random number distributions, that have been implemented by TRNG. Each subsection presents the public interface of one random number distribution. The part of the public interface, that is mandatory for a random number distribution, will not be discussed in detail, read section 3.2 instead.

Classes for continuous random number distributions are implemented as template classes. The template argument determines the `result_type` and might be either `float`, `double`, or `long double`, where `double` is the default.

Additionally to the requirements in section 3.2 each random number distribution class provides member functions that calculate its probability distribution function, its cumulative distribution function and in the case of continuous distributions its inverse cumulative distribution function as well. These member functions have the signatures

```
result_type pdf(result_type x) const;
result_type cdf(result_type x) const;
result_type icdf(result_type x) const;
```

and for discrete random variables

```
result_type pdf(int x) const;
result_type cdf(int x) const;
```

The concept of a random number distribution requires two functions that take a random number engine as its argument and generate a random variable with some specific distribution by calling `operator()` of the given random number engine. Note, the concept of a random number distribution does not specify how often `operator()` is called. This allows the implementer of a random number distribution to choose between various algorithms [33] that transform uniform random numbers into non-uniform distributed numbers. Some of these algorithms transform exactly one uniform random number into one non-uniform number, while some other algorithms have to call `operator()` more than once. How often `operator()` is called may even vary at runtime. If not otherwise stated, all random number distributions in TRNG are implemented in such a way that `operator()` is called exactly once. Because of this special feature it is much more easy to write parallel Monte Carlo simulations that give the same result (and statistical error) independent of the number of parallel processes. We say such algorithms play fair, see section 2.3 and 6.

4.2.1 Uniform distributions

TRNG provides three different classes for generating uniformly distributed random numbers with distribution function

$$p(x|a,b) = \begin{cases} 1/(b-a) & \text{if } a \le x < b \\ 0 & \text{otherwise}\,. \end{cases}$$

parameters	$a, b \in \mathbb{R}$ with $a < b$
support	$[a, b)$
mean	$(a+b)/2$
variance	$(b-a)^2/12$

The class `uniform_dist` generates random numbers in the range $[a, b)$. Valid parameters for this distribution are $a, b \in \mathbb{R}$ with $a < b$.

Many Monte Carlo simulations consume random numbers uniformly distributed in $[0, 1)$ that can be generated using class `uniform_dist` with parameters $a = 0$ and $b = 1$. However, the uniform distribution in $[0, 1)$ is so common that TRNG has a specialized class `uniform01_dist` for this case. The class `uniform01_dist` might be faster than `uniform_dist` with parameters $a = 0$ and $b = 1$.

Class uniform_int_dist is a variant of uniform_dist for integer valued random variables. It provides random numbers with distribution function

$$p(x|a,b) = \begin{cases} 1/(b-a) & \text{if } a \leq x < b \\ 0 & \text{otherwise} \end{cases} \qquad \text{for } x \in \mathbb{Z}.$$

Valid parameters for this distribution are $a, b \in \mathbb{Z}$ with $a < b$.

The class uniform_dist is declared in the header file trng/uniform_dist.hpp and its public interface is given as follows:

```
namespace trng {

  template<typename float_t=double>
  class uniform_dist {
  public:
    using result_type = float_t;
    class param_type {
    public:
      result_type a() const;
      void a(result_type);
      result_type b() const;
      void b(result_type);
      param_type(result_type a, result_type b);
    };
    uniform_dist(result_type a, result_type b);
    explicit uniform_dist(const param_type &);
    void reset();
    template<typename R>
    result_type operator()(R &);
    template<typename R>
    result_type operator()(R &, const param_type &)
    result_type min() const;
    result_type max() const;
    const param_type & param() const;
    void param(const param_type &);
    result_type a() const;
    void a(result_type);
    result_type b();
    void b(result_type);
    result_type pdf(result_type x) const;
    result_type cdf(result_type x) const;
    result_type icdf(result_type x) const;
  };

  template<typename float_t>
  bool operator==(const typename uniform_dist<float_t>::param_type &,
  const typename uniform_dist<float_t>::param_type &);
  template<typename float_t>
  bool operator!=(const typename uniform_dist<float_t>::param_type &,
  const typename uniform_dist<float_t>::param_type &);

  template<typename char_t, typename traits_t, typename float_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &,
  const typename uniform_dist<float_t>::param_type &);
  template<typename char_t, typename traits_t, typename float_t>
```

```
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &,
  typename uniform_dist<float_t>::param_type &);

  template<typename float_t>
  bool operator==(const uniform_dist<float_t> &, const uniform_dist<float_t> &);
  template<typename float_t>
  bool operator!=(const uniform_dist<float_t> &, const uniform_dist<float_t> &);

  template<typename char_t, typename traits_t, typename float_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const uniform_dist<float_t> &);
  template<typename char_t, typename traits_t, typename float_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, uniform_dist<float_t> &);
}
```

The class `uniform01_dist` is declared in the header file `trng/uniform01_dist.hpp` and its public interface is given as follows:

```
namespace trng {

  template<typename float_t=double>
  class uniform01_dist {
  public:
    using result_type = float_t;
    class param_type;
    uniform01_dist();
    explicit uniform01_dist(const param_type &);
    void reset();
    template<typename R>
    result_type operator()(R &);
    template<typename R>
    result_type operator()(R &, const param_type &);
    result_type min() const;
    result_type max() const;
    const param_type & param() const;
    void param(const param_type &);
    result_type pdf(result_type x) const;
    result_type cdf(result_type x) const;
    result_type icdf(result_type x) const;
  };

  template<typename float_t>
  bool operator==(const typename uniform01_dist<float_t>::param_type &,
  const typename uniform01_dist<float_t>::param_type &);
  template<typename float_t>
  bool operator!=(const typename uniform01_dist<float_t>::param_type &,
  const typename uniform01_dist<float_t>::param_type &);

  template<typename char_t, typename traits_t, typename float_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &,
  const typename uniform01_dist<float_t>::param_type &);
  template<typename char_t, typename traits_t, typename float_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &,
  typename uniform01_dist<float_t>::param_type &);
```

```
  template<typename float_t>
  bool operator==(const uniform01_dist<float_t> &, const uniform01_dist<float_t> &);
  template<typename float_t>
  bool operator!=(const uniform01_dist<float_t> &, const uniform01_dist<float_t> &);

  template<typename char_t, typename traits_t, typename float_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const uniform01_dist<float_t> &)
  template<typename char_t, typename traits_t, typename float_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, uniform01_dist<float_t> &);
}
```

The class `uniform_int_dist` is declared in the header file `trng/uniform_int_dist.hpp` and its public interface is given as follows:

```
namespace trng {

  class uniform_int_dist {
  public:
    typedef int result_type;
    class param_type {
    public:
      result_type a() const;
      void a(result_type);
      result_type b() const;
      void b(result_type);
      param_type(result_type a, result_type b);
    };
    uniform_int_dist(result_type a, result_type b);
    explicit uniform_int_dist(const param_type &)
    void reset();
    template<typename R>
    result_type operator()(R &);
    template<typename R>
    result_type operator()(R &, const param_type &);
    result_type min() const;
    result_type max() const;
    const param_type & param() const;
    void param(const param_type &);
    result_type a() const;
    void a(result_type);
    result_type b() const;
    void b(result_type);
    double pdf(result_type x) const;
    double cdf(result_type x) const;
  };

  bool operator==(const uniform_int_dist::param_type &, const uniform_int_dist::param_type &);
  bool operator!=(const uniform_int_dist::param_type &, const uniform_int_dist::param_type &);

  template<typename char_t, typename traits_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const uniform_int_dist::param_type &);
  template<typename char_t, typename traits_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, uniform_int_dist::param_type &);
```

```
  bool operator==(const uniform_int_dist &, const uniform_int_dist &);
  bool operator!=(const uniform_int_dist &, const uniform_int_dist &);

  template<typename char_t, typename traits_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const uniform_int_dist &);
  template<typename char_t, typename traits_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, uniform_int_dist &);
}
```

4.2.2 Exponential distribution

Class exponential_dist provides random numbers with exponential distribution with mean μ. The probability distribution function reads

$$p(x|\mu) = \begin{cases} \frac{1}{\mu}\mathrm{e}^{-x/\mu} & \text{if } x \geq 0 \\ 0 & \text{otherwise}\,. \end{cases}$$

parameter	$\mu \in \mathbb{R}$ with $\mu > 0$
support	$[0, \infty)$
mean	μ
variance	μ^2

Valid parameter for this distribution is $\mu \in \mathbb{R}$ with $\mu > 0$.

The class exponential_dist is declared in the header file trng/exponential_dist.hpp and its public interface is given as follows:

```
namespace trng {

  template<typename float_t=double>
  class exponential_dist {
  public:
    using result_type = float_t;
    class param_type {
    public:
      result_type mu() const;
      void mu(result_type);
      explicit param_type(result_type mu);
    };
    explicit exponential_dist(result_type mu);
    explicit exponential_dist(const param_type &);
    void reset();
    template<typename R>
    result_type operator()(R &);
    template<typename R>
    result_type operator()(R &, const param_type &);
    result_type min() const;
    result_type max() const;
    const param_type & param() const;
    void param(const param_type &);
    result_type mu() const;
    void mu(result_type);
    result_type pdf(result_type) const;
    result_type cdf(result_type) const;
    result_type icdf(result_type) const;
  };
```

```
  template<typename float_t>
  bool operator==(const typename exponential_dist<float_t>::param_type &,
  const typename exponential_dist<float_t>::param_type &);
  template<typename float_t>
  bool operator!=(const typename exponential_dist<float_t>::param_type &,
  const typename exponential_dist<float_t>::param_type &);

  template<typename char_t, typename traits_t, typename float_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &,
  const typename exponential_dist<float_t>::param_type &);
  template<typename char_t, typename traits_t, typename float_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &,
  typename exponential_dist<float_t>::param_type &);

  template<typename float_t>
  bool operator==(const exponential_dist<float_t> &, const exponential_dist<float_t> &);
  template<typename float_t>
  bool operator!=(const exponential_dist<float_t> &, const exponential_dist<float_t> &);

  template<typename char_t, typename traits_t, typename float_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const exponential_dist<float_t> &);
  template<typename char_t, typename traits_t, typename float_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, exponential_dist<float_t> &);
}
```

4.2.3 Two-sided exponential distribution

Class `twosided_exponential_dist` provides random numbers with two-sided exponential distribution with parameter μ. The probability distribution function reads

$$p(x|\mu) = \frac{1}{2\mu} \mathrm{e}^{-|x|/\mu}$$

parameter	$\mu \in \mathbb{R}$ with $\mu > 0$
support	$(-\infty, \infty)$
mean	0
variance	$2\mu^2$

Valid parameter for this distribution is $\mu \in \mathbb{R}$ with $\mu > 0$.

The class `twosided_exponential_dist` is declared in the header file `trng/twosided_exponential_dist.hpp` and its public interface is given as follows:

```
namespace trng {

  template<typename float_t=double>
  class twosided_exponential_dist {
  public:
    using result_type = float_t;
    class param_type {
    public:
      result_type mu() const;
      void mu(result_type);
      explicit param_type(result_type mu);
    };
```

```
    explicit twosided_exponential_dist(result_type mu);
    explicit twosided_exponential_dist(const param_type &);
    void reset();
    template<typename R>
    result_type operator()(R &);
    template<typename R>
    result_type operator()(R &, const param_type &);
    result_type min() const;
    result_type max() const;
    const param_type & param() const;
    void param(const param_type &);
    result_type mu() const;
    void mu(result_type);
    result_type pdf(result_type) const;
    result_type cdf(result_type) const;
    result_type icdf(result_type) const;
  };

  template<typename float_t>
  bool operator==(const typename exponential_dist<float_t>::param_type &,
  const typename exponential_dist<float_t>::param_type &);
  template<typename float_t>
  bool operator!=(const typename exponential_dist<float_t>::param_type &,
  const typename exponential_dist<float_t>::param_type &);

  template<typename char_t, typename traits_t, typename float_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &,
  const typename exponential_dist<float_t>::param_type &);
  template<typename char_t, typename traits_t, typename float_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &,
  typename exponential_dist<float_t>::param_type &);

  template<typename float_t>
  bool operator==(const exponential_dist<float_t> &, const exponential_dist<float_t> &);
  template<typename float_t>
  bool operator!=(const exponential_dist<float_t> &, const exponential_dist<float_t> &);

  template<typename char_t, typename traits_t, typename float_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const exponential_dist<float_t> &);
  template<typename char_t, typename traits_t, typename float_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, exponential_dist<float_t> &);
}
```

4.2.4 Normal distributions

parameters	$\mu, \sigma \in \mathbb{R}$, with $\sigma > 0$
support	$(-\infty, \infty)$
mean	μ
variance	σ^2

There are two classes for producing random numbers with normal distribution, `normal_dist` and `correlated_normal_dist` . Class `normal_dist` provides uncorrelated random numbers with normal distribution with mean μ and standard deviation σ. The probability distribution function reads

$$p(x|\mu,\sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \mathrm{e}^{-(x-\mu)^2/(2\sigma^2)}\,.$$

Valid parameters for this distribution are $\mu, \sigma \in \mathbb{R}$ with $\sigma > 0$. The normal distribution is also known as Gaussian distribution.

The class `normal_dist` is declared in the header file `trng/normal_dist.hpp` and its public interface is given as follows:

```
namespace trng {

  template<typename float_t=double>
  class normal_dist {
  public:
    using result_type = float_t;
    class param_type {
    public:
      result_type mu() const;
      void mu(result_type);
      result_type sigma() const;
      void sigma(result_type);
      param_type(result_type mu, result_type sigma);
    };
    normal_dist(result_type mu, result_type sigma);
    explicit normal_dist(const param_type &);
    void reset();
    template<typename R>
    result_type operator()(R &);
    template<typename R>
    result_type operator()(R &, const param_type &);
    result_type min() const;
    result_type max() const;
    const param_type & param() const;
    void param(const param_type &);
    result_type mu() const;
    void mu(result_type);
    result_type sigma() const;
    void sigma(result_type);
    result_type pdf(result_type) const;
    result_type cdf(result_type) const;
    result_type icdf(result_type) const;
  };

  template<typename float_t>
  bool operator==(const typename normal_dist<float_t>::param_type &,
  const typename normal_dist<float_t>::param_type &);
  template<typename float_t>
  bool operator!=(const typename normal_dist<float_t>::param_type &,
  const typename normal_dist<float_t>::param_type &);

  template<typename char_t, typename traits_t, typename float_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &,
  const typename normal_dist<float_t>::param_type &);
  template<typename char_t, typename traits_t, typename float_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &,
  typename normal_dist<float_t>::param_type &);
```

```
  template<typename float_t>
  bool operator==(const normal_dist<float_t> &, const normal_dist<float_t> &);
  template<typename float_t>
  bool operator!=(const normal_dist<float_t> &, const normal_dist<float_t> &);

  template<typename char_t, typename traits_t, typename float_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const normal_dist<float_t> &);
  template<typename char_t, typename traits_t, typename float_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, normal_dist<float_t> &);
}
```

If $\mathbf{x} = (x_1, x_2, \ldots x_d)$ are d random variables, then the multivariate normal density function for $\mathbf{x}$ is

$$p(\mathbf{x}|\mathbf{V}) = \frac{1}{\sqrt{(2\pi)^d \det \mathbf{V}}} \exp\left(-\frac{1}{2}\mathbf{x}^T\mathbf{V}^{-1}\mathbf{x}\right) . \tag{4.2}$$

Each variable $x_1, x_2, \ldots x_d$ has mean zero and the the covariance matrix of $x_1, x_2, \ldots x_d$ is given by the symmetric positive definite $d \times d$ matrix $\mathbf{V}$. Class `correlated_normal_dist` provides correlated random numbers with normal distribution by the transformation of uncorrelated random numbers [17].

The class `correlated_normal_dist` is declared in the header file `trng/correlated_normal_dist.hpp` and its public interface is given as follows:

```
namespace trng {

  template<typename float_t=double>
  class correlated_normal_dist {
  public:
    using result_type = float_t;
    class param_type {
    public:
      template<typename iter>
      param_type(iter first, iter last);
    };
    template<typename iter>
    correlated_normal_dist(iter first, iter last);
    explicit correlated_normal_dist(const param_type &);
    void reset();
    template<typename R>
    result_type operator()(R &);
    template<typename R>
    result_type operator()(R &, const param_type &);
    result_type min() const;
    result_type max() const;
    const param_type & param() const;
    void param(const param_type &p_new);
  };

  template<typename float_t>
  bool operator==(const typename correlated_normal_dist<float_t>::param_type &,
  const typename correlated_normal_dist<float_t>::param_type &);
  template<typename float_t>
  bool operator!=(const typename correlated_normal_dist<float_t>::param_type &,
  const typename correlated_normal_dist<float_t>::param_type &);
```

```
  template<typename char_t, typename traits_t, template float_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &,
  const typename correlated_normal_dist<float_t>::param_type &);
  template<typename char_t, typename traits_t, template float_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &,
  typename correlated_normal_dist<float_t>::param_type &);

  template<typename float_t>
  bool operator==(const correlated_normal_dist<float_t> &,
  const correlated_normal_dist<float_t> &);
  template<typename float_t>
  bool operator!=(const correlated_normal_dist<float_t> &,
  const correlated_normal_dist<float_t> &);

  template<typename char_t, typename traits_t, template float_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &,
  const correlated_normal_dist<float_t> &);
  template<typename char_t, typename traits_t, template float_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &,
  correlated_normal_dist<float_t> &);
}
```

The covariance matrix $\mathbf{V}$ has to be passed to the constructor of `correlated_normal_dist` by two iterators. It is not checked, if the matrix is positive definite. The call operator `operator()` returns a single random number and has complexity $\mathcal{O}(d)$. As a consequence, the generation of a tuple of d correlated random numbers takes $\mathcal{O}\left(d^2\right)$ operations.

Successive calls return random numbers with variance $\mathbf{V}_{1,1}$, $\mathbf{V}_{2,2}$ and so on, until the `operator()` has been called d times, which returns a random number with variance $\mathbf{V}_{d,d}$. A sequence of further calls of `operator()` will return random numbers with the same sequences of variances. The method `reset` resets the internal state of the distribution such that, of further calls of `operator()` will return random numbers starting with a number with variance $\mathbf{V}_{1,1}$. Listing 4.1 illustrates the usage of class `correlated_normal_dist`.

Listing 4.1: Demonstration program illustrating the usage of `correlated_normal_dist`.

```
#include <cstdlib>
#include <iostream>
#include <iomanip>
#include <vector>
#include <trng/lcg64.hpp>
#include <trng/correlated_normal_dist.hpp>

double covariance(const std::vector<double> &v1, const std::vector<double> &v2) {
  const std::vector<double>::size_type n{v1.size()};
  double m1{0.0}, m2{0.0}, c{0.0};
  for (std::vector<double>::size_type i{0}; i < n; ++i) {
    m1 += v1[i] / n;
    m2 += v2[i] / n;
  }
  for (std::vector<double>::size_type i{0}; i < n; ++i)
    c += (v1[i] - m1) * (v2[i] - m2) / n;
```

```
  return c;
}

int main() {
  const int d{4};
  // covariance matrix
  const double sig[d][d]{{2.0, -0.5, 0.3, -0.3},
                         {-0.5, 3.0, -0.3, 0.3},
                         {0.3, -0.3, 1.0, -0.3},
                         {-0.3, 0.3, -0.3, 1.0}};
  trng::correlated_normal_dist<> D(&sig[0][0], &sig[d - 1][d - 1] + 1);
  trng::lcg64 R;

  std::vector<double> x1, x2, x3, x4;
  // generate 4-tuples of correlated normal variables
  for (int i{0}; i < 1000000; ++i) {
    x1.push_back(D(R));
    x2.push_back(D(R));
    x3.push_back(D(R));
    x4.push_back(D(R));
  }
  // print (empirical) covariance matrix
  std::cout << std::setprecision(4) << covariance(x1, x1) << '\t' << covariance(x1, x2) << '\t'
            << covariance(x1, x3) << '\t' << covariance(x1, x4) << '\n'
            << covariance(x2, x1) << '\t' << covariance(x2, x2) << '\t' << covariance(x2, x3)
            << '\t' << covariance(x2, x4) << '\n'
            << covariance(x3, x1) << '\t' << covariance(x3, x2) << '\t' << covariance(x3, x3)
            << '\t' << covariance(x3, x4) << '\n'
            << covariance(x4, x1) << '\t' << covariance(x4, x2) << '\t' << covariance(x4, x3)
            << '\t' << covariance(x4, x4) << '\n';
  return EXIT_SUCCESS;
}
```

4.2.5 Truncated normal distribution

parameters	$\mu, \sigma, a, b \in \mathbb{R}$, with $\sigma > 0$, $a < b$
support	$[a, b]$
mean	$\mu + \frac{\phi(\frac{a-\mu}{\sigma}) - \phi(\frac{b-\mu}{\sigma})}{\Phi(\frac{b-\mu}{\sigma}) - \Phi(\frac{a-\mu}{\sigma})}\sigma$
variance	$\sigma^2 \left[1 + \frac{\frac{a-\mu}{\sigma}\phi(\frac{a-\mu}{\sigma}) - \frac{b-\mu}{\sigma}\phi(\frac{b-\mu}{\sigma})}{\Phi(\frac{b-\mu}{\sigma}) - \Phi(\frac{a-\mu}{\sigma})} - \left(\frac{\phi(\frac{a-\mu}{\sigma}) - \phi(\frac{b-\mu}{\sigma})}{\Phi(\frac{b-\mu}{\sigma}) - \Phi(\frac{a-\mu}{\sigma})}\right)^2\right]$

The class `truncated_normal_dist` provides random numbers with a truncated normal distribution with parameters μ, σ, a and b. The probability distribution function reads

$$p(x|\mu, \sigma, a, b) = \frac{\frac{1}{\sigma}\phi\left(\frac{x-\mu}{\sigma}\right)}{\Phi\left(\frac{b-\mu}{\sigma}\right) - \Phi\left(\frac{a-\mu}{\sigma}\right)}$$

where $\phi(x)$ denotes the probability density function of the standard normal distribution and $\Phi(x)$ its cumulative distribution function. Valid parameters for this distribution are $\mu, \sigma, a, b, \in \mathbb{R}$ with $\sigma > 0$ and $a < b$.

The class `truncated_normal_dist` is declared in the header file `trng/truncated_normal_dist.hpp` and its public interface is given as follows:

```
namespace trng {

  template<typename float_t=double>
  class truncated_normal_dist {
  public:
    using result_type = float_t;
    class param_type {
    public:
      result_type mu() const;
      void mu(result_type);
      result_type sigma() const;
      void sigma(result_type);
      result_type a() const;
      void a(result_type);
      result_type b() const;
      void b(result_type);
      param_type(result_type mu, result_type sigma, result_type a, result_type b);
    };
    truncated_normal_dist(result_type mu, result_type sigma,
    result_type a, result_type b);
    explicit truncated_normal_dist(const param_type &);
    void reset();
    template<typename R>
    result_type operator()(R &);
    template<typename R>
    result_type operator()(R &, const param_type &);
    result_type min() const;
    result_type max() const;
    const param_type & param() const;
    void param(const param_type &);
    result_type mu() const;
    void mu(result_type);
    result_type sigma() const;
    void sigma(result_type);
    result_type a() const;
    void a(result_type);
    result_type b() const;
    void b(result_type);
    result_type pdf(result_type) const;
    result_type cdf(result_type) const;
    result_type icdf(result_type) const;
  };

  template<typename float_t>
  bool operator==(const typename truncated_normal_dist<float_t>::param_type &,
  const typename truncated_normal_dist<float_t>::param_type &);
  template<typename float_t>
  bool operator!=(const typename truncated_normal_dist<float_t>::param_type &,
  const typename truncated_normal_dist<float_t>::param_type &);

  template<typename char_t, typename traits_t, typename float_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &,
  const typename truncated_normal_dist<float_t>::param_type &);
  template<typename char_t, typename traits_t, typename float_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &,
```

```
  typename truncated_normal_dist<float_t>::param_type &);

  template<typename float_t>
  bool operator==(const truncated_normal_dist<float_t> &, const truncated_normal_dist<float_t> &);
  template<typename float_t>
  bool operator!=(const truncated_normal_dist<float_t> &, const truncated_normal_dist<float_t> &);

  template<typename char_t, typename traits_t, typename float_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const truncated_normal_dist<float_t> &);
  template<typename char_t, typename traits_t, typename float_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, truncated_normal_dist<float_t> &);

}
```

4.2.6 Maxwell distribution

The class maxwell_dist provides random numbers with Maxwell distribution with the parameter θ. The probability distribution function reads

$$p(x|\theta) = \sqrt{\frac{2}{\pi}}\frac{x^2 \mathrm{e}^{-x^2/(2\theta^2)}}{\theta^3}\,.$$

parameters	$\theta \in \mathbb{R}$, with $\theta > 0$
support	$(0, \infty)$
mean	$2\theta\sqrt{2/\pi}$
variance	$\theta^2(3\pi - 8)/\pi$

Valid parameters for this distribution are $\theta \in \mathbb{R}$ with $\theta > 0$. The Maxwell distribution is also know as Maxwell-Boltzmann distribution.

The class maxwell_dist is declared in the header file trng/maxwell_dist.hpp and its public interface is given as follows:

```
namespace trng {

  template<typename float_t=double>
  class maxwell_dist {
  public:
    using result_type = float_t;
    class param_type {
    public:
      result_type theta() const;
      void theta(result_type);
      param_type(result_type theta);
    };
    maxwell_dist(result_type theta);
    explicit maxwell_dist(const param_type &);
    void reset();
    template<typename R>
    result_type operator()(R &);
    template<typename R>
    result_type operator()(R &, const param_type &);
    result_type min() const;
    result_type max() const;
    const param_type & param() const;
    void param(const param_type &);
    result_type theta() const;
    void theta(result_type);
```

```
    result_type pdf(result_type) const;
    result_type cdf(result_type) const;
    result_type icdf(result_type) const;
  };

  template<typename float_t>
  bool operator==(const typename maxwell_dist<float_t>::param_type &,
  const typename maxwell_dist<float_t>::param_type &);
  template<typename float_t>
  bool operator!=(const typename maxwell_dist<float_t>::param_type &,
  const typename maxwell_dist<float_t>::param_type &);

  template<typename char_t, typename traits_t, typename float_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &,
  const typename maxwell_dist<float_t>::param_type &);
  template<typename char_t, typename traits_t, typename float_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &,
  typename maxwell_dist<float_t>::param_type &);

  template<typename float_t>
  bool operator==(const maxwell_dist<float_t> &, const maxwell_dist<float_t> &);
  template<typename float_t>
  bool operator!=(const maxwell_dist<float_t> &, const maxwell_dist<float_t> &);

  template<typename char_t, typename traits_t, typename float_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const maxwell_dist<float_t> &);
  template<typename char_t, typename traits_t, typename float_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, maxwell_dist<float_t> &);
}
```

4.2.7 Cauchy distribution

The class cauchy_dist provides random numbers with Cauchy distribution with parameters θ and η. The probability distribution function reads

$$p(x|\theta,\eta) = \frac{1}{\theta\pi\left(1+\left(\frac{x-\eta}{\theta}\right)^2\right)}\,.$$

parameters	$\theta, \eta \in \mathbb{R}$, with $\theta > 0$
support	$(-\infty, \infty)$
mean	not defined
variance	not defined

Valid parameters for this distribution are $\theta, \eta \in \mathbb{R}$ with $\theta > 0$. The Cauchy distribution is also know as Lorentz distribution or Breit-Wigner distribution.

The class cauchy_dist is declared in the header file trng/cauchy_dist.hpp and its public interface is given as follows:

```
namespace trng {

  template<typename float_t=double>
  class cauchy_dist {
  public:
    using result_type = float_t;
```

```
    class param_type {
    public:
      result_type theta() const;
      void theta(result_type);
      result_type eta() const;
      void eta(result_type);
      param_type(result_type theta, result_type eta);
    };
    cauchy_dist(result_type theta, result_type eta);
    explicit cauchy_dist(const param_type &);
    void reset();
    template<typename R>
    result_type operator()(R &);
    template<typename R>
    result_type operator()(R &, const param_type &);
    result_type min() const;
    result_type max() const;
    const param_type & param() const;
    void param(const param_type &);
    result_type theta() const;
    void theta(result_type);
    result_type eta() const;
    void eta(result_type);
    result_type pdf(result_type) const;
    result_type cdf(result_type) const;
    result_type icdf(result_type) const;
  };

  template<typename float_t>
  bool operator==(const typename cauchy_dist<float_t>::param_type &,
  const typename cauchy_dist<float_t>::param_type &);
  template<typename float_t>
  bool operator!=(const typename cauchy_dist<float_t>::param_type &,
  const typename cauchy_dist<float_t>::param_type &);

  template<typename char_t, typename traits_t, typename float_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &,
  const typename cauchy_dist<float_t>::param_type &);
  template<typename char_t, typename traits_t, typename float_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &,
  typename cauchy_dist<float_t>::param_type &);

  template<typename float_t>
  bool operator==(const cauchy_dist<float_t> &, const cauchy_dist<float_t> &);
  template<typename float_t>
  bool operator!=(const cauchy_dist<float_t> &, const cauchy_dist<float_t> &);

  template<typename char_t, typename traits_t, typename float_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const cauchy_dist<float_t> &);
  template<typename char_t, typename traits_t, typename float_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, cauchy_dist<float_t> &);
}
```

4.2.8 Logistic distribution

Class `logistic_dist` provides random numbers with Logistic distribution with parameters θ and η. The probability distribution function reads

$$p(x|\theta,\eta) = \frac{e^{-(x-\eta)/\theta}}{\theta\left(1+e^{-(x-\eta)/\theta}\right)^2}\,.$$

parameters	$\theta, \eta \in \mathbb{R}$, with $\theta > 0$
support	$(-\infty, \infty)$
mean	η
variance	$\pi^2\theta^2/3$

Valid parameters for this distribution are $\theta, \eta \in \mathbb{R}$ with $\theta > 0$.

The class `logistic_dist` is declared in the header file `trng/logistic_dist.hpp` and its public interface is given as follows:

```
namespace trng {

  template<typename float_t=double>
  class logistic_dist {
  public:
    typedef double result_type;
    class param_type {
    public:
      result_type theta() const;
      void theta(result_type);
      result_type eta() const;
      void eta(result_type);
      param_type(result_type theta, result_type eta);
    };
    logistic_dist(result_type theta, result_type eta);
    explicit logistic_dist(const param_type &);
    void reset();
    template<typename R>
    result_type operator()(R &);
    template<typename R>
    result_type operator()(R &, const param_type &);
    result_type min() const;
    result_type max() const;
    const param_type & param() const;
    void param(const param_type &);
    result_type theta() const;
    void theta(result_type);
    result_type eta() const;
    void eta(result_type);
    result_type pdf(result_type) const;
    result_type cdf(result_type) const;
    result_type icdf(result_type) const;
  };

  template<typename float_t>
  bool operator==(const typename logistic_dist<float_t>::param_type &,
  const typename logistic_dist<float_t>::param_type &);
  template<typename float_t>
  bool operator!=(const typename logistic_dist<float_t>::param_type &,
  const typename logistic_dist<float_t>::param_type &);

  template<typename char_t, typename traits_t, typename float_t>
  std::basic_ostream<char_t, traits_t> &
```

```
  operator<<(std::basic_ostream<char_t, traits_t> &,
  const typename logistic_dist<float_t>::param_type &);
  template<typename char_t, typename traits_t, typename float_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &,
  typename logistic_dist<float_t>::param_type &);

  template<typename float_t>
  bool operator==(const logistic_dist<float_t> &, const logistic_dist<float_t> &);
  template<typename float_t>
  bool operator!=(const logistic_dist<float_t> &, const logistic_dist<float_t> &);

  template<typename char_t, typename traits_t>
  std::basic_ostream<char_t, traits_t, typename float_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const logistic_dist<float_t> &);
  template<typename char_t, typename traits_t, typename float_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, logistic_dist<float_t> &);
}
```

4.2.9 Lognormal distribution

Class `lognormal_dist` provides random numbers with lognormal distribution with parameters μ and σ. The probability distribution function reads

parameters	$\mu, \sigma \in \mathbb{R}$, with $\sigma > 0$
support	$(0, \infty)$
mean	$\mathrm{e}^{\mu+\sigma^2/2}$
variance	$(\mathrm{e}^{\sigma^2} - 1)\mathrm{e}^{\mu/2+\sigma^2}$

$$p(x|\mu,\sigma) = \begin{cases} 0 & \text{for } x \leq 0 \\ \dfrac{1}{x\sqrt{2\pi\sigma^2}}\mathrm{e}^{-(\ln x-\mu)^2/(2\sigma^2)} & \text{for } x > 0\,. \end{cases}$$

Valid parameters for this distribution are $\mu, \sigma \in \mathbb{R}$ with $\sigma > 0$.

The class `lognormal_dist` is declared in the header file `trng/lognormal_dist.hpp` and its public interface is given as follows:

```
namespace trng {

  template<typename float_t=double>
  class lognormal_dist {
  public:
    using result_type = float_t;
    class param_type {
    public:
      result_type mu() const;
      void mu(result_type);
      result_type sigma() const;
      void sigma(result_type);
      param_type(result_type mu, result_type sigma);
    };
    lognormal_dist(result_type mu, result_type sigma);
    explicit lognormal_dist(const param_type &);
    void reset();
    template<typename R>
    result_type operator()(R &);
    template<typename R>
```

```
    result_type operator()(R &, const param_type &);
    result_type min() const;
    result_type max() const;
    const param_type & param() const;
    void param(const param_type &);
    result_type mu() const;
    void mu(result_type);
    result_type sigma() const;
    void sigma(result_type);
    result_type pdf(result_type) const;
    result_type cdf(result_type) const;
    result_type icdf(result_type) const;
  };

  template<typename float_t>
  bool operator==(const typename lognormal_dist<float_t>::param_type &,
  const typename lognormal_dist<float_t>::param_type &);
  template<typename float_t>
  bool operator!=(const typename lognormal_dist<float_t>::param_type &,
  const typename lognormal_dist<float_t>::param_type &);

  template<typename char_t, typename traits_t, typename float_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &,
  const typename lognormal_dist<float_t>::param_type &);
  template<typename char_t, typename traits_t, typename float_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &,
  typename lognormal_dist<float_t>::param_type &);

  template<typename float_t>
  bool operator==(const lognormal_dist<float_t> &, const lognormal_dist<float_t> &);
  template<typename float_t>
  bool operator!=(const lognormal_dist<float_t> &, const lognormal_dist<float_t> &);

  template<typename char_t, typename traits_t, typename float_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const lognormal_dist<float_t> &);
  template<typename char_t, typename traits_t, typename float_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, lognormal_dist<float_t> &);
}
```

4.2.10 Pareto distribution

Class `pareto_dist` provides random numbers with Pareto distribution with parameters γ and θ. The probability distribution function reads

$$p(x|\gamma,\theta) = \begin{cases} 0 & \text{for } x < 0 \\ \dfrac{\gamma}{\theta}\left(1+\dfrac{x}{\theta}\right)^{-\gamma-1} & \text{for } x \geq 0\,. \end{cases}$$

Valid parameters for this distribution are $\gamma, \theta \in \mathbb{R}$ with $\gamma > 0$ and $\theta > 0$. In the mathematics literature, one

parameters	$\theta, \gamma \in (0,\infty)$
support	$[0,\infty)$
mean	$\theta/(\gamma-1)$
variance	$\dfrac{\theta^2\gamma}{(\gamma-1)^2(\gamma-2)}$

The mean and the variance are defined only if $\gamma > 1$ and $\gamma > 2$, respectively.

can find two different kinds of probability distributions that are referred to as the Pareto distribution. Section 4.2.11 introduces another probability distribution that is also sometimes called the Pareto distribution.

The class `pareto_dist` is declared in the header file `trng/pareto_dist.hpp` and its public interface is given as follows:

```
namespace trng {

  template<typename float_t=double>
  class pareto_dist {
  public:
    using result_type = float_t;
    class param_type {
    public:
      result_type gamma() const;
      void gamma(result_type);
      result_type theta() const;
      void theta(result_type);
      param_type(result_type gamma, result_type theta);
    };
    pareto_dist(result_type gamma, result_type theta);
    explicit pareto_dist(const param_type &);
    void reset();
    template<typename R>
    result_type operator()(R &);
    template<typename R>
    result_type operator()(R &, const param_type &);
    result_type min() const;
    result_type max() const;
    const param_type & param() const;
    void param(const param_type &);
    result_type gamma() const;
    void gamma(result_type);
    result_type theta() const;
    void theta(result_type);
    result_type pdf(result_type) const;
    result_type cdf(result_type) const;
    result_type icdf(result_type) const;
  };

  template<typename float_t>
  bool operator==(const typename pareto_dist<float_t>::param_type &,
  const typename pareto_dist<float_t>::param_type &);
  template<typename float_t>
  bool operator!=(const typename pareto_dist<float_t>::param_type &,
  const typename pareto_dist<float_t>::param_type &);

  template<typename char_t, typename traits_t, typename float_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &,
  const typename pareto_dist<float_t>::param_type &);
  template<typename char_t, typename traits_t, typename float_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &,
  typename pareto_dist<float_t>::param_type &);

  template<typename float_t>
```

```
  bool operator==(const pareto_dist<float_t> &, const pareto_dist<float_t> &);
  template<typename float_t>
  bool operator!=(const pareto_dist<float_t> &, const pareto_dist<float_t> &);

  template<typename char_t, typename traits_t, typename float_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const pareto_dist<float_t> &);
  template<typename char_t, typename traits_t, typename float_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, pareto_dist<float_t> &);
}
```

4.2.11 Power-law distribution

Class powerlaw_dist provides random numbers with power-law distribution with parameters γ and θ. This distribution is related to the Pareto distribution and its probability distribution function reads

$$p(x|\gamma,\theta) = \begin{cases} 0 & \text{for } x < \theta \\ \dfrac{\gamma}{\theta}\left(\dfrac{x}{\theta}\right)^{-\gamma-1} & \text{for } x \geq \theta\,. \end{cases}$$

Valid parameters for this distribution are $\gamma, \theta \in \mathbb{R}$ with $\gamma > 0$ and $\theta > 0$.

parameters	$\theta, \gamma \in (0, \infty)$
support	$[\theta, \infty)$
mean	$\gamma\theta/(\gamma-1)$
variance	$\dfrac{\theta^2\gamma}{(\gamma-1)^2(\gamma-2)}$

The mean and the variance are defined only if $\gamma > 1$ and $\gamma > 2$, respectively.

The class powerlaw_dist is declared in the header file trng/powerlaw_dist.hpp and its public interface is given as follows:

```
namespace trng {

  template<typename float_t=double>
  class powerlaw_dist {
  public:
    using result_type = float_t;
    class param_type {
    public:
      result_type gamma() const;
      void gamma(result_type);
      result_type theta() const;
      void theta(result_type);
      param_type(result_type gamma, result_type theta);
    };
    powerlaw_dist(result_type gamma, result_type theta);
    explicit powerlaw_dist(const param_type &);
    void reset();
    template<typename R>
    result_type operator()(R &);
    template<typename R>
    result_type operator()(R &, const param_type &);
    result_type min() const;
    result_type max() const;
    const param_type & param() const;
    void param(const param_type &);
    result_type gamma() const;
```

```
    void gamma(result_type);
    result_type theta() const;
    void theta(result_type);
    result_type pdf(result_type) const;
    result_type cdf(result_type) const;
    result_type icdf(result_type) const;
  };

  template<typename float_t>
  bool operator==(const typename powerlaw_dist::param_type &,
  const typename powerlaw_dist::param_type &);
  template<typename float_t>
  bool operator!=(const typename powerlaw_dist::param_type &,
  const typename powerlaw_dist::param_type &);

  template<typename char_t, typename traits_t, typename float_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &,
  const typename powerlaw_dist<float_t>::param_type &);
  template<typename char_t, typename traits_t, typename float_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &,
  typename powerlaw_dist<float_t>::param_type &);

  template<typename float_t>
  bool operator==(const powerlaw_dist<float_t> &, const powerlaw_dist<float_t> &);
  template<typename float_t>
  bool operator!=(const powerlaw_dist<float_t> &, const powerlaw_dist<float_t> &);

  template<typename char_t, typename traits_t, typename float_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const powerlaw_dist<float_t> &);
  template<typename char_t, typename traits_t, typename float_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, powerlaw_dist<float_t> &);
}
```

4.2.12 Tent distribution

parameters	$m, d \in \mathbb{R}, d > 0$
support	$(m-d, m+d)$
mean	m
variance	$d^2/6$

Class `tent_dist` provides random numbers with tent distribution with parameters m and d. This distribution is symmetric around m and its support is the interval $(m-d, m+d)$. The probability distribution function reads

$$p(x|m,d) = \begin{cases} \dfrac{1+(x-m)/d}{d} & \text{for } m-d \leq x \leq m \\ \dfrac{1-(x-m)/d}{d} & \text{for } m \leq x \leq m+d \\ 0 & \text{else} \,. \end{cases}$$

Valid parameters for this distribution are $m, d \in \mathbb{R}$ with $d > 0$.

