Tina's Random Number Generator Library

version 4.0

Heiko Bauke

September 22, 2006

Contents

1 TRNG in a nutshell

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 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 and clusters. 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 is the parallelization of the PRNG itself.

What do we expect from a library of PRNGs for parallel applications?

- 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.
- The library should provide methods for generating random variables 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 proposal [4] for an extensible random number generator facility, that will be part of the forthcoming revision of the ISO C++ standard. The TRNG library features an object oriented design, is easy to use and has been speed optimized. The implementation does not depend on any communication library or hardware architecture. It 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 Standard or OpenMP. All generators, that are implemented by TRNG, have been subjected to thorough statistical tests in sequential and parallel setups.

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.

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 ourself with something weaker than random numbers, *pseudo-random* numbers. We define in an informal manner a stream of PRNs r_i as follows:

- PRNs are generated by a deterministic rule.
- A stream of PRNs r_i cannot be distinguished from a true random sequence by means of 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.

The objective in PRNG design is to find a transition algorithm f 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 [17]. 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, which make the Monte Carlo simulation unreliable.

2.2 General parallelization techniques for PRNGs

In parallel applications we need to generate streams $t_{j,i}$ of random numbers, where $j = 0, 1, \ldots, p-1$ numbers the streams for each of the p processes. We require statistical independency of the $t_{j,i}$ within each stream and between the streams. Four different parallelization techniques are used in practice:

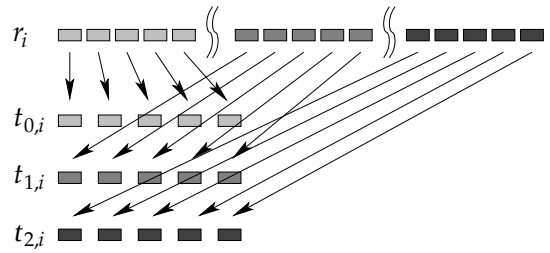

Figure 2.1: Parallelization by block splitting.

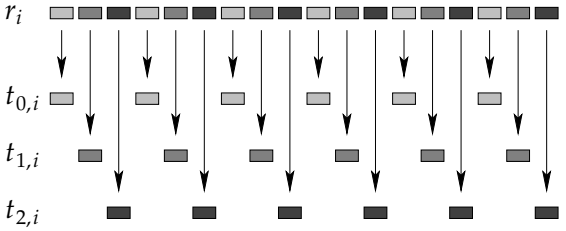

Figure 2.2: Parallelization by leapfrogging.

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" [17].

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 [40]:

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

where b_j is the $(j+2)$th prime number. Another variant uses different multipliers a for different streams [28]. The theoretical foundation of these methods is weak, and empirical tests have revealed serious correlations between streams [31]. 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) of a sequential PRNG into consecutive blocks of length M such that

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

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 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 [32, 7]. Block splitting is illustrated in Figure 2.1.

Leapfrog: The leapfrog method distributes a sequence (r) 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 kth element of the original sequence. Again this excludes many popular PRNGs.

The first two methods 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 to use 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), 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 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 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 one may use one PRNG to construct the bond percolation clusters and another PRNG to decide to flip the cluster.

2.4 Linear feedback shift register sequences

Numerous recipes for f in (2.1) have been discussed in the literature, see [17, 22] and references therein. One of the oldest and most popular PRNGs is the linear congruential generator (LCG)

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

introduced by Lehmer [24]. The additive constant b may be zero. Knuth [16] proposed a generalization of Lehmer's method known as multiple recurrence generator (MRG)

$$r_i = a_1 r_{i-1} + a_2 r_{i-2} + \ldots + a_n r_{i-n} \bmod m\,. \tag{2.6}$$

In the theory of finite fields, a sequence of type (2.6) 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 discussion of LFSR sequences including proofs can be found in [11, 14, 25, 26, 9, 48].

Since the all zero tuple $(0, 0, \ldots, 0)$ is a fix point of (2.6), 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 [17]:

Theorem 1 The LFSR sequence (2.6) over $\mathbb{F}_m$ has period $m^n - 1$, if and only if the characteristic polynomial

$$f(x) = x^n - a_1 x^{n-1} - a_2 x^{n-2} - \ldots - a_n \tag{2.7}$$

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 $\phi(m^n - 1)/n = \mathcal{O}\left(m^n/(n \ln(n \ln m))\right)$ [47], 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 [14], and most computer algebra systems offer a procedure for this.

Theorem 2 Let (r) be an LFSR sequence (2.6) 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 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 [5, 6].

Theorem 3 Let (r) be an LFSR sequence (2.6) with period $T = m^n - 1$ and let α be a complex mth root of unity. Then

$$C(h) := \sum_{i=1}^{T} \alpha^{r_i} \cdot \overline{\alpha}^{r_{i+h}} = \begin{cases} T & \text{if } h = 0 \bmod T \\ -1 & \text{if } h \neq 0 \bmod T \end{cases} . \tag{2.8}$$

$C(h)$ 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 [11, 14].