The class `tent_dist` is declared in the header file `trng/tent_dist.hpp` and its public interface is given as follows:

```
namespace trng {

  template<typename float_t=double>
  class tent_dist {
  public:
    using result_type = float_t;
    class param_type {
    public:
      result_type m() const;
      void m(result_type);
      result_type d() const;
      void d(result_type);
      param_type(result_type m, result_type d);
    };
    tent_dist(result_type m, result_type d);
    explicit tent_dist(const param_type &);
    void reset();
    template<typename R>
    result_type operator()(R &);
    template<typename R>
    result_type operator()(R &, const param_type &);
    result_type min() const;
    result_type max() const;
    const param_type & param() const;
    void param(const param_type &);
    result_type m() const;
    void m(result_type);
    result_type d() const;
    void d(result_type);
    result_type pdf(result_type) const;
    result_type cdf(result_type) const;
    result_type icdf(result_type) const;
  };

  template<typename float_t>
  bool operator==(const typename tent_dist<float_t>::param_type &,
  const typename tent_dist<float_t>::param_type &);
  template<typename float_t>
  bool operator!=(const typename tent_dist<float_t>::param_type &,
  const typename tent_dist<float_t>::param_type &);

  template<typename char_t, typename traits_t, typename float_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &,
  const typename tent_dist<float_t>::param_type &);
  template<typename char_t, typename traits_t, typename float_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &,
  typename tent_dist<float_t>::param_type &);

  template<typename float_t>
  bool operator==(const tent_dist<float_t> &, const tent_dist<float_t> &);
  template<typename float_t>
  bool operator!=(const tent_dist<float_t> &, const tent_dist<float_t> &);

  template<typename char_t, typename traits_t, typename float_t>
  std::basic_ostream<char_t, traits_t> &
```

```
  operator<<(std::basic_ostream<char_t, traits_t> &, const tent_dis<float_t>t &);
  template<typename char_t, typename traits_t, typename float_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, tent_dist<float_t> &);
}
```

4.2.13 Weibull distribution

Class `weibull_dist` provides random numbers with Weibull distribution with parameters β and θ. The probability distribution function reads

$$p(x|\theta,\beta) = \begin{cases} 0 & \text{for } x < \theta \\ \dfrac{\beta}{\theta}\left(\dfrac{x}{\theta}\right)^{\beta-1} \mathrm{e}^{-(x/\theta)^\beta} & \text{for } x \geq \theta\,. \end{cases}$$

parameters	$\beta, \theta \in (0, \infty)$
support	$(0, \infty)$
mean	$\theta\Gamma\left(1+\frac{1}{\beta}\right)$
variance	$\theta^2\left[\Gamma\left(1+\frac{2}{\beta}\right) - \Gamma^2\left(1+\frac{1}{\beta}\right)\right]$

Valid parameters for this distribution are $\theta, \beta \in \mathbb{R}$ with $\theta > 0$ and $\beta > 0$. For $\beta = 1$ Weibull distribution degenerates to an exponential distribution and for $\beta = 2$ and $\theta = \sqrt{2}\cdot\sigma$ this distribution is also known as Rayleigh distribution with parameter σ.

The class `weibull_dist` is declared in the header file `trng/weibull_dist.hpp` and its public interface is given as follows:

```
namespace trng {

  template<typename float_t=double>
  class weibull_dist {
  public:
    using result_type = float_t;
    class param_type {
    public:
      result_type theta() const;
      void theta(result_type);
      result_type beta() const;
      void beta(result_type);
      param_type(result_type theta, result_type beta);
    };
    weibull_dist(result_type theta, result_type beta);
    explicit weibull_dist(const param_type &);
    void reset();
    template<typename R>
    result_type operator()(R &);
    template<typename R>
    result_type operator()(R &, const param_type &);
    result_type min() const;
    result_type max() const;
    const param_type & param() const;
    void param(const param_type &);
    result_type beta() const;
    void beta(result_type);
    result_type theta() const;
    void theta(result_type);
    result_type pdf(result_type) const;
    result_type cdf(result_type) const;
    result_type icdf(result_type) const;
```

```
  };

  template<typename float_t>
  bool operator==(const typename weibull_dist<float_t>::param_type &,
  const typename weibull_dist<float_t>::param_type &);
  template<typename float_t>
  bool operator!=(const typename weibull_dist<float_t>::param_type &,
  const typename weibull_dist<float_t>::param_type &);

  template<typename char_t, typename traits_t, typename float_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &,
  const typename weibull_dist<float_t>::param_type &);
  template<typename char_t, typename traits_t, typename float_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &,
  typename weibull_dist<float_t>::param_type &);

  template<typename float_t>
  bool operator==(const weibull_dist<float_t> &, const weibull_dist<float_t> &);
  template<typename float_t>
  bool operator!=(const weibull_dist<float_t> &, const weibull_dist<float_t> &);

  template<typename char_t, typename traits_t, typename float_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const weibull_dist<float_t> &);
  template<typename char_t, typename traits_t, typename float_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, weibull_dist<float_t> &);
}
```

4.2.14 Extreme value distribution

Class `extreme_value_dist` provides random numbers with extreme value distribution (also known as Gumbel distribution) with parameters θ and η. The probability distribution function reads

$$p(x|\theta, \eta) = \frac{1}{\theta} \exp\left(\frac{\eta - x}{\theta} - \exp\frac{\eta - x}{\theta}\right) .$$

parameters	$\theta, \eta \in \mathbb{R}, \theta > 0$
support	$(-\infty, \infty)$
mean	$\eta - \gamma\theta$
variance	$\pi^2\theta^2/6$

γ denotes the Euler-Mascheroni constant $\gamma = 0.57721\ldots$

Valid parameters for this distribution are $\theta, \eta \in \mathbb{R}$ with $\theta > 0$.

The class `extreme_value_dist` is declared in the header file `trng/extreme_value_dist.hpp` and its public interface is given as follows:

```
namespace trng {

  template<typename float_t=double>
  class extreme_value_dist {
  public:
    using result_type = float_t;
    class param_type {
    public:
      result_type theta() const;
      void theta(result_type);
```

```
      result_type eta() const;
      void eta(result_type);
      param_type(result_type theta, result_type eta);
    };
    extreme_value_dist(result_type theta, result_type eta);
    explicit extreme_value_dist(const param_type &);
    void reset();
    template<typename R>
    result_type operator()(R &);
    template<typename R>
    result_type operator()(R &, const param_type &);
    result_type min() const;
    result_type max() const;
    const param_type & param() const;
    void param(const param_type &);
    result_type theta() const;
    void theta(result_type);
    result_type eta() const;
    void eta(result_type);
    result_type pdf(result_type) const;
    result_type cdf(result_type) const;
    result_type icdf(result_type) const;
  };

  template<typename float_t>
  bool operator==(const typename extreme_value_dist<float_t>::param_type &,
  const typename extreme_value_dist<float_t>::param_type &);
  template<typename float_t>
  bool operator!=(const typename extreme_value_dist<float_t>::param_type &,
  const typename extreme_value_dist<float_t>::param_type &);

  template<typename char_t, typename traits_t, typename float_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &,
  const typename extreme_value_dist<float_t>::param_type &);
  template<typename char_t, typename traits_t, typename float_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &,
  typename extreme_value_dist<float_t>::param_type &);

  template<typename float_t>
  bool operator==(const extreme_value_dist<float_t> &, const extreme_value_dist<float_t> &);
  template<typename float_t>
  bool operator!=(const extreme_value_dist<float_t> &, const extreme_value_dist<float_t> &);

  template<typename char_t, typename traits_t, typename float_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const extreme_value_dist<float_t> &);
  template<typename char_t, typename traits_t, typename float_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, extreme_value_dist<float_t> &);
}
```

Note that the definition of the extreme value distribution differs slightly from the one that has been introduced in C++11, see also section 6.4 and [28]. However, it is not difficult to switch from the C++ standard library to TRNG and vice versa. More precisely

```
trng::extreme_value_dist<> D1(theta, eta);
std::extreme_value_distribution<> D2(eta, -theta);
```

yield two equivalent distributions.

4.2.15 Γ-distribution

Class gamma_dist provides random numbers with Γ-distribution with parameters θ and κ. The probability distribution function reads

parameters	$\kappa, \theta \in (0, \infty)$
support	$[0, \infty)$
mean	$\kappa\theta$
variance	$\kappa\theta^2$

$$p(x|\theta,\kappa) = \begin{cases} 0 & \text{if } x < 0 \\ \dfrac{1}{\theta\Gamma(\kappa)} \left(\dfrac{x}{\theta}\right)^{\kappa-1} \mathrm{e}^{-x/\theta} & \text{if } x \geq 0\,. \end{cases}$$

Valid parameters for this distribution are $\kappa, \theta \in \mathbb{R}$ with $\kappa \geq 1$ and $\theta > 0$. Note, Γ-distribution is defined for arbitrary $\kappa \geq 0$, but class gamma_dist can handle only Γ-distributions with $\kappa \geq 1$ correctly. For $\kappa = 1$ the Γ-distribution degenerates to an exponential distribution.

The class gamma_dist is declared in the header file trng/gamma_dist.hpp and its public interface is given as follows:

```
namespace trng {

  template<typename float_t=double>
  class gamma_dist {
  public:
    using result_type = float_t;
    class param_type {
    public:
      result_type kappa() const;
      void kappa(result_type);
      result_type theta() const;
      void theta(result_type);
      param_type(result_type kappa, result_type theta);
    };
    gamma_dist(result_type kappa, result_type theta);
    explicit gamma_dist(const param_type &);
    void reset();
    template<typename R>
    result_type operator()(R &);
    template<typename R>
    result_type operator()(R &, const param_type &);
    result_type min() const;
    result_type max() const;
    const param_type & param() const;
    void param(const param_type &);
    result_type kappa() const;
    void kappa(result_type);
    result_type theta() const;
    void theta(result_type);
    result_type pdf(result_type) const;
    result_type cdf(result_type) const;
    result_type icdf(result_type) const;
  };
```

```
  template<typename float_t>
  bool operator==(const typename gamma_dist<float_t>::param_type &,
  const typename gamma_dist<float_t>::param_type &);
  template<typename float_t>
  bool operator!=(const typename gamma_dist<float_t>::param_type &,
  const typename gamma_dist<float_t>::param_type &);

  template<typename char_t, typename traits_t, typename float_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &,
  const typename gamma_dist<float_t>::param_type &);
  template<typename char_t, typename traits_t, typename float_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &,
  typename gamma_dist<float_t>::param_type &);

  template<typename float_t>
  bool operator==(const gamma_dist<float_t> &, const gamma_dist<float_t> &);
  template<typename float_t>
  bool operator!=(const gamma_dist<float_t> &, const gamma_dist<float_t> &);

  template<typename char_t, typename traits_t, typename float_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const gamma_dist<float_t> &);
  template<typename char_t, typename traits_t, typename float_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, gamma_dist<float_t> &);
}
```

4.2.16 B-distribution

Class `beta_dist` provides random numbers with B-distribution with parameters α and β. The probability distribution function reads with the Beta function $\mathrm{B}(\alpha, \beta)$

$$p(x|\alpha,\beta) = \begin{cases} 0 & \text{if } x < 0 \text{ or } x > 1 \\ \dfrac{1}{\mathrm{B}(\alpha,\beta)} x^{\alpha-1}(1-x)^{\beta-1} & \text{else}\,. \end{cases}$$

parameters	$\alpha, \beta \in (0, \infty)$
support	$[0, 1]$
mean	$\alpha/(\alpha+\beta)$
variance	$\alpha\beta/(\alpha+\beta+1)/(\alpha+\beta)^2$

Valid parameters for this distribution are $\alpha, \beta \in \mathbb{R}$ with $\alpha > 0$ and $\beta > 0$.

The class `beta_dist` is declared in the header file `trng/beta_dist.hpp` and its public interface is given as follows:

```
namespace trng {

  template<typename float_t=double>
  class beta_dist {
  public:
    using result_type = float_t;
    class param_type {
    public:
      result_type alpha() const;
      void alpha(result_type);
```

```
      result_type beta() const;
      void beta(result_type);
      param_type(result_type alpha, result_type beta);
    };
    beta_dist(result_type alpha, result_type beta);
    explicit beta_dist(const param_type &);
    void reset();
    template<typename R>
    result_type operator()(R &);
    template<typename R>
    result_type operator()(R &, const param_type &);
    result_type min() const;
    result_type max() const;
    const param_type & param() const;
    void param(const param_type &);
    result_type alpha() const;
    void alpha(result_type);
    result_type beta() const;
    void beta(result_type);
    result_type pdf(result_type) const;
    result_type cdf(result_type) const;
    result_type icdf(result_type) const;
  };

  template<typename float_t>
  bool operator==(const typename beta_dist<float_t>::param_type &,
  const typename beta_dist<float_t>::param_type &);
  template<typename float_t>
  bool operator!=(const typename beta_dist<float_t>::param_type &,
  const typename beta_dist<float_t>::param_type &);

  template<typename char_t, typename traits_t, typename float_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &,
  const typename beta_dist<float_t>::param_type &);
  template<typename char_t, typename traits_t, typename float_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &,
  typename beta_dist<float_t>::param_type &);

  template<typename float_t>
  bool operator==(const beta_dist<float_t> &, const beta_dist<float_t> &);
  template<typename float_t>
  bool operator!=(const beta_dist<float_t> &, const beta_dist<float_t> &);

  template<typename char_t, typename traits_t, typename float_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const beta_dist<float_t> &);
  template<typename char_t, typename traits_t, typename float_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, beta_dist<float_t> &);
}
```

4.2.17 χ^2-distribution

parameter	$\nu \in \mathbb{N}$
support	$(0, \infty)$
mean	ν
variance	2ν

Class chi_square_dist provides random numbers with χ^2-distribution with ν degrees of freedom. The probability distribution function reads

$$p(x|\nu) = \begin{cases} 0 & \text{if } x < 0 \\ \dfrac{x^{\nu/2-1} \mathrm{e}^{-x/2}}{2^{\nu/2}\,\Gamma(\nu/2)} & \text{if } x \geq 0\,. \end{cases}$$

A valid parameter for this distribution is $\nu \in \mathbb{N}$ with $\nu \geq 1$. Note, χ^2-distribution is a special case of Γ-distribution with $\kappa = \nu/2$ and $\theta = 2$.

The class chi_square_dist is declared in the header file trng/chi_square_dist.hpp and its public interface is given as follows:

```
namespace trng {

  template<typename float_t=double>
  class chi_square_dist {
  public:
    using result_type = float_t;
    class param_type {
    public:
      int nu() const;
      void nu(int);
      explicit param_type(int nu);
    };
    explicit chi_square_dist(int nu);
    explicit chi_square_dist(const param_type &);
    void reset();
    template<typename R>
    result_type operator()(R &);
    template<typename R>
    result_type operator()(R &, const param_type &);
    result_type min() const;
    result_type max() const;
    const param_type & param() const;
    void param(const param_type &);
    int nu() const;
    void nu(int);
    result_type pdf(result_type) const;
    result_type cdf(result_type) const;
    result_type icdf(result_type) const;
  };

  template<typename float_t>
  bool operator==(const typename chi_square_dist<float_t>::param_type &,
  const typename chi_square_dist<float_t>::param_type &);
  template<typename float_t>
  bool operator!=(const typename chi_square_dist<float_t>::param_type &,
  const typename chi_square_dist<float_t>::param_type &);

  template<typename char_t, typename traits_t, typename float_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &,
  const typename chi_square_dist<float_t>::param_type &);
  template<typename char_t, typename traits_t, typename float_t>
  std::basic_istream<char_t, traits_t> &
```

```
  operator>>(std::basic_istream<char_t, traits_t> &,
  typename chi_square_dist<float_t>::param_type &);

  template<typename float_t>
  bool operator==(const chi_square_dist<float_t> &, const chi_square_dist<float_t> &);
  template<typename float_t>
  bool operator!=(const chi_square_dist<float_t> &, const chi_square_dist<float_t> &);

  template<typename char_t, typename traits_t, typename float_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const chi_square_dist<float_t> &);
  template<typename char_t, typename traits_t, typename float_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, chi_square_dis<float_t>t &);
}
```

4.2.18 Student-t distribution

Class `student_t_dist` provides random numbers with Student-t distribution with ν degrees of freedom. The probability distribution function reads

$$p(x|\nu) = \frac{\Gamma(\frac{\nu+1}{2})}{\sqrt{\nu\pi}\,\Gamma(\frac{\nu}{2})}\left(1+\frac{x^2}{\nu}\right)^{-(\frac{\nu+1}{2})} .$$

parameter	$\nu \in \mathbb{N}$
support	$(-\infty, \infty)$
mean	0
variance	$\frac{\nu-1}{\nu-3}$

A valid parameter for this distribution is $\nu \in \mathbb{N}$ with $\nu \geq 1$.

The class `student_t_dist` is declared in the header file `trng/student_t_dist.hpp` and its public interface is given as follows:

```
namespace trng {

  template<typename float_t=double>
  class student_t_dist {
  public:
    using result_type = float_t;
    class param_type {
    public:
      int nu() const;
      void nu(int);
      explicit param_type(int nu);
    };
    explicit student_t_dist(int nu);
    explicit student_t_dist(const param_type &);
    void reset();
    template<typename R>
    result_type operator()(R &);
    template<typename R>
    result_type operator()(R &, const param_type &);
    result_type min() const;
    result_type max() const;
    const param_type & param() const;
    void param(const param_type &);
    int nu() const;
    void nu(int);
    result_type pdf(result_type) const;
```

```
    result_type cdf(result_type) const;
    result_type icdf(result_type) const;
  };

  template<typename float_t>
  bool operator==(const typename student_t_dist<float_t>::param_type &,
  const typename student_t_dist<float_t>::param_type &);
  template<typename float_t>
  bool operator!=(const typename student_t_dist<float_t>::param_type &,
  const typename student_t_dist<float_t>::param_type &);

  template<typename char_t, typename traits_t, typename float_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &,
  const typename student_t_dist<float_t>::param_type &);
  template<typename char_t, typename traits_t, typename float_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &,
  typename student_t_dist<float_t>::param_type &);

  template<typename float_t>
  bool operator==(const student_t_dist &, const student_t_dist<float_t> &);
  template<typename float_t>
  bool operator!=(const student_t_dist &, const student_t_dist<float_t> &);

  template<typename char_t, typename traits_t, typename float_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const student_t_dist<float_t> &);
  template<typename char_t, typename traits_t, typename float_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, student_t_dist<float_t> &);
}
```

4.2.19 Snedecor-F distribution

Class snedecor_fsnedecor_f_dist provides random numbers with Snedecor-F distribution (or Fisher-Snedecor distribution) with parameters n and m. The probability distribution function reads

parameter	$n, m \in \mathbb{N}$
support	$[0, \infty)$
mean	$\frac{m}{m-2}$
variance	$\frac{2m^2(m+n-2)}{n(m-2)^2(m-4)}$

$$p(x|n,m) = \begin{cases} 0 & \text{if } x < 0 \\ \dfrac{\Gamma((n+m)/2)}{\Gamma(n/2)\Gamma(m/2)} \dfrac{n^{n/2} m^{m/2} x^{n/2-1}}{(m+nx)^{(n+m)/2}} & \text{if } x \geq 0\,. \end{cases}$$

Valid parameters for this distribution are $n, m \in \mathbb{N}$ with $n, m \geq 1$.

The class `snedecor_f_dist` is declared in the header file `trng/snedecor_f_dist.hpp` and its public interface is given as follows:

```
namespace trng {

  template<typename float_t=double>
  class snedecor_f_dist {
  public:
    using result_type = float_t;
```

```
    class param_type {
    public:
      int n() const;
      void n(int);
      int m() const;
      void m(int);
      param_type(int n, int m);
    };
    snedecor_f_dist(int n, int m);
    explicit snedecor_f_dist(const param_type &);
    void reset();
    template<typename R>
    result_type operator()(R &);
    template<typename R>
    result_type operator()(R &, const param_type &);
    result_type min() const;
    result_type max() const;
    const param_type & param() const;
    void param(const param_type &);
    int n() const;
    void n(int);
    int m() const;
    void m(int);
    result_type pdf(result_type) const;
    result_type cdf(result_type) const;
    result_type icdf(result_type) const;
  };

  template<typename float_t>
  bool operator==(const typename snedecor_f_dist<float_t>::param_type &,
  const typename snedecor_f_dist<float_t>::param_type &);
  template<typename float_t>
  bool operator!=(const typename snedecor_f_dist<float_t>::param_type &,
  const typename snedecor_f_dist<float_t>::param_type &);

  template<typename char_t, typename traits_t, typename float_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &,
  const typename snedecor_f_dist<float_t>::param_type &);
  template<typename char_t, typename traits_t, typename float_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &,
  typename snedecor_f_dist<float_t>::param_type &);

  template<typename float_t>
  bool operator==(const snedecor_f_dist<float_t> &, const snedecor_f_dist<float_t> &);
  template<typename float_t>
  bool operator!=(const snedecor_f_dist<float_t> &, const snedecor_f_dist<float_t> &);

  template<typename char_t, typename traits_t, typename float_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const snedecor_f_dist<float_t> &);
  template<typename char_t, typename traits_t, typename float_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, snedecor_f_dist<float_t> &);
}
```

4.2.20 Rayleigh distribution

Class `rayleigh_dist` provides random numbers with Rayleigh distribution with parameter ν. The probability distribution function reads

$$p(x|\nu) = \begin{cases} 0 & \text{if } x \leq 0 \\ \dfrac{x}{\nu^2}\mathrm{e}^{-x^2/(2\nu^2)} & \text{if } x > 0\,. \end{cases}$$

parameter	$\nu \in (0, \infty)$
support	$(0, \infty)$
mean	$\nu\sqrt{\pi/2}$
variance	$(4-\pi)\nu^2/2$

A valid parameter for this distribution is $\nu > 0$.

The class `rayleigh_dist` is declared in the header file `trng/rayleigh_dist.hpp` and its public interface is given as follows:

```
namespace trng {

  template<typename float_t=double>
  class rayleigh_dist {
  public:
    using result_type = float_t;
    class param_type {
    public:
      result_type nu() const;
      void nu(result_type nu_new);
      explicit param_type(result_type nu);
    };

    explicit rayleigh_dist(result_type nu);
    explicit rayleigh_dist(const param_type &);
    void reset();
    template<typename R>
    result_type operator()(R &);
    template<typename R>
    result_type operator()(R &, const param_type &);
    result_type min() const;
    result_type max() const;
    param_type param() const { return p; }
    void param(const param_type &);
    result_type nu() const;
    void nu(result_type);
    result_type pdf(result_type x) const;
    result_type cdf(result_type x) const;
    result_type icdf(result_type x) const;
  };

  template<typename float_t>
  bool operator==(const typename rayleigh_dist<float_t>::param_type &,
  const typename rayleigh_dist<float_t>::param_type &);
  template<typename float_t>
  bool operator!=(const typename rayleigh_dist<float_t>::param_type &,
  const typename rayleigh_dist<float_t>::param_type &);

  template<typename char_t, typename traits_t, typename float_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &,
  const typename rayleigh_dist<float_t>::param_type &);
  template<typename char_t, typename traits_t, typename float_t>
```

```
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &,
  typename rayleigh_dist<float_t>::param_type &);

  template<typename float_t>
  bool operator==(const rayleigh_dist<float_t> &, const rayleigh_dist<float_t> &);
  template<typename float_t>
  bool operator!=(const rayleigh_dist<float_t> &, const rayleigh_dist<float_t> &);

  template<typename char_t, typename traits_t, typename float_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const rayleigh_dist<float_t> &);
  template<typename char_t, typename traits_t, typename float_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, rayleigh_dist<float_t> &);
}
```

4.2.21 Bernoulli distribution

The template class bernoulli_dist provides random objects with Bernoulli distribution with parameter p. The probability distribution function reads

$$P(x|p) = \begin{cases} p & \text{if } x = 0 \text{ (head)} \\ 1-p & \text{if } x = 1 \text{ (tail)} \\ 0 & \text{else}\,. \end{cases}$$

parameter	$p \in [0,1]$
support	$0, 1$
mean	$p/2$
variance	$p^2/12$

A valid parameter for this distribution is $p \in [0,1]$. In contrast to other random distribution classes any default-constructible type (not only floating point types) may be utilized for the template parameter T.

The class bernoulli_dist is declared in the header file trng/bernoulli_dist.hpp and its public interface is given as follows:

```
namespace trng {

  template<typename T>
  class bernoulli_dist {
  public:
    typedef T result_type;

    class param_type {
    public:
      double p() const;
      void p(double);
      T head() const;
      void head(const T &);
      T tail() const;
      void tail(const T &);
      param_type(double p, const T &head, const T &tail);
    };
```

The one-parameter constructor bernoulli_dist(double p) initializes “head” to 0 (or false if T is bool) and “tail” to 1 (or true if T is bool) if T is an arithmetic type, i. e., either a floating point type, an integer type or bool. Using the one-parameter constructor with a non-arithmetic type T leads to compile-time errors.

```
    explicit bernoulli_dist(double p);
    explicit bernoulli_dist(double p, const T &head, const T &tail);
    explicit bernoulli_dist(const param_type &);
    void reset();
    template<typename R>
    T operator()(R &);
    template<typename R>
    T operator()(R &, const param_type &);
```

Method `min` returns "head" and method `max` returns "tail".

```
    T min() const;
    T max() const;
    const param_type & param() const;
    void param(const param_type &);
    double p() const;
    void p(double);
    T head() const;
    void head(const T &);
    T tail() const;
    void tail(const T &);
```

Method `pdf` will return p if its argument is "head", $1 - p$ if its argument is "tail" and 0 otherwise.

```
    double pdf(const T &) const;
```

Method `cdf` will return p if its argument is "head", 1 if its argument is "tail" and 0 otherwise.

```
    double cdf(const T &) const;
  };

  template<typename T>
  bool operator==(const typename bernoulli_dist<T>::param_type &,
  const typename bernoulli_dist<T>::param_type &);
  template<typename T>
  bool operator!=(const typename bernoulli_dist<T>::param_type &,
  const typename bernoulli_dist<T>::param_type &);

  template<typename char_t, typename traits_t, typename T>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &,
  const typename bernoulli_dist<T>::param_type &);
  template<typename char_t, typename traits_t, typename T>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &,
  typename bernoulli_dist<T>::param_type &);

  template<typename T>
  bool operator==(const bernoulli_dist<T> &, const bernoulli_dist<T> &);
  template<typename T>
  bool operator!=(const bernoulli_dist<T> &, const bernoulli_dist<T> &);

  template<typename char_t, typename traits_t, typename T>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const bernoulli_dist<T> &);
  template<typename char_t, typename traits_t, typename T>
  std::basic_istream<char_t, traits_t> &
```

```
  operator>>(std::basic_istream<char_t, traits_t> &, bernoulli_dist<T> &);
}
```

Listing 4.2: Class bernoulli_dist in action.

```
#include <cstdlib>
#include <iostream>
#include <iomanip>
#include <vector>
#include <trng/lcg64.hpp>
#include <trng/bernoulli_dist.hpp>

typedef enum { head = 0, tail = 1 } coin;

int main() {
  // discrete distribution object
  trng::bernoulli_dist<coin> biased_coin(0.51, head, tail);
  // random number generator
  trng::lcg64 r;
  // draw some random numbers
  std::vector<int> count(2, 0);
  const int samples{100000};
  for (int i = 0; i < samples; ++i) {
    const int x{biased_coin(r)};  // draw a random number
    ++count[x];                   // count
  }
  // print results
  std::cout << "value\t\tprobability\tcount\t\tempirical probability\n"
            << "=====\t\t===========\t=====\t\t=====================\n";
  for (std::vector<int>::size_type i = 0; i < count.size(); ++i)
    std::cout << std::setprecision(3) << i << "\t\t" << biased_coin.pdf(static_cast<coin>(i))
              << "\t\t" << count[i] << "\t\t" << static_cast<double>(count[i]) / samples
              << '\n';
  return EXIT_SUCCESS;
}
```

4.2.22 Binomial distribution

Class binomial_dist provides random integers with binomial distribution with parameters p and n. The probability distribution function reads

$$P(x|p,n) = \begin{cases} \binom{n}{x} p^x (1-p)^{n-x} & \text{if } x \in \{0,1,\ldots,n\} \\ 0 & \text{else}\,. \end{cases}$$

parameters	$p \in [0,1]$, $n \in \mathbb{N}$
support	$0, 1, \ldots, n$
mean	np
variance	$np(1-p)$

Valid parameters for this distribution are $p \in [0,1]$ and $n \in \mathbb{N}$.

The class binomial_dist is declared in the header file trng/binomial_dist.hpp and its public interface is given as follows:

```
namespace trng {
```

```
  class binomial_dist {
  public:
    typedef int result_type;

    class param_type {
    public:
      double p() const;
      void p(double);
      int n() const;
      void n(int);
      param_type(double p, int n);
    };

    binomial_dist(double p, int n);
    explicit binomial_dist(const param_type &);
    void reset();
    template<typename R>
    int operator()(R &);
    template<typename R>
    int operator()(R &, const param_type &);
    int min() const;
    int max() const;
    const param_type & param() const;
    void param(const param_type &);
    double p() const;
    void p(double);
    int n() const;
    void n(int);
    double pdf(int) const;
    double cdf(int) const;
  };

  bool operator==(const binomial_dist::param_type &, const binomial_dist::param_type &);
  bool operator!=(const binomial_dist::param_type &, const binomial_dist::param_type &);

  template<typename char_t, typename traits_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const binomial_dist::param_type &);
  template<typename char_t, typename traits_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, binomial_dist::param_type &);

  bool operator==(const binomial_dist &, const binomial_dist &);
  bool operator!=(const binomial_dist &, const binomial_dist &);

  template<typename char_t, typename traits_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const binomial_dist &);
  template<typename char_t, typename traits_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, binomial_dist &);
}
```

4.2.23 Negative binomial distribution

Class `negative_binomial_dist` provides random integers with negative binomial distribution with parameters p and r. This distribution is also known as gamma–Poisson (mixture) distribution. The probability distribution function reads

parameters	$p \in [0,1], r \in \mathbb{N}$
support	$0, 1, \ldots$
mean	$r(1-p)/p$
variance	$r(1-p)/p^2$

$$P(x|p,r) = \begin{cases} \dfrac{\Gamma(r+x)}{x!\Gamma(r)} p^r (1-p)^x & \text{if } x \in \{0,1,\ldots\} \\ 0 & \text{else}\,. \end{cases}$$

Valid parameters for this distribution are $p \in [0,1]$ and $r \in (0,\infty)$.

The class `negative_binomial_dist` is declared in the header file `trng/negative_binomial_dist.hpp` and its public interface is given as follows:

```
namespace trng {

  class negative_binomial_dist {
  public:
    typedef int result_type;

    class param_type {
    public:
      double p() const;
      void p(double);
      int r() const;
      void r(int);
      param_type(double p, double r);
    };

    negative_binomial_dist(double p, double r);
    explicit negative_binomial_dist(const param_type &);
    void reset();
    template<typename R>
    int operator()(R &);
    template<typename R>
    int operator()(R &, const param_type &);
    int min() const;
    int max() const;
    const param_type & param() const;
    void param(const param_type &);
    double p() const;
    void p(double);
    double r() const;
    void r(double);
    double pdf(int) const;
    double cdf(int) const;
  };

  bool operator==(const negative_binomial_dist::param_type &,
  const negative_binomial_dist::param_type &);
  bool operator!=(const negative_binomial_dist::param_type &,
  const negative_binomial_dist::param_type &);

  template<typename char_t, typename traits_t>
```

```
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const negative_binomial_dist::param_type &);
  template<typename char_t, typename traits_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, negative_binomial_dist::param_type &);

  bool operator==(const negative_binomial_dist &, const negative_binomial_dist &);
  bool operator!=(const negative_binomial_dist &, const negative_binomial_dist &);

  template<typename char_t, typename traits_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const negative_binomial_dist &);
  template<typename char_t, typename traits_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, negative_binomial_dist &);
}
```

4.2.24 Hypergeometric distribution

Class hypergeometric_dist provides random integers with hypergeometric distribution with parameters n, m and d. The probability distribution function reads

parameters	$n \in \mathbb{N}, m \in \{0,1,\ldots,n\}, d \in \{1,2,\ldots,n\}$
support	$\max(0,d-n+m),\ldots,\min(d,m)$
mean	dm/n
variance	$d\frac{m}{n}\left(1-\frac{m}{n}\right)\frac{n-d}{n-1}$

$$P(x|n,m,d) = \begin{cases} \dfrac{\binom{m}{x}\binom{n-m}{d-x}}{\binom{n}{d}} & \text{if } x \in \{\max(0,d-n+m),\ldots,\min(d,m)\}\,, \\ 0 & \text{else}\,. \end{cases}$$

Valid parameters for this distribution are $n \in \mathbb{N}$, $m \in \{0,1,\ldots,n\}$, and $d \in \{1,2,\ldots,n\}$,

The class hypergeometric_dist is declared in the header file trng/hypergeometric_dist.hpp and its public interface is given as follows:

```
namespace trng {

  class hypergeometric_dist {
  public:
    typedef int result_type;

    class param_type {
    public:
      int n() const;
      void n(int);
      int m() const;
      void m(int);
      int d() const;
      void d(int);
      param_type(int n, int m, int d);
    };
```

```
    hypergeometric_dist(double n, int m, int d);
    explicit hypergeometric_dist(const param_type &);
    void reset();
    template<typename R>
    int operator()(R &);
    template<typename R>
    int operator()(R &, const param_type &);
    int min() const;
    int max() const;
    const param_type & param() const;
    void param(const param_type &);
    int n() const;
    void n(int);
    int m() const;
    void m(int);
    int d() const;
    void d(int);
    double pdf(int) const;
    double cdf(int) const;
  };

  bool operator==(const hypergeometric_dist::param_type &,
  const hypergeometric_dist::param_type &);
  bool operator!=(const hypergeometric_dist::param_type &,
  const hypergeometric_dist::param_type &);

  template<typename char_t, typename traits_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const hypergeometric_dist::param_type &);
  template<typename char_t, typename traits_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, hypergeometric_dist::param_type &);

  bool operator==(const hypergeometric_dist &, const hypergeometric_dist &);
  bool operator!=(const hypergeometric_dist &, const hypergeometric_dist &);

  template<typename char_t, typename traits_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const hypergeometric_dist &);
  template<typename char_t, typename traits_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, hypergeometric_dist &);
}
```

4.2.25 Geometric distribution

parameter	$p \in (0,1)$
support	$0, 1, \ldots$
mean	$(1-p)/p$
variance	$(1-p)/p^2$

Class `geometric_dist` provides random integers with geometric distribution with parameter p. The probability distribution function reads

$$P(x|p) = p(1-p)^x \qquad \text{for } x \in \{0, 1, \ldots\}.$$

A valid parameter p is $p \in (0,1)$.

The class `geometric_dist` is declared in the header file `trng/geometric_dist.hpp` and its public interface is given as follows:

```
namespace trng {

  class geometric_dist {
  public:
    typedef int result_type;

    class param_type {
    public:
      double p() const;
      void p(double);
      explicit param_type(double p);
    };

    explicit geometric_dist(double p);
    explicit geometric_dist(const param_type &);
    void reset();
    template<typename R>
    int operator()(R &);
    template<typename R>
    int operator()(R &, const param_type &);
    int min() const;
    int max() const;
    const param_type & param() const;
    void param(const param_type &);
    double p() const;
    void p(double);
    double pdf(int) const;
    double cdf(int) const;
  };

  bool operator==(const geometric_dist::param_type &, const geometric_dist::param_type &);
  bool operator!=(const geometric_dist::param_type &, const geometric_dist::param_type &);

  template<typename char_t, typename traits_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const geometric_dist::param_type &);
  template<typename char_t, typename traits_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, geometric_dist::param_type &);

  bool operator==(const geometric_dist &, const geometric_dist &);
  bool operator!=(const geometric_dist &, const geometric_dist &);

  template<typename char_t, typename traits_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const geometric_dist &);
  template<typename char_t, typename traits_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, geometric_dist &);
}
```

4.2.26 Poisson distribution

Class `poisson_dist` provides random integers with Poisson distribution with mean μ. The probability distribution function reads

parameter	$\mu \in [0, \infty)$
support	$0, 1, \ldots$
mean	μ
variance	μ

$$P(x|\mu) = \frac{e^{-\mu}\mu^x}{x!} \qquad \text{for } x \in \{0, 1, \ldots\}.$$

A valid parameter μ is $\mu \in [0, \infty)$.

The class `poisson_dist` is declared in the header file `trng/poisson_dist.hpp` and its public interface is given as follows:

```
namespace trng {

  class poisson_dist {
  public:
    typedef int result_type;

    class param_type {
    public:
      double mu() const;
      void mu(double);
      explicit param_type(double mu);
    };

    explicit poisson_dist(double mu);
    explicit poisson_dist(const param_type &);
    void reset();
    template<typename R>
    int operator()(R &);
    template<typename R>
    int operator()(R &, const param_type &);
    int min() const;
    int max() const;
    const param_type & param() const;
    void param(const param_type &);
    double mu() const;
    void mu(double);
    double pdf(int) const;
    double cdf(int) const;
  };

  bool operator==(const poisson_dist::param_type &, const poisson_dist::param_type &);
  bool operator!=(const poisson_dist::param_type &, const poisson_dist::param_type &);

  template<typename char_t, typename traits_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const poisson_dist::param_type &);
  template<typename char_t, typename traits_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, poisson_dist::param_type &);

  bool operator==(const poisson_dist &, const poisson_dist &);
  bool operator!=(const poisson_dist &, const poisson_dist &);

  template<typename char_t, typename traits_t>
  std::basic_ostream<char_t, traits_t> &
```

```
  operator<<(std::basic_ostream<char_t, traits_t> &, const poisson_dist &);
  template<typename char_t, typename traits_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, poisson_dist &);
}
```

4.2.27 Zero-truncated Poisson distribution

parameter	$\mu \in [0, \infty)$
support	$1, 2, \ldots$
mean	$\frac{\mu e^{\mu}}{1-e^{\mu}}$
variance	$\frac{\mu e^{\mu}}{1-e^{\mu}}\left(1 - \frac{\mu}{1-e^{\mu}}\right)$

Class zero_truncated_poisson_dist provides random integers with zero-truncated Poisson distribution (also known as the conditional Poisson distribution or the positive Poisson distribution) with parameter μ. It is the conditional probability distribution of a Poisson-distributed random variable, given that the value of the random variable is not zero. The probability distribution function reads

$$P(x|\mu) = \frac{e^{-\mu}\mu^{x}}{x!(1-e^{-\mu})} \qquad \text{for } x \in \{1, 2, \ldots\}.$$

A valid parameter μ is $\mu \in [0, \infty)$.