2.4.1 Parallelization of LFSR sequences

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 A the linear recurrence (2.6) can be written as a vector matrix product.

$$\begin{pmatrix} r_{i-(n-1)} \\ \vdots \\ r_{i-1} \\ r_i \end{pmatrix} = \underbrace{\begin{pmatrix} 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \\ a_n & a_{n-1} & \dots & a_1 \end{pmatrix}}_{A} \begin{pmatrix} r_{i-n} \\ \vdots \\ r_{i-2} \\ r_{i-1} \end{pmatrix} \bmod m \tag{2.9}$$

From this formula it follows immediately that the M-fold successive iteration of (2.6) may be written as

$$\begin{pmatrix} r_{i-(n-1)} \\ \vdots \\ r_{i-1} \\ r_i \end{pmatrix} = A^M \begin{pmatrix} r_{i-M-(n-1)} \\ \vdots \\ r_{i-M-1} \\ r_{i-M} \end{pmatrix} \bmod m\,. \tag{2.10}$$

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

$$\begin{pmatrix} r_{pi+j-(n-1)} \\ \vdots \\ r_{pi+j-1} \\ r_{pi+j} \end{pmatrix} = A^p \begin{pmatrix} r_{p(i-1)+j-(n-1)} \\ \vdots \\ r_{p(i-1)+j-1} \\ r_{p(i-1)+j} \end{pmatrix} \bmod m \tag{2.11}$$

is no option, because 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 [14]. This will provide us with a very efficient way to generate leapfrog sequences.

Theorem 4 Let (r) 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.

$$t_{j,i} = r_{pi+j}\,. \tag{2.12}$$

Then (t_j) 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) and (t_j) are not just cyclic shifts of each other, except when

$$p = m^h \bmod (m^n - 1) \tag{2.13}$$

for some $0 \leq h < n$. If $\gcd(m^n - 1, p) > 1$ the sequence (t_j) 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.

While Theorem 4 ensures that leapfrog sequences are not just cyclic shifts of the base sequence (unless (2.13) holds), the leapfrog sequences are cyclic shifts of each other. This is true for general sequences, not just LFSR sequences. Consider an arbitrary sequence (r) with period T. If $\gcd(T, p) = 1$, all leapfrog sequences $(t_1), (t_2), \ldots, (t_p)$ are cyclic shifts of each other, i. e. for every pair of leapfrog sequences (t_{j_1}) and (t_{j_2}) of a common base sequence (r) 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 $\lfloor T/p \rfloor$.

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) 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) by solving the linear system

$$\begin{pmatrix} t_{i+n} \\ t_{i+n+1} \\ \vdots \\ t_{i+2n-1} \end{pmatrix} = \begin{pmatrix} t_{i+n-1} & \ldots & t_{i+1} & t_i \\ t_{i+n} & \ldots & t_{i+2} & t_{i+1} \\ \vdots & \ddots & \vdots & \vdots \\ t_{i+2n-2} & \ldots & t_{i+n} & t_{i+n-1} \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} \bmod m \tag{2.14}$$

over $\mathbb{F}_m$.

Starting from the base sequence we determine $2n$ values of the sequence (t) by applying the leapfrog rule. Then we solve (2.14) 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 (2.14) 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.

2.4.2 Choice of modulus

LFSR sequences can be defined over any prime field, and in particular LFSR sequences over $\mathbb{F}_2$ with sparse feedback polynomials are popular sources of PRNs [15, 49, 17] 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 mod-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 [8, 12, 44, 49] that can be blamed to the small size of the underlying field [1]. 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 $2^n - 1$.

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 [33] 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.

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.

2.5 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 (2.6) leads to the concentration of the sampling points on n-dimensional hyper-planes [13, 19], see also top of Figure 2.3. This phenomenon, first noticed by Marsaglia in 1968 [27], constitutes one of the well known tests of randomness applied to PRNGs, the so-called spectral test [17]. The spectral test checks the behavior of

a generator when its outputs are used to form d-tuples. LFSR sequences do fail the spectral test in dimensions larger than the register size n. Closely related to this mechanism are the observed correlations in other empirical tests like the birthday spacings test and the collision test [21, 23]. 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:

Theorem 6 Let (q) 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 (r) with

$$r_i = \begin{cases} g^{q_i} \bmod m & \text{if } q_i > 0 \\ 0 & \text{if } q_i = 0 \end{cases} \tag{2.15}$$

is a pseudo-noise sequence, too.

The proof of this theorem is trivial: since g is a generator of $\mathbb{F}_m^*$, the map (2.15) is bijective. We call delinearized generators based on Theorem 6 YARN generators (yet another random number).

The linearity is completely destroyed by the map (2.15), 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). 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 nonlinear as it can get.

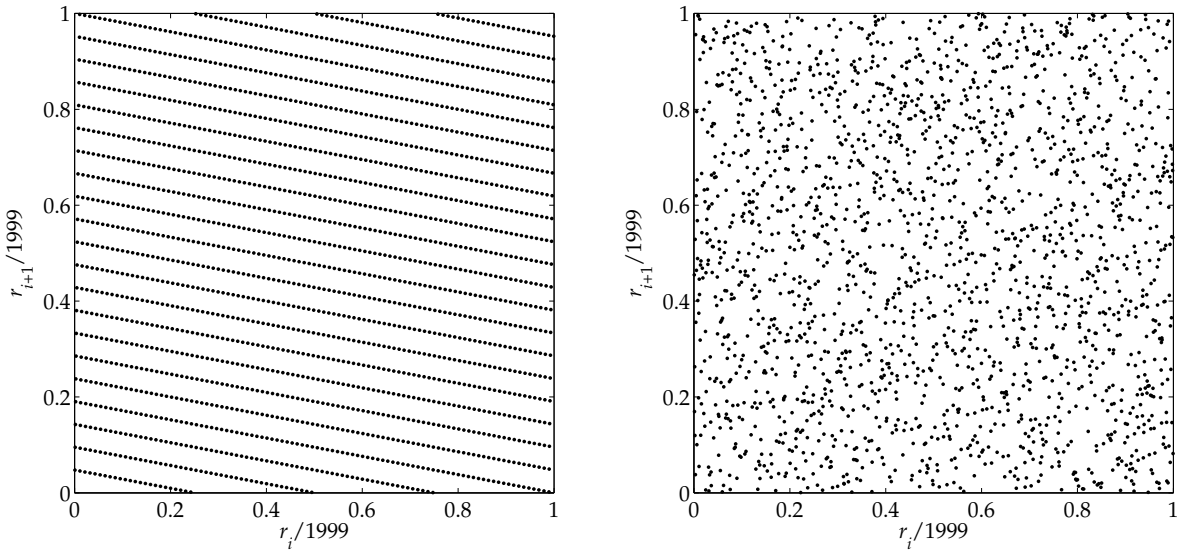

Figure 2.3: Exponentiation of a generating element in a prime field is an effective way to destroy the linear structures of LFSR sequences. Both pictures show the full period of the generator. Top: $r_i = 95 \cdot r_{i-i} \bmod 1999$. Bottom: $r_i = 1099^{q_i} \bmod 1999$ with $q_i = 95 \cdot q_{i-i}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

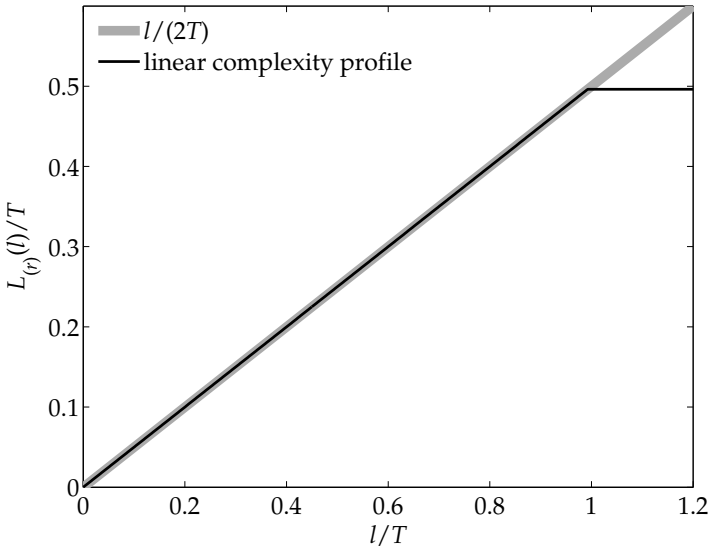

Figure 2.4: Linear complexity profile $L_{(r)}(l)$ of a YARN sequence, produced by the recurrence $q_i = 173 \cdot q_{i-1} + 219 \cdot q_{i-2} \bmod 317$ and $r_i = 151^{q_i} \bmod 317$. The period of this sequence equals $T = 317^2 - 1$.

3 Basic concepts

TRNG consists of a loose bunch of classes. These classes can be divided into to 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 [4] 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 as introduced in [4]. But in TRNG the notion of a random number engine is extended to a "parallel random number engine". To fulfil the requirements of a parallel random number engine, a class has to fulfil all the requirements of a random number engine and in addition some further requirements that make them applicable for parallel Monte Carlo simulations. Random number engine concept and parallel random number engine concept will be discussed in detail in section 3.1.

A random number engine is not very useful by itself. Therefore 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 with some specific distribution. The general concept of an random number distribution is discussed in section 3.2.

3.1 Random number engines

To be a random number engine, a class has to fulfil a set of requirements that we will summarize as follows. For details see [4]. 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
`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)

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. An engine's state 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 engine's state x_i is advanced to its *successor 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(s, p)`	`void`	pre: `s` and `p` are of type `unsigned int` with `s` $<$ `p`. If `s` $\geq$ `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 `p` $-$ 1. The complexity of `operator()` will not change.	polynomial in size of state, `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^{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`

x_{i+1}, and

- the *generation algorithm* GA by which an engine's state is mapped to a value of type `result_type`.

Furthermore, a random number engine shall fulfil 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 like `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 [4]. 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. Note that it is demanded that leapfrog is implemented in such a way, that the complexity of `operator()` will not depend on, in how many sub-streams a stream has been split. That means a valid implementation will not implement leapfrog by calculating all random numbers of a stream and then throw away bunches of numbers to derive the random numbers of a sub-stream. This requirement restricts number of pseudo-random number generator algorithms that are proper for parallel random number engines.

3.2 Random number distributions

To model the concept of a random number distribution a class has to fulfil a set of requirements that we will summarize as follows. Refer to [4] 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 a distribution's parameters `p` taken as a whole.

Furthermore a random number distribution shall fulfil 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

Tina's Random Number Generator Library provides to the user several 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 all implemented random number engines are documented. Each subsection describes the public interface of one random number engine. The part of the public interface, that is mandatory for a random number engine, will not be discussed in detail. Read section 3.1 instead.

4.1.1 LCG64

The class `trng::lcg64` implements a linear congruential generator, for which the transition algorithm reads [24, 17]

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

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 $a \bmod 4 = 1$ [17]. 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 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 high order bits. The class `trng::lcg64` should be avoided whenever the randomness of lower bits have a significant impact to the simulation. In [18] L'Ecuyer warns about multiplicative linear congruential generators (MLCG) with $r_{i+1} = a \cdot r_i \bmod m$:

> 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 [8].

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()() const;
    static const result_type min;
    static const 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 [20].