The class zero_truncated_poisson_dist is declared in the header file trng/zero_truncated_poisson_dist.hpp and its public interface is given as follows:

```
namespace trng {

  class zero_truncated_poisson_dist {
  public:
    typedef int result_type;

    class param_type {
    public:
      double mu() const;
      void mu(double);
      explicit param_type(double mu);
    };

    explicit zero_truncated_poisson_dist(double mu);
    explicit zero_truncated_poisson_dist(const param_type &);
    void reset();
    template<typename R>
    int operator()(R &);
    template<typename R>
    int operator()(R &, const param_type &);
    int min() const;
    int max() const;
    const param_type & param() const;
    void param(const param_type &);
    double mu() const;
    void mu(double);
    double pdf(int) const;
    double cdf(int) const;
  };
```

```
  bool operator==(const zero_truncated_poisson_dist::param_type &,
  const zero_truncated_poisson_dist::param_type &);
  bool operator!=(const zero_truncated_poisson_dist::param_type &,
  const zero_truncated_poisson_dist::param_type &);

  template<typename char_t, typename traits_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const zero_truncated_poisson_dist::param_type &);
  template<typename char_t, typename traits_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, zero_truncated_poisson_dist::param_type &);

  bool operator==(const zero_truncated_poisson_dist &, const zero_truncated_poisson_dist &);
  bool operator!=(const zero_truncated_poisson_dist &, const zero_truncated_poisson_dist &);

  template<typename char_t, typename traits_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const zero_truncated_poisson_dist &);
  template<typename char_t, typename traits_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, zero_truncated_poisson_dist &);
}
```

4.2.28 Discrete distribution

The general probability distribution function for integers in $[0, 1, \ldots, n-1]$ is determined by a set of n non-negative weights p_i ($i = 0, 1, \ldots, n-1$) and reads

$$P(x|\{p_i\}) = \frac{p_x}{\sum_{i=0}^{n-1} p_i} \qquad \text{for } x \in \{0, 1, \ldots, n-1\}.$$

TRNG provides two classes for the generation of random integers with a general discrete distribution, class `discrete_dist` and `fast_discrete_dist`. Both classes provide basically the same interface but they are implemented by different internal data structures and feature different performance characteristics.

The classes `discrete_dist` and `fast_discrete_dist` have several different constructors. The constructor `discrete_dist(int  n)` (`fast_discrete_dist(int n)`) sets up a flat distribution of n integers, each integer has the same statistical weight. Another way to construct an object of the class `discrete_dist` (`fast_discrete_dist`) is to pass the weights p_i to the constructor `discrete_dist(iter first, iter last);` (`fast_discrete_dist(iter first, iter last);`) by some iterator range.

Drawing a random number from a general discrete distribution is a $\mathcal{O}(\log n)$ operation for `discrete_dist`, while `fast_discrete_dist` is able to carryout this operation in constant time. For small n the performance difference is negligible, but for large n ($n \gtrapprox 1\,000$) becomes more and more important and therefore `fast_discrete_dist` will be used in most cases.

The method `param(int, double)` allows to change relative probability of a single relative probability p_i after an object of the type `discrete_dist` has been constructed. This will cause an update of the internal data structures that costs $\mathcal{O}(\log n)$ operation. Note that `fast_discrete_dist` does not allow to change relative probabilities and does not provide a method `param(int, double)`. This is the price we have to pay for performance.

The class `discrete_dist` is declared in the header file `trng/discrete_dist.hpp` and its public interface is given as follows:

```
namespace trng {

  class discrete_dist {
  public:
    typedef int result_type;
    class param_type {
    public:
      template<typename iter>
      explicit param_type(iter first, iter last);
    };

    discrete_dist(int n);
    template<typename iter>
    discrete_dist(iter first, iter last);
    explicit discrete_dist(const param_type &);
    void reset();
    template<typename R>
    int operator()(R &);
    template<typename R>
    int operator()(R &, const param_type &);
    int min() const;
    int max() const;
    const param_type & param() const;
    void param(const param_type &);
    void param(int, double);
    double pdf(int) const;
    double cdf(int) const;
  };

  bool operator==(const discrete_dist::param_type &, const discrete_dist::param_type &);
  bool operator!=(const discrete_dist::param_type &, const discrete_dist::param_type &);

  template<typename char_t, typename traits_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const discrete_dist::param_type &);
  template<typename char_t, typename traits_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, discrete_dist::param_type &);

  bool operator==(const discrete_dist &, const discrete_dist &);
  bool operator!=(const discrete_dist &, const discrete_dist &);

  template<typename char_t, typename traits_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const discrete_dist &);
  template<typename char_t, typename traits_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, discrete_dist &);

}
```

The files `discrete_dist.cc` (see Listing 4.3) and `discrete_dist_c_style.cc` in the TRNG source distribution demonstrate the usage of the class `discrete_dist` in detail.

The class `fast_discrete_dist` is declared in the header file `trng/fast_discrete_dist.hpp` and its public interface is given as follows:

```
namespace trng {

  class fast_discrete_dist {
  public:
    typedef int result_type;
    class param_type {
    public:
      template<typename iter>
      explicit param_type(iter first, iter last);
    };

    fast_discrete_dist(int n);
    template<typename iter>
    fast_discrete_dist(iter first, iter last);
    explicit fast_discrete_dist(const param_type &);
    void reset();
    template<typename R>
    int operator()(R &);
    template<typename R>
    int operator()(R &, const param_type &);
    int min() const;
    int max() const;
    const param_type & param() const;
    void param(const param_type &);
    double pdf(int) const;
    double cdf(int) const;
  };

  bool operator==(const fast_discrete_dist::param_type &,
  const fast_discrete_dist::param_type &);
  bool operator!=(const fast_discrete_dist::param_type &,
  const fast_discrete_dist::param_type &);

  template<typename char_t, typename traits_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const fast_discrete_dist::param_type &);
  template<typename char_t, typename traits_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, fast_discrete_dist::param_type &);

  bool operator==(const fast_discrete_dist &, const fast_discrete_dist &);
  bool operator!=(const fast_discrete_dist &, const fast_discrete_dist &);

  template<typename char_t, typename traits_t>
  std::basic_ostream<char_t, traits_t> &
  operator<<(std::basic_ostream<char_t, traits_t> &, const fast_discrete_dist &);
  template<typename char_t, typename traits_t>
  std::basic_istream<char_t, traits_t> &
  operator>>(std::basic_istream<char_t, traits_t> &, fast_discrete_dist &);

}
```

Listing 4.3: Class `discrete_dist` in action.

```
#include <cstdlib>
#include <iostream>
#include <iomanip>
#include <vector>
#include <trng/lcg64.hpp>
#include <trng/discrete_dist.hpp>

int main() {
  // stores relative probabilities
  const std::vector<double> p{1., 3.25, 5., 6.5, 7., 2.};
  // discrete distribution object
  trng::discrete_dist dist(p.begin(), p.end());
  // random number generator
  trng::lcg64 r;
  // draw some random numbers
  std::vector<int> count(p.size(), 0);
  const int samples{10000};
  for (int i{0}; i < samples; ++i) {
    const int x{dist(r)};  // draw a random number
    ++count[x];            // count
  }
  // print results
  std::cout << "value\t\tprobability\tcount\t\tempirical probability\n"
            << "=====\t\t===========\t=====\t\t=====================\n";
  for (std::vector<int>::size_type i{0}; i < count.size(); ++i) {
    std::cout << std::setprecision(3) << i << "\t\t" << dist.pdf(static_cast<int>(i)) << "\t\t"
              << count[i] << "\t\t" << static_cast<double>(count[i]) / samples << '\n';
  }
  return EXIT_SUCCESS;
}
```

4.3 Function template `generate_canonical`

In this section we describe a function template introduced by [11]. Each function instantiated from the template `generate_canonical` maps the result of a single invocation of a supplied uniform random number generator to one member of the set $\mathcal{L}$ (described below) such that, if the values produced by the generator are uniformly distributed, the results of the instantiation are distributed as uniformly as possible according to the uniformity requirements described below.

Let $\mathcal{L}$ consist of all values t of type `result_type` such that:

- If `result_type` is a floating-point type, `result_type(0)` $< t <$ `result_type(1)`.
- If `result_type` is a signed or an unsigned integral type, then the value t lays in the range

  `numeric_limits<result_type>::min()` $\leq t \leq$ `numeric_limits<result_type>::max()` .

Obtaining a value in $\mathcal{L}$ can be a useful step in the process of transforming a value generated by a uniform random number generator into a value that can be delivered by a random number distribution. The function template

```
template<class result_type, class UniformRandomNumberGenerator>
result_type generate_canonical(UniformRandomNumberGenerator &g);
```

returns a value from $\mathcal{L}$ by exactly one invocation of g, see [11] for details.

4.4 CUDA support

TRNG may be utilized in parallel Monte Carlo simulations. It does not depend on a specific parallelization technique, e. g., POSIX threads, MPI or others. TRNG also supports CUDA. CUDA is a parallel architecture and programming model for general purpose computations on graphics processing units (GPUs). GPU computing is enabled by the CUDA programming model that provides a set of abstractions that enable to express data parallelism and task parallelism. This programming model is implemented by equipping the sequential C++ programming language with extensions for parallel execution of so-called kernel functions on a GPU and providing an application programming interface. GPU kernel functions are implemented by a subset of the C++ programming language. See the [2, 30] for details.

Because there are some C++ features that can not be used in GPU functions not all TRNG classes and functions can be utilized in GPU code. For example, only parallel random number engines may be used in GPU code, see Table 4.1. One may call the methods `split`, `jump` and `jump2` or one of the call-operators of parallel random number engines. Other parallel random number engine methods are not callable from GPU code, not even the constructor. Thus, a parallel random number engine instance has to be constructed in CPU code and later to be copied to the GPU before it may be used on the GPU, see Listing 6.6 for an example.

The function template `generate_canonical` and random number distributions may be used for GPU code in the same way as in CPU code without any restrictions. Except the following distributions: `correlated_normal_dist`, `binomial_dist`, `hypergeometric_dist`, `geometric_dist`, `poisson_dist`, `zero_truncated_poisson_dist` and `discrete_dist`, they provide no CUDA support at all. These restrictions might be lifted in future TRNG releases.

5 Installation

5.1 Prerequisites

To make the installation procedure portable and comfortable, TRNG utilizes the CMake build configuration generator. For a proper installation you will need

- CMake version 3.10 or later,
- a recent C++ compiler that implements the C++11 language standard and
- a make tool or an integrated environment with cmake support, e. g., Microsoft Visual Studio, Clion, Xcode or Eclipse.

TRNG comes with numerous sample programs that illustrate the usage of the TRNG library. Some of these sample programs will use external libraries, i. e.:

- Boost C++ libraries, [8]
- an implementation of the Message Passing Interface (MPI) standard (various open source implementations can be found at [59, 55])
- Intel Threading Building Blocks [26].

If you want to compile all sample programs and unit tests, you will have to install these libraries as well. But TRNG does not depend on any of the libraries listed above.

5.2 Compilation

CMake can generate configurations for various build systems, e. g., Makefiles, which are typically employed on Unix-like systems, Visual Studio project files on Windows, or project files for various other integrated development environments. For example, Clion and Visual Studio 2019 come with build-in CMake support [13] and CMake is included in most Linux distributions. After the sources have been extracted from the source archive or have been cloned via git, the build configuration needs to be generated by CMake. In the following, the installation procedure on a typical Unix-like environment (BSD, Linux, Cygwin, etc.) will be given. For compilation in an integrated development environment read the documentation of your preferred tool. For Microsoft Visual Studio this is described in the Visual Studio documentation [13].

On a Unix-like box, just call the `cmake` tool to find your C++ compiler and to generate a set of Makefiles. It is good practice to setup an out-of source build in a separate directory. For this purpose, Makefiles are generated by the following sequence of shell commands

```
bauke@hal:~/trng-4.24$ mkdir build
bauke@hal:~/trng-4.24$ cd build
bauke@hal:~/trng-4.24/build$ cmake ..
```

The `cmake` tool may be controlled by various options and shell variables, see [12] for details. If no options are provided to TRNG will be installed in the `/usr/local` hierarchy. Call

```
bauke@hal:~/trng-4.24/build$ cmake --help
```

to get an overview about all options. Here a complex example: to compile TRNG with the Intel C++ compiler `icpc` and to install the library and the header files in `/opt/trng` call

```
bauke@hal:~/trng-4.24/build$ CXX=icpc cmake -DCMAKE_INSTALL_PREFIX=/opt/trng ..
```

After TRNG has been configured and Makefiles have been generated by CMake, the library can be compiled and installed by the `make` tool.

```
bauke@hal:~/trng-4.24/build$ cmake --build .
bauke@hal:~/trng-4.24/build$ cmake --build . --target install
```

Compilation of the TRNG library generates a static as well as a shared library unless TRNG is compiled on Windows only static libraries are supported by CMake. Depending on your system further steps might be necessary to make the TRNG shared library known to the dynamic linker. On a Linux system the system administrator has to call `ldconfig` or you might set the `LD_LIBRARY_PATH` environment variable. See also the ld.so man page for further information.

In the source directory `examples` you will find some example programs. These sources are compiled also during the compilation of the TRNG library provided that all required third party libraries (Boost etc.) have been found by the CMake tool.

A distributable package can be generated as a last optional build step. Calling the `cmake` utility with the target `package`

```
bauke@hal:~/trng-4.24/build$ cmake --build . --target package
```

yields on a Linux host a Debian package, a RPM package and a zipped tar archive. (RPM packages are created only if the `rpmbuild` tool has been found by CMake.) On all other operating systems only a zip file is created.

5.3 Running unit tests

When the TRNG library is built also a set of unit test is compiled, provided that the Boost test library has been found. To run these tests with verbose output run the CTest tool:

```
bauke@hal:~/trng-4.24/build$ ctest -V --progress
```

6 Examples

6.1 Hello world!

In listing 6.1 we present the simplest nontrivial C++ program that produces pseudo-random numbers by TRNG. Whenever one generates random numbers with TRNG at least two header files have to be included, one for a random number engine and one for a distribution function, see lines 4 and 5 in listing 6.1. In lines 9 and 11 respectively a random number engine and a random number distribution are declared. The parameters of a random number distribution object have to be specified by its declaration. In our example random numbers with a normal distribution with mean 6 and standard deviation of 2 are generated. Distribution parameters can be changed at run-time, if necessary. In the loop in lines 13 and 14 the random number engine object `R` and the random number distribution object `normal` are used to generate 1000 random numbers.

The program `hello_world.cc` has to be linked to the TRNG library. Using the GNU C++ compiler we transform the sources by

```
bauke@hal:~$ g++ -o hello_world hello_world.cc -ltrng4
```

into an executable.

In a second example we want to calculate an approximate value for π by a parallel Monte Carlo calculation. The general idea of this calculation is to choose random points in a square with edge length R. Some of these points fall into a sector of a circle in the square, see Figure 6.1. The value of π can be approximated by considering the fraction of points that fall into the

Listing 6.1: A simple TRNG sample program `hello_world.cc` that generates 1000 random variables with normal distribution.

```
1  #include <cstdlib>
2  #include <iostream>
3  // include TRNG header files
4  #include <trng/yarn2.hpp>
5  #include <trng/normal_dist.hpp>
6
7  int main() {
8    // random number engine
9    trng::yarn2 R;
10   // normal distribution with mean 6 and standard deviation 2
11   trng::normal_dist<> normal(6.0, 2.0);
12   // generate 1000 normal distributed random numbers
13   for (int i{0}; i < 100000; ++i)
14     std::cout << normal(R) << '\n';
15   return EXIT_SUCCESS;
16 }
```

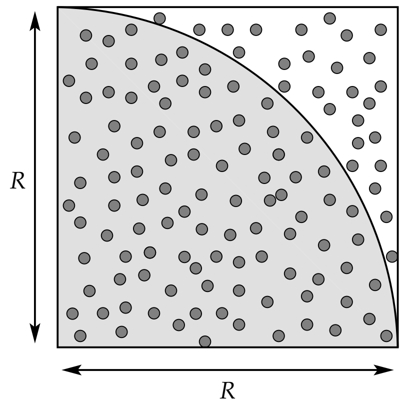


Figure 6.1: The numerical value of π can be estimated by throwing random points into a square.

circle. From the relation

$$\frac{\text{number of points in circle}}{\text{number of points in square}} \approx \frac{\pi R^2/4}{R^2} = \frac{\pi}{4}$$

we conclude

$$\pi \approx 4\frac{\text{number of points in circle}}{\text{number of points in square}}\,.$$

In listing 6.2 we use this equation to estimate π. In the for-loop in lines 12 to 16 a random x-coordinate and a random y-coordinate are chosen. Both coordinates are independently uniformly distributed in $[0, 1)$. If $\sqrt{x^2 + y^2} < 1$, or equivalently $x^2 + y^2 < 1$, the point (x, y) lies within the circle. The program draws a huge number of points from the square and counts the number of points lying within the circle and at the end of the program the fraction $4 \cdot (\text{points in circle})/(\text{points in square})$ is shown as an estimate for π.

Listing 6.2: Sequential Monte Carlo calculation of π.

```
1  #include <cstdlib>
2  #include <iostream>
3  #include <trng/yarn2.hpp>
4  #include <trng/uniform01_dist.hpp>
5
6  int main() {
7    const long samples{1000000l};  // total number of points in square
8    long in{0l};                   // no points in circle
9    trng::yarn2 r;                 // random number engine
10   trng::uniform01_dist<> u;      // random number distribution
11   // throw random points into square
12   for (long i{0}; i < samples; ++i) {
13     const double x{u(r)}, y{u(r)};  // choose random x- and y-coordinates
14     if (x * x + y * y <= 1.0)       // is point in circle?
15       ++in;                         // increase counter
16   }
17   std::cout << "pi = " << 4.0 * in / samples << std::endl;
18   return EXIT_SUCCESS;
19 }
```

6.2 Hello parallel world!

TRNG is a very flexible random number generator library. It allows for sequential as well as for parallel applications. The library does not depend on any particular communication library. It may be utilized with Message Passing Interface (MPI), OpenMP, and as well as with POSIX threads, or any other communication library. This section gives a short tutorial on writing parallel Monte Carlo applications with TRNG and various parallel programming models, e. g. MPI or OpenMP. Here we cannot give an introduction to MPI or OpenMP readers who are not familiar with parallel programming may consult [60, 5, 64, 65] instead.

How can we parallelize the Monte Carlo calculation of π? A striking feature of the Monte Carlo π calculation algorithm (from the previous section): the placement of some point in the square does not affect the placement of other points. In other words: throwing N points into a square is an embarrassingly parallel process. Everything that matters, is the fraction of points in the square that had been placed into the circle. Keeping this fact in mind the Monte Carlo calculation of π can be parallelized easily via the block splitting method or the leapfrog method.

6.2.1 Block splitting

Let us apply the block splitting parallelization technique as introduced in section 2. A total of N points has to be selected by p processes. We number the points from 0 to $N-1$ and the processes from 0 to $p-1$ respectively. The number of a process is called its rank. To distribute the workload equally, we split the entire set of N points into p consecutive blocks of about N/p points. To be specific, a process with rank r selects the points with numbers

$$\lfloor N \cdot r/p \rfloor \qquad \text{to} \qquad \lfloor N \cdot (r+1)/p \rfloor - 1\,,$$

where $\lfloor \cdot \rfloor$ denotes rounding to zero. Each point is determined by two coordinates and a process with rank r consumes

$$2\left(\lfloor N \cdot (r+1)/p \rfloor - \lfloor N \cdot r/p \rfloor\right)$$

random numbers, which are generated by the same random number engine.

All concurrent processes generate random points by their own local copy of the same random number engine. Of course, if all these engines start from the same initial state, they will produce the same sequence of random numbers. For that reason each process jumps $2\lfloor N \cdot r/p \rfloor$ steps ahead, before any random numbers are consumed. This ensures that sequences of random numbers of two different processes never overlap, and furthermore, the outcome of the parallelized program is the same as for the sequential in the previous section, even in its statistical errors.

Listing 6.3 presents an implementation of the parallel Monte Carlo computation of π by MPI, while in listing 6.4 an implementation presented that is based on OpenMP. Note the parenthesis within the argument of the `jump` method in lines 15 and 17 respectively. Together with the C++ rounding rules they are the C++ equivalent to the $\lfloor \cdot \rfloor$ function.

There is one important conceptual difference between the MPI version and the OpenMP implementation. While MPI is based on a distributed memory model, OpenMP can utilize shared memory. For that reason the MPI program counts how many points lie in the circle for each process in a process local variable `in`. At the end of the computation the process local variables have to be summed up by `MPI::COMM_WORLD.Reduce` to the (process local) variable

Listing 6.3: Parallel Monte Carlo calculation of π using block splitting and MPI.

```
#include <trng/yarn2.hpp>
#include <trng/uniform01_dist.hpp>

int main(int argc, char *argv[]) {
  const long samples{1000000l};  // total number of points in square
  MPI_Init(&argc, &argv);        // initialise MPI environment
  int size, rank;
  MPI_Comm_size(MPI_COMM_WORLD, &size);  // get total number of processes
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);  // get rank of current process
  long in{0};                            // number of points in circle
  trng::yarn2 r;                         // random number engine
  trng::uniform01_dist<> u;              // random number distribution
  r.jump(2 * (rank * samples / size));   // jump ahead
  // throw random points into square and distribute workload over all processes
  for (long i{rank * samples / size}; i < (rank + 1) * samples / size; ++i) {
    const double x{u(r)}, y{u(r)};  // choose random x- and y-coordinates
    if (x * x + y * y <= 1.0)       // is point in circle?
      ++in;                         // increase counter
  }
  // calculate sum of all local variables 'in' and storre result in 'in_all' on process 0
  long in_all;
  MPI_Reduce(&in, &in_all, 1, MPI_LONG, MPI_SUM, 0, MPI_COMM_WORLD);
  if (rank == 0)  // print result
    std::cout << "pi = " << 4.0 * in_all / samples << std::endl;
  MPI_Finalize();  // quit MPI
  return EXIT_SUCCESS;
}
```

Listing 6.4: Parallel Monte Carlo calculation of π using block splitting and OpenMP.

```
#include <trng/yarn2.hpp>
#include <trng/uniform01_dist.hpp>

int main() {
  const long samples{1000000l};  // total number of points in square
  long in{0l};                   // number of points in circle
  // distribute workload over all processes and make a global reduction
#pragma omp parallel reduction(+ : in) default(none)
  {
    trng::yarn2 r;                          // random number engine
    const int size{omp_get_num_threads()};  // get total number of processes
    const int rank{omp_get_thread_num()};   // get rank of current process
    trng::uniform01_dist<> u;               // random number distribution
    r.jump(2 * (rank * samples / size));    // jump ahead
    // throw random points into square
    for (long i{rank * samples / size}; i < (rank + 1) * samples / size; ++i) {
      const double x{u(r)}, y{u(r)};  // choose random x- and y-coordinates
      if (x * x + y * y <= 1.0)       // is point in circle?
        ++in;                         // increase thread-local counter
    }
  }
  // print result
  std::cout << "pi = " << 4.0 * in / samples << std::endl;
  return EXIT_SUCCESS;
}
```

Listing 6.5: Parallel Monte Carlo calculation of π using block splitting and Intel Threading Building Blocks.

```
#include <trng/uniform01_dist.hpp>
#include <tbb/blocked_range.h>
#include <tbb/parallel_reduce.h>

class parallel_pi {
  trng::uniform01_dist<> u;  // random number distribution
  const trng::yarn2 &r;
  long in;

public:
  void operator()(const tbb::blocked_range<long> &range) {
    trng::yarn2 r_local(r);           // local copy of random number engine
    r_local.jump(2 * range.begin());  // jump ahead
    for (long i{range.begin()}; i != range.end(); ++i) {
      const double x{u(r_local)}, y{u(r_local)};  // choose random x- and y-coordinates
      if (x * x + y * y <= 1.0)                   // is point in circle?
        ++in;                                     // increase thread-local counter
    }
  }
  // join threds and counters
  void join(const parallel_pi &other) { in += other.in; }
  long in_circle() const { return in; }
  explicit parallel_pi(const trng::yarn2 &r) : r{r}, in{0} {}
  explicit parallel_pi(const parallel_pi &other, tbb::split) : r{other.r}, in{0} {}
};

int main() {
  const long samples{1000000l};   // total number of points in square
  trng::yarn2 r;                  // random number engine
  parallel_pi pi(r);              // functor for parallel reduce
  // parallel MC computation of pi
  tbb::parallel_reduce(tbb::blocked_range<long>(0, samples), pi, tbb::auto_partitioner());
  // print result
  std::cout << "pi = " << 4.0 * pi.in_circle() / samples << std::endl;
  return EXIT_SUCCESS;
}
```

`in_all` on the process with rank zero. In a OpenMP program this global reduction can be avoided by using a shared memory variable. But here concurrent write accesses to `in` have to be prevented by the pragma `omp critical` in lines 23 to 24.

Listing 6.5 shows another block splitting Monte Carlo calculation of π that is based on the Intel Threading Building Blocks [26, 65]. To give a detailed introduction to this excellent C++ library is beyond the scope of the TRNG documentation. The reader should note the following special features of the Intel Threading Building Blocks and listing 6.5. The (thread) parallel computation is based on the function `tbb::parallel_reduce`. This function requires a class object that implements the task that has to be parallelized. However, the programmer does not specify how the global task is divided into smaller subtasks. Work distribution, load balancing and reduction of the global result (number of points in the square) are handled by the Intel Threading Building Blocks library.

Listing 6.6 shows a block splitting Monte Carlo calculation of π using CUDA. For CUDA we have to leap frog the random number engines in host memory and to copy random number engines to device memory before the parallel Monte Carlo calculation can be carried out.

Listing 6.6: Parallel Monte Carlo calculation of π using block splitting and CUDA.

```
#include <cstdlib>
#include <iostream>
#include <vector>
#include <trng/yarn5s.hpp>
#include <trng/uniform01_dist.hpp>

__global__ void parallel_pi(long samples, long *in, trng::yarn5s r) {
  long rank = threadIdx.x;
  long size = blockDim.x;
  r.jump(2 * (rank * samples / size));  // jump ahead
  trng::uniform01_dist<float> u;        // random number distribution
  in[rank] = 0;                         // local number of points in circle
  for (long i = rank * samples / size; i < (rank + 1) * samples / size; ++i) {
    const float x = u(r), y = u(r);  // choose random x- and y-coordinates
    if (x * x + y * y <= 1)          // is point in circle?
      ++in[rank];                    // increase thread-local counter
  }
}

int main(int argc, char *argv[]) {
  const long samples{1000000l};  // total number of points in square
  const int size{128};           // number of threads
  long *in_device;
  cudaMalloc(&in_device, size * sizeof(*in_device));
  trng::yarn5s r;
  // start parallel Monte Carlo
  parallel_pi<<<1, size>>>(samples, in_device, r);
  // gather results
  std::vector<long> in(size);
  cudaMemcpy(in.data(), in_device, size * sizeof(*in), cudaMemcpyDeviceToHost);
  cudaFree(in_device);
  long sum{0};
  for (int rank{0}; rank < size; ++rank)
    sum += in[rank];
  // print result
  std::cout << "pi = " << 4.0 * sum / samples << std::endl;
  return EXIT_SUCCESS;
}
```

6.2.2 Leapfrog

Leapfrog is a convenient approach to derive p non overlapping streams of pseudo-random numbers from a single base stream. As defined in section 3.1 each parallel random number engine provides a `split` method for leapfrog. If `split(p, s)` is called, the internal parameters of the random number engine are changed in such a way that future calls to `operator()` will generate the sth sub-stream of `p` sub-streams. Sub-streams are numbered from 0 to `p` -1. Changing line 15 or line 17 in listing 6.3 or listing 6.4 respectively, which reads

```
r.jump(2*(rank*samples/size));  // jump ahead
```

into

```
r.split(size, rank);            // choose sub-stream no. rank out of size streams
```

Listing 6.7: Parallel Monte Carlo calculation of π using leapfrog and MPI.

```
#include <trng/yarn2.hpp>
#include <trng/uniform01_dist.hpp>

int main(int argc, char *argv[]) {
  const long samples{1000000l};  // total number of points in square
  MPI_Init(&argc, &argv);        // initialize MPI environment
  int size, rank;
  MPI_Comm_size(MPI_COMM_WORLD, &size);  // get total number of processes
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);  // get rank of current process
  trng::yarn2 rx, ry;                    // random number engines for x- and y-coordinates
  // split PRN sequences by leapfrog method
  rx.split(2, 0);            // choose sub-stream no. 0 out of 2 streams
  ry.split(2, 1);            // choose sub-stream no. 1 out of 2 streams
  rx.split(size, rank);      // choose sub-stream no. rank out of size streams
  ry.split(size, rank);      // choose sub-stream no. rank out of size streams
  long in{0l};               // number of points in circle
  trng::uniform01_dist<> u;  // random number distribution
  // throw random points into square and distribute workload over all processes
  for (long i{rank}; i < samples; i += size) {
    const double x{u(rx)}, y{u(ry)};  // choose random x- and y-coordinates
    if (x * x + y * y <= 1.0)         // is point in circle?
      ++in;                           // increase counter
  }
  // calculate sum of all local variables 'in' and storre result in 'in_all' on process 0
  long in_all;
  MPI_Reduce(&in, &in_all, 1, MPI_LONG, MPI_SUM, 0, MPI_COMM_WORLD);
  if (rank == 0)  // print result
    std::cout << "pi = " << 4.0 * in_all / samples << std::endl;
  MPI_Finalize();  // quit MPI
  return EXIT_SUCCESS;
}
```

provides different statistically independent sub-streams of pseudo-random numbers to each process.

But note, the pseudo-random numbers of the base stream are now utilized in a completely different fashion. The sequential program and also the two on block splitting based programs from section 6.2.1 determine the position of a point (its x- and y-coordinate) by two consecutive pseudo-random numbers of the base sequence. After calling `split(size, rank)` consecutive calls to `operator()` will return pseudo-random numbers that are no longer neighboring numbers of the base sequence. In fact they have a distance of `size` with respect to the original sequence of pseudo-random numbers. For that reason the proposed replacement of the call of the `jump` method to a call to the `split` method will result in another value for the approximation of π with another statistical error.

To prevent this issue, we use the fact that the leapfrog method can be applied several times to a sequence of pseudo-random numbers by successive calls to `split`. Each time `split` is invoked the sequence is split into further sub-sequences. In listing 6.7 and listing 6.8 it is shown how this works. Both programs start with two random number engines of the same kind.

```
trng::yarn2 rx, ry;                    // random number engines for x- and y-coordinates
```

Listing 6.8: Parallel Monte Carlo calculation of π using leapfrog and OpenMP.

```
#include <trng/yarn2.hpp>
#include <trng/uniform01_dist.hpp>

int main() {
  const long samples{1000000l};  // total number of points in square
  long in{0l};                   // no points in circle
  // distribute workload over all processes and make a global reduction
#pragma omp parallel reduction(+ : in) default(none)
  {
    trng::yarn2 rx, ry;                      // random number engines for x- and y-coordinates
    const int size{omp_get_num_threads()};  // get total number of processes
    const int rank{omp_get_thread_num()};   // get rank of current process
    // split PRN sequences by leapfrog method
    rx.split(2, 0);            // choose sub-stream no. 0 out of 2 streams
    ry.split(2, 1);            // choose sub-stream no. 1 out of 2 streams
    rx.split(size, rank);      // choose sub-stream no. rank out of size streams
    ry.split(size, rank);      // choose sub-stream no. rank out of size streams
    trng::uniform01_dist<> u;  // random number distribution
    // throw random points into square
    for (long i{rank}; i < samples; i += size) {
      const double x{u(rx)}, y{u(ry)};  // choose random x- and y-coordinates
      if (x * x + y * y <= 1.0)         // is point in circle?
        ++in;                           // increase thread-local counter
    }
  }
  // print result
  std::cout << "pi = " << 4.0 * in / samples << std::endl;
  return EXIT_SUCCESS;
}
```

Later all x- and y-coordinates will be determined exclusively by one of these random number engines. But without any manipulations of the internal status via `jump` or `split` method, both engines will return the same sequences of pseudo-random numbers. Therefore, if the coordinates of each point are chosen by calling `operator()` of `rx` and `ry` once, all points will lie on the diagonal of the square. For that reason the sequences are split by

```
rx.split(2, 0);                      // choose sub-stream no. 0 out of 2 streams
ry.split(2, 1);                      // choose sub-stream no. 1 out of 2 streams
```

into two non overlapping sequences. Now successive calls to `operator()` will return different sequences of pseudo-random numbers and the points are uniformly distributed over the square. But still each process consumes the same two sequences of random numbers. However, this can be solved by calling the `split` method a second time.

```
rx.split(size, rank);                // choose sub-stream no. rank out of size streams
ry.split(size, rank);                // choose sub-stream no. rank out of size streams
```

6.2.3 Block splitting or leapfrog?

TRNG provides two powerful techniques for parallelizing streams of pseudo-random numbers, namely block splitting and leapfrog. Which one to choose, depends highly on the structure of your Monte Carlo algorithm and your needs.

In the simplest case, each process of a parallel Monte Carlo application with a fixed number of processes p (that does not change at run time) has just to equipped with some source of pseudo-random numbers and the only requirement on the p streams of pseudo-random numbers is that they do not overlap with any stream of pseudo-random numbers on any other process. In this case it is sufficient to use a single random number engine of the same type for each of the p process. Different streams are deviated by the leapfrog method and calling the `split` method of a pseudo-random number engine object after these random number engines have been initialized with the same parameters and the same seed. Of course with this simple minded approach the outcome of a Monte Carlo application (and the actual statistical errors) will depend on the number of processes.

On the other hand it is often desirable to design a parallel Monte Carlo algorithm in such a way that its outcome is independent of the number of processes. That means the Monte Carlo algorithm plays fair, see also section 2.3. Usually this additional constraint can be fulfilled by a creative combination of block splitting, leapfrog method and using more than one random number engine per processor. The previous sections gave already some elementary examples, how this can be achieved. But in general this can be quite intricate. Therefore we give some general guidelines.

- Identify the inherently parallel parts of the Monte Carlo algorithm. Which steps of the Monte Carlo algorithm cannot be parallelized?
- Break the parallelizable tasks into p (p number of processes) smaller sub-parts of approximately equal size.
- Is the number of pseudo-random numbers consumed by a parallelizable task (before it is divided into subparts) constant or does it change at runtime? If it is constant, break up the sequence of a single pseudo-random number engine into sub-streams in such a way that mimics the way in which the parallelizable task is split into independent sub-problems. This can always be achieved by calling the `split` or the `jump` method of a random number engine object.
- If the number of pseudo-random numbers consumed by a parallelizable task is not constant, or cannot be determined a priori, e. g. because this number itself is a function of the random number sequence, an upper bound for this number may be estimated. With this number a Monte Carlo algorithm can often be parallelized as if the number of consumed random numbers was fixed.

To make this advise somewhat more clear, we give a further example. Imagine the simulation of a site percolation process [70] on a two-dimensional square lattice of size $N = N_x \times N_y$. In site percolation each site of the lattice is occupied with probability P independently of the other sites and clusters of neighboring occupied sites are constructed afterward. Once these clusters are known, one can answer for a particular realization of occupied sites a lot of questions that arise in percolation theory. Is there a spanning cluster that connects the lower line of the grid and its upper line? What is the size of the largest cluster? And so on. How can we parallelize such a Monte Carlo simulation for site percolation?

The easiest way is not to parallelize at all. At least not the analysis of a single realization of occupied sites itself. Usually one is not interested in the analysis of a single realization of occupied sites by itself, but one wants to know statistical properties of site percolation (or another problem) that arise after averaging over many, lets say M, realizations of systems of the same kind. It is quite natural to spread the workload over p processors in such a way that

Listing 6.9: Sketch of a coarse-grained parallel Monte Carlo simulation of site percolation via MPI. The program creates many realizations of lattices with randomly occupied sites. Each realization is generated by a single process.

```
#include <cstdlib>
#include <trng/yarn2.hpp>
#include <trng/uniform01_dist.hpp>
#include "mpi.h"

const int number_of_realizations{1000};
const int Nx{250}, Ny{200};  // grid size
const int number_of_PRNs_per_sweep{Nx * Ny};
int site[Nx][Ny];      // lattice
const double P{0.46};  // occupation probability

int main(int argc, char *argv[]) {
  MPI_Init(&argc, &argv);  // initialize MPI environment
  int size, rank;
  MPI_Comm_size(MPI_COMM_WORLD, &size);  // get total number of processes
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);  // get rank of current process
  trng::yarn2 R;                         // random number engine
  trng::uniform01_dist<> u;              // random number distribution
  // skip random numbers that are consumed by other processes
  R.jump(rank * number_of_PRNs_per_sweep);
  for (int i{rank}; i < number_of_realizations; i += size) {
    // consume Nx * Ny pseudo-random numbers
    for (int x{0}; x < Nx; ++x)
      for (int y{0}; y < Ny; ++y)
        if (u(R) < P)
          site[x][y] = 1;  // site is occupied
        else
          site[x][y] = 0;  // site is not occupied
    // skip random numbers that are consumed by other processes
    R.jump((size - 1) * number_of_PRNs_per_sweep);
    // analyze lattice
    // ... source omitted
  }
  MPI_Finalize();  // quit MPI
  return EXIT_SUCCESS;
}
```

each process analyzes each pth lattice of the M lattices. If we number the processes by its rank from 0 to $p-1$ and the lattices form 0 to $M-1$, each process starts with a lattice which number equals the process' rank. Thereafter each process can skip $p-1$ lattices, because these are handled by other processes, and continue with the next lattice. Of course each process has not only to skip the work that is done by other processes, but also the pseudo-random numbers that would be consumed by analyzing the skipped lattices. Listing 6.9 gives a sketch of such a parallelized site percolation program.

Unfortunately it is not always possible to parallelize a Monte Carlo simulation in such a coarse-grained fashion like in the last example. Sometimes (e. g. in the Swendson-Wang-cluster-algorithm [71, 57]) the generation and the analysis of a single lattice has to be parallelized by itself. For that reason we split the lattice into $p_x \times p_y$ sub-lattices in such a way that the number of parallel processes p equals $p_x \times p_y$ and $p_x \approx p_y$. Each process is responsible for one of the

Listing 6.10: Sketch of a fine-grained parallel Monte Carlo simulation of site percolation via MPI. The program creates many realizations of lattices with randomly occupied sites. Each realization is generated by all processes together, workload is distributed by domain decomposition.

```
#include <cstdlib>
#include <new>
#include <trng/yarn2.hpp>
#include <trng/uniform01_dist.hpp>
#include "mpi.h"

const int number_of_realizations{1000};
const int Nx{250}, Ny{200};  // grid size
const double P{0.46};        // occupation probability

int main(int argc, char *argv[]) {
  MPI_Init(&argc, &argv);  // initialize MPI environment
  int size;
  MPI_Comm_size(MPI_COMM_WORLD, &size);  // get total number of processes
  // create a two-dimensional Cartesian communicator
  int dims[2]{0, 0};             // number of processes in each domension
  int coords[2];                 // coordinates of current process within the grid
  int periods[2]{false, false};  // no periodic boundary conditions
  // calculate a balanced grid partitioning such that  size = dims[0] * dims[1]
  MPI_Dims_create(size, 2, dims);
  MPI_Comm Comm;
  MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, true, &Comm);
  int rank;
  MPI_Comm_rank(Comm, &rank);              // get rank of current process
  MPI_Cart_coords(Comm, rank, 2, coords);  // get coordinates of current process
  // determine section of current process
  int x0{coords[0] * Nx / dims[0]}, x1{(coords[0] + 1) * Nx / dims[0]}, Nxl{x1 - x0},
      y0{coords[1] * Ny / dims[1]}, y1{(coords[1] + 1) * Ny / dims[1]}, Nyl{y1 - y0};
  int *site{new int[Nxl * Nyl]};  // allocate memory to storre a sublattice
  trng::yarn2 R;                  // random number engine
  trng::uniform01_dist<> u;       // random number distribution
  // skip random numbers that are consumed by other processes
  R.jump(Nx * y0 + x0);
  for (int i{0}; i < number_of_realizations; ++i) {
    // consume Nxl * Nyl pseudo-random numbers
    int *s{site};
    for (int y{y0}; y < y1; ++y) {
      for (int x{x0}; x < x1; ++x) {
        if (u(R) < P)
          *s = 1;  // site is occupied
        else
          *s = 0;  // site is not occupied
        ++s;
      }
      // skip random numbers that are consumed by other processes
      R.jump(Nx - Nxl);
    }
    // skip random numbers that are consumed by other processes
    R.jump(Nx * (Ny - Nyl));
    // analyze lattice
    // ... source omitted
  }
  delete[] site;
  MPI_Finalize();  // quit MPI
  return EXIT_SUCCESS;
}
```

sub-lattices and uses the same random number engine. This generic parallelization paradigm is also known as domain decomposition.

To make the site percolation lattice generation independent of the number processes and thus independent of the details of the lattice partition, some numbers within the stream of pseudo-random numbers of the random number engine have to be skipped by the `jump` method. If we determine the state (occupied or not occupied) of the sites in a row-major fashion, the `jump` method has to be called, whenever a process has filled a row of its sub-lattice. Of course each process has to skip a certain amount of pseudo-random numbers at the start of the simulation, too.

Listing 6.10 shows the outline of a fine-grained parallel Monte Carlo simulation of site percolation via MPI, where each single lattice generation is done in parallel via domain decomposition. This program shows two noteworthy implementation details. First the program uses a runtime generated Cartesian communicator rather than the standard communicator `MPI::COMM_WOLD` as seen in the MPI examples so far. Such a communicator reflects the special topology of the domain decomposition and eases its implementation significantly. The number of sub-lattices in each dimension, p_x and p_y respectively, is determined by `MPI::Compute_dims`, see [60, 5] for details. Its result (returned in the field `dims`) determines the topology of the Cartesian communicator `Comm`. Another nice feature of the example code in listing 6.10 is that it does not assume the number of sites in any dimension is a multiple of the number of sub-lattices in this dimension. So the sizes of the sub-lattices can vary slightly from process to process. The precise range of coordinates that each process is responsible for is calculated in lines 24 and 25.

Skipping numbers in a pseudo-random number sequence via `jump` is not for free. Of course it is so smart that it can jump ahead without actually generating the numbers that have to be skipped. But the complexity of `jump` grows logarithmically in its argument. If the domain decomposition is coarse-grained enough, the overhead introduced by skipping numbers via `jump` can be neglected. But if the number of processes that generate a site percolation lattice becomes larger and larger, at a certain point this overhead can no longer be ignored and it starts to limit the speedup achievable by parallelization. Finding the right level of granularity is a general problem in parallel computing. On one hand one wants to use a large number of processes to attain a large speedup, on the other hand, the relative portion of the inherent sequential part of a program and the overhead introduced by the parallelization grow with the number of processes as well. This fact is also known as Amdahl's law.

6.3 Using TRNG with STL and Boost

Whenever large scale Monte Carlo applications are written, they will not base on TRNG solely, but also on other libraries, e. g. the C++ Standard Template Library (STL) or Boost [8]. In this section we show, how to use TRNG in combination with the STL, especially its containers and algorithms. We assume you are familiar with the concepts of the C++ STL, otherwise we suggest to read [56].

Imagine a C++ array or an STL container like a vector or a list of integers that has to be populated by random numbers with a given distribution. This can be achieved by a simple loop.

```
trng::yarn2 R;                        // random number engine
trng::uniform_int_dist U(0, 100);     // random number distribution
```

```
std::vector<long> v(10);             // vector of long with 10 elements
for (std::vector<long>::iterator i(v.begin()), end(v.end()); i!=end; ++i)
  *i=U(R);      // generate a random number form distribution U by engine R
```

This loop looks innocent, but it is not. Its error-prone and it its not obvious what is actually effected by the loop. The loop is error-prone because the programmer has to take care that the type of the iterator `i` fits to the container. Things become much more handy, if STL algorithms like `std::generate` are used.

The template function `std::generate` takes an iterator range and a function object that takes no arguments as its arguments. The prototype of this function reads

```
namespace std {

  template <class ForwardIterator, class Generator>
  void generate(ForwardIterator first, ForwardIterator last, Generator gen);

}
```

and it assigns the result of invoking gen to each element in the range [`first`, `last`). Random number distributions as introduced in section 3.2 do not meet the requirements of `std::generate`, because their overloaded call operator requires at least one argument, namely a random number engine, see Table 3.2. For that reason we need a function adapter that makes random number distributions compatible with `std::generate`, e. g., or `std::bind` or a lambda function. Employing the template class `std::bind`, an STL container v can be filled by

```
trng::yarn2 R;                       // random number engine
trng::uniform_int_dist U(0, 100);    // random number distribution
std::vector<long> v(10);             // vector of long with 10 elements
std::generate(v.begin(), v.end(), std::bind(U, std::ref(R));
```

The statement

```
std::bind(U, std::ref(R))
```

returns a temporary function object whose call operator requires no arguments. The function `std::ref` assures that the temporary function object holds a reference to the random number engine `R`, otherwise it would contain a copy of `R`. Omitting `std::ref` may have unexpected side effects, e. g. the loop

```
for (int i(0); i < 10; ++i)
  std::generate(v.begin(), v.end(), std::bind(U, R));
```

would fill the vector v ten times with random numbers, each time with the same set of random numbers. Because `std::bind` generates at each call to `std::generate` a copy of the random number engine `R` and this copy determines the random values in v, but not the random number engine `R` itself. As a consequence of this copy process `std::generate` generates random numbers by a random number engine that starts with the same internal state in each cycle of the loop.

Listing 6.11 demonstrates all the techniques for binding function arguments that have been discussed in this section. Additionally it shows that TRNG random number engine meet the requirements of the STL functions `std::random_shuffle` and `std::shuffle` directly, no function adaption via `std::bind` is needed.[1]

[1]Note that `std::random_shuffle` has been removed from the C++ standard library in C++17.

Listing 6.11: This demo program demonstrates the interplay of TRNG, the C++ STL.

```
#include <cstdlib>
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
#include <trng/yarn2.hpp>
#include <trng/uniform_int_dist.hpp>

// print an iterator range to stdout
template<typename iter>
void print_range(iter i1, iter i2) {
  while (i1 != i2)
    std::cout << (*(i1++)) << '\t';
  std::cout << "\n\n";
}

int main() {
  trng::yarn2 R;
  trng::uniform_int_dist U(0, 100);
  std::vector<long> v(10);

  std::cout << "random number generation by call operator\n";
  for (auto &val : v)
    val = U(R);
  print_range(v.begin(), v.end());
  std::vector<long> w(12);
  std::cout << "random number generation by std::generate\n";
  std::generate(w.begin(), w.end(), std::bind(U, std::ref(R)));
  print_range(w.begin(), w.end());
  std::cout << "random number generation by std::generate\n";
  std::generate(w.begin(), w.end(), std::bind(U, std::ref(R)));
  print_range(w.begin(), w.end());
  std::cout << "same sequence as above, but in a random shuffled order\n";
  std::shuffle(w.begin(), w.end(), R);
  print_range(w.begin(), w.end());
  return EXIT_SUCCESS;
}
```

6.4 Using TRNG with C++ standard library random number facility

Random number engines and distributions from TRNG and the C++11 (or later) standard library [27, 28] have the same interfaces and can therefore may be utilized in combination. This meas, for example, random numbers may be generated by using a random number distribution of the C++11 standard library and a TRNG random number engine, see listing 6.12.

There are some probability distributions that are implemented by TRNG random number distribution classes as well as by random number distribution classes from the C++11 standard library. There is, however, a crucial difference between TRNG distributions and C++11 distributions. TRNG distributions consume *exactly* one random number from a random number engine to generated a random number from a desired distribution. With C++11 distributions the number of consumed random numbers may be larger or may even vary. Thus, C++11 random number distributions should not be utilized in parallel Monte Carlo simulations.

In particular, it is not possible to write parallel Monte Carlo simulations that play fair, see section 2.3.

Listing 6.12: TRNG random number generators and distributions may be mixed with C++11 random number generators and distributions.

```
#include <iostream>
#include <random>
#include <trng/lcg64.hpp>
#include <trng/normal_dist.hpp>

int main() {
  std::mt19937 R_cpp11;
  trng::lcg64 R_trng;
  std::normal_distribution<> N_cpp11;
  trng::normal_dist<> N_trng(0, 1);
  for (int i{0}; i < 10000; ++i) {
    std::cout << N_cpp11(R_cpp11) << '\t';
    std::cout << N_cpp11(R_trng) << '\t';
    std::cout << N_trng(R_cpp11) << '\t';
    std::cout << N_trng(R_trng) << '\n';
  }
  return EXIT_SUCCESS;
}
```

7 Implementation details and efficiency

Random number engines `trng::mrgn`, `trng::mrgns`, `trng::yarnn`, and `trng::yarnns` utilize LFSR sequences

$$r_i = a_1 \cdot r_{i-1} + a_2 \cdot r_{i-2} + \ldots + a_n \cdot r_{i-n} \bmod m \tag{7.1}$$

over a prime field $\mathbb{F}_m$. The modulus m may be any prime. But LFSR sequences over $\mathbb{F}_2$ have found much more proliferation in the random number generation business than LFSR sequences over other prime fields. LFSR sequences over general prime fields have been proposed in the literature [25, 36, 33] as PRNGs. But so far, they found less attention by practitioners because it is not straight forward to implement LFSR sequences over $\mathbb{F}_m$ efficiently, if m is a large prime, especially if m of the order of the largest in a single computer word representable integer. For that reason, we present some implementation techniques.

We assume that all integer arithmetic is done in w-bit registers and $m < 2^{w-1}$. Under this condition addition of modulo m can be done without overflow problems. But multiplying two $(w-1)$-bit integers modulo m is not straightforward because the intermediate product has $2(w-1)$ significant bits and cannot be stored in a w-bit register. For the special case $a_k < \sqrt{m}$ Schrage [67] showed how to calculate $a_k \cdot r_{i-k} \bmod m$ without overflow. Based on this technique a portable implementation of LFSR sequences with coefficients $a_k < \sqrt{m}$ is presented in [37]. For parallel PRNGs this methods do not apply because the leapfrog method may yield coefficients that violate this condition. Knuth [33, section 3.2.1.1] proposed a generalization of Schrage's method for arbitrary positive factors less than m, but this method requires up to twelve multiplications and divisions and is therefore not very efficient.

The only way to implement (2.9) without additional measures to circumvent overflow problems is to restrict m to $m < 2^{w/2}$. On machines with 32-bit registers, 16 random bits per number is not enough for some applications. Fortunately today's C compiler provide fast 64-bit-arithmetic even on 32-CPUs and genuine 64-CPUs become more and more common. This allows us to increase m to 32.

7.1 Efficient modular reduction

Since the modulo operation in (2.9) is usually slower than other integer operations like addition, multiplication, Boolean operations or shifting, it has a significant impact on the total performance of PRNGs based on LFSR sequences. If the modulus is a Mersenne Prime $m = 2^e - 1$, however, the modulo operation can be done using only a few additions, Boolean operations and shift operations [61].

A summand $s = a_k \cdot r_{i-k}$ in (2.9) will never exceed $(m-1)^2 = (2^e - 2)^2$ and for each positive integer $s \in [0, (2^e - 1)^2]$ there is a unique decomposition of s into

$$s = r \cdot 2^e + q \quad \text{with} \quad 0 \le q < 2^e\,. \tag{7.2}$$

From this decomposition we conclude

$$\begin{aligned} s - r \cdot 2^e &= q \\ s - r(2^e - 1) &= q + r \\ s \bmod (2^e - 1) &= q + r \bmod (2^e - 1) \end{aligned}$$

and r and q are bounded form above by

$$q < 2^e \quad \text{and} \quad r \leq \lfloor (2^e - 2)^2 / 2^e \rfloor < 2^e - 2$$

respectively, and therefore

$$q + r < 2^e + 2^e - 2 = 2m\,.$$

So if $m = 2^e - 1$ and $s \leq (m-1)^2$, $x = s \bmod m$ can be calculated solely by shift operations, Boolean operations and addition, viz

$$x = (s \bmod 2^e) + \lfloor s/2^e \rfloor\,. \tag{7.3}$$

If (7.3) yields a value $x \geq m$ we simply subtract m.

From a computational point of view Mersenne Prime moduli are optimal and we propose to choose the modulus $m = 2^{31} - 1$. This is the largest positive integer that can be represented by a signed 32-bit integer variable, and it is also a Mersenne Prime. On the other hand our theoretical considerations favor Sophie-Germain Prime moduli, for which (7.3) does not apply directly. But one can generalize (7.3) to moduli $2^e - k$ [47]. Again we start from a decomposition of s into

$$s = r \cdot 2^e + q \quad \text{with} \quad 0 \leq q < 2^e\,, \tag{7.4}$$

and conclude

$$\begin{aligned} s - r \cdot 2^e &= q \\ s - r(2^e - k) &= q + kr \\ s \bmod (2^e - k) &= q + kr \bmod (2^e - k)\,. \end{aligned}$$

The sum $s' = q + kr$ exceeds the modulus at most by a factor $k+1$, because by applying

$$q < 2^e \quad \text{and} \quad r \leq \lfloor (2^e - k - 1)^2 / 2^e \rfloor < 2^e - k - 1$$

we get the bound

$$q + kr < 2^e + k(2^e - k - 1) = (k+1)m\,.$$

In addition by the decomposition of $s' = q + kr$

$$s' = r' \cdot 2^e + q' \quad \text{with} \quad 0 \leq q' < 2^e\,,$$

it follows

$$s \bmod (2^e - k) = s' \bmod (2^e - k) = q' + kr' \bmod (2^e - k)\,,$$

and this time the bounds

$$q' < 2^e \quad \text{and} \quad r' \leq \lfloor (k+1)(2^e - k)/2^e \rfloor < k + 1$$

and

$$q' + kr' < 2^e + k(k+1) = m + k(k+2)\,.$$

hold. Therefore if $m = 2^e - k$, $s \leq (m-k)^2$ and $k(k+2) \leq m$, $x = s \bmod m$ can be calculated solely by shift operations, Boolean operations and addition, viz

$$\begin{aligned} s' &= (s \bmod 2^e) + k\lfloor s/2^e \rfloor \\ x &= (s' \bmod 2^e) + k\lfloor s'/2^e \rfloor\,. \end{aligned} \tag{7.5}$$

If (7.5) yields a value $x \geq m$, a single subtraction of m will complete the modular reduction. To carry out (7.5) twice as many operations as for (7.3) are needed. But (7.5) applies for all moduli $m = 2^e - k$ with $k(k+2) \leq m$.

7.2 Fast delinearization

YARN generators hide linear structures of LFSR sequences q_i by raising a generating element g to the power $g^{q_i} \bmod m$. This can be done efficiently by binary exponentiation, which takes $\mathcal{O}(\log m)$ steps. But considering LFSR sequences with only a few feedback taps ($n \leq 6$) and $m \approx 2^{31}$ even fast exponentiation is significantly more expensive than a single iteration of (2.9). Therefore we propose to implement exponentiation by table look up. If m is a $2e'$-bit number we apply the decomposition

$$\begin{aligned} q_i &= q_{i,1} \cdot 2^{e'} + q_{i,0} \quad \text{with} \\ q_{i,1} &= \lfloor q_i/2^{e'} \rfloor\,, \quad q_{i,0} = q_i \bmod 2^{e'} \end{aligned} \tag{7.6}$$

and use the identity

$$r_i = g^{q_i} \bmod m = (g^{2^{e'}})^{q_{i,1}} \cdot g^{q_{i,0}} \bmod m \tag{7.7}$$

to calculate $g^{q_i} \bmod m$ by two table look-ups and one multiplication modulo m. If $m < 2^{31}$ the tables for $(g^{2^{e'}})^{q_{i,1}} \bmod m$ and $g^{q_{i,0}} \bmod m$ have 2^{16} and 2^{15} entries respectively and fit easily into the cache of modern CPUs.

7.3 Performance

By TRNG we provide an optimized PRNG library. The implementation uses 64-bit-arithmetic, fast modular reduction (7.3) and (7.5) and exponentiation by table look-up (7.7) to implement PRNGs based on LFSR sequences over prime fields, with Mersenne or Sophie-Germain Prime modulus. PRNGs of TRNG are able to compete with other sequential PRNGs in terms of speed and statistical properties but do support block splitting and leapfrog, too. Table 7.1 shows some benchmark results. For this benchmark 2^{26} PRNs were generated and the execution time was measured to compute how many PRNs each PRNG is able to generate per second. Apparently the performance of the PRNGs of TRNG compete quite well with popular PRNGs like the Mersenne Twister (`trng::mt19937`, `std::mt19937` and `boost::mt19937`), lagged Fibonacci generators (LFSR sequences over $\mathbb{F}_2$) or RANLUX that can be found in the Boost library [8].

Table 7.1: Performance of various random number engines from TRNG, the C++ Standard Library and Boost. Test program was compiled and executed on a Intel Core i7-1051U 1.80 GHz in 64-bit mode using an Intel C++ compiler version 19.1.3.304 and the optimization option -O3.

generator	PRNs per second
TRNG	
trng::lcg64	$1068.8 \cdot 10^6$
trng::lcg64_shift	$842.4 \cdot 10^6$
trng::lcg64_count_shift	$640.8 \cdot 10^6$
trng::mrg2	$310.6 \cdot 10^6$
trng::mrg3	$266.4 \cdot 10^6$
trng::mrg3s	$212.3 \cdot 10^6$
trng::mrg4	$217.5 \cdot 10^6$
trng::mrg5	$164.3 \cdot 10^6$
trng::mrg5s	$175.0 \cdot 10^6$
trng::yarn2	$239.1 \cdot 10^6$
trng::yarn3	$226.6 \cdot 10^6$
trng::yarn3s	$177.2 \cdot 10^6$
trng::yarn4	$178.3 \cdot 10^6$
trng::yarn5	$122.6 \cdot 10^6$
trng::yarn5s	$108.7 \cdot 10^6$
trng::mt19937	$347.9 \cdot 10^6$
trng::mt19937_64	$268.4 \cdot 10^6$
trng::lagfib2xor_19937_64	$929.1 \cdot 10^6$
trng::lagfib4xor_19937_64	$636.9 \cdot 10^6$
trng::lagfib2plus_19937_64	$926.2 \cdot 10^6$
trng::lagfib4plus_19937_64	$612.6 \cdot 10^6$
trng::xoshiro256plus	$917.1 \cdot 10^6$
C++ Standard Library	
std::minstd_rand0	$216.3 \cdot 10^6$
std::minstd_rand	$227.0 \cdot 10^6$
std::mt19937	$276.1 \cdot 10^6$
std::mt19937_64	$319.7 \cdot 10^6$
std::ranlux24_base	$181.2 \cdot 10^6$
std::ranlux48_base	$209.3 \cdot 10^6$
std::ranlux24	$16.2 \cdot 10^6$
std::ranlux48	$5.9 \cdot 10^6$
std::knuth_b	$60.8 \cdot 10^6$
Boost Library	
boost::minstd_rand	$238.6 \cdot 10^6$
boost::ecuyer1988	$201.8 \cdot 10^6$
boost::kreutzer1986	$187.2 \cdot 10^6$
boost::hellekalek1995	$5.6 \cdot 10^6$
boost::mt11213b	$534.4 \cdot 10^6$
boost::mt19937	$413.3 \cdot 10^6$
boost::lagged_fibonacci607	$675.4 \cdot 10^6$

generator	PRNs per second
boost::lagged_fibonacci1279	$641.7 \cdot 10^6$
boost::lagged_fibonacci2281	$640.9 \cdot 10^6$
boost::lagged_fibonacci3217	$642.3 \cdot 10^6$
boost::lagged_fibonacci4423	$634.2 \cdot 10^6$
boost::lagged_fibonacci9689	$618.3 \cdot 10^6$
boost::lagged_fibonacci19937	$596.0 \cdot 10^6$
boost::lagged_fibonacci23209	$595.3 \cdot 10^6$
boost::lagged_fibonacci44497	$579.8 \cdot 10^6$

8 Quality and statistical tests

Sequences of PRNs are sequences of deterministic numbers that try to mimic true random numbers and, one may wonder, how close sequences produced by a TRNG can come to sequences of real random numbers? This question can be answered (at least partly) by statistical tests. One can apply a battery of tests on a generator, and the more tests a generator can pass, the better its quality. One distinguishes empirical and theoretical test procedures.

Empirical tests take a finite sequence of PRNs and compute certain statistics, e. g. chi-square or Kolmogorov-Smirnov statistics, to judge the generator as "random" or not. The test statistic is a random variate with a probability distribution that can be calculated under the assumption that the test statistic is a function of true random numbers. This probability distribution is used to judge a finite sequence of PRNs as possibly random or non-random. For example in an actual test we may find a value of the test statistic that is so large (or small) that such a value or a larger (or smaller) value can be found by chance for true random numbers with a probability of 5 % only. In this case we assume the PRNG has failed the test and its sequence of PRNs behaves non-random. But note, we may be wrong, there is a 5 % probability that we have just seen normal statistical deviations. Therefore a statistical test should be applied several times. If the PRNG fails more often than it can be explained by normal statistical deviations, it has a serious flaw and should be rejected as non-random.

While empirical tests focus only on the statistical properties of a finite stream of PRNs and ignore all the details of the underlying PRNG algorithm, theoretical tests analyze the PRNG algorithm itself by number-theoretic methods and establish a priori characteristics of the PRN sequence. These a priori characteristics may be used to choose good parameter sets for a certain class of PRNGs, e. g. the coefficients of the LFSR sequences in the random number engines `trng::mrg`*n* and `trng::yarn`*n* (see section 4.1) have been found by an extensive computer search [37] and give good results in the spectral test [33], the most important theoretical test for this class of generators.

On one hand the more kinds of statistical test procedures a PRNG masters, the more we will trust its statistical properties. On the other hand statistical test can never prove that an finite sequence of numbers is "random" or not. Knuth writes in [33]:

> "In practice, we apply about half a dozen different kinds of statistical tests on a sequence, and if is passes them satisfactorily we consider it to be random—it is then presumed innocent until proven guilty."

All PRNGs of TRNG and sub-streams of them have been subject to different statistical tests as presented below. Empirical tests of the PRNGs of TNRG by other researchers have been carried out in [3] and [48]. In respect of these tests the generator you find in TRNG are comparable to other well-known high-quality generators like the Mersenne twister generator [49]. The tables in this section present results of various statistical tests of streams of pseudo-random numbers that are generated by PRNGs of TRNG with default parameters and no leapfrog splitting. All statistical tests are implemented by an extendend version [9] of the dieharder test suite [10] that incorporates the generators of the TRNG library. A detailed description of the statistical tests can be found on the Dieharder web site [10] or in [33] and [1]. Diehard offers many parameters to tweak the sensitivity of the statitical tests. In order to make it easier to compare test results for TRNG random number engines to results for other generators, the following tables are generated with the Dieharder's default settings. TRNG users may run their own tests with custom parameters if desired, see [9] for the source code of the applied tets.

There are a few things that are worth noting about the test results. The engine `trng::lcg64` fails in many tests which just illustrates the known weaknesses of linear congruential generators. The non-linear output mapping of `trng::lcg64_shift`, however, eliminates these issues very effectively. The engines `mrg`*n* and `yarn`*n* perform very well. They fail, however, all the test `diehard_dna`. This implementation of George Marsaglia's DNA test assumes that the PRNG generates pseudo random integers with at least 32 bits. Therefore the test is actually not applicable to the engines `mrg`*n* and `yarn`*n*, which yield only 31-bit integers due to their design.

Listing 8.1: Test results for random number engine `trng::lcg64`.

```
#=============================================================================#
#          dieharder version 3.31.2beta Copyright 2003 Robert G. Brown        #
#=============================================================================#
   rng_name    |rands/second|   Seed   |  k ints/sec|k doubles/sec|
     trng_lcg64|  3.98e+08  |2324135212|    398168  |     405465  |
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
   diehard_birthdays|   0|       100|     100|0.22883165|  PASSED
      diehard_operm5|   0|   1000000|     100|0.21457528|  PASSED
  diehard_rank_32x32|   0|     40000|     100|0.88535548|  PASSED
    diehard_rank_6x8|   0|    100000|     100|0.00000000|  FAILED
   diehard_bitstream|   0|   2097152|     100|0.00000000|  FAILED
        diehard_opso|   0|   2097152|     100|0.00000000|  FAILED
        diehard_oqso|   0|   2097152|     100|0.00000000|  FAILED
         diehard_dna|   0|   2097152|     100|0.00000000|  FAILED
diehard_count_1s_str|   0|    256000|     100|0.00000000|  FAILED
diehard_count_1s_byt|   0|    256000|     100|0.00000000|  FAILED
 diehard_parking_lot|   0|     12000|     100|0.51759383|  PASSED
    diehard_2dsphere|   2|      8000|     100|0.56127461|  PASSED
    diehard_3dsphere|   3|      4000|     100|0.72776399|  PASSED
     diehard_squeeze|   0|    100000|     100|0.73487690|  PASSED
        diehard_runs|   0|    100000|     100|0.83104160|  PASSED
        diehard_runs|   0|    100000|     100|0.82311998|  PASSED
       diehard_craps|   0|    200000|     100|0.70157406|  PASSED
       diehard_craps|   0|    200000|     100|0.22917743|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.00000000|  FAILED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.00000000|  FAILED
         sts_monobit|   1|    100000|     100|0.02474264|  PASSED
            sts_runs|   2|    100000|     100|0.41235478|  PASSED
          sts_serial|   1|    100000|     100|0.29240776|  PASSED
          sts_serial|   2|    100000|     100|0.00000022|  FAILED
          sts_serial|   3|    100000|     100|0.00000000|  FAILED
          sts_serial|   3|    100000|     100|0.00000069|  FAILED
          sts_serial|   4|    100000|     100|0.00000000|  FAILED
          sts_serial|   4|    100000|     100|0.00004013|   WEAK
          sts_serial|   5|    100000|     100|0.00000000|  FAILED
          sts_serial|   5|    100000|     100|0.00000000|  FAILED
          sts_serial|   6|    100000|     100|0.00000000|  FAILED
          sts_serial|   6|    100000|     100|0.00000000|  FAILED
          sts_serial|   7|    100000|     100|0.00000000|  FAILED
          sts_serial|   7|    100000|     100|0.00000000|  FAILED
          sts_serial|   8|    100000|     100|0.00000000|  FAILED
          sts_serial|   8|    100000|     100|0.00000000|  FAILED
          sts_serial|   9|    100000|     100|0.00000000|  FAILED
          sts_serial|   9|    100000|     100|0.00000000|  FAILED
          sts_serial|  10|    100000|     100|0.00000000|  FAILED
          sts_serial|  10|    100000|     100|0.00000000|  FAILED
```

```
           sts_serial|  11|    100000|      100|0.00000000|  FAILED
           sts_serial|  11|    100000|      100|0.00000000|  FAILED
           sts_serial|  12|    100000|      100|0.00000000|  FAILED
           sts_serial|  12|    100000|      100|0.00000000|  FAILED
           sts_serial|  13|    100000|      100|0.00000000|  FAILED
           sts_serial|  13|    100000|      100|0.00000000|  FAILED
           sts_serial|  14|    100000|      100|0.00000000|  FAILED
           sts_serial|  14|    100000|      100|0.00000000|  FAILED
           sts_serial|  15|    100000|      100|0.00000000|  FAILED
           sts_serial|  15|    100000|      100|0.00000000|  FAILED
           sts_serial|  16|    100000|      100|0.00000000|  FAILED
           sts_serial|  16|    100000|      100|0.00000000|  FAILED
          rgb_bitdist|   1|    100000|      100|0.00000000|  FAILED
          rgb_bitdist|   2|    100000|      100|0.00000000|  FAILED
          rgb_bitdist|   3|    100000|      100|0.00000000|  FAILED
          rgb_bitdist|   4|    100000|      100|0.00000000|  FAILED
          rgb_bitdist|   5|    100000|      100|0.00000000|  FAILED
          rgb_bitdist|   6|    100000|      100|0.00000000|  FAILED
          rgb_bitdist|   7|    100000|      100|0.00000000|  FAILED
          rgb_bitdist|   8|    100000|      100|0.00000000|  FAILED
          rgb_bitdist|   9|    100000|      100|0.00000054|  FAILED
          rgb_bitdist|  10|    100000|      100|0.00856489|  PASSED
          rgb_bitdist|  11|    100000|      100|0.41250241|  PASSED
          rgb_bitdist|  12|    100000|      100|0.21641090|  PASSED
rgb_minimum_distance|   2|     10000|     1000|0.52728976|  PASSED
rgb_minimum_distance|   3|     10000|     1000|0.78830670|  PASSED
rgb_minimum_distance|   4|     10000|     1000|0.24397744|  PASSED
rgb_minimum_distance|   5|     10000|     1000|0.27337143|  PASSED
    rgb_permutations|   2|    100000|      100|0.65073784|  PASSED
    rgb_permutations|   3|    100000|      100|0.73284507|  PASSED
    rgb_permutations|   4|    100000|      100|0.99239283|  PASSED
    rgb_permutations|   5|    100000|      100|0.16236905|  PASSED
      rgb_lagged_sum|   0|   1000000|      100|0.29770092|  PASSED
      rgb_lagged_sum|   1|   1000000|      100|0.98149568|  PASSED
      rgb_lagged_sum|   2|   1000000|      100|0.76120464|  PASSED
      rgb_lagged_sum|   3|   1000000|      100|0.77141569|  PASSED
      rgb_lagged_sum|   4|   1000000|      100|0.19354913|  PASSED
      rgb_lagged_sum|   5|   1000000|      100|0.85151893|  PASSED
      rgb_lagged_sum|   6|   1000000|      100|0.92747266|  PASSED
      rgb_lagged_sum|   7|   1000000|      100|0.09980915|  PASSED
      rgb_lagged_sum|   8|   1000000|      100|0.84368530|  PASSED
      rgb_lagged_sum|   9|   1000000|      100|0.89020318|  PASSED
      rgb_lagged_sum|  10|   1000000|      100|0.94396737|  PASSED
      rgb_lagged_sum|  11|   1000000|      100|0.59436152|  PASSED
      rgb_lagged_sum|  12|   1000000|      100|0.86857366|  PASSED
      rgb_lagged_sum|  13|   1000000|      100|0.69960240|  PASSED
      rgb_lagged_sum|  14|   1000000|      100|0.17786268|  PASSED
      rgb_lagged_sum|  15|   1000000|      100|0.98195772|  PASSED
      rgb_lagged_sum|  16|   1000000|      100|0.03190425|  PASSED
      rgb_lagged_sum|  17|   1000000|      100|0.35665074|  PASSED
      rgb_lagged_sum|  18|   1000000|      100|0.98868335|  PASSED
      rgb_lagged_sum|  19|   1000000|      100|0.74397447|  PASSED
      rgb_lagged_sum|  20|   1000000|      100|0.69059022|  PASSED
      rgb_lagged_sum|  21|   1000000|      100|0.77613051|  PASSED
      rgb_lagged_sum|  22|   1000000|      100|0.71232639|  PASSED
      rgb_lagged_sum|  23|   1000000|      100|0.19486729|  PASSED
      rgb_lagged_sum|  24|   1000000|      100|0.62367847|  PASSED
      rgb_lagged_sum|  25|   1000000|      100|0.03598248|  PASSED
      rgb_lagged_sum|  26|   1000000|      100|0.86507031|  PASSED
      rgb_lagged_sum|  27|   1000000|      100|0.04344101|  PASSED
      rgb_lagged_sum|  28|   1000000|      100|0.94014091|  PASSED
      rgb_lagged_sum|  29|   1000000|      100|0.42401075|  PASSED
      rgb_lagged_sum|  30|   1000000|      100|0.40726659|  PASSED
      rgb_lagged_sum|  31|   1000000|      100|0.63399901|  PASSED
      rgb_lagged_sum|  32|   1000000|      100|0.80720423|  PASSED
     rgb_kstest_test|   0|     10000|     1000|0.21983008|  PASSED
     dab_bytedistrib|   0|  51200000|        1|1.00000000|  FAILED
             dab_dct| 256|     50000|        1|0.00000000|  FAILED
Skipping test 207
Preparing to run test 208.  ntuple = 0
```

```
       dab_filltree2|   0|   5000000|       1|0.00000000|  FAILED
       dab_filltree2|   1|   5000000|       1|0.00000000|  FAILED
Preparing to run test 209.  ntuple = 0
        dab_monobit2|  12|  65000000|       1|1.00000000|  FAILED
Preparing to run test 210.  ntuple = 0
#=============================================================================#
#   mean  |  stddev |  error-rate (best = 0.0, worst = 0.5)
#=============================================================================#
0.131540  |0.152170  |
```

Listing 8.2: Test results for random number engine `trng::lcg64_shift`.

```
#=============================================================================#
#          dieharder version 3.31.2beta Copyright 2003 Robert G. Brown       #
#=============================================================================#
   rng_name    |rands/second|   Seed   |  k ints/sec|k doubles/sec|
trng_lcg64_shift|  3.25e+08  |3446090337|    325150  |     341401  |
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
   diehard_birthdays|   0|       100|     100|0.46811564|  PASSED
      diehard_operm5|   0|   1000000|     100|0.97747747|  PASSED
  diehard_rank_32x32|   0|     40000|     100|0.73431820|  PASSED
    diehard_rank_6x8|   0|    100000|     100|0.74389729|  PASSED
   diehard_bitstream|   0|   2097152|     100|0.08176060|  PASSED
        diehard_opso|   0|   2097152|     100|0.88453992|  PASSED
        diehard_oqso|   0|   2097152|     100|0.24965591|  PASSED
         diehard_dna|   0|   2097152|     100|0.45476082|  PASSED
diehard_count_1s_str|   0|    256000|     100|0.62125819|  PASSED
diehard_count_1s_byt|   0|    256000|     100|0.92226141|  PASSED
 diehard_parking_lot|   0|     12000|     100|0.33862926|  PASSED
    diehard_2dsphere|   2|      8000|     100|0.92727871|  PASSED
    diehard_3dsphere|   3|      4000|     100|0.64505493|  PASSED
     diehard_squeeze|   0|    100000|     100|0.34169225|  PASSED
        diehard_runs|   0|    100000|     100|0.59900311|  PASSED
        diehard_runs|   0|    100000|     100|0.21190294|  PASSED
       diehard_craps|   0|    200000|     100|0.58475844|  PASSED
       diehard_craps|   0|    200000|     100|0.77494445|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.51868282|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.97915532|  PASSED
         sts_monobit|   1|    100000|     100|0.12297650|  PASSED
            sts_runs|   2|    100000|     100|0.05475608|  PASSED
          sts_serial|   1|    100000|     100|0.41231819|  PASSED
          sts_serial|   2|    100000|     100|0.66161569|  PASSED
          sts_serial|   3|    100000|     100|0.62328057|  PASSED
          sts_serial|   3|    100000|     100|0.04507747|  PASSED
          sts_serial|   4|    100000|     100|0.99970747|   WEAK
          sts_serial|   4|    100000|     100|0.37053979|  PASSED
          sts_serial|   5|    100000|     100|0.98107642|  PASSED
          sts_serial|   5|    100000|     100|0.86047627|  PASSED
          sts_serial|   6|    100000|     100|0.15140809|  PASSED
          sts_serial|   6|    100000|     100|0.02282848|  PASSED
          sts_serial|   7|    100000|     100|0.18752824|  PASSED
          sts_serial|   7|    100000|     100|0.65659295|  PASSED
          sts_serial|   8|    100000|     100|0.49265221|  PASSED
          sts_serial|   8|    100000|     100|0.43109258|  PASSED
          sts_serial|   9|    100000|     100|0.32604993|  PASSED
          sts_serial|   9|    100000|     100|0.47678946|  PASSED
          sts_serial|  10|    100000|     100|0.14393008|  PASSED
          sts_serial|  10|    100000|     100|0.08478875|  PASSED
          sts_serial|  11|    100000|     100|0.23276385|  PASSED
          sts_serial|  11|    100000|     100|0.51476401|  PASSED
          sts_serial|  12|    100000|     100|0.94767643|  PASSED
          sts_serial|  12|    100000|     100|0.22477282|  PASSED
          sts_serial|  13|    100000|     100|0.95171104|  PASSED
          sts_serial|  13|    100000|     100|0.99763519|   WEAK
          sts_serial|  14|    100000|     100|0.75134240|  PASSED
          sts_serial|  14|    100000|     100|0.55977396|  PASSED
          sts_serial|  15|    100000|     100|0.22884332|  PASSED
          sts_serial|  15|    100000|     100|0.53905607|  PASSED
```

```
         sts_serial|  16|    100000|     100|0.93487670|  PASSED
         sts_serial|  16|    100000|     100|0.26844067|  PASSED
        rgb_bitdist|   1|    100000|     100|0.69597111|  PASSED
        rgb_bitdist|   2|    100000|     100|0.53747116|  PASSED
        rgb_bitdist|   3|    100000|     100|0.61512579|  PASSED
        rgb_bitdist|   4|    100000|     100|0.97614483|  PASSED
        rgb_bitdist|   5|    100000|     100|0.80809337|  PASSED
        rgb_bitdist|   6|    100000|     100|0.41125163|  PASSED
        rgb_bitdist|   7|    100000|     100|0.67358603|  PASSED
        rgb_bitdist|   8|    100000|     100|0.70258637|  PASSED
        rgb_bitdist|   9|    100000|     100|0.54582044|  PASSED
        rgb_bitdist|  10|    100000|     100|0.83384828|  PASSED
        rgb_bitdist|  11|    100000|     100|0.90352116|  PASSED
        rgb_bitdist|  12|    100000|     100|0.79713497|  PASSED
rgb_minimum_distance|   2|     10000|    1000|0.86277250|  PASSED
rgb_minimum_distance|   3|     10000|    1000|0.87843710|  PASSED
rgb_minimum_distance|   4|     10000|    1000|0.13950609|  PASSED
rgb_minimum_distance|   5|     10000|    1000|0.00502820|  PASSED
    rgb_permutations|   2|    100000|     100|0.83885179|  PASSED
    rgb_permutations|   3|    100000|     100|0.27295976|  PASSED
    rgb_permutations|   4|    100000|     100|0.91783140|  PASSED
    rgb_permutations|   5|    100000|     100|0.91163391|  PASSED
      rgb_lagged_sum|   0|   1000000|     100|0.93939226|  PASSED
      rgb_lagged_sum|   1|   1000000|     100|0.42139682|  PASSED
      rgb_lagged_sum|   2|   1000000|     100|0.62939949|  PASSED
      rgb_lagged_sum|   3|   1000000|     100|0.56091359|  PASSED
      rgb_lagged_sum|   4|   1000000|     100|0.49996043|  PASSED
      rgb_lagged_sum|   5|   1000000|     100|0.98159904|  PASSED
      rgb_lagged_sum|   6|   1000000|     100|0.80116434|  PASSED
      rgb_lagged_sum|   7|   1000000|     100|0.92091181|  PASSED
      rgb_lagged_sum|   8|   1000000|     100|0.35513552|  PASSED
      rgb_lagged_sum|   9|   1000000|     100|0.13152739|  PASSED
      rgb_lagged_sum|  10|   1000000|     100|0.93808921|  PASSED
      rgb_lagged_sum|  11|   1000000|     100|0.89831034|  PASSED
      rgb_lagged_sum|  12|   1000000|     100|0.50588204|  PASSED
      rgb_lagged_sum|  13|   1000000|     100|0.41965439|  PASSED
      rgb_lagged_sum|  14|   1000000|     100|0.40825785|  PASSED
      rgb_lagged_sum|  15|   1000000|     100|0.25938220|  PASSED
      rgb_lagged_sum|  16|   1000000|     100|0.56044096|  PASSED
      rgb_lagged_sum|  17|   1000000|     100|0.96187658|  PASSED
      rgb_lagged_sum|  18|   1000000|     100|0.67253781|  PASSED
      rgb_lagged_sum|  19|   1000000|     100|0.29076228|  PASSED
      rgb_lagged_sum|  20|   1000000|     100|0.14735346|  PASSED
      rgb_lagged_sum|  21|   1000000|     100|0.94320175|  PASSED
      rgb_lagged_sum|  22|   1000000|     100|0.54837865|  PASSED
      rgb_lagged_sum|  23|   1000000|     100|0.97201252|  PASSED
      rgb_lagged_sum|  24|   1000000|     100|0.37334658|  PASSED
      rgb_lagged_sum|  25|   1000000|     100|0.53664037|  PASSED
      rgb_lagged_sum|  26|   1000000|     100|0.91196769|  PASSED
      rgb_lagged_sum|  27|   1000000|     100|0.85504024|  PASSED
      rgb_lagged_sum|  28|   1000000|     100|0.47755512|  PASSED
      rgb_lagged_sum|  29|   1000000|     100|0.19500970|  PASSED
      rgb_lagged_sum|  30|   1000000|     100|0.88541635|  PASSED
      rgb_lagged_sum|  31|   1000000|     100|0.66863823|  PASSED
      rgb_lagged_sum|  32|   1000000|     100|0.91330177|  PASSED
     rgb_kstest_test|   0|     10000|    1000|0.33307896|  PASSED
     dab_bytedistrib|   0|  51200000|       1|0.34309155|  PASSED
             dab_dct| 256|     50000|       1|0.77858514|  PASSED
Skipping test 207
Preparing to run test 208.  ntuple = 0
        dab_filltree2|   0|   5000000|       1|0.65590257|  PASSED
        dab_filltree2|   1|   5000000|       1|0.21196507|  PASSED
Preparing to run test 209.  ntuple = 0
         dab_monobit2|  12|  65000000|       1|0.51133151|  PASSED
Preparing to run test 210.  ntuple = 0
#=============================================================================#
#   mean  |  stddev |  error-rate (best = 0.0, worst = 0.5)
#=============================================================================#
0.223971  |0.140406  |
```