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

```
    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);
```

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(result_type);
```

Parallel random number engine requirements:

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

Furthermore the class `trng::lcg64` provides a function that returns a string with the name of the random number engine and an operator `operator()` that makes `trng::lcg64` applicable to `std::random_shuffle`. In terms of the C++ Standard Template Library `trng::lcg64` models a "random number generator".

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

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 &);
}
```

4.1.2 MRG2

The class `trng::mrg2` implements a multiple recursive generator (also known as linear feedback shift register generator) over the prime field $\mathbb{F}_{2^{31}-1}$ with two feedback taps, for which the transition algorithm reads

$$r_i = a_1 \cdot r_{i-1} + a_2 \cdot r_{i-2} \bmod (2^{31} - 1)\,.$$

The prime modulus of $m = 2^{31} - 1$ was chosen for performance reasons, see chapter 7, and is a fix parameter of the random number engine `trng::mrg2`. The state of this generator at time i is given by (r_{i-1}, r_{i-2}). The maximal period of the generator is $m^2 - 1 \approx 2^{62} \approx 4.61 \cdot 10^{18}$.

In general, the modulus m of a linear feedback shift register generator with n feedback taps

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

has to be prime. The maximal period $m^n - 1$ is attained if and only if [17] the characteristic polynomial

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

is *primitive* modulo m. A polynomial of degree n is primitive modulo m if and only if it is irreducible (i. e., cannot be factorized over $\mathbb{F}_m$) and its roots are primitive elements of the extension field $\mathbb{F}_{m^n}$. Primitive polynomials modulo m can be very efficiently be constructed if the factorization of $m^n - 1$ is known [14], and most computer algebra systems offer a procedure for this. All these facts about linear feedback shift registers do also apply to the classes `trng::mrg3`, `trng::mrg4`, `trng::mrg5`, `trng::yarn2`, `trng::yarn3`, `trng::yarn4` and `trng::yarn5` that will be introduced in the forthcoming sections and have to be considered if one wants to chose other parameter sets for these generators than provided by TRNG.

The class `trng::mrg2` is declared in the header file `trng/mrg2.hpp` and its public interface is given as follows:

```
namespace trng {

  class mrg2 {
  public:
```

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

```
    typedef long result_type;
    result_type operator()() const;
    static const result_type min;
    static const 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 two parameter sets for a_1 and a_2, which are taken from [19] which are chosen to give a generator with good statistical properties.

$$a_1 = 1\,498\,809\,829\,, \quad a_2 = 1\,160\,990\,996$$

```
    static const parameter_type LEcuyer1;
```

$$a_1 = 46\,325\,, \quad a_2 = 1\,084\,587$$

```
    static const parameter_type LEcuyer2;
```

An instance of class `trng::mrg2` can be instantiated by various constructors as specified for a random number engine. Additionally a non-default parameter set may be chosen.

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

Class `trng::mrg2` provides all necessary seeding functions (see Table 3.1) and an additional function that sets (r_{i-1}, r_{i-2}).

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

Parallel random number engine requirements:

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

Class `trng::mrg2` provides also a function that returns a string with the name of the random number engine and an operator `operator()` that makes `trng::mrg2` applicable to `std::random_shuffle`. In terms of the C++ Standard Template Library `trng::mrg2` models a "random number generator".

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

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

```
  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 &);
}
```

4.1.3 MRG3

The class `trng::mrg3` implements a multiple recursive generator with three feedback taps over the prime field $\mathbb{F}_{2^{31}-1}$, for which the transition algorithm reads

$$r_i = a_1 \cdot r_{i-1} + a_2 \cdot r_{i-2} + a_3 \cdot r_{i-3} \bmod (2^{31} - 1)\,.$$

The prime modulus of $m = 2^{31} - 1$ was chosen for performance reasons and is a fix parameter of the PRNG. The state of this generator at time i is given by $(r_{i-1}, r_{i-2}, r_{i-3})$. The maximal period of the generator is $m^3 - 1 \approx 2^{93} \approx 9.90 \cdot 10^{27}$.

The class `trng::mrg3` is declared in the header file `trng/mrg3.hpp` and its public interface is given as follows:

```
namespace trng {

  class mrg3 {
  public:
```

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

```
    typedef long result_type;
    result_type operator()() const;
    static const result_type min;
    static const 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 three parameter sets for a_1, a_2 and a_3, which are taken from [19] and chosen to give good statistical properties.

$$a_1 = 2\,021\,422\,057\,, \quad a_2 = 1\,826\,992\,351\,, \quad a_3 = 1\,977\,753\,457$$

```
    static const parameter_type LEcuyer1;
```

$$a_1 = 1\,476\,728\,729\,, \quad a_2 = 0\,, \quad a_3 = 1\,155\,643\,113$$

```
    static const parameter_type LEcuyer2;
```

$$a_1 = 65\,338\,, \quad a_2 = 0\,, \quad a_3 = 64\,636$$

```
    static const parameter_type LEcuyer3;
```

An instance of class `trng::mrg3` can be instantiated by various constructors as specified for a random number engine. Additionally a non-default parameter set may be chosen.

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

Class `trng::mrg3` provides all necessary seeding functions (see Table 3.1) and an additional function that sets $(r_{i-1}, r_{i-2}, r_{i-3})$.

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

Parallel random number engine requirements:

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

Class `trng::mrg3` provides also a function that returns a string with the name of the random number engine and an operator `operator()` that makes `trng::mrg3` applicable to `std::random_shuffle`. In terms of the C++ Standard Template Library `trng::mrg3` models a "random number generator".