Listing 8.3: Test results for random number engine `trng::lcg64_count_shift`.

```
#=============================================================================#
#            dieharder version 3.31.2beta Copyright 2003 Robert G. Brown      #
#=============================================================================#
   rng_name    |rands/second|   Seed   |  k ints/sec|k doubles/sec|
trng_lcg64_count_shift|  3.18e+08  |3118895655|    318258   |     317238   |
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
   diehard_birthdays|   0|       100|     100|0.99479962|  PASSED
      diehard_operm5|   0|   1000000|     100|0.94717530|  PASSED
  diehard_rank_32x32|   0|     40000|     100|0.54663220|  PASSED
    diehard_rank_6x8|   0|    100000|     100|0.38749076|  PASSED
   diehard_bitstream|   0|   2097152|     100|0.67441833|  PASSED
        diehard_opso|   0|   2097152|     100|0.31549464|  PASSED
        diehard_oqso|   0|   2097152|     100|0.79952052|  PASSED
         diehard_dna|   0|   2097152|     100|0.02370751|  PASSED
diehard_count_1s_str|   0|    256000|     100|0.10373962|  PASSED
diehard_count_1s_byt|   0|    256000|     100|0.97926558|  PASSED
 diehard_parking_lot|   0|     12000|     100|0.10437273|  PASSED
    diehard_2dsphere|   2|      8000|     100|0.78782733|  PASSED
    diehard_3dsphere|   3|      4000|     100|0.30386806|  PASSED
     diehard_squeeze|   0|    100000|     100|0.83723483|  PASSED
        diehard_runs|   0|    100000|     100|0.29686258|  PASSED
        diehard_runs|   0|    100000|     100|0.40735691|  PASSED
       diehard_craps|   0|    200000|     100|0.60023718|  PASSED
       diehard_craps|   0|    200000|     100|0.86847850|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.53847115|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.95278803|  PASSED
         sts_monobit|   1|    100000|     100|0.55189877|  PASSED
            sts_runs|   2|    100000|     100|0.77199571|  PASSED
          sts_serial|   1|    100000|     100|0.26799109|  PASSED
          sts_serial|   2|    100000|     100|0.78064827|  PASSED
          sts_serial|   3|    100000|     100|0.11264792|  PASSED
          sts_serial|   3|    100000|     100|0.09402199|  PASSED
          sts_serial|   4|    100000|     100|0.00536119|  PASSED
          sts_serial|   4|    100000|     100|0.16655331|  PASSED
          sts_serial|   5|    100000|     100|0.02846078|  PASSED
          sts_serial|   5|    100000|     100|0.80312914|  PASSED
          sts_serial|   6|    100000|     100|0.00694571|  PASSED
          sts_serial|   6|    100000|     100|0.91950937|  PASSED
          sts_serial|   7|    100000|     100|0.22540561|  PASSED
          sts_serial|   7|    100000|     100|0.62951007|  PASSED
          sts_serial|   8|    100000|     100|0.22381596|  PASSED
          sts_serial|   8|    100000|     100|0.98097024|  PASSED
          sts_serial|   9|    100000|     100|0.87958434|  PASSED
          sts_serial|   9|    100000|     100|0.26560557|  PASSED
          sts_serial|  10|    100000|     100|0.59175321|  PASSED
          sts_serial|  10|    100000|     100|0.27271635|  PASSED
          sts_serial|  11|    100000|     100|0.19323614|  PASSED
          sts_serial|  11|    100000|     100|0.13587161|  PASSED
          sts_serial|  12|    100000|     100|0.02703775|  PASSED
          sts_serial|  12|    100000|     100|0.64353639|  PASSED
          sts_serial|  13|    100000|     100|0.24275388|  PASSED
          sts_serial|  13|    100000|     100|0.42725413|  PASSED
          sts_serial|  14|    100000|     100|0.01313964|  PASSED
          sts_serial|  14|    100000|     100|0.30492917|  PASSED
          sts_serial|  15|    100000|     100|0.29718277|  PASSED
          sts_serial|  15|    100000|     100|0.67338582|  PASSED
          sts_serial|  16|    100000|     100|0.23094450|  PASSED
          sts_serial|  16|    100000|     100|0.86074664|  PASSED
         rgb_bitdist|   1|    100000|     100|0.12802284|  PASSED
         rgb_bitdist|   2|    100000|     100|0.27921612|  PASSED
         rgb_bitdist|   3|    100000|     100|0.98647211|  PASSED
         rgb_bitdist|   4|    100000|     100|0.18168649|  PASSED
         rgb_bitdist|   5|    100000|     100|0.72037822|  PASSED
         rgb_bitdist|   6|    100000|     100|0.16742041|  PASSED
         rgb_bitdist|   7|    100000|     100|0.65639582|  PASSED
         rgb_bitdist|   8|    100000|     100|0.43552181|  PASSED
```

```
          rgb_bitdist|   9|    100000|     100|0.80319567|  PASSED
          rgb_bitdist|  10|    100000|     100|0.12677528|  PASSED
          rgb_bitdist|  11|    100000|     100|0.09550224|  PASSED
          rgb_bitdist|  12|    100000|     100|0.54128996|  PASSED
rgb_minimum_distance|   2|     10000|    1000|0.40518125|  PASSED
rgb_minimum_distance|   3|     10000|    1000|0.23409387|  PASSED
rgb_minimum_distance|   4|     10000|    1000|0.17705177|  PASSED
rgb_minimum_distance|   5|     10000|    1000|0.22733922|  PASSED
    rgb_permutations|   2|    100000|     100|0.71917781|  PASSED
    rgb_permutations|   3|    100000|     100|0.70594878|  PASSED
    rgb_permutations|   4|    100000|     100|0.92037693|  PASSED
    rgb_permutations|   5|    100000|     100|0.33379531|  PASSED
      rgb_lagged_sum|   0|   1000000|     100|0.88816843|  PASSED
      rgb_lagged_sum|   1|   1000000|     100|0.37612519|  PASSED
      rgb_lagged_sum|   2|   1000000|     100|0.02834617|  PASSED
      rgb_lagged_sum|   3|   1000000|     100|0.33679497|  PASSED
      rgb_lagged_sum|   4|   1000000|     100|0.96925338|  PASSED
      rgb_lagged_sum|   5|   1000000|     100|0.35442443|  PASSED
      rgb_lagged_sum|   6|   1000000|     100|0.33425826|  PASSED
      rgb_lagged_sum|   7|   1000000|     100|0.83851644|  PASSED
      rgb_lagged_sum|   8|   1000000|     100|0.26403452|  PASSED
      rgb_lagged_sum|   9|   1000000|     100|0.50703905|  PASSED
      rgb_lagged_sum|  10|   1000000|     100|0.70235889|  PASSED
      rgb_lagged_sum|  11|   1000000|     100|0.71773742|  PASSED
      rgb_lagged_sum|  12|   1000000|     100|0.80460021|  PASSED
      rgb_lagged_sum|  13|   1000000|     100|0.48751203|  PASSED
      rgb_lagged_sum|  14|   1000000|     100|0.17104808|  PASSED
      rgb_lagged_sum|  15|   1000000|     100|0.65187787|  PASSED
      rgb_lagged_sum|  16|   1000000|     100|0.24837117|  PASSED
      rgb_lagged_sum|  17|   1000000|     100|0.17038198|  PASSED
      rgb_lagged_sum|  18|   1000000|     100|0.26229980|  PASSED
      rgb_lagged_sum|  19|   1000000|     100|0.92348090|  PASSED
      rgb_lagged_sum|  20|   1000000|     100|0.78766513|  PASSED
      rgb_lagged_sum|  21|   1000000|     100|0.05198213|  PASSED
      rgb_lagged_sum|  22|   1000000|     100|0.70681542|  PASSED
      rgb_lagged_sum|  23|   1000000|     100|0.98132965|  PASSED
      rgb_lagged_sum|  24|   1000000|     100|0.10870557|  PASSED
      rgb_lagged_sum|  25|   1000000|     100|0.87675459|  PASSED
      rgb_lagged_sum|  26|   1000000|     100|0.22492381|  PASSED
      rgb_lagged_sum|  27|   1000000|     100|0.54984973|  PASSED
      rgb_lagged_sum|  28|   1000000|     100|0.18130703|  PASSED
      rgb_lagged_sum|  29|   1000000|     100|0.65104194|  PASSED
      rgb_lagged_sum|  30|   1000000|     100|0.89058579|  PASSED
      rgb_lagged_sum|  31|   1000000|     100|0.40023484|  PASSED
      rgb_lagged_sum|  32|   1000000|     100|0.36036936|  PASSED
     rgb_kstest_test|   0|     10000|    1000|0.14996736|  PASSED
     dab_bytedistrib|   0|  51200000|       1|0.73470144|  PASSED
             dab_dct| 256|     50000|       1|0.28302370|  PASSED
Skipping test 207
Preparing to run test 208.  ntuple = 0
        dab_filltree2|   0|   5000000|       1|0.54898122|  PASSED
        dab_filltree2|   1|   5000000|       1|0.80560502|  PASSED
Preparing to run test 209.  ntuple = 0
         dab_monobit2|  12|  65000000|       1|0.89538597|  PASSED
Preparing to run test 210.  ntuple = 0
#=============================================================================#
#   mean   |  stddev |  error-rate (best = 0.0, worst = 0.5)
#=============================================================================#
0.199983  |0.126859  |
```

Listing 8.4: Test results for random number engine `trng::mrg2`.

```
#=============================================================================#
#          dieharder version 3.31.2beta Copyright 2003 Robert G. Brown        #
#=============================================================================#
   rng_name    |rands/second|   Seed   |  k ints/sec|k doubles/sec|
      trng_mrg2|  2.38e+08  | 293173125|    237563  |     235499  |
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
```

```
    diehard_birthdays|   0|       100|     100|0.81145710|  PASSED
       diehard_operm5|   0|   1000000|     100|0.51831897|  PASSED
   diehard_rank_32x32|   0|     40000|     100|0.96638905|  PASSED
     diehard_rank_6x8|   0|    100000|     100|0.99509371|   WEAK
    diehard_bitstream|   0|   2097152|     100|0.39687320|  PASSED
         diehard_opso|   0|   2097152|     100|0.53627788|  PASSED
         diehard_oqso|   0|   2097152|     100|0.59513263|  PASSED
          diehard_dna|   0|   2097152|     100|0.00000000|  FAILED
diehard_count_1s_str|   0|    256000|     100|0.83791757|  PASSED
diehard_count_1s_byt|   0|    256000|     100|0.15073500|  PASSED
 diehard_parking_lot|   0|     12000|     100|0.33358085|  PASSED
    diehard_2dsphere|   2|      8000|     100|0.01437800|  PASSED
    diehard_3dsphere|   3|      4000|     100|0.57566959|  PASSED
     diehard_squeeze|   0|    100000|     100|0.99996423|   WEAK
        diehard_runs|   0|    100000|     100|0.72554575|  PASSED
        diehard_runs|   0|    100000|     100|0.95575129|  PASSED
       diehard_craps|   0|    200000|     100|0.40909579|  PASSED
       diehard_craps|   0|    200000|     100|0.96240682|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.26138508|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.78268209|  PASSED
         sts_monobit|   1|    100000|     100|0.02960280|  PASSED
            sts_runs|   2|    100000|     100|0.45751580|  PASSED
          sts_serial|   1|    100000|     100|0.30242544|  PASSED
          sts_serial|   2|    100000|     100|0.20755329|  PASSED
          sts_serial|   3|    100000|     100|0.22997812|  PASSED
          sts_serial|   3|    100000|     100|0.98315989|  PASSED
          sts_serial|   4|    100000|     100|0.34638771|  PASSED
          sts_serial|   4|    100000|     100|0.97768514|  PASSED
          sts_serial|   5|    100000|     100|0.53408069|  PASSED
          sts_serial|   5|    100000|     100|0.80673438|  PASSED
          sts_serial|   6|    100000|     100|0.91503703|  PASSED
          sts_serial|   6|    100000|     100|0.86287512|  PASSED
          sts_serial|   7|    100000|     100|0.92993431|  PASSED
          sts_serial|   7|    100000|     100|0.59999256|  PASSED
          sts_serial|   8|    100000|     100|0.53366237|  PASSED
          sts_serial|   8|    100000|     100|0.29444933|  PASSED
          sts_serial|   9|    100000|     100|0.00963824|  PASSED
          sts_serial|   9|    100000|     100|0.09154335|  PASSED
          sts_serial|  10|    100000|     100|0.16566075|  PASSED
          sts_serial|  10|    100000|     100|0.37009801|  PASSED
          sts_serial|  11|    100000|     100|0.18856036|  PASSED
          sts_serial|  11|    100000|     100|0.50897396|  PASSED
          sts_serial|  12|    100000|     100|0.83593420|  PASSED
          sts_serial|  12|    100000|     100|0.24278743|  PASSED
          sts_serial|  13|    100000|     100|0.92539468|  PASSED
          sts_serial|  13|    100000|     100|0.32446917|  PASSED
          sts_serial|  14|    100000|     100|0.84792813|  PASSED
          sts_serial|  14|    100000|     100|0.98077692|  PASSED
          sts_serial|  15|    100000|     100|0.42985984|  PASSED
          sts_serial|  15|    100000|     100|0.05543002|  PASSED
          sts_serial|  16|    100000|     100|0.76850295|  PASSED
          sts_serial|  16|    100000|     100|0.66695225|  PASSED
         rgb_bitdist|   1|    100000|     100|0.41281567|  PASSED
         rgb_bitdist|   2|    100000|     100|0.42053888|  PASSED
         rgb_bitdist|   3|    100000|     100|0.51584740|  PASSED
         rgb_bitdist|   4|    100000|     100|0.79193124|  PASSED
         rgb_bitdist|   5|    100000|     100|0.53392696|  PASSED
         rgb_bitdist|   6|    100000|     100|0.92787456|  PASSED
         rgb_bitdist|   7|    100000|     100|0.85733317|  PASSED
         rgb_bitdist|   8|    100000|     100|0.34561769|  PASSED
         rgb_bitdist|   9|    100000|     100|0.49205570|  PASSED
         rgb_bitdist|  10|    100000|     100|0.14334039|  PASSED
         rgb_bitdist|  11|    100000|     100|0.26937408|  PASSED
         rgb_bitdist|  12|    100000|     100|0.75458640|  PASSED
rgb_minimum_distance|   2|     10000|    1000|0.46502045|  PASSED
rgb_minimum_distance|   3|     10000|    1000|0.88771554|  PASSED
rgb_minimum_distance|   4|     10000|    1000|0.10973417|  PASSED
rgb_minimum_distance|   5|     10000|    1000|0.09506323|  PASSED
    rgb_permutations|   2|    100000|     100|0.47460946|  PASSED
    rgb_permutations|   3|    100000|     100|0.87765088|  PASSED
```

```
    rgb_permutations|   4|    100000|     100|0.90527862|  PASSED
    rgb_permutations|   5|    100000|     100|0.58949756|  PASSED
      rgb_lagged_sum|   0|   1000000|     100|0.03708825|  PASSED
      rgb_lagged_sum|   1|   1000000|     100|0.96652376|  PASSED
      rgb_lagged_sum|   2|   1000000|     100|0.20667080|  PASSED
      rgb_lagged_sum|   3|   1000000|     100|0.87199215|  PASSED
      rgb_lagged_sum|   4|   1000000|     100|0.98591080|  PASSED
      rgb_lagged_sum|   5|   1000000|     100|0.52390121|  PASSED
      rgb_lagged_sum|   6|   1000000|     100|0.99891559|   WEAK
      rgb_lagged_sum|   7|   1000000|     100|0.46631327|  PASSED
      rgb_lagged_sum|   8|   1000000|     100|0.00064564|   WEAK
      rgb_lagged_sum|   9|   1000000|     100|0.54264546|  PASSED
      rgb_lagged_sum|  10|   1000000|     100|0.64354408|  PASSED
      rgb_lagged_sum|  11|   1000000|     100|0.43052392|  PASSED
      rgb_lagged_sum|  12|   1000000|     100|0.12947289|  PASSED
      rgb_lagged_sum|  13|   1000000|     100|0.40660381|  PASSED
      rgb_lagged_sum|  14|   1000000|     100|0.05407115|  PASSED
      rgb_lagged_sum|  15|   1000000|     100|0.43949745|  PASSED
      rgb_lagged_sum|  16|   1000000|     100|0.64883608|  PASSED
      rgb_lagged_sum|  17|   1000000|     100|0.29885007|  PASSED
      rgb_lagged_sum|  18|   1000000|     100|0.90009736|  PASSED
      rgb_lagged_sum|  19|   1000000|     100|0.74713123|  PASSED
      rgb_lagged_sum|  20|   1000000|     100|0.01256825|  PASSED
      rgb_lagged_sum|  21|   1000000|     100|0.61113240|  PASSED
      rgb_lagged_sum|  22|   1000000|     100|0.20481414|  PASSED
      rgb_lagged_sum|  23|   1000000|     100|0.01462602|  PASSED
      rgb_lagged_sum|  24|   1000000|     100|0.62197653|  PASSED
      rgb_lagged_sum|  25|   1000000|     100|0.54555124|  PASSED
      rgb_lagged_sum|  26|   1000000|     100|0.37113145|  PASSED
      rgb_lagged_sum|  27|   1000000|     100|0.81834504|  PASSED
      rgb_lagged_sum|  28|   1000000|     100|0.76463844|  PASSED
      rgb_lagged_sum|  29|   1000000|     100|0.85876154|  PASSED
      rgb_lagged_sum|  30|   1000000|     100|0.94088166|  PASSED
      rgb_lagged_sum|  31|   1000000|     100|0.48054621|  PASSED
      rgb_lagged_sum|  32|   1000000|     100|0.98218869|  PASSED
     rgb_kstest_test|   0|     10000|    1000|0.41773337|  PASSED
     dab_bytedistrib|   0|  51200000|       1|0.15934943|  PASSED
             dab_dct| 256|     50000|       1|0.88860952|  PASSED
Skipping test 207
Preparing to run test 208.  ntuple = 0
       dab_filltree2|   0|   5000000|       1|0.73013566|  PASSED
       dab_filltree2|   1|   5000000|       1|0.05083696|  PASSED
Preparing to run test 209.  ntuple = 0
        dab_monobit2|  12|  65000000|       1|0.93331821|  PASSED
Preparing to run test 210.  ntuple = 0
#=============================================================================#
#   mean   |  stddev |  error-rate (best = 0.0, worst = 0.5)
#=============================================================================#
0.195536  |0.166823  |
```

Listing 8.5: Test results for random number engine `trng::mrg3`.

```
#=============================================================================#
#          dieharder version 3.31.2beta Copyright 2003 Robert G. Brown       #
#=============================================================================#
   rng_name    |rands/second|   Seed   |  k ints/sec|k doubles/sec|
      trng_mrg3|  1.21e+08  |1952899518|    120743  |     122723  |
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
   diehard_birthdays|   0|       100|     100|0.52073911|  PASSED
      diehard_operm5|   0|   1000000|     100|0.90069820|  PASSED
  diehard_rank_32x32|   0|     40000|     100|0.46113137|  PASSED
    diehard_rank_6x8|   0|    100000|     100|0.80164486|  PASSED
   diehard_bitstream|   0|   2097152|     100|0.61694098|  PASSED
        diehard_opso|   0|   2097152|     100|0.69044317|  PASSED
        diehard_oqso|   0|   2097152|     100|0.45274954|  PASSED
         diehard_dna|   0|   2097152|     100|0.00000000|  FAILED
diehard_count_1s_str|   0|    256000|     100|0.78291434|  PASSED
diehard_count_1s_byt|   0|    256000|     100|0.93887955|  PASSED
```

```
   diehard_parking_lot|   0|     12000|     100|0.66719833|  PASSED
      diehard_2dsphere|   2|      8000|     100|0.43290368|  PASSED
      diehard_3dsphere|   3|      4000|     100|0.29343368|  PASSED
       diehard_squeeze|   0|    100000|     100|0.79544911|  PASSED
          diehard_runs|   0|    100000|     100|0.76359623|  PASSED
          diehard_runs|   0|    100000|     100|0.00182929|   WEAK
         diehard_craps|   0|    200000|     100|0.12555311|  PASSED
         diehard_craps|   0|    200000|     100|0.52635290|  PASSED
   marsaglia_tsang_gcd|   0|  10000000|     100|0.96525861|  PASSED
   marsaglia_tsang_gcd|   0|  10000000|     100|0.67128151|  PASSED
           sts_monobit|   1|    100000|     100|0.76072457|  PASSED
              sts_runs|   2|    100000|     100|0.86602116|  PASSED
            sts_serial|   1|    100000|     100|0.05316046|  PASSED
            sts_serial|   2|    100000|     100|0.33603454|  PASSED
            sts_serial|   3|    100000|     100|0.42013982|  PASSED
            sts_serial|   3|    100000|     100|0.63269459|  PASSED
            sts_serial|   4|    100000|     100|0.73409041|  PASSED
            sts_serial|   4|    100000|     100|0.66854125|  PASSED
            sts_serial|   5|    100000|     100|0.60998404|  PASSED
            sts_serial|   5|    100000|     100|0.71075497|  PASSED
            sts_serial|   6|    100000|     100|0.58518662|  PASSED
            sts_serial|   6|    100000|     100|0.96370445|  PASSED
            sts_serial|   7|    100000|     100|0.84033364|  PASSED
            sts_serial|   7|    100000|     100|0.85857503|  PASSED
            sts_serial|   8|    100000|     100|0.20588806|  PASSED
            sts_serial|   8|    100000|     100|0.88661483|  PASSED
            sts_serial|   9|    100000|     100|0.86226034|  PASSED
            sts_serial|   9|    100000|     100|0.58711555|  PASSED
            sts_serial|  10|    100000|     100|0.39852302|  PASSED
            sts_serial|  10|    100000|     100|0.12936376|  PASSED
            sts_serial|  11|    100000|     100|0.89432106|  PASSED
            sts_serial|  11|    100000|     100|0.61019277|  PASSED
            sts_serial|  12|    100000|     100|0.62447398|  PASSED
            sts_serial|  12|    100000|     100|0.54109152|  PASSED
            sts_serial|  13|    100000|     100|0.18488958|  PASSED
            sts_serial|  13|    100000|     100|0.28122461|  PASSED
            sts_serial|  14|    100000|     100|0.30979341|  PASSED
            sts_serial|  14|    100000|     100|0.97834754|  PASSED
            sts_serial|  15|    100000|     100|0.88327755|  PASSED
            sts_serial|  15|    100000|     100|0.88836428|  PASSED
            sts_serial|  16|    100000|     100|0.55827519|  PASSED
            sts_serial|  16|    100000|     100|0.05988389|  PASSED
           rgb_bitdist|   1|    100000|     100|0.73316646|  PASSED
           rgb_bitdist|   2|    100000|     100|0.23362776|  PASSED
           rgb_bitdist|   3|    100000|     100|0.84062517|  PASSED
           rgb_bitdist|   4|    100000|     100|0.47789424|  PASSED
           rgb_bitdist|   5|    100000|     100|0.98964518|  PASSED
           rgb_bitdist|   6|    100000|     100|0.43319992|  PASSED
           rgb_bitdist|   7|    100000|     100|0.15062280|  PASSED
           rgb_bitdist|   8|    100000|     100|0.06439724|  PASSED
           rgb_bitdist|   9|    100000|     100|0.16558963|  PASSED
           rgb_bitdist|  10|    100000|     100|0.68659529|  PASSED
           rgb_bitdist|  11|    100000|     100|0.00148126|   WEAK
           rgb_bitdist|  12|    100000|     100|0.83733037|  PASSED
  rgb_minimum_distance|   2|     10000|    1000|0.88554973|  PASSED
  rgb_minimum_distance|   3|     10000|    1000|0.49724744|  PASSED
  rgb_minimum_distance|   4|     10000|    1000|0.89910887|  PASSED
  rgb_minimum_distance|   5|     10000|    1000|0.21561819|  PASSED
      rgb_permutations|   2|    100000|     100|0.00879716|  PASSED
      rgb_permutations|   3|    100000|     100|0.14292589|  PASSED
      rgb_permutations|   4|    100000|     100|0.16170562|  PASSED
      rgb_permutations|   5|    100000|     100|0.74644601|  PASSED
        rgb_lagged_sum|   0|   1000000|     100|0.66867813|  PASSED
        rgb_lagged_sum|   1|   1000000|     100|0.78908067|  PASSED
        rgb_lagged_sum|   2|   1000000|     100|0.45358935|  PASSED
        rgb_lagged_sum|   3|   1000000|     100|0.32975344|  PASSED
        rgb_lagged_sum|   4|   1000000|     100|0.54388662|  PASSED
        rgb_lagged_sum|   5|   1000000|     100|0.38186023|  PASSED
        rgb_lagged_sum|   6|   1000000|     100|0.99678738|   WEAK
        rgb_lagged_sum|   7|   1000000|     100|0.69183865|  PASSED
```

```
      rgb_lagged_sum|   8|   1000000|     100|0.90547494|  PASSED
      rgb_lagged_sum|   9|   1000000|     100|0.84618457|  PASSED
      rgb_lagged_sum|  10|   1000000|     100|0.26335279|  PASSED
      rgb_lagged_sum|  11|   1000000|     100|0.30999503|  PASSED
      rgb_lagged_sum|  12|   1000000|     100|0.47211537|  PASSED
      rgb_lagged_sum|  13|   1000000|     100|0.86548180|  PASSED
      rgb_lagged_sum|  14|   1000000|     100|0.40420728|  PASSED
      rgb_lagged_sum|  15|   1000000|     100|0.71411361|  PASSED
      rgb_lagged_sum|  16|   1000000|     100|0.08745708|  PASSED
      rgb_lagged_sum|  17|   1000000|     100|0.66478305|  PASSED
      rgb_lagged_sum|  18|   1000000|     100|0.27440242|  PASSED
      rgb_lagged_sum|  19|   1000000|     100|0.22997842|  PASSED
      rgb_lagged_sum|  20|   1000000|     100|0.48595073|  PASSED
      rgb_lagged_sum|  21|   1000000|     100|0.13805271|  PASSED
      rgb_lagged_sum|  22|   1000000|     100|0.81873706|  PASSED
      rgb_lagged_sum|  23|   1000000|     100|0.60164383|  PASSED
      rgb_lagged_sum|  24|   1000000|     100|0.92192398|  PASSED
      rgb_lagged_sum|  25|   1000000|     100|0.73527270|  PASSED
      rgb_lagged_sum|  26|   1000000|     100|0.92636718|  PASSED
      rgb_lagged_sum|  27|   1000000|     100|0.48781537|  PASSED
      rgb_lagged_sum|  28|   1000000|     100|0.21302907|  PASSED
      rgb_lagged_sum|  29|   1000000|     100|0.61415694|  PASSED
      rgb_lagged_sum|  30|   1000000|     100|0.03647843|  PASSED
      rgb_lagged_sum|  31|   1000000|     100|0.54163003|  PASSED
      rgb_lagged_sum|  32|   1000000|     100|0.85894057|  PASSED
     rgb_kstest_test|   0|     10000|    1000|0.23460411|  PASSED
     dab_bytedistrib|   0|  51200000|       1|0.95315547|  PASSED
             dab_dct| 256|     50000|       1|0.56050854|  PASSED
Skipping test 207
Preparing to run test 208.  ntuple = 0
        dab_filltree2|   0|   5000000|       1|0.35369817|  PASSED
        dab_filltree2|   1|   5000000|       1|0.25765151|  PASSED
Preparing to run test 209.  ntuple = 0
         dab_monobit2|  12|  65000000|       1|0.59453925|  PASSED
Preparing to run test 210.  ntuple = 0
#=============================================================================#
#   mean  |  stddev |  error-rate (best = 0.0, worst = 0.5)
#=============================================================================#
0.231450  |0.149255  |
```

Listing 8.6: Test results for random number engine `trng::mrg3s`.

```
#=============================================================================#
#          dieharder version 3.31.2beta Copyright 2003 Robert G. Brown        #
#=============================================================================#
   rng_name    |rands/second|   Seed   |  k ints/sec|k doubles/sec|
     trng_mrg3s|  1.51e+08  |3506756067|    151073  |     151457  |
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
   diehard_birthdays|   0|       100|     100|0.58747644|  PASSED
      diehard_operm5|   0|   1000000|     100|0.72155380|  PASSED
  diehard_rank_32x32|   0|     40000|     100|0.69474284|  PASSED
    diehard_rank_6x8|   0|    100000|     100|0.18289971|  PASSED
   diehard_bitstream|   0|   2097152|     100|0.69597158|  PASSED
        diehard_opso|   0|   2097152|     100|0.62917201|  PASSED
        diehard_oqso|   0|   2097152|     100|0.83879731|  PASSED
         diehard_dna|   0|   2097152|     100|0.00000000|  FAILED
diehard_count_1s_str|   0|    256000|     100|0.76224242|  PASSED
diehard_count_1s_byt|   0|    256000|     100|0.94216392|  PASSED
 diehard_parking_lot|   0|     12000|     100|0.46927986|  PASSED
    diehard_2dsphere|   2|      8000|     100|0.57960248|  PASSED
    diehard_3dsphere|   3|      4000|     100|0.83834394|  PASSED
     diehard_squeeze|   0|    100000|     100|0.91448178|  PASSED
        diehard_runs|   0|    100000|     100|0.47712643|  PASSED
        diehard_runs|   0|    100000|     100|0.69683873|  PASSED
       diehard_craps|   0|    200000|     100|0.44417420|  PASSED
       diehard_craps|   0|    200000|     100|0.19245412|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.99995734|   WEAK
 marsaglia_tsang_gcd|   0|  10000000|     100|0.83823815|  PASSED
```

```
         sts_monobit|   1|    100000|     100|0.16304623|  PASSED
            sts_runs|   2|    100000|     100|0.86721244|  PASSED
          sts_serial|   1|    100000|     100|0.81505073|  PASSED
          sts_serial|   2|    100000|     100|0.79724603|  PASSED
          sts_serial|   3|    100000|     100|0.99972551|   WEAK
          sts_serial|   3|    100000|     100|0.88962171|  PASSED
          sts_serial|   4|    100000|     100|0.79642439|  PASSED
          sts_serial|   4|    100000|     100|0.18263382|  PASSED
          sts_serial|   5|    100000|     100|0.93578803|  PASSED
          sts_serial|   5|    100000|     100|0.51280899|  PASSED
          sts_serial|   6|    100000|     100|0.19413883|  PASSED
          sts_serial|   6|    100000|     100|0.01355168|  PASSED
          sts_serial|   7|    100000|     100|0.33654302|  PASSED
          sts_serial|   7|    100000|     100|0.98493756|  PASSED
          sts_serial|   8|    100000|     100|0.74959537|  PASSED
          sts_serial|   8|    100000|     100|0.19556558|  PASSED
          sts_serial|   9|    100000|     100|0.97622724|  PASSED
          sts_serial|   9|    100000|     100|0.62495570|  PASSED
          sts_serial|  10|    100000|     100|0.58915552|  PASSED
          sts_serial|  10|    100000|     100|0.99161190|  PASSED
          sts_serial|  11|    100000|     100|0.98966445|  PASSED
          sts_serial|  11|    100000|     100|0.94785820|  PASSED
          sts_serial|  12|    100000|     100|0.91958125|  PASSED
          sts_serial|  12|    100000|     100|0.32437081|  PASSED
          sts_serial|  13|    100000|     100|0.94658383|  PASSED
          sts_serial|  13|    100000|     100|0.72193939|  PASSED
          sts_serial|  14|    100000|     100|0.98054565|  PASSED
          sts_serial|  14|    100000|     100|0.96519896|  PASSED
          sts_serial|  15|    100000|     100|0.86544023|  PASSED
          sts_serial|  15|    100000|     100|0.67751804|  PASSED
          sts_serial|  16|    100000|     100|0.39277625|  PASSED
          sts_serial|  16|    100000|     100|0.62359076|  PASSED
         rgb_bitdist|   1|    100000|     100|0.79579811|  PASSED
         rgb_bitdist|   2|    100000|     100|0.99827010|   WEAK
         rgb_bitdist|   3|    100000|     100|0.69422261|  PASSED
         rgb_bitdist|   4|    100000|     100|0.87352457|  PASSED
         rgb_bitdist|   5|    100000|     100|0.58400200|  PASSED
         rgb_bitdist|   6|    100000|     100|0.63209872|  PASSED
         rgb_bitdist|   7|    100000|     100|0.91376039|  PASSED
         rgb_bitdist|   8|    100000|     100|0.41602918|  PASSED
         rgb_bitdist|   9|    100000|     100|0.73025476|  PASSED
         rgb_bitdist|  10|    100000|     100|0.96657593|  PASSED
         rgb_bitdist|  11|    100000|     100|0.95414687|  PASSED
         rgb_bitdist|  12|    100000|     100|0.40076115|  PASSED
rgb_minimum_distance|   2|     10000|    1000|0.65374488|  PASSED
rgb_minimum_distance|   3|     10000|    1000|0.03078637|  PASSED
rgb_minimum_distance|   4|     10000|    1000|0.32209098|  PASSED
rgb_minimum_distance|   5|     10000|    1000|0.26938992|  PASSED
    rgb_permutations|   2|    100000|     100|0.98845868|  PASSED
    rgb_permutations|   3|    100000|     100|0.97417972|  PASSED
    rgb_permutations|   4|    100000|     100|0.96610613|  PASSED
    rgb_permutations|   5|    100000|     100|0.34580978|  PASSED
      rgb_lagged_sum|   0|   1000000|     100|0.77840398|  PASSED
      rgb_lagged_sum|   1|   1000000|     100|0.85266817|  PASSED
      rgb_lagged_sum|   2|   1000000|     100|0.39979380|  PASSED
      rgb_lagged_sum|   3|   1000000|     100|0.55636396|  PASSED
      rgb_lagged_sum|   4|   1000000|     100|0.85735364|  PASSED
      rgb_lagged_sum|   5|   1000000|     100|0.69800827|  PASSED
      rgb_lagged_sum|   6|   1000000|     100|0.57882827|  PASSED
      rgb_lagged_sum|   7|   1000000|     100|0.68610953|  PASSED
      rgb_lagged_sum|   8|   1000000|     100|0.66357986|  PASSED
      rgb_lagged_sum|   9|   1000000|     100|0.76708950|  PASSED
      rgb_lagged_sum|  10|   1000000|     100|0.94882183|  PASSED
      rgb_lagged_sum|  11|   1000000|     100|0.92445746|  PASSED
      rgb_lagged_sum|  12|   1000000|     100|0.24513777|  PASSED
      rgb_lagged_sum|  13|   1000000|     100|0.18611101|  PASSED
      rgb_lagged_sum|  14|   1000000|     100|0.98494619|  PASSED
      rgb_lagged_sum|  15|   1000000|     100|0.94472704|  PASSED
      rgb_lagged_sum|  16|   1000000|     100|0.99718915|   WEAK
      rgb_lagged_sum|  17|   1000000|     100|0.24070317|  PASSED
```

```
      rgb_lagged_sum|  18|   1000000|     100|0.69061461|  PASSED
      rgb_lagged_sum|  19|   1000000|     100|0.89649877|  PASSED
      rgb_lagged_sum|  20|   1000000|     100|0.40606659|  PASSED
      rgb_lagged_sum|  21|   1000000|     100|0.85923241|  PASSED
      rgb_lagged_sum|  22|   1000000|     100|0.47417913|  PASSED
      rgb_lagged_sum|  23|   1000000|     100|0.54392817|  PASSED
      rgb_lagged_sum|  24|   1000000|     100|0.34741219|  PASSED
      rgb_lagged_sum|  25|   1000000|     100|0.99535227|   WEAK
      rgb_lagged_sum|  26|   1000000|     100|0.82862160|  PASSED
      rgb_lagged_sum|  27|   1000000|     100|0.94676841|  PASSED
      rgb_lagged_sum|  28|   1000000|     100|0.05940732|  PASSED
      rgb_lagged_sum|  29|   1000000|     100|0.38670468|  PASSED
      rgb_lagged_sum|  30|   1000000|     100|0.37738836|  PASSED
      rgb_lagged_sum|  31|   1000000|     100|0.64988349|  PASSED
      rgb_lagged_sum|  32|   1000000|     100|0.17643835|  PASSED
     rgb_kstest_test|   0|     10000|    1000|0.70590221|  PASSED
     dab_bytedistrib|   0|  51200000|       1|0.69130579|  PASSED
             dab_dct| 256|     50000|       1|0.46782183|  PASSED
Skipping test 207
Preparing to run test 208.  ntuple = 0
        dab_filltree2|   0|   5000000|       1|0.57819923|  PASSED
        dab_filltree2|   1|   5000000|       1|0.68591353|  PASSED
Preparing to run test 209.  ntuple = 0
         dab_monobit2|  12|  65000000|       1|0.07281346|  PASSED
Preparing to run test 210.  ntuple = 0
#=============================================================================#
#   mean   |  stddev |  error-rate (best = 0.0, worst = 0.5)
#=============================================================================#
0.224068  |0.135649  |
```

Listing 8.7: Test results for random number engine `trng::mrg4`.

```
#=============================================================================#
#          dieharder version 3.31.2beta Copyright 2003 Robert G. Brown        #
#=============================================================================#
   rng_name    |rands/second|   Seed   |  k ints/sec|k doubles/sec|
      trng_mrg4|  1.03e+08  |3751699289|    102778  |     106185  |
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
   diehard_birthdays|   0|       100|     100|0.99573435|   WEAK
      diehard_operm5|   0|   1000000|     100|0.85929754|  PASSED
  diehard_rank_32x32|   0|     40000|     100|0.32775900|  PASSED
    diehard_rank_6x8|   0|    100000|     100|0.56035116|  PASSED
   diehard_bitstream|   0|   2097152|     100|0.87396654|  PASSED
        diehard_opso|   0|   2097152|     100|0.61469274|  PASSED
        diehard_oqso|   0|   2097152|     100|0.85965881|  PASSED
         diehard_dna|   0|   2097152|     100|0.00000000|  FAILED
diehard_count_1s_str|   0|    256000|     100|0.67630258|  PASSED
diehard_count_1s_byt|   0|    256000|     100|0.59899835|  PASSED
 diehard_parking_lot|   0|     12000|     100|0.43004782|  PASSED
    diehard_2dsphere|   2|      8000|     100|0.40140622|  PASSED
    diehard_3dsphere|   3|      4000|     100|0.63510393|  PASSED
     diehard_squeeze|   0|    100000|     100|0.95367534|  PASSED
        diehard_runs|   0|    100000|     100|0.53889758|  PASSED
        diehard_runs|   0|    100000|     100|0.94696448|  PASSED
       diehard_craps|   0|    200000|     100|0.55562462|  PASSED
       diehard_craps|   0|    200000|     100|0.65992648|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.91678904|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.77694097|  PASSED
         sts_monobit|   1|    100000|     100|0.55695227|  PASSED
            sts_runs|   2|    100000|     100|0.89697657|  PASSED
          sts_serial|   1|    100000|     100|0.81935082|  PASSED
          sts_serial|   2|    100000|     100|0.22742126|  PASSED
          sts_serial|   3|    100000|     100|0.98938643|  PASSED
          sts_serial|   3|    100000|     100|0.10651361|  PASSED
          sts_serial|   4|    100000|     100|0.69371425|  PASSED
          sts_serial|   4|    100000|     100|0.74065624|  PASSED
          sts_serial|   5|    100000|     100|0.22691957|  PASSED
          sts_serial|   5|    100000|     100|0.30715690|  PASSED
```

```
          sts_serial|   6|    100000|     100|0.18933425|  PASSED
          sts_serial|   6|    100000|     100|0.97266314|  PASSED
          sts_serial|   7|    100000|     100|0.03995393|  PASSED
          sts_serial|   7|    100000|     100|0.36177138|  PASSED
          sts_serial|   8|    100000|     100|0.53864244|  PASSED
          sts_serial|   8|    100000|     100|0.89417043|  PASSED
          sts_serial|   9|    100000|     100|0.65600490|  PASSED
          sts_serial|   9|    100000|     100|0.20825221|  PASSED
          sts_serial|  10|    100000|     100|0.73988002|  PASSED
          sts_serial|  10|    100000|     100|0.28473969|  PASSED
          sts_serial|  11|    100000|     100|0.50000804|  PASSED
          sts_serial|  11|    100000|     100|0.65766587|  PASSED
          sts_serial|  12|    100000|     100|0.83491155|  PASSED
          sts_serial|  12|    100000|     100|0.74993021|  PASSED
          sts_serial|  13|    100000|     100|0.98739467|  PASSED
          sts_serial|  13|    100000|     100|0.95543585|  PASSED
          sts_serial|  14|    100000|     100|0.22284194|  PASSED
          sts_serial|  14|    100000|     100|0.10226738|  PASSED
          sts_serial|  15|    100000|     100|0.19439411|  PASSED
          sts_serial|  15|    100000|     100|0.71202173|  PASSED
          sts_serial|  16|    100000|     100|0.70947629|  PASSED
          sts_serial|  16|    100000|     100|0.67857580|  PASSED
         rgb_bitdist|   1|    100000|     100|0.75762457|  PASSED
         rgb_bitdist|   2|    100000|     100|0.41814086|  PASSED
         rgb_bitdist|   3|    100000|     100|0.22966690|  PASSED
         rgb_bitdist|   4|    100000|     100|0.25269156|  PASSED
         rgb_bitdist|   5|    100000|     100|0.61610971|  PASSED
         rgb_bitdist|   6|    100000|     100|0.94067944|  PASSED
         rgb_bitdist|   7|    100000|     100|0.52688024|  PASSED
         rgb_bitdist|   8|    100000|     100|0.98701254|  PASSED
         rgb_bitdist|   9|    100000|     100|0.66300430|  PASSED
         rgb_bitdist|  10|    100000|     100|0.53832990|  PASSED
         rgb_bitdist|  11|    100000|     100|0.32306367|  PASSED
         rgb_bitdist|  12|    100000|     100|0.70129702|  PASSED
rgb_minimum_distance|   2|     10000|    1000|0.78519819|  PASSED
rgb_minimum_distance|   3|     10000|    1000|0.50442917|  PASSED
rgb_minimum_distance|   4|     10000|    1000|0.01457515|  PASSED
rgb_minimum_distance|   5|     10000|    1000|0.00636057|  PASSED
    rgb_permutations|   2|    100000|     100|0.99038654|  PASSED
    rgb_permutations|   3|    100000|     100|0.98195663|  PASSED
    rgb_permutations|   4|    100000|     100|0.16945802|  PASSED
    rgb_permutations|   5|    100000|     100|0.91178516|  PASSED
      rgb_lagged_sum|   0|   1000000|     100|0.18116097|  PASSED
      rgb_lagged_sum|   1|   1000000|     100|0.91283941|  PASSED
      rgb_lagged_sum|   2|   1000000|     100|0.72470077|  PASSED
      rgb_lagged_sum|   3|   1000000|     100|0.48092296|  PASSED
      rgb_lagged_sum|   4|   1000000|     100|0.72561860|  PASSED
      rgb_lagged_sum|   5|   1000000|     100|0.37497745|  PASSED
      rgb_lagged_sum|   6|   1000000|     100|0.26344743|  PASSED
      rgb_lagged_sum|   7|   1000000|     100|0.79749900|  PASSED
      rgb_lagged_sum|   8|   1000000|     100|0.75526152|  PASSED
      rgb_lagged_sum|   9|   1000000|     100|0.99413159|  PASSED
      rgb_lagged_sum|  10|   1000000|     100|0.69670849|  PASSED
      rgb_lagged_sum|  11|   1000000|     100|0.65977149|  PASSED
      rgb_lagged_sum|  12|   1000000|     100|0.53362225|  PASSED
      rgb_lagged_sum|  13|   1000000|     100|0.90280901|  PASSED
      rgb_lagged_sum|  14|   1000000|     100|0.55469374|  PASSED
      rgb_lagged_sum|  15|   1000000|     100|0.15546763|  PASSED
      rgb_lagged_sum|  16|   1000000|     100|0.61780161|  PASSED
      rgb_lagged_sum|  17|   1000000|     100|0.94322388|  PASSED
      rgb_lagged_sum|  18|   1000000|     100|0.67494278|  PASSED
      rgb_lagged_sum|  19|   1000000|     100|0.99808285|   WEAK
      rgb_lagged_sum|  20|   1000000|     100|0.48504159|  PASSED
      rgb_lagged_sum|  21|   1000000|     100|0.18463226|  PASSED
      rgb_lagged_sum|  22|   1000000|     100|0.07248522|  PASSED
      rgb_lagged_sum|  23|   1000000|     100|0.51778805|  PASSED
      rgb_lagged_sum|  24|   1000000|     100|0.09929599|  PASSED
      rgb_lagged_sum|  25|   1000000|     100|0.24886791|  PASSED
      rgb_lagged_sum|  26|   1000000|     100|0.87364821|  PASSED
      rgb_lagged_sum|  27|   1000000|     100|0.08284150|  PASSED
```

```
      rgb_lagged_sum|  28|   1000000|     100|0.92506950|  PASSED
      rgb_lagged_sum|  29|   1000000|     100|0.28105955|  PASSED
      rgb_lagged_sum|  30|   1000000|     100|0.09715932|  PASSED
      rgb_lagged_sum|  31|   1000000|     100|0.64167247|  PASSED
      rgb_lagged_sum|  32|   1000000|     100|0.36461017|  PASSED
     rgb_kstest_test|   0|     10000|    1000|0.90468984|  PASSED
     dab_bytedistrib|   0|  51200000|       1|0.11690202|  PASSED
             dab_dct| 256|     50000|       1|0.05977969|  PASSED
Skipping test 207
Preparing to run test 208.  ntuple = 0
       dab_filltree2|   0|   5000000|       1|0.13059056|  PASSED
       dab_filltree2|   1|   5000000|       1|0.16274249|  PASSED
Preparing to run test 209.  ntuple = 0
        dab_monobit2|  12|  65000000|       1|0.88534377|  PASSED
Preparing to run test 210.  ntuple = 0
#=============================================================================#
#   mean  |  stddev |  error-rate (best = 0.0, worst = 0.5)
#=============================================================================#
0.202309  |0.150519  |
```

Listing 8.8: Test results for random number engine `trng::mrg5`.

```
#=============================================================================#
#          dieharder version 3.31.2beta Copyright 2003 Robert G. Brown        #
#=============================================================================#
   rng_name    |rands/second|   Seed   |  k ints/sec|k doubles/sec|
      trng_mrg5|  2.00e+08  | 317009865|    199652  |     187143  |
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
   diehard_birthdays|   0|       100|     100|0.07496255|  PASSED
      diehard_operm5|   0|   1000000|     100|0.12306214|  PASSED
  diehard_rank_32x32|   0|     40000|     100|0.35028121|  PASSED
    diehard_rank_6x8|   0|    100000|     100|0.79035596|  PASSED
   diehard_bitstream|   0|   2097152|     100|0.89933764|  PASSED
        diehard_opso|   0|   2097152|     100|0.65796178|  PASSED
        diehard_oqso|   0|   2097152|     100|0.76639111|  PASSED
         diehard_dna|   0|   2097152|     100|0.00000000|  FAILED
diehard_count_1s_str|   0|    256000|     100|0.85600445|  PASSED
diehard_count_1s_byt|   0|    256000|     100|0.98520234|  PASSED
 diehard_parking_lot|   0|     12000|     100|0.35536156|  PASSED
    diehard_2dsphere|   2|      8000|     100|0.29551928|  PASSED
    diehard_3dsphere|   3|      4000|     100|0.53987473|  PASSED
     diehard_squeeze|   0|    100000|     100|0.90924904|  PASSED
        diehard_runs|   0|    100000|     100|0.45922265|  PASSED
        diehard_runs|   0|    100000|     100|0.99797233|   WEAK
       diehard_craps|   0|    200000|     100|0.23175176|  PASSED
       diehard_craps|   0|    200000|     100|0.68297462|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.29763029|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.61517344|  PASSED
         sts_monobit|   1|    100000|     100|0.45859509|  PASSED
            sts_runs|   2|    100000|     100|0.93037104|  PASSED
          sts_serial|   1|    100000|     100|0.99403467|  PASSED
          sts_serial|   2|    100000|     100|0.85923407|  PASSED
          sts_serial|   3|    100000|     100|0.98200684|  PASSED
          sts_serial|   3|    100000|     100|0.57063760|  PASSED
          sts_serial|   4|    100000|     100|0.68615271|  PASSED
          sts_serial|   4|    100000|     100|0.86355768|  PASSED
          sts_serial|   5|    100000|     100|0.26353002|  PASSED
          sts_serial|   5|    100000|     100|0.01708308|  PASSED
          sts_serial|   6|    100000|     100|0.47512179|  PASSED
          sts_serial|   6|    100000|     100|0.31609985|  PASSED
          sts_serial|   7|    100000|     100|0.62703475|  PASSED
          sts_serial|   7|    100000|     100|0.50472525|  PASSED
          sts_serial|   8|    100000|     100|0.77273335|  PASSED
          sts_serial|   8|    100000|     100|0.86284620|  PASSED
          sts_serial|   9|    100000|     100|0.36604451|  PASSED
          sts_serial|   9|    100000|     100|0.37963037|  PASSED
          sts_serial|  10|    100000|     100|0.82035464|  PASSED
          sts_serial|  10|    100000|     100|0.50427940|  PASSED
```

```
            sts_serial|  11|    100000|     100|0.72986313|  PASSED
            sts_serial|  11|    100000|     100|0.36314989|  PASSED
            sts_serial|  12|    100000|     100|0.02038200|  PASSED
            sts_serial|  12|    100000|     100|0.00142498|   WEAK
            sts_serial|  13|    100000|     100|0.08197842|  PASSED
            sts_serial|  13|    100000|     100|0.73944198|  PASSED
            sts_serial|  14|    100000|     100|0.07389248|  PASSED
            sts_serial|  14|    100000|     100|0.89152797|  PASSED
            sts_serial|  15|    100000|     100|0.10020482|  PASSED
            sts_serial|  15|    100000|     100|0.94825550|  PASSED
            sts_serial|  16|    100000|     100|0.87126691|  PASSED
            sts_serial|  16|    100000|     100|0.04158881|  PASSED
           rgb_bitdist|   1|    100000|     100|0.41495666|  PASSED
           rgb_bitdist|   2|    100000|     100|0.77588442|  PASSED
           rgb_bitdist|   3|    100000|     100|0.59680370|  PASSED
           rgb_bitdist|   4|    100000|     100|0.48931374|  PASSED
           rgb_bitdist|   5|    100000|     100|0.74304510|  PASSED
           rgb_bitdist|   6|    100000|     100|0.40334205|  PASSED
           rgb_bitdist|   7|    100000|     100|0.31159838|  PASSED
           rgb_bitdist|   8|    100000|     100|0.91718654|  PASSED
           rgb_bitdist|   9|    100000|     100|0.41960531|  PASSED
           rgb_bitdist|  10|    100000|     100|0.98819909|  PASSED
           rgb_bitdist|  11|    100000|     100|0.18620620|  PASSED
           rgb_bitdist|  12|    100000|     100|0.70588564|  PASSED
rgb_minimum_distance|   2|     10000|    1000|0.29819604|  PASSED
rgb_minimum_distance|   3|     10000|    1000|0.46820871|  PASSED
rgb_minimum_distance|   4|     10000|    1000|0.52169834|  PASSED
rgb_minimum_distance|   5|     10000|    1000|0.12604026|  PASSED
    rgb_permutations|   2|    100000|     100|0.71271095|  PASSED
    rgb_permutations|   3|    100000|     100|0.98140932|  PASSED
    rgb_permutations|   4|    100000|     100|0.98315878|  PASSED
    rgb_permutations|   5|    100000|     100|0.80663620|  PASSED
      rgb_lagged_sum|   0|   1000000|     100|0.31192386|  PASSED
      rgb_lagged_sum|   1|   1000000|     100|0.81822397|  PASSED
      rgb_lagged_sum|   2|   1000000|     100|0.95823413|  PASSED
      rgb_lagged_sum|   3|   1000000|     100|0.28590126|  PASSED
      rgb_lagged_sum|   4|   1000000|     100|0.29681063|  PASSED
      rgb_lagged_sum|   5|   1000000|     100|0.10954522|  PASSED
      rgb_lagged_sum|   6|   1000000|     100|0.80614555|  PASSED
      rgb_lagged_sum|   7|   1000000|     100|0.46055936|  PASSED
      rgb_lagged_sum|   8|   1000000|     100|0.15307280|  PASSED
      rgb_lagged_sum|   9|   1000000|     100|0.97246173|  PASSED
      rgb_lagged_sum|  10|   1000000|     100|0.27485010|  PASSED
      rgb_lagged_sum|  11|   1000000|     100|0.14822553|  PASSED
      rgb_lagged_sum|  12|   1000000|     100|0.51116470|  PASSED
      rgb_lagged_sum|  13|   1000000|     100|0.89897520|  PASSED
      rgb_lagged_sum|  14|   1000000|     100|0.29761389|  PASSED
      rgb_lagged_sum|  15|   1000000|     100|0.78175464|  PASSED
      rgb_lagged_sum|  16|   1000000|     100|0.45493032|  PASSED
      rgb_lagged_sum|  17|   1000000|     100|0.05338751|  PASSED
      rgb_lagged_sum|  18|   1000000|     100|0.95279489|  PASSED
      rgb_lagged_sum|  19|   1000000|     100|0.45709467|  PASSED
      rgb_lagged_sum|  20|   1000000|     100|0.11352361|  PASSED
      rgb_lagged_sum|  21|   1000000|     100|0.76179405|  PASSED
      rgb_lagged_sum|  22|   1000000|     100|0.71430388|  PASSED
      rgb_lagged_sum|  23|   1000000|     100|0.19041535|  PASSED
      rgb_lagged_sum|  24|   1000000|     100|0.58562007|  PASSED
      rgb_lagged_sum|  25|   1000000|     100|0.99132000|  PASSED
      rgb_lagged_sum|  26|   1000000|     100|0.77877111|  PASSED
      rgb_lagged_sum|  27|   1000000|     100|0.84290820|  PASSED
      rgb_lagged_sum|  28|   1000000|     100|0.47863164|  PASSED
      rgb_lagged_sum|  29|   1000000|     100|0.67339525|  PASSED
      rgb_lagged_sum|  30|   1000000|     100|0.59590769|  PASSED
      rgb_lagged_sum|  31|   1000000|     100|0.36997986|  PASSED
      rgb_lagged_sum|  32|   1000000|     100|0.90857623|  PASSED
     rgb_kstest_test|   0|     10000|    1000|0.68916694|  PASSED
     dab_bytedistrib|   0|  51200000|       1|0.06045585|  PASSED
             dab_dct| 256|     50000|       1|0.82066409|  PASSED
Skipping test 207
Preparing to run test 208.  ntuple = 0
```

```
       dab_filltree2|   0|   5000000|       1|0.73867204|  PASSED
       dab_filltree2|   1|   5000000|       1|0.61249913|  PASSED
Preparing to run test 209.  ntuple = 0
        dab_monobit2|  12|  65000000|       1|0.93655589|  PASSED
Preparing to run test 210.  ntuple = 0
#=============================================================================#
#   mean  |  stddev |  error-rate (best = 0.0, worst = 0.5)
#=============================================================================#
0.197724  |0.137387  |
```

Listing 8.9: Test results for random number engine `trng::mrg5s`.

```
#=============================================================================#
#          dieharder version 3.31.2beta Copyright 2003 Robert G. Brown        #
#=============================================================================#
   rng_name    |rands/second|   Seed   |  k ints/sec|k doubles/sec|
     trng_mrg5s|  1.39e+08  | 990537952|    138644  |     136113  |
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
   diehard_birthdays|   0|       100|     100|0.26330156|  PASSED
      diehard_operm5|   0|   1000000|     100|0.62158732|  PASSED
  diehard_rank_32x32|   0|     40000|     100|0.95706740|  PASSED
    diehard_rank_6x8|   0|    100000|     100|0.31774455|  PASSED
   diehard_bitstream|   0|   2097152|     100|0.54385938|  PASSED
        diehard_opso|   0|   2097152|     100|0.54982561|  PASSED
        diehard_oqso|   0|   2097152|     100|0.65737135|  PASSED
         diehard_dna|   0|   2097152|     100|0.00000000|  FAILED
diehard_count_1s_str|   0|    256000|     100|0.99424696|  PASSED
diehard_count_1s_byt|   0|    256000|     100|0.96645742|  PASSED
 diehard_parking_lot|   0|     12000|     100|0.06357627|  PASSED
    diehard_2dsphere|   2|      8000|     100|0.66029687|  PASSED
    diehard_3dsphere|   3|      4000|     100|0.63061937|  PASSED
     diehard_squeeze|   0|    100000|     100|0.51743151|  PASSED
        diehard_runs|   0|    100000|     100|0.05451619|  PASSED
        diehard_runs|   0|    100000|     100|0.13700089|  PASSED
       diehard_craps|   0|    200000|     100|0.25532791|  PASSED
       diehard_craps|   0|    200000|     100|0.02946607|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.91150871|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.48545967|  PASSED
         sts_monobit|   1|    100000|     100|0.71563583|  PASSED
            sts_runs|   2|    100000|     100|0.19357251|  PASSED
          sts_serial|   1|    100000|     100|0.51302341|  PASSED
          sts_serial|   2|    100000|     100|0.11535712|  PASSED
          sts_serial|   3|    100000|     100|0.20739440|  PASSED
          sts_serial|   3|    100000|     100|0.48877815|  PASSED
          sts_serial|   4|    100000|     100|0.42668636|  PASSED
          sts_serial|   4|    100000|     100|0.81721237|  PASSED
          sts_serial|   5|    100000|     100|0.65501674|  PASSED
          sts_serial|   5|    100000|     100|0.97716905|  PASSED
          sts_serial|   6|    100000|     100|0.19443938|  PASSED
          sts_serial|   6|    100000|     100|0.24991475|  PASSED
          sts_serial|   7|    100000|     100|0.33859471|  PASSED
          sts_serial|   7|    100000|     100|0.37987704|  PASSED
          sts_serial|   8|    100000|     100|0.49314740|  PASSED
          sts_serial|   8|    100000|     100|0.78104638|  PASSED
          sts_serial|   9|    100000|     100|0.96000738|  PASSED
          sts_serial|   9|    100000|     100|0.59712108|  PASSED
          sts_serial|  10|    100000|     100|0.97052899|  PASSED
          sts_serial|  10|    100000|     100|0.87028389|  PASSED
          sts_serial|  11|    100000|     100|0.40585932|  PASSED
          sts_serial|  11|    100000|     100|0.82902288|  PASSED
          sts_serial|  12|    100000|     100|0.53575799|  PASSED
          sts_serial|  12|    100000|     100|0.03381144|  PASSED
          sts_serial|  13|    100000|     100|0.83245972|  PASSED
          sts_serial|  13|    100000|     100|0.67292424|  PASSED
          sts_serial|  14|    100000|     100|0.73483393|  PASSED
          sts_serial|  14|    100000|     100|0.87232437|  PASSED
          sts_serial|  15|    100000|     100|0.68602810|  PASSED
          sts_serial|  15|    100000|     100|0.25593098|  PASSED
```

```
           sts_serial|  16|    100000|     100|0.71934523|  PASSED
           sts_serial|  16|    100000|     100|0.66009559|  PASSED
          rgb_bitdist|   1|    100000|     100|0.12281638|  PASSED
          rgb_bitdist|   2|    100000|     100|0.86198879|  PASSED
          rgb_bitdist|   3|    100000|     100|0.09965475|  PASSED
          rgb_bitdist|   4|    100000|     100|0.92565556|  PASSED
          rgb_bitdist|   5|    100000|     100|0.17828149|  PASSED
          rgb_bitdist|   6|    100000|     100|0.29280086|  PASSED
          rgb_bitdist|   7|    100000|     100|0.23383715|  PASSED
          rgb_bitdist|   8|    100000|     100|0.67897519|  PASSED
          rgb_bitdist|   9|    100000|     100|0.36522577|  PASSED
          rgb_bitdist|  10|    100000|     100|0.94219508|  PASSED
          rgb_bitdist|  11|    100000|     100|0.22431946|  PASSED
          rgb_bitdist|  12|    100000|     100|0.95928636|  PASSED
rgb_minimum_distance|   2|     10000|    1000|0.41699181|  PASSED
rgb_minimum_distance|   3|     10000|    1000|0.53315933|  PASSED
rgb_minimum_distance|   4|     10000|    1000|0.01860963|  PASSED
rgb_minimum_distance|   5|     10000|    1000|0.86072835|  PASSED
     rgb_permutations|   2|    100000|     100|0.95661259|  PASSED
     rgb_permutations|   3|    100000|     100|0.44540877|  PASSED
     rgb_permutations|   4|    100000|     100|0.93699892|  PASSED
     rgb_permutations|   5|    100000|     100|0.58590718|  PASSED
       rgb_lagged_sum|   0|   1000000|     100|0.42054523|  PASSED
       rgb_lagged_sum|   1|   1000000|     100|0.98122501|  PASSED
       rgb_lagged_sum|   2|   1000000|     100|0.92270019|  PASSED
       rgb_lagged_sum|   3|   1000000|     100|0.42525454|  PASSED
       rgb_lagged_sum|   4|   1000000|     100|0.80826975|  PASSED
       rgb_lagged_sum|   5|   1000000|     100|0.73168521|  PASSED
       rgb_lagged_sum|   6|   1000000|     100|0.93939924|  PASSED
       rgb_lagged_sum|   7|   1000000|     100|0.37229676|  PASSED
       rgb_lagged_sum|   8|   1000000|     100|0.72937457|  PASSED
       rgb_lagged_sum|   9|   1000000|     100|0.78617304|  PASSED
       rgb_lagged_sum|  10|   1000000|     100|0.71140645|  PASSED
       rgb_lagged_sum|  11|   1000000|     100|0.23129609|  PASSED
       rgb_lagged_sum|  12|   1000000|     100|0.76243801|  PASSED
       rgb_lagged_sum|  13|   1000000|     100|0.10056615|  PASSED
       rgb_lagged_sum|  14|   1000000|     100|0.23613549|  PASSED
       rgb_lagged_sum|  15|   1000000|     100|0.07815034|  PASSED
       rgb_lagged_sum|  16|   1000000|     100|0.80127778|  PASSED
       rgb_lagged_sum|  17|   1000000|     100|0.83693958|  PASSED
       rgb_lagged_sum|  18|   1000000|     100|0.96569603|  PASSED
       rgb_lagged_sum|  19|   1000000|     100|0.56888945|  PASSED
       rgb_lagged_sum|  20|   1000000|     100|0.63464066|  PASSED
       rgb_lagged_sum|  21|   1000000|     100|0.79135747|  PASSED
       rgb_lagged_sum|  22|   1000000|     100|0.07452702|  PASSED
       rgb_lagged_sum|  23|   1000000|     100|0.08480770|  PASSED
       rgb_lagged_sum|  24|   1000000|     100|0.85521052|  PASSED
       rgb_lagged_sum|  25|   1000000|     100|0.65701297|  PASSED
       rgb_lagged_sum|  26|   1000000|     100|0.01056421|  PASSED
       rgb_lagged_sum|  27|   1000000|     100|0.93172133|  PASSED
       rgb_lagged_sum|  28|   1000000|     100|0.39623934|  PASSED
       rgb_lagged_sum|  29|   1000000|     100|0.43507918|  PASSED
       rgb_lagged_sum|  30|   1000000|     100|0.08696490|  PASSED
       rgb_lagged_sum|  31|   1000000|     100|0.00985709|  PASSED
       rgb_lagged_sum|  32|   1000000|     100|0.43375366|  PASSED
      rgb_kstest_test|   0|     10000|    1000|0.92455891|  PASSED
      dab_bytedistrib|   0|  51200000|       1|0.58744212|  PASSED
              dab_dct| 256|     50000|       1|0.00534334|  PASSED
Skipping test 207
Preparing to run test 208.  ntuple = 0
        dab_filltree2|   0|   5000000|       1|0.14240843|  PASSED
        dab_filltree2|   1|   5000000|       1|0.14162723|  PASSED
Preparing to run test 209.  ntuple = 0
         dab_monobit2|  12|  65000000|       1|0.05351860|  PASSED
Preparing to run test 210.  ntuple = 0
#=============================================================================#
#   mean  |  stddev |  error-rate (best = 0.0, worst = 0.5)
#=============================================================================#
0.200003  |0.158749  |
```

Listing 8.10: Test results for random number engine `trng::yarn2`.

```
#=============================================================================#
#            dieharder version 3.31.2beta Copyright 2003 Robert G. Brown      #
#=============================================================================#
   rng_name    |rands/second|   Seed   |  k ints/sec|k doubles/sec|
     trng_yarn2|  1.63e+08  | 582740618|    163196  |     148084  |
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
   diehard_birthdays|   0|       100|     100|0.99450776|  PASSED
      diehard_operm5|   0|   1000000|     100|0.97519193|  PASSED
  diehard_rank_32x32|   0|     40000|     100|0.21602687|  PASSED
    diehard_rank_6x8|   0|    100000|     100|0.01660551|  PASSED
   diehard_bitstream|   0|   2097152|     100|0.65204510|  PASSED
        diehard_opso|   0|   2097152|     100|0.87020552|  PASSED
        diehard_oqso|   0|   2097152|     100|0.32930700|  PASSED
         diehard_dna|   0|   2097152|     100|0.00000000|  FAILED
diehard_count_1s_str|   0|    256000|     100|0.29229995|  PASSED
diehard_count_1s_byt|   0|    256000|     100|0.54291035|  PASSED
 diehard_parking_lot|   0|     12000|     100|0.83939978|  PASSED
    diehard_2dsphere|   2|      8000|     100|0.85089272|  PASSED
    diehard_3dsphere|   3|      4000|     100|0.67676595|  PASSED
     diehard_squeeze|   0|    100000|     100|0.97656928|  PASSED
        diehard_runs|   0|    100000|     100|0.42622137|  PASSED
        diehard_runs|   0|    100000|     100|0.67542119|  PASSED
       diehard_craps|   0|    200000|     100|0.38075210|  PASSED
       diehard_craps|   0|    200000|     100|0.69062819|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.83100825|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.76564625|  PASSED
         sts_monobit|   1|    100000|     100|0.08458055|  PASSED
            sts_runs|   2|    100000|     100|0.97874715|  PASSED
          sts_serial|   1|    100000|     100|0.96126586|  PASSED
          sts_serial|   2|    100000|     100|0.95367981|  PASSED
          sts_serial|   3|    100000|     100|0.39375738|  PASSED
          sts_serial|   3|    100000|     100|0.96913308|  PASSED
          sts_serial|   4|    100000|     100|0.89721642|  PASSED
          sts_serial|   4|    100000|     100|0.47601690|  PASSED
          sts_serial|   5|    100000|     100|0.33410553|  PASSED
          sts_serial|   5|    100000|     100|0.48906214|  PASSED
          sts_serial|   6|    100000|     100|0.77286630|  PASSED
          sts_serial|   6|    100000|     100|0.04221389|  PASSED
          sts_serial|   7|    100000|     100|0.91312114|  PASSED
          sts_serial|   7|    100000|     100|0.44643403|  PASSED
          sts_serial|   8|    100000|     100|0.72753916|  PASSED
          sts_serial|   8|    100000|     100|0.76701933|  PASSED
          sts_serial|   9|    100000|     100|0.93974728|  PASSED
          sts_serial|   9|    100000|     100|0.75853264|  PASSED
          sts_serial|  10|    100000|     100|0.76933588|  PASSED
          sts_serial|  10|    100000|     100|0.77215886|  PASSED
          sts_serial|  11|    100000|     100|0.89502328|  PASSED
          sts_serial|  11|    100000|     100|0.37451644|  PASSED
          sts_serial|  12|    100000|     100|0.98011960|  PASSED
          sts_serial|  12|    100000|     100|0.92558015|  PASSED
          sts_serial|  13|    100000|     100|0.83636019|  PASSED
          sts_serial|  13|    100000|     100|0.46272346|  PASSED
          sts_serial|  14|    100000|     100|0.96408206|  PASSED
          sts_serial|  14|    100000|     100|0.98627966|  PASSED
          sts_serial|  15|    100000|     100|0.96271416|  PASSED
          sts_serial|  15|    100000|     100|0.98752593|  PASSED
          sts_serial|  16|    100000|     100|0.16173884|  PASSED
          sts_serial|  16|    100000|     100|0.40651137|  PASSED
         rgb_bitdist|   1|    100000|     100|0.82795907|  PASSED
         rgb_bitdist|   2|    100000|     100|0.99107603|  PASSED
         rgb_bitdist|   3|    100000|     100|0.96936507|  PASSED
         rgb_bitdist|   4|    100000|     100|0.68911062|  PASSED
         rgb_bitdist|   5|    100000|     100|0.81964314|  PASSED
         rgb_bitdist|   6|    100000|     100|0.87604120|  PASSED
         rgb_bitdist|   7|    100000|     100|0.65787025|  PASSED
         rgb_bitdist|   8|    100000|     100|0.98872657|  PASSED
```

```
          rgb_bitdist|   9|    100000|     100|0.53843892|  PASSED
          rgb_bitdist|  10|    100000|     100|0.58350840|  PASSED
          rgb_bitdist|  11|    100000|     100|0.87861150|  PASSED
          rgb_bitdist|  12|    100000|     100|0.25693011|  PASSED
rgb_minimum_distance|   2|     10000|    1000|0.82001234|  PASSED
rgb_minimum_distance|   3|     10000|    1000|0.25121816|  PASSED
rgb_minimum_distance|   4|     10000|    1000|0.21420573|  PASSED
rgb_minimum_distance|   5|     10000|    1000|0.68615323|  PASSED
    rgb_permutations|   2|    100000|     100|0.17118068|  PASSED
    rgb_permutations|   3|    100000|     100|0.80016388|  PASSED
    rgb_permutations|   4|    100000|     100|0.69819781|  PASSED
    rgb_permutations|   5|    100000|     100|0.88722101|  PASSED
      rgb_lagged_sum|   0|   1000000|     100|0.60758562|  PASSED
      rgb_lagged_sum|   1|   1000000|     100|0.76146501|  PASSED
      rgb_lagged_sum|   2|   1000000|     100|0.68078034|  PASSED
      rgb_lagged_sum|   3|   1000000|     100|0.11814457|  PASSED
      rgb_lagged_sum|   4|   1000000|     100|0.68845760|  PASSED
      rgb_lagged_sum|   5|   1000000|     100|0.93161405|  PASSED
      rgb_lagged_sum|   6|   1000000|     100|0.63063411|  PASSED
      rgb_lagged_sum|   7|   1000000|     100|0.74128294|  PASSED
      rgb_lagged_sum|   8|   1000000|     100|0.78530311|  PASSED
      rgb_lagged_sum|   9|   1000000|     100|0.88910966|  PASSED
      rgb_lagged_sum|  10|   1000000|     100|0.24094190|  PASSED
      rgb_lagged_sum|  11|   1000000|     100|0.29751071|  PASSED
      rgb_lagged_sum|  12|   1000000|     100|0.71528930|  PASSED
      rgb_lagged_sum|  13|   1000000|     100|0.22285628|  PASSED
      rgb_lagged_sum|  14|   1000000|     100|0.22555601|  PASSED
      rgb_lagged_sum|  15|   1000000|     100|0.76008025|  PASSED
      rgb_lagged_sum|  16|   1000000|     100|0.72862859|  PASSED
      rgb_lagged_sum|  17|   1000000|     100|0.76724369|  PASSED
      rgb_lagged_sum|  18|   1000000|     100|0.22362228|  PASSED
      rgb_lagged_sum|  19|   1000000|     100|0.80324996|  PASSED
      rgb_lagged_sum|  20|   1000000|     100|0.83568850|  PASSED
      rgb_lagged_sum|  21|   1000000|     100|0.75702714|  PASSED
      rgb_lagged_sum|  22|   1000000|     100|0.99711067|   WEAK
      rgb_lagged_sum|  23|   1000000|     100|0.26290660|  PASSED
      rgb_lagged_sum|  24|   1000000|     100|0.88094551|  PASSED
      rgb_lagged_sum|  25|   1000000|     100|0.45771207|  PASSED
      rgb_lagged_sum|  26|   1000000|     100|0.55273113|  PASSED
      rgb_lagged_sum|  27|   1000000|     100|0.47041399|  PASSED
      rgb_lagged_sum|  28|   1000000|     100|0.94308347|  PASSED
      rgb_lagged_sum|  29|   1000000|     100|0.31256352|  PASSED
      rgb_lagged_sum|  30|   1000000|     100|0.51463719|  PASSED
      rgb_lagged_sum|  31|   1000000|     100|0.83937078|  PASSED
      rgb_lagged_sum|  32|   1000000|     100|0.22083941|  PASSED
     rgb_kstest_test|   0|     10000|    1000|0.58008482|  PASSED
     dab_bytedistrib|   0|  51200000|       1|0.55462589|  PASSED
             dab_dct| 256|     50000|       1|0.97539309|  PASSED
Skipping test 207
Preparing to run test 208.  ntuple = 0
        dab_filltree2|   0|   5000000|       1|0.42559215|  PASSED
        dab_filltree2|   1|   5000000|       1|0.11061420|  PASSED
Preparing to run test 209.  ntuple = 0
         dab_monobit2|  12|  65000000|       1|0.63911479|  PASSED
Preparing to run test 210.  ntuple = 0
#=============================================================================#
#   mean   |  stddev |  error-rate (best = 0.0, worst = 0.5)
#=============================================================================#
0.194738  |0.145951  |
```

Listing 8.11: Test results for random number engine `trng::yarn3`.

```
#=============================================================================#
#          dieharder version 3.31.2beta Copyright 2003 Robert G. Brown        #
#=============================================================================#
   rng_name    |rands/second|   Seed   |  k ints/sec|k doubles/sec|
     trng_yarn3|  1.03e+08  |3875421949|    103112  |      98056  |
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
```

```
    diehard_birthdays|   0|       100|     100|0.22384685|  PASSED
       diehard_operm5|   0|   1000000|     100|0.90338662|  PASSED
   diehard_rank_32x32|   0|     40000|     100|0.67202683|  PASSED
     diehard_rank_6x8|   0|    100000|     100|0.98664699|  PASSED
    diehard_bitstream|   0|   2097152|     100|0.08259643|  PASSED
         diehard_opso|   0|   2097152|     100|0.03190653|  PASSED
         diehard_oqso|   0|   2097152|     100|0.61768703|  PASSED
          diehard_dna|   0|   2097152|     100|0.00000000|  FAILED
diehard_count_1s_str|   0|    256000|     100|0.48507690|  PASSED
diehard_count_1s_byt|   0|    256000|     100|0.06392099|  PASSED
 diehard_parking_lot|   0|     12000|     100|0.95668914|  PASSED
    diehard_2dsphere|   2|      8000|     100|0.94682190|  PASSED
    diehard_3dsphere|   3|      4000|     100|0.80237391|  PASSED
     diehard_squeeze|   0|    100000|     100|0.77404412|  PASSED
        diehard_runs|   0|    100000|     100|0.67576899|  PASSED
        diehard_runs|   0|    100000|     100|0.34657816|  PASSED
       diehard_craps|   0|    200000|     100|0.73363658|  PASSED
       diehard_craps|   0|    200000|     100|0.82177930|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.42938702|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.69448881|  PASSED
         sts_monobit|   1|    100000|     100|0.29246611|  PASSED
            sts_runs|   2|    100000|     100|0.44620715|  PASSED
          sts_serial|   1|    100000|     100|0.71233872|  PASSED
          sts_serial|   2|    100000|     100|0.60277590|  PASSED
          sts_serial|   3|    100000|     100|0.24544705|  PASSED
          sts_serial|   3|    100000|     100|0.03944418|  PASSED
          sts_serial|   4|    100000|     100|0.25638728|  PASSED
          sts_serial|   4|    100000|     100|0.87910015|  PASSED
          sts_serial|   5|    100000|     100|0.89364332|  PASSED
          sts_serial|   5|    100000|     100|0.66750649|  PASSED
          sts_serial|   6|    100000|     100|0.95063771|  PASSED
          sts_serial|   6|    100000|     100|0.89232046|  PASSED
          sts_serial|   7|    100000|     100|0.17805852|  PASSED
          sts_serial|   7|    100000|     100|0.05758383|  PASSED
          sts_serial|   8|    100000|     100|0.98606728|  PASSED
          sts_serial|   8|    100000|     100|0.54274548|  PASSED
          sts_serial|   9|    100000|     100|0.92018142|  PASSED
          sts_serial|   9|    100000|     100|0.41529645|  PASSED
          sts_serial|  10|    100000|     100|0.82123207|  PASSED
          sts_serial|  10|    100000|     100|0.36396874|  PASSED
          sts_serial|  11|    100000|     100|0.32461290|  PASSED
          sts_serial|  11|    100000|     100|0.50475642|  PASSED
          sts_serial|  12|    100000|     100|0.19573767|  PASSED
          sts_serial|  12|    100000|     100|0.76961347|  PASSED
          sts_serial|  13|    100000|     100|0.44545165|  PASSED
          sts_serial|  13|    100000|     100|0.87602364|  PASSED
          sts_serial|  14|    100000|     100|0.42784909|  PASSED
          sts_serial|  14|    100000|     100|0.46626647|  PASSED
          sts_serial|  15|    100000|     100|0.93404208|  PASSED
          sts_serial|  15|    100000|     100|0.88747522|  PASSED
          sts_serial|  16|    100000|     100|0.33419525|  PASSED
          sts_serial|  16|    100000|     100|0.93029579|  PASSED
         rgb_bitdist|   1|    100000|     100|0.33799306|  PASSED
         rgb_bitdist|   2|    100000|     100|0.20639654|  PASSED
         rgb_bitdist|   3|    100000|     100|0.71877338|  PASSED
         rgb_bitdist|   4|    100000|     100|0.95390953|  PASSED
         rgb_bitdist|   5|    100000|     100|0.02770524|  PASSED
         rgb_bitdist|   6|    100000|     100|0.59121952|  PASSED
         rgb_bitdist|   7|    100000|     100|0.93179391|  PASSED
         rgb_bitdist|   8|    100000|     100|0.20987615|  PASSED
         rgb_bitdist|   9|    100000|     100|0.70480073|  PASSED
         rgb_bitdist|  10|    100000|     100|0.99733460|   WEAK
         rgb_bitdist|  11|    100000|     100|0.97822903|  PASSED
         rgb_bitdist|  12|    100000|     100|0.75042909|  PASSED
rgb_minimum_distance|   2|     10000|    1000|0.82168287|  PASSED
rgb_minimum_distance|   3|     10000|    1000|0.42195596|  PASSED
rgb_minimum_distance|   4|     10000|    1000|0.24626825|  PASSED
rgb_minimum_distance|   5|     10000|    1000|0.32736395|  PASSED
    rgb_permutations|   2|    100000|     100|0.22728399|  PASSED
    rgb_permutations|   3|    100000|     100|0.85137878|  PASSED
```

```
    rgb_permutations|   4|    100000|     100|0.74599660|  PASSED
    rgb_permutations|   5|    100000|     100|0.27205623|  PASSED
      rgb_lagged_sum|   0|   1000000|     100|0.33213586|  PASSED
      rgb_lagged_sum|   1|   1000000|     100|0.55904643|  PASSED
      rgb_lagged_sum|   2|   1000000|     100|0.98162953|  PASSED
      rgb_lagged_sum|   3|   1000000|     100|0.96874278|  PASSED
      rgb_lagged_sum|   4|   1000000|     100|0.47043121|  PASSED
      rgb_lagged_sum|   5|   1000000|     100|0.98581814|  PASSED
      rgb_lagged_sum|   6|   1000000|     100|0.65272746|  PASSED
      rgb_lagged_sum|   7|   1000000|     100|0.84539181|  PASSED
      rgb_lagged_sum|   8|   1000000|     100|0.89336450|  PASSED
      rgb_lagged_sum|   9|   1000000|     100|0.77526049|  PASSED
      rgb_lagged_sum|  10|   1000000|     100|0.62751640|  PASSED
      rgb_lagged_sum|  11|   1000000|     100|0.93513480|  PASSED
      rgb_lagged_sum|  12|   1000000|     100|0.26424058|  PASSED
      rgb_lagged_sum|  13|   1000000|     100|0.99012476|  PASSED
      rgb_lagged_sum|  14|   1000000|     100|0.60188066|  PASSED
      rgb_lagged_sum|  15|   1000000|     100|0.85567448|  PASSED
      rgb_lagged_sum|  16|   1000000|     100|0.03674328|  PASSED
      rgb_lagged_sum|  17|   1000000|     100|0.17654411|  PASSED
      rgb_lagged_sum|  18|   1000000|     100|0.37392773|  PASSED
      rgb_lagged_sum|  19|   1000000|     100|0.88307285|  PASSED
      rgb_lagged_sum|  20|   1000000|     100|0.69512943|  PASSED
      rgb_lagged_sum|  21|   1000000|     100|0.36518232|  PASSED
      rgb_lagged_sum|  22|   1000000|     100|0.37373840|  PASSED
      rgb_lagged_sum|  23|   1000000|     100|0.73981101|  PASSED
      rgb_lagged_sum|  24|   1000000|     100|0.88471815|  PASSED
      rgb_lagged_sum|  25|   1000000|     100|0.20599786|  PASSED
      rgb_lagged_sum|  26|   1000000|     100|0.48933302|  PASSED
      rgb_lagged_sum|  27|   1000000|     100|0.91682571|  PASSED
      rgb_lagged_sum|  28|   1000000|     100|0.35281769|  PASSED
      rgb_lagged_sum|  29|   1000000|     100|0.99364068|  PASSED
      rgb_lagged_sum|  30|   1000000|     100|0.66177694|  PASSED
      rgb_lagged_sum|  31|   1000000|     100|0.27833544|  PASSED
      rgb_lagged_sum|  32|   1000000|     100|0.45135453|  PASSED
     rgb_kstest_test|   0|     10000|    1000|0.09743583|  PASSED
     dab_bytedistrib|   0|  51200000|       1|0.81837241|  PASSED
             dab_dct| 256|     50000|       1|0.15014667|  PASSED
Skipping test 207
Preparing to run test 208.  ntuple = 0
       dab_filltree2|   0|   5000000|       1|0.05926335|  PASSED
       dab_filltree2|   1|   5000000|       1|0.56241564|  PASSED
Preparing to run test 209.  ntuple = 0
        dab_monobit2|  12|  65000000|       1|0.31359673|  PASSED
Preparing to run test 210.  ntuple = 0
#=============================================================================#
#   mean  |  stddev |  error-rate (best = 0.0, worst = 0.5)
#=============================================================================#
0.190446  |0.134897  |
```