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

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

```
  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 &);
}
```

4.1.4 MRG3S

The class `trng::mrg3s` implements a multiple recursive generator with three feedback taps over the prime field $\mathbb{F}_{2^{31}-21069}$, for which the transition algorithm reads

$$r_i = a_1 \cdot r_{i-1} + a_2 \cdot r_{i-2} + a_3 \cdot r_{i-3} \bmod (2^{31} - 21069)\,.$$

The prime modulus of $m = 2^{31} - 21069$ is a Sophie-Germain Prime and the period of this generator is a product of only three prime factors. The modulus was also chosen for performance reasons, see chapter 7, and is a fix parameter of the PRNG. The state of this generator at time i is given by $(r_{i-1}, r_{i-2}, r_{i-3})$. The maximal period of the generator is $m^3 - 1 \approx 2^{93} \approx 9.90 \cdot 10^{27}$.

The class `trng::mrg3s` is declared in the header file `trng/mrg3s.hpp` and its public interface is given as follows:

```
namespace trng {

  class mrg3s {
  public:
```

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

```
    typedef long result_type;
    result_type operator()() const;
    static const result_type min;
    static const 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 only one parameter set for a_1, a_2 and a_3.

$$a_1 = 1\,287\,767\,370\,, \quad a_2 = 1\,045\,931\,779\,, \quad a_3 = 58\,150\,106$$

```
    static const parameter_type trng1;
```

An instance of class `trng::mrg3s` can be instantiated by various constructors as specified for a random number engine. Additionally a non-default parameter set may be chosen.

```
    explicit mrg3s(parameter_type=trng1);
    explicit mrg3s(unsigned long, parameter_type=trng1);
    template<typename gen>
    explicit mrg3s(gen &, parameter_type P=trng1);
```

Class `trng::mrg3s` provides all necessary seeding functions (see Table 3.1) and an additional function that sets $(r_{i-1}, r_{i-2}, r_{i-3})$.

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

Parallel random number engine requirements:

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

Class `trng::mrg3s` provides also a function that returns a string with the name of the random number engine and an operator `operator()` that makes `trng::mrg3s` applicable to `std::random_shuffle`. In terms of the C++ Standard Template Library `trng::mrg3s` models a "random number generator".

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

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

```
  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 &);
}
```

4.1.5 MRG4

The class `trng::mrg4` implements a multiple recursive generator with four feedback taps over the prime field $\mathbb{F}_{2^{31}-1}$, for which the transition algorithm reads

$$r_i = a_1 \cdot r_{i-1} + a_2 \cdot r_{i-2} + a_3 \cdot r_{i-3} + a_4 \cdot r_{i-4} \bmod (2^{31} - 1)\,.$$

The prime modulus of $m = 2^{31} - 1$ was chosen for performance reasons and is a fix parameter of the PRNG. The state of this generator at time i is given by $(r_{i-1}, r_{i-2}, r_{i-3}, r_{i-4})$. The maximal period of the generator is $m^4 - 1 \approx 2^{124} \approx 2.13 \cdot 10^{37}$.

The class `trng::mrg4` is declared in the header file `trng/mrg4.hpp` and its public interface is given as follows:

```
namespace trng {

  class mrg4 {
  public:
```

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

```
    typedef long result_type;
    result_type operator()() const;
    static const result_type min;
    static const 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 two parameter sets for a_1, a_2, a_3 and a_4, which are taken from [19].

$$a_1 = 2\,001\,982\,722\,, \quad a_2 = 1\,412\,284\,257\,, \quad a_3 = 1\,155\,380\,217\,, \quad a_4 = 1\,668\,339\,922$$

```
    static const parameter_type LEcuyer1;
```

$$a_1 = 64\,886\,, \quad a_2 = 0\,, \quad a_3 = 0\,, \quad a_4 = 64\,322$$

```
    static const parameter_type LEcuyer2;
```

An instance of class `trng::mrg4` can be instantiated by various constructors as specified for a random number engine. Additionally a non-default parameter set may be chosen.

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

Class `trng::mrg4` provides all necessary seeding functions (see Table 3.1) and an additional function that sets $(r_{i-1}, r_{i-2}, r_{i-3}, r_{i-4})$.

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

Parallel random number engine requirements:

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

Class `trng::mrg4` provides also a function that returns a string with the name of the random number engine and an operator `operator()` that makes `trng::mrg4` applicable to `std::random_shuffle`. In terms of the C++ Standard Template Library `trng::mrg4` models a "random number generator".

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

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

```
  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 &);
}
```

4.1.6 MRG5

The class `trng::mrg5` implements a multiple recursive generator with five feedback taps over the prime field $\mathbb{F}_{2^{31}-1}$, for which the transition algorithm reads

$$r_i = a_1 \cdot r_{i-1} + a_2 \cdot r_{i-2} + a_3 \cdot r_{i-3} + a_4 \cdot r_{i-4} + a_5 \cdot r_{i-5} \bmod (2^{31} - 1)\,.$$

The prime modulus of $m = 2^{31} - 1$ was chosen for performance reasons and is a fix parameter of the PRNG. The state of this generator at time i is given by $(r_{i-1}, r_{i-2}, r_{i-3}, r_{i-4}, r_{i-5})$. The maximal period of the generator is $m^5 - 1 \approx 2^{155} \approx 4.57 \cdot 10^{46}$.

The class `trng::mrg5` is declared in the header file `trng/mrg5.hpp` and its public interface is given as follows:

```
namespace trng {

  class mrg5 {
  public:
```

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

```
    typedef long result_type;
    result_type operator()() const;
    static const result_type min;
    static const 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 two parameter sets for a_1, a_2, a_3, a_4 and a_5, which are taken from [19].

$$a_1 = 107\,374\,182\,, \quad a_2 = 0\,, \quad a_3 = 0\,, \quad a_4 = 0\,, \quad a_5 = 104\,480$$

```
    static const parameter_type LEcuyer1;
```

An instance of class `trng::mrg5` can be instantiated by various constructors as specified for a random number engine. Additionally a non-default parameter set may be chosen.

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

Class `trng::mrg5` provides all necessary seeding functions (see Table 3.1) and an additional function that sets $(r_{i-1}, r_{i-2}, r_{i-3}, r_{i-4}, r_{i-5})$.

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

Parallel random number engine requirements:

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

Class `trng::mrg5` provides also a function that returns a string with the name of the random number engine and an operator `operator()` that makes `trng::mrg5` applicable to `std::random_shuffle`. In terms of the C++ Standard Template Library `trng::mrg5` models a "random number generator".

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

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

```
  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 &);
}
```

4.1.7 MRG5S

The class `trng::mrg5s` implements a multiple recursive generator with five feedback taps over the prime field $\mathbb{F}_{2^{31}-22641}$, for which the transition algorithm reads

$$r_i = a_1 \cdot r_{i-1} + a_2 \cdot r_{i-2} + a_3 \cdot r_{i-3} + a_4 \cdot r_{i-4} + a_5 \cdot r_{i-5} \bmod (2^{31} - 22\,641)\,.$$

The prime modulus of $m = 2^{31} - 22\,641$ is a Sophie-Germain Prime and the period of this generator is a product of only three prime factors. The prime modulus was also chosen for performance reasons and is a fix parameter of the PRNG. The state of this generator at time i is given by $(r_{i-1}, r_{i-2}, r_{i-3}, r_{i-4}, r_{i-5})$. The maximal period of the generator is $m^5 - 1 \approx 2^{155} \approx 4.57 \cdot 10^{46}$.

The class `trng::mrg5s` is declared in the header file `trng/mrg5s.hpp` and its public interface is given as follows:

```
namespace trng {

  class mrg5s {
  public:
```

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

```
    typedef long result_type;
    result_type operator()() const;
    static const result_type min;
    static const 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 only one parameter set for a_1, a_2, a_3, a_4 and a_5.

$$a_1 = 2\,068\,619\,238\,, \quad a_2 = 2\,138\,332\,912\,, \quad a_3 = 671\,754\,166\,,$$
$$a_4 = 1\,442\,240\,992\,, \quad a_5 = 1\,526\,958\,817$$

```
    static const parameter_type trng1;
```

An instance of class `trng::mrg5s` can be instantiated by various constructors as specified for a random number engine. Additionally a non-default parameter set may be chosen.

```
    explicit mrg5s(parameter_type=trng1);
    explicit mrg5s(unsigned long, parameter_type=trng1);
    template<typename gen>
    explicit mrg5s(gen &, parameter_type P=trng1);
```

Class `trng::mrg5s` provides all necessary seeding functions (see Table 3.1) and an additional function that sets $(r_{i-1}, r_{i-2}, r_{i-3}, r_{i-4}, r_{i-5})$.

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

Parallel random number engine requirements:

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

Class `trng::mrg5s` provides also a function that returns a string with the name of the random number engine and an operator `operator()` that makes `trng::mrg5s` applicable to `std::random_shuffle`. In terms of the C++ Standard Template Library `trng::mrg5s` models a "random number generator".

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

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

```
  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 &);
}
```

4.1.8 YARN2

The class `trng::yarn2` implements a so-called YARN generator (yet another random number generator), that is based on a multiple recursive generator with two feedback taps, for which the transition algorithm reads

$$r_i = a_1 \cdot r_{i-1} + a_2 \cdot r_{i-2} \bmod (2^{31} - 1)\,.$$

This generator has a prime modulus of $m = 2^{31} - 1$, which was chosen for performance reasons and is a fix parameter of the PRNG. The state of this generator at time i is given by (r_{i-1}, r_{i-2}). The maximal period of the generator is $m^2 - 1 \approx 2^{62} \approx 4.61 \cdot 10^{18}$.

While a pure multiple recursive generator returns 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 $m = 2^{31} - 1$ this mapping reads

$$q_i = \begin{cases} b^{r_i} \bmod (2^{31} - 1) & \text{if } r_i > 0 \\ 0 & \text{if } r_i = 0 \end{cases},$$

where b is a generating element of the multiplicative group modulo $2^{31} - 1$. 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.

The class `trng::yarn2` is declared in the header file `trng/yarn2.hpp` and its public interface is given as follows:

```
namespace trng {

  class yarn2 {
  public:
```

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

```
    typedef long result_type;
    result_type operator()() const;
```

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

```
    static const result_type min;
    static const 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 two parameter sets for a_1 and a_2, which are taken from [19], and b.

$$a_1 = 1\,498\,809\,829\,, \quad a_2 = 1\,160\,990\,996\,, \quad b = 123\,567\,893$$

```
    static const parameter_type LEcuyer1;
```

$$a_1 = 46\,325\,, \quad a_2 = 1\,084\,587\,, \quad b = 123\,567\,893$$

```
    static const parameter_type LEcuyer2;
```

An instance of class `trng::yarn2` can be instantiated by various constructors as specified for a random number engine. Additionally a non-default parameter set may be chosen.

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

Class `trng::yarn2` provides all necessary seeding functions (see Table 3.1) and an additional function that sets (r_{i-1}, r_{i-2}).

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

Parallel random number engine requirements:

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

Class `trng::yarn2` provides also a function that returns a string with the name of the random number engine and an operator `operator()` that makes `trng::yarn2` applicable to `std::random_shuffle`. In terms of the C++ Standard Template Library `trng::yarn2` models a "random number generator".

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

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