Listing 8.12: Test results for random number engine `trng::yarn3s`.

```
#=============================================================================#
#          dieharder version 3.31.2beta Copyright 2003 Robert G. Brown        #
#=============================================================================#
   rng_name    |rands/second|   Seed   |  k ints/sec|k doubles/sec|
    trng_yarn3s|  1.13e+08  |1841431493|    112547  |     102773  |
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
   diehard_birthdays|   0|       100|     100|0.01000086|  PASSED
      diehard_operm5|   0|   1000000|     100|0.58884666|  PASSED
  diehard_rank_32x32|   0|     40000|     100|0.99456184|  PASSED
    diehard_rank_6x8|   0|    100000|     100|0.96194846|  PASSED
   diehard_bitstream|   0|   2097152|     100|0.82834158|  PASSED
        diehard_opso|   0|   2097152|     100|0.49826579|  PASSED
        diehard_oqso|   0|   2097152|     100|0.86291787|  PASSED
         diehard_dna|   0|   2097152|     100|0.00000000|  FAILED
diehard_count_1s_str|   0|    256000|     100|0.35922661|  PASSED
diehard_count_1s_byt|   0|    256000|     100|0.98613479|  PASSED
```

```
  diehard_parking_lot|   0|     12000|       100|0.96238980|  PASSED
     diehard_2dsphere|   2|      8000|       100|0.69385552|  PASSED
     diehard_3dsphere|   3|      4000|       100|0.02755190|  PASSED
      diehard_squeeze|   0|    100000|       100|0.92449600|  PASSED
         diehard_runs|   0|    100000|       100|0.13445493|  PASSED
         diehard_runs|   0|    100000|       100|0.43729950|  PASSED
        diehard_craps|   0|    200000|       100|0.25993192|  PASSED
        diehard_craps|   0|    200000|       100|0.06662435|  PASSED
  marsaglia_tsang_gcd|   0|  10000000|       100|0.13091439|  PASSED
  marsaglia_tsang_gcd|   0|  10000000|       100|0.38980567|  PASSED
          sts_monobit|   1|    100000|       100|0.05678718|  PASSED
             sts_runs|   2|    100000|       100|0.96823702|  PASSED
           sts_serial|   1|    100000|       100|0.25785975|  PASSED
           sts_serial|   2|    100000|       100|0.76926837|  PASSED
           sts_serial|   3|    100000|       100|0.93562264|  PASSED
           sts_serial|   3|    100000|       100|0.70446810|  PASSED
           sts_serial|   4|    100000|       100|0.98091806|  PASSED
           sts_serial|   4|    100000|       100|0.85882203|  PASSED
           sts_serial|   5|    100000|       100|0.74362000|  PASSED
           sts_serial|   5|    100000|       100|0.27783869|  PASSED
           sts_serial|   6|    100000|       100|0.94723480|  PASSED
           sts_serial|   6|    100000|       100|0.79535255|  PASSED
           sts_serial|   7|    100000|       100|0.35852489|  PASSED
           sts_serial|   7|    100000|       100|0.33243432|  PASSED
           sts_serial|   8|    100000|       100|0.10137264|  PASSED
           sts_serial|   8|    100000|       100|0.25791323|  PASSED
           sts_serial|   9|    100000|       100|0.87613358|  PASSED
           sts_serial|   9|    100000|       100|0.91995664|  PASSED
           sts_serial|  10|    100000|       100|0.13107940|  PASSED
           sts_serial|  10|    100000|       100|0.13334207|  PASSED
           sts_serial|  11|    100000|       100|0.04856285|  PASSED
           sts_serial|  11|    100000|       100|0.65902003|  PASSED
           sts_serial|  12|    100000|       100|0.22036443|  PASSED
           sts_serial|  12|    100000|       100|0.84326675|  PASSED
           sts_serial|  13|    100000|       100|0.07924162|  PASSED
           sts_serial|  13|    100000|       100|0.07798293|  PASSED
           sts_serial|  14|    100000|       100|0.40065610|  PASSED
           sts_serial|  14|    100000|       100|0.84891900|  PASSED
           sts_serial|  15|    100000|       100|0.31915426|  PASSED
           sts_serial|  15|    100000|       100|0.85867835|  PASSED
           sts_serial|  16|    100000|       100|0.19309679|  PASSED
           sts_serial|  16|    100000|       100|0.35896857|  PASSED
          rgb_bitdist|   1|    100000|       100|0.84757482|  PASSED
          rgb_bitdist|   2|    100000|       100|0.79444034|  PASSED
          rgb_bitdist|   3|    100000|       100|0.19292066|  PASSED
          rgb_bitdist|   4|    100000|       100|0.75674834|  PASSED
          rgb_bitdist|   5|    100000|       100|0.94338213|  PASSED
          rgb_bitdist|   6|    100000|       100|0.67047344|  PASSED
          rgb_bitdist|   7|    100000|       100|0.71364769|  PASSED
          rgb_bitdist|   8|    100000|       100|0.56115877|  PASSED
          rgb_bitdist|   9|    100000|       100|0.32013936|  PASSED
          rgb_bitdist|  10|    100000|       100|0.18043171|  PASSED
          rgb_bitdist|  11|    100000|       100|0.70758434|  PASSED
          rgb_bitdist|  12|    100000|       100|0.01037276|  PASSED
rgb_minimum_distance|   2|     10000|      1000|0.74324484|  PASSED
rgb_minimum_distance|   3|     10000|      1000|0.77959924|  PASSED
rgb_minimum_distance|   4|     10000|      1000|0.56691962|  PASSED
rgb_minimum_distance|   5|     10000|      1000|0.76840943|  PASSED
     rgb_permutations|   2|    100000|       100|0.05325431|  PASSED
     rgb_permutations|   3|    100000|       100|0.51628725|  PASSED
     rgb_permutations|   4|    100000|       100|0.62527604|  PASSED
     rgb_permutations|   5|    100000|       100|0.93191884|  PASSED
       rgb_lagged_sum|   0|   1000000|       100|0.97038950|  PASSED
       rgb_lagged_sum|   1|   1000000|       100|0.26015066|  PASSED
       rgb_lagged_sum|   2|   1000000|       100|0.51427685|  PASSED
       rgb_lagged_sum|   3|   1000000|       100|0.60554564|  PASSED
       rgb_lagged_sum|   4|   1000000|       100|0.65839700|  PASSED
       rgb_lagged_sum|   5|   1000000|       100|0.72346294|  PASSED
       rgb_lagged_sum|   6|   1000000|       100|0.81576133|  PASSED
       rgb_lagged_sum|   7|   1000000|       100|0.39591424|  PASSED
```

```
      rgb_lagged_sum|   8|   1000000|     100|0.17314349|  PASSED
      rgb_lagged_sum|   9|   1000000|     100|0.17810746|  PASSED
      rgb_lagged_sum|  10|   1000000|     100|0.94180985|  PASSED
      rgb_lagged_sum|  11|   1000000|     100|0.24520748|  PASSED
      rgb_lagged_sum|  12|   1000000|     100|0.51424586|  PASSED
      rgb_lagged_sum|  13|   1000000|     100|0.92079173|  PASSED
      rgb_lagged_sum|  14|   1000000|     100|0.48450085|  PASSED
      rgb_lagged_sum|  15|   1000000|     100|0.68855483|  PASSED
      rgb_lagged_sum|  16|   1000000|     100|0.47257614|  PASSED
      rgb_lagged_sum|  17|   1000000|     100|0.36108100|  PASSED
      rgb_lagged_sum|  18|   1000000|     100|0.64758110|  PASSED
      rgb_lagged_sum|  19|   1000000|     100|0.99713244|   WEAK
      rgb_lagged_sum|  20|   1000000|     100|0.53402517|  PASSED
      rgb_lagged_sum|  21|   1000000|     100|0.20841477|  PASSED
      rgb_lagged_sum|  22|   1000000|     100|0.56278766|  PASSED
      rgb_lagged_sum|  23|   1000000|     100|0.48987903|  PASSED
      rgb_lagged_sum|  24|   1000000|     100|0.95134239|  PASSED
      rgb_lagged_sum|  25|   1000000|     100|0.19230976|  PASSED
      rgb_lagged_sum|  26|   1000000|     100|0.11030790|  PASSED
      rgb_lagged_sum|  27|   1000000|     100|0.99942244|   WEAK
      rgb_lagged_sum|  28|   1000000|     100|0.71898396|  PASSED
      rgb_lagged_sum|  29|   1000000|     100|0.27898719|  PASSED
      rgb_lagged_sum|  30|   1000000|     100|0.69658273|  PASSED
      rgb_lagged_sum|  31|   1000000|     100|0.33703832|  PASSED
      rgb_lagged_sum|  32|   1000000|     100|0.91333157|  PASSED
     rgb_kstest_test|   0|     10000|    1000|0.33470138|  PASSED
     dab_bytedistrib|   0|  51200000|       1|0.12607218|  PASSED
             dab_dct| 256|     50000|       1|0.39238939|  PASSED
Skipping test 207
Preparing to run test 208.  ntuple = 0
        dab_filltree2|   0|   5000000|       1|0.73562685|  PASSED
        dab_filltree2|   1|   5000000|       1|0.74943974|  PASSED
Preparing to run test 209.  ntuple = 0
         dab_monobit2|  12|  65000000|       1|0.34833019|  PASSED
Preparing to run test 210.  ntuple = 0
#=============================================================================#
#   mean   |  stddev  |  error-rate (best = 0.0, worst = 0.5)
#=============================================================================#
0.172636  |0.147323  |
```

Listing 8.13: Test results for random number engine `trng::yarn4`.

```
#=============================================================================#
#          dieharder version 3.31.2beta Copyright 2003 Robert G. Brown        #
#=============================================================================#
   rng_name    |rands/second|   Seed   |  k ints/sec|k doubles/sec|
     trng_yarn4|  8.11e+07  |1568923093|     81116  |      81063  |
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
   diehard_birthdays|   0|       100|     100|0.03080055|  PASSED
      diehard_operm5|   0|   1000000|     100|0.33524636|  PASSED
  diehard_rank_32x32|   0|     40000|     100|0.61770011|  PASSED
    diehard_rank_6x8|   0|    100000|     100|0.82321598|  PASSED
   diehard_bitstream|   0|   2097152|     100|0.43460535|  PASSED
        diehard_opso|   0|   2097152|     100|0.90279598|  PASSED
        diehard_oqso|   0|   2097152|     100|0.76796698|  PASSED
         diehard_dna|   0|   2097152|     100|0.00000000|  FAILED
diehard_count_1s_str|   0|    256000|     100|0.96708638|  PASSED
diehard_count_1s_byt|   0|    256000|     100|0.40857672|  PASSED
 diehard_parking_lot|   0|     12000|     100|0.15982204|  PASSED
    diehard_2dsphere|   2|      8000|     100|0.17754492|  PASSED
    diehard_3dsphere|   3|      4000|     100|0.46328676|  PASSED
     diehard_squeeze|   0|    100000|     100|0.06490510|  PASSED
        diehard_runs|   0|    100000|     100|0.50444482|  PASSED
        diehard_runs|   0|    100000|     100|0.83486842|  PASSED
       diehard_craps|   0|    200000|     100|0.34926772|  PASSED
       diehard_craps|   0|    200000|     100|0.58092588|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.97613149|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.99337368|  PASSED
```

```
         sts_monobit|   1|    100000|     100|0.72182663|  PASSED
            sts_runs|   2|    100000|     100|0.33712517|  PASSED
          sts_serial|   1|    100000|     100|0.61868856|  PASSED
          sts_serial|   2|    100000|     100|0.06245739|  PASSED
          sts_serial|   3|    100000|     100|0.06046640|  PASSED
          sts_serial|   3|    100000|     100|0.00472140|   WEAK
          sts_serial|   4|    100000|     100|0.34065971|  PASSED
          sts_serial|   4|    100000|     100|0.45431020|  PASSED
          sts_serial|   5|    100000|     100|0.20339663|  PASSED
          sts_serial|   5|    100000|     100|0.78951575|  PASSED
          sts_serial|   6|    100000|     100|0.23622953|  PASSED
          sts_serial|   6|    100000|     100|0.96119696|  PASSED
          sts_serial|   7|    100000|     100|0.56328552|  PASSED
          sts_serial|   7|    100000|     100|0.52965870|  PASSED
          sts_serial|   8|    100000|     100|0.44371467|  PASSED
          sts_serial|   8|    100000|     100|0.56644625|  PASSED
          sts_serial|   9|    100000|     100|0.43512280|  PASSED
          sts_serial|   9|    100000|     100|0.09993698|  PASSED
          sts_serial|  10|    100000|     100|0.69461545|  PASSED
          sts_serial|  10|    100000|     100|0.80052060|  PASSED
          sts_serial|  11|    100000|     100|0.30436839|  PASSED
          sts_serial|  11|    100000|     100|0.78550752|  PASSED
          sts_serial|  12|    100000|     100|0.45142376|  PASSED
          sts_serial|  12|    100000|     100|0.03204033|  PASSED
          sts_serial|  13|    100000|     100|0.54425116|  PASSED
          sts_serial|  13|    100000|     100|0.61357014|  PASSED
          sts_serial|  14|    100000|     100|0.82862203|  PASSED
          sts_serial|  14|    100000|     100|0.63768875|  PASSED
          sts_serial|  15|    100000|     100|0.38610118|  PASSED
          sts_serial|  15|    100000|     100|0.09086976|  PASSED
          sts_serial|  16|    100000|     100|0.73488237|  PASSED
          sts_serial|  16|    100000|     100|0.79172819|  PASSED
         rgb_bitdist|   1|    100000|     100|0.14076543|  PASSED
         rgb_bitdist|   2|    100000|     100|0.81051946|  PASSED
         rgb_bitdist|   3|    100000|     100|0.53179522|  PASSED
         rgb_bitdist|   4|    100000|     100|0.69721795|  PASSED
         rgb_bitdist|   5|    100000|     100|0.10807605|  PASSED
         rgb_bitdist|   6|    100000|     100|0.11250718|  PASSED
         rgb_bitdist|   7|    100000|     100|0.95198973|  PASSED
         rgb_bitdist|   8|    100000|     100|0.19235854|  PASSED
         rgb_bitdist|   9|    100000|     100|0.88236294|  PASSED
         rgb_bitdist|  10|    100000|     100|0.83408341|  PASSED
         rgb_bitdist|  11|    100000|     100|0.91113554|  PASSED
         rgb_bitdist|  12|    100000|     100|0.12060204|  PASSED
rgb_minimum_distance|   2|     10000|    1000|0.95607604|  PASSED
rgb_minimum_distance|   3|     10000|    1000|0.81836945|  PASSED
rgb_minimum_distance|   4|     10000|    1000|0.76918933|  PASSED
rgb_minimum_distance|   5|     10000|    1000|0.41710206|  PASSED
    rgb_permutations|   2|    100000|     100|0.38311556|  PASSED
    rgb_permutations|   3|    100000|     100|0.38784439|  PASSED
    rgb_permutations|   4|    100000|     100|0.21420995|  PASSED
    rgb_permutations|   5|    100000|     100|0.76363017|  PASSED
      rgb_lagged_sum|   0|   1000000|     100|0.86626959|  PASSED
      rgb_lagged_sum|   1|   1000000|     100|0.26174856|  PASSED
      rgb_lagged_sum|   2|   1000000|     100|0.19025941|  PASSED
      rgb_lagged_sum|   3|   1000000|     100|0.89587648|  PASSED
      rgb_lagged_sum|   4|   1000000|     100|0.40838953|  PASSED
      rgb_lagged_sum|   5|   1000000|     100|0.78408129|  PASSED
      rgb_lagged_sum|   6|   1000000|     100|0.93796733|  PASSED
      rgb_lagged_sum|   7|   1000000|     100|0.97721719|  PASSED
      rgb_lagged_sum|   8|   1000000|     100|0.57280565|  PASSED
      rgb_lagged_sum|   9|   1000000|     100|0.47359615|  PASSED
      rgb_lagged_sum|  10|   1000000|     100|0.60352872|  PASSED
      rgb_lagged_sum|  11|   1000000|     100|0.75303114|  PASSED
      rgb_lagged_sum|  12|   1000000|     100|0.88653523|  PASSED
      rgb_lagged_sum|  13|   1000000|     100|0.57779011|  PASSED
      rgb_lagged_sum|  14|   1000000|     100|0.40738968|  PASSED
      rgb_lagged_sum|  15|   1000000|     100|0.80247793|  PASSED
      rgb_lagged_sum|  16|   1000000|     100|0.10057491|  PASSED
      rgb_lagged_sum|  17|   1000000|     100|0.05815288|  PASSED
```

```
      rgb_lagged_sum|  18|   1000000|     100|0.90762026|  PASSED
      rgb_lagged_sum|  19|   1000000|     100|0.88985943|  PASSED
      rgb_lagged_sum|  20|   1000000|     100|0.95625341|  PASSED
      rgb_lagged_sum|  21|   1000000|     100|0.67040901|  PASSED
      rgb_lagged_sum|  22|   1000000|     100|0.89888407|  PASSED
      rgb_lagged_sum|  23|   1000000|     100|0.62288717|  PASSED
      rgb_lagged_sum|  24|   1000000|     100|0.09273349|  PASSED
      rgb_lagged_sum|  25|   1000000|     100|0.51561263|  PASSED
      rgb_lagged_sum|  26|   1000000|     100|0.23041854|  PASSED
      rgb_lagged_sum|  27|   1000000|     100|0.22572093|  PASSED
      rgb_lagged_sum|  28|   1000000|     100|0.85534903|  PASSED
      rgb_lagged_sum|  29|   1000000|     100|0.22439221|  PASSED
      rgb_lagged_sum|  30|   1000000|     100|0.02640871|  PASSED
      rgb_lagged_sum|  31|   1000000|     100|0.62799418|  PASSED
      rgb_lagged_sum|  32|   1000000|     100|0.78349945|  PASSED
     rgb_kstest_test|   0|     10000|    1000|0.98152804|  PASSED
     dab_bytedistrib|   0|  51200000|       1|0.42409508|  PASSED
             dab_dct| 256|     50000|       1|0.09418888|  PASSED
Skipping test 207
Preparing to run test 208.  ntuple = 0
       dab_filltree2|   0|   5000000|       1|0.98232162|  PASSED
       dab_filltree2|   1|   5000000|       1|0.38604609|  PASSED
Preparing to run test 209.  ntuple = 0
        dab_monobit2|  12|  65000000|       1|0.85780377|  PASSED
Preparing to run test 210.  ntuple = 0
#=============================================================================#
#   mean  |  stddev |  error-rate (best = 0.0, worst = 0.5)
#=============================================================================#
0.198288  |0.148321  |
```

Listing 8.14: Test results for random number engine `trng::yarn5`.

```
#=============================================================================#
#          dieharder version 3.31.2beta Copyright 2003 Robert G. Brown        #
#=============================================================================#
   rng_name    |rands/second|   Seed   |  k ints/sec|k doubles/sec|
     trng_yarn5|  1.18e+08  |3973401754|    118093  |     108812  |
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
   diehard_birthdays|   0|       100|     100|0.96851619|  PASSED
      diehard_operm5|   0|   1000000|     100|0.77705928|  PASSED
  diehard_rank_32x32|   0|     40000|     100|0.71916815|  PASSED
    diehard_rank_6x8|   0|    100000|     100|0.19958362|  PASSED
   diehard_bitstream|   0|   2097152|     100|0.83244274|  PASSED
        diehard_opso|   0|   2097152|     100|0.93842380|  PASSED
        diehard_oqso|   0|   2097152|     100|0.88851584|  PASSED
         diehard_dna|   0|   2097152|     100|0.00000000|  FAILED
diehard_count_1s_str|   0|    256000|     100|0.96024947|  PASSED
diehard_count_1s_byt|   0|    256000|     100|0.19531214|  PASSED
 diehard_parking_lot|   0|     12000|     100|0.99225529|  PASSED
    diehard_2dsphere|   2|      8000|     100|0.53881606|  PASSED
    diehard_3dsphere|   3|      4000|     100|0.42874744|  PASSED
     diehard_squeeze|   0|    100000|     100|0.18151766|  PASSED
        diehard_runs|   0|    100000|     100|0.99971758|   WEAK
        diehard_runs|   0|    100000|     100|0.66360517|  PASSED
       diehard_craps|   0|    200000|     100|0.99982908|   WEAK
       diehard_craps|   0|    200000|     100|0.95838048|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.06708803|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.04600809|  PASSED
         sts_monobit|   1|    100000|     100|0.43485022|  PASSED
            sts_runs|   2|    100000|     100|0.99884377|   WEAK
          sts_serial|   1|    100000|     100|0.73299338|  PASSED
          sts_serial|   2|    100000|     100|0.62802508|  PASSED
          sts_serial|   3|    100000|     100|0.97461752|  PASSED
          sts_serial|   3|    100000|     100|0.96990197|  PASSED
          sts_serial|   4|    100000|     100|0.01820225|  PASSED
          sts_serial|   4|    100000|     100|0.02480012|  PASSED
          sts_serial|   5|    100000|     100|0.80024061|  PASSED
          sts_serial|   5|    100000|     100|0.78859310|  PASSED
```

```
           sts_serial|  6|    100000|      100|0.26838402|  PASSED
           sts_serial|  6|    100000|      100|0.19058730|  PASSED
           sts_serial|  7|    100000|      100|0.59999056|  PASSED
           sts_serial|  7|    100000|      100|0.86556749|  PASSED
           sts_serial|  8|    100000|      100|0.47440369|  PASSED
           sts_serial|  8|    100000|      100|0.33032595|  PASSED
           sts_serial|  9|    100000|      100|0.64130322|  PASSED
           sts_serial|  9|    100000|      100|0.96633098|  PASSED
           sts_serial| 10|    100000|      100|0.45042429|  PASSED
           sts_serial| 10|    100000|      100|0.38854850|  PASSED
           sts_serial| 11|    100000|      100|0.56509309|  PASSED
           sts_serial| 11|    100000|      100|0.94034773|  PASSED
           sts_serial| 12|    100000|      100|0.83225520|  PASSED
           sts_serial| 12|    100000|      100|0.46472799|  PASSED
           sts_serial| 13|    100000|      100|0.51276591|  PASSED
           sts_serial| 13|    100000|      100|0.58936298|  PASSED
           sts_serial| 14|    100000|      100|0.18652953|  PASSED
           sts_serial| 14|    100000|      100|0.12927023|  PASSED
           sts_serial| 15|    100000|      100|0.55511939|  PASSED
           sts_serial| 15|    100000|      100|0.63841685|  PASSED
           sts_serial| 16|    100000|      100|0.88727036|  PASSED
           sts_serial| 16|    100000|      100|0.77474238|  PASSED
          rgb_bitdist|  1|    100000|      100|0.46750769|  PASSED
          rgb_bitdist|  2|    100000|      100|0.55359203|  PASSED
          rgb_bitdist|  3|    100000|      100|0.41282470|  PASSED
          rgb_bitdist|  4|    100000|      100|0.84876230|  PASSED
          rgb_bitdist|  5|    100000|      100|0.70342837|  PASSED
          rgb_bitdist|  6|    100000|      100|0.71417121|  PASSED
          rgb_bitdist|  7|    100000|      100|0.07018578|  PASSED
          rgb_bitdist|  8|    100000|      100|0.53712412|  PASSED
          rgb_bitdist|  9|    100000|      100|0.74274346|  PASSED
          rgb_bitdist| 10|    100000|      100|0.38284038|  PASSED
          rgb_bitdist| 11|    100000|      100|0.76744659|  PASSED
          rgb_bitdist| 12|    100000|      100|0.74781396|  PASSED
rgb_minimum_distance|  2|     10000|     1000|0.62039866|  PASSED
rgb_minimum_distance|  3|     10000|     1000|0.38660417|  PASSED
rgb_minimum_distance|  4|     10000|     1000|0.07330626|  PASSED
rgb_minimum_distance|  5|     10000|     1000|0.60587847|  PASSED
    rgb_permutations|  2|    100000|      100|0.56273791|  PASSED
    rgb_permutations|  3|    100000|      100|0.72699513|  PASSED
    rgb_permutations|  4|    100000|      100|0.55480554|  PASSED
    rgb_permutations|  5|    100000|      100|0.58271215|  PASSED
      rgb_lagged_sum|  0|   1000000|      100|0.57999871|  PASSED
      rgb_lagged_sum|  1|   1000000|      100|0.09264998|  PASSED
      rgb_lagged_sum|  2|   1000000|      100|0.24250515|  PASSED
      rgb_lagged_sum|  3|   1000000|      100|0.96074953|  PASSED
      rgb_lagged_sum|  4|   1000000|      100|0.43701309|  PASSED
      rgb_lagged_sum|  5|   1000000|      100|0.39393693|  PASSED
      rgb_lagged_sum|  6|   1000000|      100|0.25615347|  PASSED
      rgb_lagged_sum|  7|   1000000|      100|0.94863615|  PASSED
      rgb_lagged_sum|  8|   1000000|      100|0.96239753|  PASSED
      rgb_lagged_sum|  9|   1000000|      100|0.64243180|  PASSED
      rgb_lagged_sum| 10|   1000000|      100|0.28560516|  PASSED
      rgb_lagged_sum| 11|   1000000|      100|0.81873489|  PASSED
      rgb_lagged_sum| 12|   1000000|      100|0.89009976|  PASSED
      rgb_lagged_sum| 13|   1000000|      100|0.34021040|  PASSED
      rgb_lagged_sum| 14|   1000000|      100|0.76045398|  PASSED
      rgb_lagged_sum| 15|   1000000|      100|0.85583898|  PASSED
      rgb_lagged_sum| 16|   1000000|      100|0.48330069|  PASSED
      rgb_lagged_sum| 17|   1000000|      100|0.76700600|  PASSED
      rgb_lagged_sum| 18|   1000000|      100|0.93862188|  PASSED
      rgb_lagged_sum| 19|   1000000|      100|0.79537782|  PASSED
      rgb_lagged_sum| 20|   1000000|      100|0.95075707|  PASSED
      rgb_lagged_sum| 21|   1000000|      100|0.05557928|  PASSED
      rgb_lagged_sum| 22|   1000000|      100|0.26578900|  PASSED
      rgb_lagged_sum| 23|   1000000|      100|0.94403934|  PASSED
      rgb_lagged_sum| 24|   1000000|      100|0.97418822|  PASSED
      rgb_lagged_sum| 25|   1000000|      100|0.99844417|   WEAK
      rgb_lagged_sum| 26|   1000000|      100|0.66322027|  PASSED
      rgb_lagged_sum| 27|   1000000|      100|0.28015421|  PASSED
```

```
      rgb_lagged_sum|  28|   1000000|     100|0.95952141|  PASSED
      rgb_lagged_sum|  29|   1000000|     100|0.04408584|  PASSED
      rgb_lagged_sum|  30|   1000000|     100|0.54742943|  PASSED
      rgb_lagged_sum|  31|   1000000|     100|0.08193720|  PASSED
      rgb_lagged_sum|  32|   1000000|     100|0.74026086|  PASSED
     rgb_kstest_test|   0|     10000|    1000|0.44417324|  PASSED
     dab_bytedistrib|   0|  51200000|       1|0.50885886|  PASSED
             dab_dct| 256|     50000|       1|0.25541225|  PASSED
Skipping test 207
Preparing to run test 208.  ntuple = 0
       dab_filltree2|   0|   5000000|       1|0.41779399|  PASSED
       dab_filltree2|   1|   5000000|       1|0.20581331|  PASSED
Preparing to run test 209.  ntuple = 0
        dab_monobit2|  12|  65000000|       1|0.91780644|  PASSED
Preparing to run test 210.  ntuple = 0
#=============================================================================#
#   mean   |  stddev |  error-rate (best = 0.0, worst = 0.5)
#=============================================================================#
0.192392  |0.162477  |
```

Listing 8.15: Test results for random number engine `trng::yarn5s`.

```
#=============================================================================#
#          dieharder version 3.31.2beta Copyright 2003 Robert G. Brown        #
#=============================================================================#
   rng_name    |rands/second|   Seed   |  k ints/sec|k doubles/sec|
    trng_yarn5s|  1.01e+08  |4288905992|    101472  |      90059  |
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
   diehard_birthdays|   0|       100|     100|0.61754502|  PASSED
      diehard_operm5|   0|   1000000|     100|0.17154093|  PASSED
  diehard_rank_32x32|   0|     40000|     100|0.31218105|  PASSED
    diehard_rank_6x8|   0|    100000|     100|0.80482932|  PASSED
   diehard_bitstream|   0|   2097152|     100|0.88915263|  PASSED
        diehard_opso|   0|   2097152|     100|0.36549830|  PASSED
        diehard_oqso|   0|   2097152|     100|0.80121979|  PASSED
         diehard_dna|   0|   2097152|     100|0.00000000|  FAILED
diehard_count_1s_str|   0|    256000|     100|0.68549577|  PASSED
diehard_count_1s_byt|   0|    256000|     100|0.94151513|  PASSED
 diehard_parking_lot|   0|     12000|     100|0.61281762|  PASSED
    diehard_2dsphere|   2|      8000|     100|0.80966184|  PASSED
    diehard_3dsphere|   3|      4000|     100|0.73421989|  PASSED
     diehard_squeeze|   0|    100000|     100|0.71906803|  PASSED
        diehard_runs|   0|    100000|     100|0.14967360|  PASSED
        diehard_runs|   0|    100000|     100|0.23219994|  PASSED
       diehard_craps|   0|    200000|     100|0.15775996|  PASSED
       diehard_craps|   0|    200000|     100|0.13817449|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.91311035|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.29359411|  PASSED
         sts_monobit|   1|    100000|     100|0.69371168|  PASSED
            sts_runs|   2|    100000|     100|0.95478719|  PASSED
          sts_serial|   1|    100000|     100|0.24285373|  PASSED
          sts_serial|   2|    100000|     100|0.96184108|  PASSED
          sts_serial|   3|    100000|     100|0.92832688|  PASSED
          sts_serial|   3|    100000|     100|0.72579638|  PASSED
          sts_serial|   4|    100000|     100|0.38704399|  PASSED
          sts_serial|   4|    100000|     100|0.90057166|  PASSED
          sts_serial|   5|    100000|     100|0.61415878|  PASSED
          sts_serial|   5|    100000|     100|0.32916179|  PASSED
          sts_serial|   6|    100000|     100|0.61848947|  PASSED
          sts_serial|   6|    100000|     100|0.98981912|  PASSED
          sts_serial|   7|    100000|     100|0.47800960|  PASSED
          sts_serial|   7|    100000|     100|0.71913993|  PASSED
          sts_serial|   8|    100000|     100|0.76128389|  PASSED
          sts_serial|   8|    100000|     100|0.84093510|  PASSED
          sts_serial|   9|    100000|     100|0.79834641|  PASSED
          sts_serial|   9|    100000|     100|0.09215156|  PASSED
          sts_serial|  10|    100000|     100|0.47353328|  PASSED
          sts_serial|  10|    100000|     100|0.71070010|  PASSED
```

```
          sts_serial|  11|    100000|     100|0.95857242|  PASSED
          sts_serial|  11|    100000|     100|0.94793695|  PASSED
          sts_serial|  12|    100000|     100|0.30466210|  PASSED
          sts_serial|  12|    100000|     100|0.39843097|  PASSED
          sts_serial|  13|    100000|     100|0.86870000|  PASSED
          sts_serial|  13|    100000|     100|0.27096365|  PASSED
          sts_serial|  14|    100000|     100|0.77389959|  PASSED
          sts_serial|  14|    100000|     100|0.08526275|  PASSED
          sts_serial|  15|    100000|     100|0.66461090|  PASSED
          sts_serial|  15|    100000|     100|0.04476795|  PASSED
          sts_serial|  16|    100000|     100|0.57832070|  PASSED
          sts_serial|  16|    100000|     100|0.94635214|  PASSED
         rgb_bitdist|   1|    100000|     100|0.90868721|  PASSED
         rgb_bitdist|   2|    100000|     100|0.92489838|  PASSED
         rgb_bitdist|   3|    100000|     100|0.22516803|  PASSED
         rgb_bitdist|   4|    100000|     100|0.20209551|  PASSED
         rgb_bitdist|   5|    100000|     100|0.39776851|  PASSED
         rgb_bitdist|   6|    100000|     100|0.77180584|  PASSED
         rgb_bitdist|   7|    100000|     100|0.87525852|  PASSED
         rgb_bitdist|   8|    100000|     100|0.17249709|  PASSED
         rgb_bitdist|   9|    100000|     100|0.73715001|  PASSED
         rgb_bitdist|  10|    100000|     100|0.95652701|  PASSED
         rgb_bitdist|  11|    100000|     100|0.87048478|  PASSED
         rgb_bitdist|  12|    100000|     100|0.81556263|  PASSED
rgb_minimum_distance|   2|     10000|    1000|0.08459170|  PASSED
rgb_minimum_distance|   3|     10000|    1000|0.39223961|  PASSED
rgb_minimum_distance|   4|     10000|    1000|0.46117881|  PASSED
rgb_minimum_distance|   5|     10000|    1000|0.96963853|  PASSED
    rgb_permutations|   2|    100000|     100|0.68748848|  PASSED
    rgb_permutations|   3|    100000|     100|0.85086731|  PASSED
    rgb_permutations|   4|    100000|     100|0.98968729|  PASSED
    rgb_permutations|   5|    100000|     100|0.57852726|  PASSED
      rgb_lagged_sum|   0|   1000000|     100|0.01605125|  PASSED
      rgb_lagged_sum|   1|   1000000|     100|0.15565474|  PASSED
      rgb_lagged_sum|   2|   1000000|     100|0.64665879|  PASSED
      rgb_lagged_sum|   3|   1000000|     100|0.67294497|  PASSED
      rgb_lagged_sum|   4|   1000000|     100|0.60174098|  PASSED
      rgb_lagged_sum|   5|   1000000|     100|0.05457774|  PASSED
      rgb_lagged_sum|   6|   1000000|     100|0.29040305|  PASSED
      rgb_lagged_sum|   7|   1000000|     100|0.64933858|  PASSED
      rgb_lagged_sum|   8|   1000000|     100|0.45177377|  PASSED
      rgb_lagged_sum|   9|   1000000|     100|0.24101724|  PASSED
      rgb_lagged_sum|  10|   1000000|     100|0.96583895|  PASSED
      rgb_lagged_sum|  11|   1000000|     100|0.87740122|  PASSED
      rgb_lagged_sum|  12|   1000000|     100|0.60253728|  PASSED
      rgb_lagged_sum|  13|   1000000|     100|0.64658215|  PASSED
      rgb_lagged_sum|  14|   1000000|     100|0.47403183|  PASSED
      rgb_lagged_sum|  15|   1000000|     100|0.00117020|   WEAK
      rgb_lagged_sum|  16|   1000000|     100|0.09559053|  PASSED
      rgb_lagged_sum|  17|   1000000|     100|0.36668962|  PASSED
      rgb_lagged_sum|  18|   1000000|     100|0.49909254|  PASSED
      rgb_lagged_sum|  19|   1000000|     100|0.46247570|  PASSED
      rgb_lagged_sum|  20|   1000000|     100|0.25655723|  PASSED
      rgb_lagged_sum|  21|   1000000|     100|0.41791979|  PASSED
      rgb_lagged_sum|  22|   1000000|     100|0.77146044|  PASSED
      rgb_lagged_sum|  23|   1000000|     100|0.44909462|  PASSED
      rgb_lagged_sum|  24|   1000000|     100|0.94590745|  PASSED
      rgb_lagged_sum|  25|   1000000|     100|0.47595911|  PASSED
      rgb_lagged_sum|  26|   1000000|     100|0.48077232|  PASSED
      rgb_lagged_sum|  27|   1000000|     100|0.66249603|  PASSED
      rgb_lagged_sum|  28|   1000000|     100|0.65660063|  PASSED
      rgb_lagged_sum|  29|   1000000|     100|0.22870532|  PASSED
      rgb_lagged_sum|  30|   1000000|     100|0.52947733|  PASSED
      rgb_lagged_sum|  31|   1000000|     100|0.08919333|  PASSED
      rgb_lagged_sum|  32|   1000000|     100|0.98542855|  PASSED
     rgb_kstest_test|   0|     10000|    1000|0.37986573|  PASSED
     dab_bytedistrib|   0|  51200000|       1|0.21347786|  PASSED
             dab_dct| 256|     50000|       1|0.81674467|  PASSED
Skipping test 207
Preparing to run test 208.  ntuple = 0
```

```
       dab_filltree2|   0|   5000000|       1|0.54753360|  PASSED
       dab_filltree2|   1|   5000000|       1|0.42368312|  PASSED
Preparing to run test 209.  ntuple = 0
        dab_monobit2|  12|  65000000|       1|0.82278974|  PASSED
Preparing to run test 210.  ntuple = 0
#=============================================================================#
#   mean  |  stddev |  error-rate (best = 0.0, worst = 0.5)
#=============================================================================#
0.216301  |0.115996  |
```