```
  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 &);
}
```

4.1.9 YARN3

The class `trng::yarn3` implements a so-called YARN generator (yet another random number generator), that is based on a multiple recursive generator with three feedback taps, for which the transition algorithm reads

$$r_i = a_1 \cdot r_{i-1} + a_2 \cdot r_{i-2} + a_3 \cdot r_{i-3} \bmod (2^{31} - 1)\,.$$

This generator has a prime modulus of $m = 2^{31} - 1$, which was chosen for performance reasons and is a fix parameter of the PRNG. The state of this generator at time i is given by $(r_{i-1}, r_{i-2}, r_{i-3})$. The maximal period of the generator is $m^2 - 1 \approx 2^{62} \approx 4.61 \cdot 10^{18}$.

While a pure multiple recursive generator returns the r_i as pseudo-random numbers directly, a YARN generator "shuffles" the output of the underlying multiple recursive generator by a bijective mapping, see section 4.1.8 for details.

The class `trng::yarn3` is declared in the header file `trng/yarn3.hpp` and its public interface is given as follows:

```
namespace trng {

  class yarn3 {
  public:
```

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

```
    typedef long result_type;
    result_type operator()() const;
    static const result_type min;
    static const 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 three parameter sets for a_1, a_2 and a_3, which are taken from [19], and b.

$$a_1 = 2\,021\,422\,057\,, \quad a_2 = 1\,826\,992\,351\,, \quad a_3 = 1\,977\,753\,457\,, \quad b = 123\,567\,893$$

```
    static const parameter_type LEcuyer1;
```

$$a_1 = 1\,476\,728\,729\,, \quad a_2 = 0\,, \quad a_3 = 1\,155\,643\,113\,, \quad b = 123\,567\,893$$

```
    static const parameter_type LEcuyer2;
```

$$a_1 = 65\,338\,, \quad a_2 = 0\,, \quad a_3 = 64\,636\,, \quad b = 123\,567\,893$$

```
    static const parameter_type LEcuyer3;
```

An instance of class `trng::yarn3` can be instantiated by various constructors as specified for a random number engine. Additionally a non-default parameter set may be chosen.

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

Class `trng::yarn3` provides all necessary seeding functions (see Table 3.1) and an additional function that sets $(r_{i-1}, r_{i-2}, r_{i-3})$.

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

Parallel random number engine requirements:

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

Class `trng::yarn3` provides also a function that returns a string with the name of the random number engine and an operator `operator()` that makes `trng::yarn3` applicable to `std::random_shuffle`. In terms of the C++ Standard Template Library `trng::yarn3` models a "random number generator".

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

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

```
  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 &);
}
```

4.1.10 YARN3S

The class `trng::yarn3s` implements a so-called YARN generator (yet another random number generator), that is based on a multiple recursive generator with three feedback taps, for which the transition algorithm reads

$$r_i = a_1 \cdot r_{i-1} + a_2 \cdot r_{i-2} + a_3 \cdot r_{i-3} \bmod (2^{31} - 21\,069)\,.$$

The prime modulus of $m = 2^{31} - 21069$ is a Sophie-Germain Prime and the period of this generator is a product of only three prime factors. The prime modulus was also chosen for performance reasons and is a fix parameter of the PRNG. The state of this generator at time i is given by $(r_{i-1}, r_{i-2}, r_{i-3})$. The maximal period of the generator is $m^2 - 1 \approx 2^{62} \approx 4.61 \cdot 10^{18}$.

While a pure multiple recursive generator returns the r_i as pseudo-random numbers directly, a YARN generator "shuffles" the output of the underlying multiple recursive generator by a bijective mapping, see section 4.1.8 for details.

The class `trng::yarn3s` is declared in the header file `trng/yarn3s.hpp` and its public interface is given as follows:

```
namespace trng {

  class yarn3s {
  public:
```

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

```
    typedef long result_type;
    result_type operator()() const;
    static const result_type min;
    static const 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 only one three parameter set for a_1, a_2 and a_3 and b.

$$a_1 = 1\,287\,767\,370\,, \quad a_2 = 1\,045\,931\,779\,, \quad a_3 = 58\,150\,106\,, \quad b = 1\,616\,076\,847$$

```
    static const parameter_type trng1;
```

An instance of class `trng::yarn3s` can be instantiated by various constructors as specified for a random number engine. Additionally a non-default parameter set may be chosen.

```
    explicit yarn3s(parameter_type=trng1);
    explicit yarn3s(unsigned long, parameter_type=trng1);
    template<typename gen>
    explicit yarn3s(gen &, parameter_type P=trng1);
```

Class `trng::yarn3s` provides all necessary seeding functions (see Table 3.1) and an additional function that sets $(r_{i-1}, r_{i-2}, r_{i-3})$.

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

Parallel random number engine requirements:

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

Class `trng::yarn3s` provides also a function that returns a string with the name of the random number engine and an operator `operator()` that makes `trng::yarn3s` applicable to `std::random_shuffle`. In terms of the C++ Standard Template Library `trng::yarn3s` models a "random number generator".

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

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