Listing 8.16: Test results for random number engine `trng::mt19937`.

```
#=============================================================================#
#          dieharder version 3.31.2beta Copyright 2003 Robert G. Brown        #
#=============================================================================#
   rng_name    |rands/second|   Seed   |  k ints/sec|k doubles/sec|
   trng_mt19937|  2.07e+08  |3499974960|    206808  |     237051  |
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
   diehard_birthdays|   0|       100|     100|0.16463040|  PASSED
      diehard_operm5|   0|   1000000|     100|0.78019960|  PASSED
  diehard_rank_32x32|   0|     40000|     100|0.62419810|  PASSED
    diehard_rank_6x8|   0|    100000|     100|0.26681693|  PASSED
   diehard_bitstream|   0|   2097152|     100|0.87776498|  PASSED
        diehard_opso|   0|   2097152|     100|0.80592270|  PASSED
        diehard_oqso|   0|   2097152|     100|0.38521353|  PASSED
         diehard_dna|   0|   2097152|     100|0.25232215|  PASSED
diehard_count_1s_str|   0|    256000|     100|0.87002043|  PASSED
diehard_count_1s_byt|   0|    256000|     100|0.79895800|  PASSED
 diehard_parking_lot|   0|     12000|     100|0.64976124|  PASSED
    diehard_2dsphere|   2|      8000|     100|0.28906317|  PASSED
    diehard_3dsphere|   3|      4000|     100|0.02406320|  PASSED
     diehard_squeeze|   0|    100000|     100|0.98738347|  PASSED
        diehard_runs|   0|    100000|     100|0.19947835|  PASSED
        diehard_runs|   0|    100000|     100|0.56870826|  PASSED
       diehard_craps|   0|    200000|     100|0.98396304|  PASSED
       diehard_craps|   0|    200000|     100|0.44771931|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.80019812|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.85964446|  PASSED
         sts_monobit|   1|    100000|     100|0.02221872|  PASSED
            sts_runs|   2|    100000|     100|0.66669930|  PASSED
          sts_serial|   1|    100000|     100|0.29769282|  PASSED
          sts_serial|   2|    100000|     100|0.88188427|  PASSED
          sts_serial|   3|    100000|     100|0.68143206|  PASSED
          sts_serial|   3|    100000|     100|0.42158619|  PASSED
          sts_serial|   4|    100000|     100|0.63373726|  PASSED
          sts_serial|   4|    100000|     100|0.29455649|  PASSED
          sts_serial|   5|    100000|     100|0.94934669|  PASSED
          sts_serial|   5|    100000|     100|0.17998159|  PASSED
          sts_serial|   6|    100000|     100|0.84948329|  PASSED
          sts_serial|   6|    100000|     100|0.60184228|  PASSED
          sts_serial|   7|    100000|     100|0.88850770|  PASSED
          sts_serial|   7|    100000|     100|0.89448526|  PASSED
          sts_serial|   8|    100000|     100|0.65698579|  PASSED
          sts_serial|   8|    100000|     100|0.84217163|  PASSED
          sts_serial|   9|    100000|     100|0.70426658|  PASSED
          sts_serial|   9|    100000|     100|0.54250339|  PASSED
          sts_serial|  10|    100000|     100|0.97850429|  PASSED
          sts_serial|  10|    100000|     100|0.81362297|  PASSED
          sts_serial|  11|    100000|     100|0.71753452|  PASSED
          sts_serial|  11|    100000|     100|0.68996582|  PASSED
          sts_serial|  12|    100000|     100|0.28502159|  PASSED
          sts_serial|  12|    100000|     100|0.65422771|  PASSED
          sts_serial|  13|    100000|     100|0.62064622|  PASSED
          sts_serial|  13|    100000|     100|0.70556224|  PASSED
          sts_serial|  14|    100000|     100|0.86532457|  PASSED
          sts_serial|  14|    100000|     100|0.91538389|  PASSED
          sts_serial|  15|    100000|     100|0.99766778|   WEAK
          sts_serial|  15|    100000|     100|0.60562273|  PASSED
```

```
          sts_serial|  16|    100000|     100|0.97784531|  PASSED
          sts_serial|  16|    100000|     100|0.55548498|  PASSED
         rgb_bitdist|   1|    100000|     100|0.97465565|  PASSED
         rgb_bitdist|   2|    100000|     100|0.62703618|  PASSED
         rgb_bitdist|   3|    100000|     100|0.53843082|  PASSED
         rgb_bitdist|   4|    100000|     100|0.51216007|  PASSED
         rgb_bitdist|   5|    100000|     100|0.96655031|  PASSED
         rgb_bitdist|   6|    100000|     100|0.30039303|  PASSED
         rgb_bitdist|   7|    100000|     100|0.04593593|  PASSED
         rgb_bitdist|   8|    100000|     100|0.64942381|  PASSED
         rgb_bitdist|   9|    100000|     100|0.53179589|  PASSED
         rgb_bitdist|  10|    100000|     100|0.98774183|  PASSED
         rgb_bitdist|  11|    100000|     100|0.47629845|  PASSED
         rgb_bitdist|  12|    100000|     100|0.60913267|  PASSED
rgb_minimum_distance|   2|     10000|    1000|0.65813233|  PASSED
rgb_minimum_distance|   3|     10000|    1000|0.78030917|  PASSED
rgb_minimum_distance|   4|     10000|    1000|0.42207539|  PASSED
rgb_minimum_distance|   5|     10000|    1000|0.87043229|  PASSED
    rgb_permutations|   2|    100000|     100|0.02943198|  PASSED
    rgb_permutations|   3|    100000|     100|0.99700270|   WEAK
    rgb_permutations|   4|    100000|     100|0.01023023|  PASSED
    rgb_permutations|   5|    100000|     100|0.11263803|  PASSED
      rgb_lagged_sum|   0|   1000000|     100|0.61830657|  PASSED
      rgb_lagged_sum|   1|   1000000|     100|0.47291520|  PASSED
      rgb_lagged_sum|   2|   1000000|     100|0.37663455|  PASSED
      rgb_lagged_sum|   3|   1000000|     100|0.99850744|   WEAK
      rgb_lagged_sum|   4|   1000000|     100|0.99915828|   WEAK
      rgb_lagged_sum|   5|   1000000|     100|0.08086062|  PASSED
      rgb_lagged_sum|   6|   1000000|     100|0.15708646|  PASSED
      rgb_lagged_sum|   7|   1000000|     100|0.78836106|  PASSED
      rgb_lagged_sum|   8|   1000000|     100|0.33221121|  PASSED
      rgb_lagged_sum|   9|   1000000|     100|0.02094641|  PASSED
      rgb_lagged_sum|  10|   1000000|     100|0.99993773|   WEAK
      rgb_lagged_sum|  11|   1000000|     100|0.69703106|  PASSED
      rgb_lagged_sum|  12|   1000000|     100|0.23776162|  PASSED
      rgb_lagged_sum|  13|   1000000|     100|0.99927233|   WEAK
      rgb_lagged_sum|  14|   1000000|     100|0.15464674|  PASSED
      rgb_lagged_sum|  15|   1000000|     100|0.17727454|  PASSED
      rgb_lagged_sum|  16|   1000000|     100|0.42913552|  PASSED
      rgb_lagged_sum|  17|   1000000|     100|0.77880194|  PASSED
      rgb_lagged_sum|  18|   1000000|     100|0.96875023|  PASSED
      rgb_lagged_sum|  19|   1000000|     100|0.61786357|  PASSED
      rgb_lagged_sum|  20|   1000000|     100|0.62658363|  PASSED
      rgb_lagged_sum|  21|   1000000|     100|0.56112353|  PASSED
      rgb_lagged_sum|  22|   1000000|     100|0.60519414|  PASSED
      rgb_lagged_sum|  23|   1000000|     100|0.18711776|  PASSED
      rgb_lagged_sum|  24|   1000000|     100|0.60617786|  PASSED
      rgb_lagged_sum|  25|   1000000|     100|0.25323999|  PASSED
      rgb_lagged_sum|  26|   1000000|     100|0.31725753|  PASSED
      rgb_lagged_sum|  27|   1000000|     100|0.93094372|  PASSED
      rgb_lagged_sum|  28|   1000000|     100|0.15759642|  PASSED
      rgb_lagged_sum|  29|   1000000|     100|0.78197104|  PASSED
      rgb_lagged_sum|  30|   1000000|     100|0.87808490|  PASSED
      rgb_lagged_sum|  31|   1000000|     100|0.04520679|  PASSED
      rgb_lagged_sum|  32|   1000000|     100|0.29080536|  PASSED
     rgb_kstest_test|   0|     10000|    1000|0.00324040|   WEAK
     dab_bytedistrib|   0|  51200000|       1|0.95590055|  PASSED
             dab_dct| 256|     50000|       1|0.85144915|  PASSED
Skipping test 207
Preparing to run test 208.  ntuple = 0
        dab_filltree2|   0|   5000000|       1|0.42268641|  PASSED
        dab_filltree2|   1|   5000000|       1|0.39651375|  PASSED
Preparing to run test 209.  ntuple = 0
         dab_monobit2|  12|  65000000|       1|0.84833926|  PASSED
Preparing to run test 210.  ntuple = 0
#=============================================================================#
#   mean   |  stddev |  error-rate (best = 0.0, worst = 0.5)
#=============================================================================#
0.181639  |0.125938  |
```

Listing 8.17: Test results for random number engine `trng::mt19937_64`.

```
#=============================================================================#
#            dieharder version 3.31.2beta Copyright 2003 Robert G. Brown          #
#=============================================================================#
   rng_name    |rands/second|   Seed   |  k ints/sec|k doubles/sec|
trng_mt19937_64|  1.51e+08  |2433103529|    150927  |     149171  |
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
   diehard_birthdays|   0|       100|     100|0.21571310|  PASSED
      diehard_operm5|   0|   1000000|     100|0.15396810|  PASSED
  diehard_rank_32x32|   0|     40000|     100|0.66703783|  PASSED
    diehard_rank_6x8|   0|    100000|     100|0.42255399|  PASSED
   diehard_bitstream|   0|   2097152|     100|0.76920004|  PASSED
        diehard_opso|   0|   2097152|     100|0.51629818|  PASSED
        diehard_oqso|   0|   2097152|     100|0.92018704|  PASSED
         diehard_dna|   0|   2097152|     100|0.63104389|  PASSED
diehard_count_1s_str|   0|    256000|     100|0.54634671|  PASSED
diehard_count_1s_byt|   0|    256000|     100|0.02476623|  PASSED
 diehard_parking_lot|   0|     12000|     100|0.50364626|  PASSED
    diehard_2dsphere|   2|      8000|     100|0.81089431|  PASSED
    diehard_3dsphere|   3|      4000|     100|0.35951679|  PASSED
     diehard_squeeze|   0|    100000|     100|0.53283860|  PASSED
        diehard_runs|   0|    100000|     100|0.84234662|  PASSED
        diehard_runs|   0|    100000|     100|0.98695750|  PASSED
       diehard_craps|   0|    200000|     100|0.18455120|  PASSED
       diehard_craps|   0|    200000|     100|0.10098367|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.93932001|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.93322405|  PASSED
         sts_monobit|   1|    100000|     100|0.41668443|  PASSED
            sts_runs|   2|    100000|     100|0.59396979|  PASSED
          sts_serial|   1|    100000|     100|0.69066679|  PASSED
          sts_serial|   2|    100000|     100|0.06458817|  PASSED
          sts_serial|   3|    100000|     100|0.45514135|  PASSED
          sts_serial|   3|    100000|     100|0.93041561|  PASSED
          sts_serial|   4|    100000|     100|0.74591453|  PASSED
          sts_serial|   4|    100000|     100|0.37441695|  PASSED
          sts_serial|   5|    100000|     100|0.05443383|  PASSED
          sts_serial|   5|    100000|     100|0.05207743|  PASSED
          sts_serial|   6|    100000|     100|0.07648128|  PASSED
          sts_serial|   6|    100000|     100|0.62107531|  PASSED
          sts_serial|   7|    100000|     100|0.96955756|  PASSED
          sts_serial|   7|    100000|     100|0.44626064|  PASSED
          sts_serial|   8|    100000|     100|0.42543728|  PASSED
          sts_serial|   8|    100000|     100|0.23043789|  PASSED
          sts_serial|   9|    100000|     100|0.29498205|  PASSED
          sts_serial|   9|    100000|     100|0.05806274|  PASSED
          sts_serial|  10|    100000|     100|0.41891966|  PASSED
          sts_serial|  10|    100000|     100|0.94225030|  PASSED
          sts_serial|  11|    100000|     100|0.83570513|  PASSED
          sts_serial|  11|    100000|     100|0.60421994|  PASSED
          sts_serial|  12|    100000|     100|0.84818457|  PASSED
          sts_serial|  12|    100000|     100|0.92400578|  PASSED
          sts_serial|  13|    100000|     100|0.88023905|  PASSED
          sts_serial|  13|    100000|     100|0.72486397|  PASSED
          sts_serial|  14|    100000|     100|0.71812497|  PASSED
          sts_serial|  14|    100000|     100|0.13466859|  PASSED
          sts_serial|  15|    100000|     100|0.85535415|  PASSED
          sts_serial|  15|    100000|     100|0.78856307|  PASSED
          sts_serial|  16|    100000|     100|0.29947622|  PASSED
          sts_serial|  16|    100000|     100|0.08024648|  PASSED
         rgb_bitdist|   1|    100000|     100|0.03166364|  PASSED
         rgb_bitdist|   2|    100000|     100|0.01091185|  PASSED
         rgb_bitdist|   3|    100000|     100|0.47576131|  PASSED
         rgb_bitdist|   4|    100000|     100|0.85385191|  PASSED
         rgb_bitdist|   5|    100000|     100|0.99175302|  PASSED
         rgb_bitdist|   6|    100000|     100|0.44858670|  PASSED
         rgb_bitdist|   7|    100000|     100|0.84057887|  PASSED
         rgb_bitdist|   8|    100000|     100|0.99997777|   WEAK
```

```
          rgb_bitdist|   9|    100000|     100|0.78612830|  PASSED
          rgb_bitdist|  10|    100000|     100|0.71366186|  PASSED
          rgb_bitdist|  11|    100000|     100|0.99980863|   WEAK
          rgb_bitdist|  12|    100000|     100|0.88154929|  PASSED
rgb_minimum_distance|   2|     10000|    1000|0.54342609|  PASSED
rgb_minimum_distance|   3|     10000|    1000|0.00898176|  PASSED
rgb_minimum_distance|   4|     10000|    1000|0.50609071|  PASSED
rgb_minimum_distance|   5|     10000|    1000|0.19251376|  PASSED
    rgb_permutations|   2|    100000|     100|0.85566904|  PASSED
    rgb_permutations|   3|    100000|     100|0.80562487|  PASSED
    rgb_permutations|   4|    100000|     100|0.87158337|  PASSED
    rgb_permutations|   5|    100000|     100|0.98512333|  PASSED
      rgb_lagged_sum|   0|   1000000|     100|0.06994686|  PASSED
      rgb_lagged_sum|   1|   1000000|     100|0.91156810|  PASSED
      rgb_lagged_sum|   2|   1000000|     100|0.58499443|  PASSED
      rgb_lagged_sum|   3|   1000000|     100|0.00332462|   WEAK
      rgb_lagged_sum|   4|   1000000|     100|0.69280240|  PASSED
      rgb_lagged_sum|   5|   1000000|     100|0.55588891|  PASSED
      rgb_lagged_sum|   6|   1000000|     100|0.71748123|  PASSED
      rgb_lagged_sum|   7|   1000000|     100|0.50459209|  PASSED
      rgb_lagged_sum|   8|   1000000|     100|0.48294011|  PASSED
      rgb_lagged_sum|   9|   1000000|     100|0.02336243|  PASSED
      rgb_lagged_sum|  10|   1000000|     100|0.16002083|  PASSED
      rgb_lagged_sum|  11|   1000000|     100|0.60380055|  PASSED
      rgb_lagged_sum|  12|   1000000|     100|0.97007418|  PASSED
      rgb_lagged_sum|  13|   1000000|     100|0.97425936|  PASSED
      rgb_lagged_sum|  14|   1000000|     100|0.80860882|  PASSED
      rgb_lagged_sum|  15|   1000000|     100|0.88115885|  PASSED
      rgb_lagged_sum|  16|   1000000|     100|0.13685530|  PASSED
      rgb_lagged_sum|  17|   1000000|     100|0.28229906|  PASSED
      rgb_lagged_sum|  18|   1000000|     100|0.77483530|  PASSED
      rgb_lagged_sum|  19|   1000000|     100|0.70887727|  PASSED
      rgb_lagged_sum|  20|   1000000|     100|0.86254332|  PASSED
      rgb_lagged_sum|  21|   1000000|     100|0.52902065|  PASSED
      rgb_lagged_sum|  22|   1000000|     100|0.28211929|  PASSED
      rgb_lagged_sum|  23|   1000000|     100|0.53938106|  PASSED
      rgb_lagged_sum|  24|   1000000|     100|0.85555850|  PASSED
      rgb_lagged_sum|  25|   1000000|     100|0.93286557|  PASSED
      rgb_lagged_sum|  26|   1000000|     100|0.92861649|  PASSED
      rgb_lagged_sum|  27|   1000000|     100|0.22437790|  PASSED
      rgb_lagged_sum|  28|   1000000|     100|0.67029069|  PASSED
      rgb_lagged_sum|  29|   1000000|     100|0.20327150|  PASSED
      rgb_lagged_sum|  30|   1000000|     100|0.42472311|  PASSED
      rgb_lagged_sum|  31|   1000000|     100|0.44737767|  PASSED
      rgb_lagged_sum|  32|   1000000|     100|0.99160488|  PASSED
     rgb_kstest_test|   0|     10000|    1000|0.32751572|  PASSED
     dab_bytedistrib|   0|  51200000|       1|0.74997200|  PASSED
             dab_dct| 256|     50000|       1|0.54693658|  PASSED
Skipping test 207
Preparing to run test 208.  ntuple = 0
        dab_filltree2|   0|   5000000|       1|0.10123976|  PASSED
        dab_filltree2|   1|   5000000|       1|0.09852725|  PASSED
Preparing to run test 209.  ntuple = 0
         dab_monobit2|  12|  65000000|       1|0.31169090|  PASSED
Preparing to run test 210.  ntuple = 0
#=============================================================================#
#   mean   |  stddev |  error-rate (best = 0.0, worst = 0.5)
#=============================================================================#
0.260314  |0.155059  |
```

Listing 8.18: Test results for random number engine `trng::xoshiro256plus`.

```
#=============================================================================#
#          dieharder version 3.31.2beta Copyright 2003 Robert G. Brown        #
#=============================================================================#
   rng_name    |rands/second|   Seed   |  k ints/sec|k doubles/sec|
trng_xoshiro256plus|  2.40e+08  | 237993098|    240286  |     263407  |
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
```

```
   diehard_birthdays|   0|       100|     100|0.87585169|  PASSED
      diehard_operm5|   0|   1000000|     100|0.31893836|  PASSED
  diehard_rank_32x32|   0|     40000|     100|0.63022920|  PASSED
    diehard_rank_6x8|   0|    100000|     100|0.30856757|  PASSED
   diehard_bitstream|   0|   2097152|     100|0.54800765|  PASSED
        diehard_opso|   0|   2097152|     100|0.52127419|  PASSED
        diehard_oqso|   0|   2097152|     100|0.88426787|  PASSED
         diehard_dna|   0|   2097152|     100|0.21516088|  PASSED
diehard_count_1s_str|   0|    256000|     100|0.23435276|  PASSED
diehard_count_1s_byt|   0|    256000|     100|0.27568791|  PASSED
 diehard_parking_lot|   0|     12000|     100|0.65655550|  PASSED
    diehard_2dsphere|   2|      8000|     100|0.18340999|  PASSED
    diehard_3dsphere|   3|      4000|     100|0.40357936|  PASSED
     diehard_squeeze|   0|    100000|     100|0.99717703|   WEAK
        diehard_runs|   0|    100000|     100|0.14088141|  PASSED
        diehard_runs|   0|    100000|     100|0.41977168|  PASSED
       diehard_craps|   0|    200000|     100|0.54157096|  PASSED
       diehard_craps|   0|    200000|     100|0.59263038|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.14016539|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.83244985|  PASSED
         sts_monobit|   1|    100000|     100|0.39504889|  PASSED
            sts_runs|   2|    100000|     100|0.14298796|  PASSED
          sts_serial|   1|    100000|     100|0.50238448|  PASSED
          sts_serial|   2|    100000|     100|0.69365129|  PASSED
          sts_serial|   3|    100000|     100|0.80922103|  PASSED
          sts_serial|   3|    100000|     100|0.99404600|  PASSED
          sts_serial|   4|    100000|     100|0.82034603|  PASSED
          sts_serial|   4|    100000|     100|0.93319053|  PASSED
          sts_serial|   5|    100000|     100|0.89566694|  PASSED
          sts_serial|   5|    100000|     100|0.70082362|  PASSED
          sts_serial|   6|    100000|     100|0.16836061|  PASSED
          sts_serial|   6|    100000|     100|0.40035425|  PASSED
          sts_serial|   7|    100000|     100|0.66221919|  PASSED
          sts_serial|   7|    100000|     100|0.99807377|   WEAK
          sts_serial|   8|    100000|     100|0.58584988|  PASSED
          sts_serial|   8|    100000|     100|0.86655813|  PASSED
          sts_serial|   9|    100000|     100|0.87990788|  PASSED
          sts_serial|   9|    100000|     100|0.76453357|  PASSED
          sts_serial|  10|    100000|     100|0.97200443|  PASSED
          sts_serial|  10|    100000|     100|0.98629460|  PASSED
          sts_serial|  11|    100000|     100|0.74184335|  PASSED
          sts_serial|  11|    100000|     100|0.19888186|  PASSED
          sts_serial|  12|    100000|     100|0.91514712|  PASSED
          sts_serial|  12|    100000|     100|0.81265084|  PASSED
          sts_serial|  13|    100000|     100|0.82432729|  PASSED
          sts_serial|  13|    100000|     100|0.57364298|  PASSED
          sts_serial|  14|    100000|     100|0.95870106|  PASSED
          sts_serial|  14|    100000|     100|0.25820387|  PASSED
          sts_serial|  15|    100000|     100|0.90282458|  PASSED
          sts_serial|  15|    100000|     100|0.90826265|  PASSED
          sts_serial|  16|    100000|     100|0.66810295|  PASSED
          sts_serial|  16|    100000|     100|0.18792699|  PASSED
         rgb_bitdist|   1|    100000|     100|0.34295745|  PASSED
         rgb_bitdist|   2|    100000|     100|0.71223099|  PASSED
         rgb_bitdist|   3|    100000|     100|0.22099283|  PASSED
         rgb_bitdist|   4|    100000|     100|0.23594899|  PASSED
         rgb_bitdist|   5|    100000|     100|0.41228542|  PASSED
         rgb_bitdist|   6|    100000|     100|0.87650443|  PASSED
         rgb_bitdist|   7|    100000|     100|0.80292716|  PASSED
         rgb_bitdist|   8|    100000|     100|0.10738575|  PASSED
         rgb_bitdist|   9|    100000|     100|0.83952005|  PASSED
         rgb_bitdist|  10|    100000|     100|0.95457924|  PASSED
         rgb_bitdist|  11|    100000|     100|0.82966253|  PASSED
         rgb_bitdist|  12|    100000|     100|0.91999581|  PASSED
rgb_minimum_distance|   2|     10000|    1000|0.71420713|  PASSED
rgb_minimum_distance|   3|     10000|    1000|0.21739662|  PASSED
rgb_minimum_distance|   4|     10000|    1000|0.53624305|  PASSED
rgb_minimum_distance|   5|     10000|    1000|0.70154164|  PASSED
    rgb_permutations|   2|    100000|     100|0.51533674|  PASSED
    rgb_permutations|   3|    100000|     100|0.99689787|   WEAK
```

```
    rgb_permutations|   4|    100000|     100|0.50169354|  PASSED
    rgb_permutations|   5|    100000|     100|0.99436505|  PASSED
      rgb_lagged_sum|   0|   1000000|     100|0.44490972|  PASSED
      rgb_lagged_sum|   1|   1000000|     100|0.43234511|  PASSED
      rgb_lagged_sum|   2|   1000000|     100|0.80021545|  PASSED
      rgb_lagged_sum|   3|   1000000|     100|0.50099644|  PASSED
      rgb_lagged_sum|   4|   1000000|     100|0.39916584|  PASSED
      rgb_lagged_sum|   5|   1000000|     100|0.45994285|  PASSED
      rgb_lagged_sum|   6|   1000000|     100|0.18291153|  PASSED
      rgb_lagged_sum|   7|   1000000|     100|0.68670632|  PASSED
      rgb_lagged_sum|   8|   1000000|     100|0.93410224|  PASSED
      rgb_lagged_sum|   9|   1000000|     100|0.66810630|  PASSED
      rgb_lagged_sum|  10|   1000000|     100|0.19125648|  PASSED
      rgb_lagged_sum|  11|   1000000|     100|0.21386016|  PASSED
      rgb_lagged_sum|  12|   1000000|     100|0.72454842|  PASSED
      rgb_lagged_sum|  13|   1000000|     100|0.68878156|  PASSED
      rgb_lagged_sum|  14|   1000000|     100|0.47111674|  PASSED
      rgb_lagged_sum|  15|   1000000|     100|0.68442562|  PASSED
      rgb_lagged_sum|  16|   1000000|     100|0.86262233|  PASSED
      rgb_lagged_sum|  17|   1000000|     100|0.38159296|  PASSED
      rgb_lagged_sum|  18|   1000000|     100|0.90039163|  PASSED
      rgb_lagged_sum|  19|   1000000|     100|0.71486845|  PASSED
      rgb_lagged_sum|  20|   1000000|     100|0.02844656|  PASSED
      rgb_lagged_sum|  21|   1000000|     100|0.83936352|  PASSED
      rgb_lagged_sum|  22|   1000000|     100|0.71569243|  PASSED
      rgb_lagged_sum|  23|   1000000|     100|0.49976734|  PASSED
      rgb_lagged_sum|  24|   1000000|     100|0.00270853|   WEAK
      rgb_lagged_sum|  25|   1000000|     100|0.02194632|  PASSED
      rgb_lagged_sum|  26|   1000000|     100|0.42793087|  PASSED
      rgb_lagged_sum|  27|   1000000|     100|0.34837809|  PASSED
      rgb_lagged_sum|  28|   1000000|     100|0.58993907|  PASSED
      rgb_lagged_sum|  29|   1000000|     100|0.35129039|  PASSED
      rgb_lagged_sum|  30|   1000000|     100|0.28901798|  PASSED
      rgb_lagged_sum|  31|   1000000|     100|0.71284278|  PASSED
      rgb_lagged_sum|  32|   1000000|     100|0.15101729|  PASSED
     rgb_kstest_test|   0|     10000|    1000|0.04478140|  PASSED
     dab_bytedistrib|   0|  51200000|       1|0.93751803|  PASSED
             dab_dct| 256|     50000|       1|0.66213904|  PASSED
Skipping test 207
Preparing to run test 208.  ntuple = 0
       dab_filltree2|   0|   5000000|       1|0.06482576|  PASSED
       dab_filltree2|   1|   5000000|       1|0.09758981|  PASSED
Preparing to run test 209.  ntuple = 0
        dab_monobit2|  12|  65000000|       1|0.94713875|  PASSED
Preparing to run test 210.  ntuple = 0
#=============================================================================#
#   mean  |  stddev |  error-rate (best = 0.0, worst = 0.5)
#=============================================================================#
0.229707  |0.139701  |
```

9 Frequently asked questions

What are the license terms for using and distributing the TRNG library? TRNG is free software. Starting from version 4.9, the TRNG library is distributed under the terms of a BSD style license (3-clause license). Earlier TRNG versions are distributed under the GNU Public License (GPL) Version 2. See also page 159.

Why is the library called TRNG? Who is Tina? Tina is the name of a Linux cluster at the Institute of Theoretical Physics at the University Magdeburg in Germany. TRNG was written to carry out Monte Carlo simulations on this parallel computer. The name Tina is a self referring acronym for "Tina is no acronym". The abbreviation TRNG stands for "Tina's Random Number Generator Library". But sometimes it is used in the literature for "true random number generator" as well, which is a technical device that generates random numbers by a physical process (e. g. radioactive decay or noise in a electric circuit).

I am confused, there are so many different PRNGs in TRNG. Which one is the best? There is nothing like the best PRNG. If a generator behaves as a good source of randomness or not can depend on your Monte Carlo application, and there are trade-offs between speed and quality. In general, it is a good idea to test if the outcome of a Monte Carlo simulation is independent of the underlying PRNG. Therefore TRNG offers so many of them.

But generally speaking, YARN generators are a good choice (see section 4.1.3). If the PRNG is the bottleneck of your Monte Carlo simulation you might try the linear congruential generator (see section 4.1.1) or in the case of a sequential simulation a lagged Fibonacci generator with four feedback taps (see section 4.1.4).

Why is TRNG written in C++? C++ provides a lot of advanced features as inline functions and static polymorphism via templates. These language features give us the power to implement a fast, portable and easy to use library of PRNGs. Other languages (as FORTRAN or C) do no offer these (or comparable) features, are significantly slower (as Java or scripting languages), or are supported by fewer platforms.

How can I use TRNG in my FORTRAN programs? Unfortunately this is not possible. TRNG makes heavy use of special C++ language features as classes, inline functions, and templates. All theses concepts have no counterpart in the FORTRAN programming language. Large parts of TRNG even do not reside in the library that you link with `-ltrng4` to your object code. Template functions and inline functions are defined exclusively in the header files.

How can I use TRNG in my C programs? Unfortunately this is not possible. Here the same statements apply as for the last question. However, it is much more easy to port a C program to C++ than porting a FORTRAN program to C++. Just comply with the following recipe.

- Rename header files *foo*.h of the C standard library into c*foo* but let other header files untouched, i. e., change

  ```
  #include <stdio.h>
  #include <math.h>
  #include <unistd.h>
  ```

  into

  ```
  #include <cstdio>
  #include <cmath>
  #include <unistd.h>
  ```

  Note, the `unistd.h` header is not part of the C standard library.
- Insert the line

  ```
  using namespace std;
  ```

  after the include directives of each source file.
- Do not use C++ function names that are C++ keywords, i. e., `class`, `new`, `public` or `private`.

This recipe will give you an ugly but valid C++ program, at least in the most cases. This modified "C" program has to be compiled by a C++ compiler now, but it is ready to benefit from the TRNG library.

How can I give feedback, report bugs, or make a feature request? Send bug reports and feature requests to the author of TRNG via e-mail to trng@mail.de or open an issue on Github [72].

I used TRNG in my research and want to give credit. How should I cite TRNG? The main concepts, which TRNG builds on, are published in Heiko Bauke and Stephan Mertens. Random numbers for large-scale distributed Monte Carlo simulations. *Physical Review E*, 75(6):066701, 2007. Please cite this publication.

License

Starting from version 4.9, the TRNG library is distributed under the terms of a BSD style license (3-clause license). Earlier TRNG versions are distributed under the GNU Public License (GPL) Version 2. The BSD license is a much more liberal license than the GPL but it is a GPL compatible license. Thus, TRNG 4.9 and later versions may be used in GPL software projects.

Bibliography

[1] Diehard tests. https://en.wikipedia.org/wiki/Diehard_tests.

[2] NVIDIA CUDA C Programming Guide, 2010.

[3] L. Yu. Barash and L. N. Shchur. PRAND: GPU accelerated parallel random number generation library: Using most reliable algorithms and applying rallelism of modern GPUs and CPUs. *Computer Physics Communications*, 185(4):1343–1353, 2014.

[4] Heiko Bauke and Stephan Mertens. Pseudo random coins show more heads than tails. *Journal of Statistical Physics*, 114(3):1149–1169, 2004.

[5] Heiko Bauke and Stephan Mertens. *Cluster Computing*. Springer, 2005.

[6] Heiko Bauke and Stephan Mertens. Random numbers for large-scale distributed Monte Carlo simulations. *Physical Review E*, 75(6):066701, 2007.

[7] David Blackman and Sebastiano Vigna. Scrambled linear pseudorandom number generators. https://arxiv.org/abs/1805.01407, 2019.

[8] Boost C++ libraries. http://www.boost.org.

[9] Robert G. Brown. Dieharder: A Random Number Test Suite (modified for TRNG). https://github.com/rabauke/dieharder/tree/trng.

[10] Robert G. Brown. Dieharder: A Random Number Test Suite. https://webhome.phy.duke.edu/~rgb/General/dieharder.php, 2021.

[11] Walter E. Brown, Mark Fischler, Jim Kowalkowski, and Marc Paterno. *Random Number Generation in C++0X: A Comprehensive Proposal, version 2*, 2006. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2032.pdf.

[12] CMake documenation. https://cmake.org/documentation/.

[13] CMake projects in Visual Studio. https://docs.microsoft.com/en-us/cpp/build/cmake-projects-in-visual-studio.

[14] Aaldert Compagner. Definitions of randomness. *American Journal of Physics*, 59(8):700–705, August 1991.

[15] Aaldert Compagner. The hierarchy of correlations in random binary sequences. *Journal of Statistical Physics*, 63:883–896, 1991.

[16] Lih-Yuan Deng and Dale Bowman. Developments in pseudo-random number generators. *Wiley Interdisciplinary Reviews: Computational Statistics*, 9(5):e1404, Aug 2017.

[17] Luc Devroye. *Non-Uniform Random Variate Generation*. Springer, 1986.

[18] Jürgen Eichenauer-Herrmann and Holger Grothe. A remark on long-range correlations in multiplicative congruential pseudo random number generators. *Numerische Mathematik*, 56(6):609–611, 1989.

[19] Alan M. Ferrenberg and D. P. Landau. Monte carlo simulations: Hidden errors from "good" random number generators. *Physical Review Letters*, 69(23):3382–3384, 1992.

[20] Jay Fillmore and Morris Marx. Linear recursive sequences. *SIAM Review*, 10(3):342–353, 1968.

[21] George Fishman. *Monte Carlo*. Springer, 1996.

[22] S. W. Golomb. *Shift Register Sequences*. Aegan Park Press, Laguna Hills, CA, revised edition, 1982.

[23] Peter Grassberger. On correlations in "good" random number generators. *Physics Letters A*, 181(1):43–46, 1993.

[24] Holger Grothe. Matrix generators for pseudo-random vector generation. *Statistische Hefte*, 28(1):233–238, Dec 1987.

[25] A. Grube. Mehrfach rekursiv-erzeugte Pseudo-Zufallszahlen. *Zeitschrift für angewandte Mathematik und Mechanik*, 53:T223–T225, 1973.

[26] Intel Threading Building Blocks. http://www.threadingbuildingblocks.org.

[27] ISO. *ISO/IEC 14882:2011 Information technology – Programming languages – C++*. ISO.

[28] Nicolai M. Josuttis. *The C++ Standard Library*. Addison Wesley, 2nd edition, 2012.

[29] Dieter Jungnickel. *Finite Fields: Structure and Arithmetics*. Bibliographisches Institut, 1993.

[30] David Kirk and Wen mei Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2010.

[31] Scott Kirkpatrick and Erich P. Stoll. A very fast shift-register sequence random number generator. *Journal of Computational Physics*, 40(2):517–526, 1981.

[32] Donald E. Knuth. *The Art of Computer Programming*, volume 2: Seminumerical Algorithms. Addison Wesley Professional, 1st edition, 1969.

[33] Donald E. Knuth. *The Art of Computer Programming*, volume 2: Seminumerical Algorithms. Addison Wesley Professional, 3rd edition, 1998.

[34] Werner Krauth. *Statistical Mechanics: Algorithms and Computations*. Oxford Master Series in Statistical, Computational, and Theoretical Physics. Oxford University Press, 2006.

[35] David P. Landau and Kurt Binder. *A Guide to Monte Carlo Simulations in Statistical Physics*. Cambridge University Press, 2nd edition, 2005.

[36] Pierre L'Ecuyer. Random numbers for simulation. *Communications of the ACM*, 33(10):85–97, 1990.

[37] Pierre L'Ecuyer. A search for good multiple recursive random number generators. *ACM Transactions on Modeling and Computer Simulation*, 3(2):87–98, 1993.

[38] Pierre L'Ecuyer. Tables of linear congruential generators of different sizes and good lattice structure. *Mathematics of Computation*, 68:249–260, 1999.

[39] Pierre L'Ecuyer. Software for uniform random number generation: Distinguishing the good and the bad. In *Proceedings of the 2001 Winter Simulation Conference*, pages 95–105. IEEE, IEEE Press, 2001.

[40] Pierre L'Ecuyer. Random number generation. In James E. Gentle, Wolfgang Härdle, and Yuichi Mori, editors, *Handbook of Computational Statistics*. Springer, 2004.

[41] Pierre L'Ecuyer and Peter Hellekalek. Random number generators: Selection criteria and testing. In *Random and Quasi-Random Point Sets*, volume 138 of *Lecture Notes in Statistics*, pages 223–266. Springer, 1998.

[42] D. H. Lehmer. Mathematical methods in large-scale computing units. In *Proc. 2nd Sympos. on Large-Scale Digital Calculating Machinery, Cambridge, MA, 1949*, pages 141–146. Harvard University Press, 1951.

[43] Rudolf Lidl and Harald Niederreiter. *Introduction to finite fields and their applications*. Cambridge University Press, 2nd edition, 1994.

[44] Rudolf Lidl and Harald Niederreiter. *Finite Fields*, volume 20 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, 2nd edition, 1997.

[45] George Marsaglia. Random numbers fall mainly in the planes. *Proceedings of the National Academy of Sciences*, 61:25–28, 1968.

[46] Michael Mascagni. Parallel linear congruential generators with prime moduli. *Prallel Computing*, 24(5–6):923–936, 1998.

[47] Michael Mascagni and Hongmei Chi. Parallel linear congruential generators with Sophie-Germain moduli. *Parallel Computing*, 30(11):1217–1231, 2004.

[48] Michael Mascagni and Lin-Yee Hin. Parallel random number generators in monte carlo derivative pricing: An application-based test. *Monte Carlo Methods and Applications*, 18(2):161–179, Jan 2012.

[49] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998.

[50] A. De Matteis and S. Pagnutti. A class of parallel random number generators. *Parallel Computing*, 13(2):193–198, 1990.

[51] A. De Matteis and S. Pagnutti. Long-range correlations in linear and non-linear random number generators. *Parallel Computing*, 14(2):207–210, 1990.

[52] Don L. McLeish. *Monte Carlo Simulation and Finance*. John Wiley & Sons, 2005.

[53] Stephan Mertens. Random number generators: A survival guide for large scale simulations. https://arxiv.org/abs/0905.4238, 2009.

[54] Stephan Mertens and Heiko Bauke. Entropy of pseudo-random-number generators. *Physical Review E*, 69:055702–1–055702–4, 2004.

[55] MPICH2. http://www-unix.mcs.anl.gov/mpi/mpich.

[56] David R. Musser, Gillmer J. Derge, and Atul Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley Professional, 2001.

[57] M. E. J. Newman and G. T. Barkema. *Monte Carlo Methods in Statistical Physics*. Oxford University Press, 1999.

[58] Random number generation with multiple streams for sequential and parallel computing. Pierre L'Ecuyer. In *2015 Winter Simulation Conference (WSC)*. IEEE.

[59] Open MPI. http://www.open-mpi.org.

[60] Peter Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann Publishers Inc, 1996.

[61] W. H. Payne, J. R. Rabung, and T. P. Bogyo. Coding the lehmer pseudo-random number generator. *Communications of the ACM*, 12(2):85–86, 1969.

[62] Ora E. Percus and Malvin H. Kalos. Random number generators for MIMD parallel processors. *Journal of Parallel and Distributed Computing*, 6:477–497, 1989.

[63] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes*. Cambridge University Press, third edition, 2007.

[64] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, 2003.

[65] James Reinders. *Intel Threading Building Blocks*. O'Reilly, 2007.

[66] Christian P. Robert and George Casella. *Monte Carlo Statistical Methods*. Springer Texts in Statistics. Springer, 2004.

[67] Linus Schrage. A more portable Fortran random number generator. *ACM Transactions on Mathematical Software*, 5(2):132–138, 1979.

[68] L. N. Shchur, J. R. Heringa, and H. W. J. Blöte. Simulation of a directed random-walk model the effect of pseudo-random-number correlations. *Physica A*, 241(3–4):579–592, 1997.

[69] Mirai Solutions. rTRNG and Valgrind: Docker & actions to the rescue. https://mirai-solutions.ch/news/2021/02/10/rtrng-4.23.1-1-valgrind-docker-actions/, 2021.

[70] Dietrich Stauffer and Ammon Aharony. *Introduction to Percolation Theory*. Taylor & Francis Ltd, 2nd edition, 1994.

[71] Robert H. Swendsen and Jian-Sheng Wang. Nonuniversal critical dynamics in monte carlo simulations. *Physical Review Letters*, 58:86–88, 1987.

[72] Tina's Random Number Generator Library. https://www.numbercrunch.de/trng/, https://github.com/rabauke/trng4/.

[73] Zhe-Xian Wan. *Lectures on Finite Fields and Galois Rings*. World Scientific, 2003.

[74] Neal Zierler. Linear recurring sequences. *J. Soc. Indust. Appl. Math.*, 7(1):31–48, 1959.

[75] Robert M. Ziff. Four-tap shift-register-sequence random-number generators. *Computers in Physics*, 12(4), 1998.

Index