```
  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 &);
}
```

4.1.11 YARN4

The class `trng::yarn4` implements a so-called YARN generator (yet another random number generator), that is based on a multiple recursive generator with four feedback taps over the prime field $\mathbb{F}_{2^{31}-1}$, for which the transition algorithm reads

$$r_i = a_1 \cdot r_{i-1} + a_2 \cdot r_{i-2} + a_3 \cdot r_{i-3} + a_4 \cdot r_{i-4} \bmod (2^{31} - 1)\,.$$

The prime modulus of $m = 2^{31} - 1$, which was chosen for performance reasons and is a fix parameter of the PRNG. The state of this generator at time i is given by $(r_{i-1}, r_{i-2}, r_{i-3}, r_{i-4})$. The maximal period of the generator is $m^4 - 1 \approx 2^{124} \approx 2.13 \cdot 10^{37}$.

While a pure multiple recursive generator returns the r_i as pseudo-random numbers directly, a YARN generator "shuffles" the output of the underlying multiple recursive generator by a bijective mapping, see section 4.1.8 for details.

The class `trng::yarn4` is declared in the header file `trng/yarn4.hpp` and its public interface is given as follows:

```
namespace trng {

  class yarn4 {
  public:
```

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

```
    typedef long result_type;
    result_type operator()() const;
    static const result_type min;
    static const 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 two parameter sets for a_1, a_2, a_3 and a_4, which are taken from [19], and b.

$$a_1 = 2\,001\,982\,722\,, \quad a_2 = 1\,412\,284\,257\,, \quad a_3 = 1\,155\,380\,217\,, \quad a_4 = 1\,668\,339\,922\,,$$
$$b = 123\,567\,893$$

```
    static const parameter_type LEcuyer1;
```

$$a_1 = 64\,886\,, \quad a_2 = 0\,, \quad a_3 = 0\,, \quad a_4 = 64\,322\,, \quad b = 123\,567\,893$$

```
    static const parameter_type LEcuyer2;
```

An instance of class `trng::yarn4` can be instantiated by various constructors as specified for a random number engine. Additionally a non-default parameter set may be chosen.

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

Class `trng::yarn4` provides all necessary seeding functions (see Table 3.1) and an additional function that sets $(r_{i-1}, r_{i-2}, r_{i-3}, r_{i-4})$.

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

Parallel random number engine requirements:

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

Class `trng::yarn4` provides also a function that returns a string with the name of the random number engine and an operator `operator()` that makes `trng::yarn4` applicable to `std::random_shuffle`. In terms of the C++ Standard Template Library `trng::yarn4` models a "random number generator".

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

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

```
  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 &);
}
```

4.1.12 YARN5

The class `trng::yarn5` implements a so-called YARN generator (yet another random number generator), that is based on a multiple recursive generator with five feedback taps over the prime field $\mathbb{F}_{2^{31}-1}$, for which the transition algorithm reads

$$r_i = a_1 \cdot r_{i-1} + a_2 \cdot r_{i-2} + a_3 \cdot r_{i-3} + a_4 \cdot r_{i-4} + a_5 \cdot r_{i-5} \bmod (2^{31} - 1)\,.$$

The prime modulus of $m = 2^{31} - 1$ was chosen for performance reasons and is a fix parameter of the PRNG. The state of this generator at time i is given by $(r_{i-1}, r_{i-2}, r_{i-3}, r_{i-4}, r_{i-5})$. The maximal period of the generator is $m^4 - 1 \approx 2^{155} \approx 4.57 \cdot 10^{46}$.

While a pure multiple recursive generator returns the r_i as pseudo-random numbers directly, a YARN generator "shuffles" the output of the underlying multiple recursive generator by a bijective mapping, see section 4.1.8 for details.

The class `trng::yarn5` is declared in the header file `trng/yarn5.hpp` and its public interface is given as follows:

```
namespace trng {

  class yarn5 {
  public:
```

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

```
    typedef long result_type;
    result_type operator()() const;
    static const result_type min;
    static const 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 two parameter sets for a_1, a_2, a_3, a_4 and a_5, which are taken from [19], and b.

$$a_1 = 107\,374\,182\,, \quad a_2 = 0\,, \quad a_3 = 0\,, \quad a_4 = 0\,, \quad a_5 = 104\,480\,,$$
$$b = 123\,567\,893$$

```
    static const parameter_type LEcuyer1;
```

An instance of class `trng::yarn5` can be instantiated by various constructors as specified for a random number engine. Additionally a non-default parameter set may be chosen.

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

Class `trng::yarn5` provides all necessary seeding functions (see Table 3.1) and an additional function that sets $(r_{i-1}, r_{i-2}, r_{i-3}, r_{i-4}, r_{i-5})$.

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

Parallel random number engine requirements:

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

Class `trng::yarn5` provides also a function that returns a string with the name of the random number engine and an operator `operator()` that makes `trng::yarn5` applicable to `std::random_shuffle`. In terms of the C++ Standard Template Library `trng::yarn5` models a "random number generator".

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

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

```
  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 &);
}
```

4.1.13 YARN5S

The class `trng::yarn5s` implements a so-called YARN generator (yet another random number generator), that is based on a multiple recursive generator with five feedback taps over the prime field $\mathbb{F}_{2^{31}-22\,641}$, for which the transition algorithm reads

$$r_i = a_1 \cdot r_{i-1} + a_2 \cdot r_{i-2} + a_3 \cdot r_{i-3} + a_4 \cdot r_{i-4} + a_5 \cdot r_{i-5} \bmod (2^{31} - 1)\,.$$

The prime modulus of $m = 2^{31} - 21069$ is a Sophie-Germain Prime and the period of this generator is a product of only three prime factors. The modulus was also chosen for performance reasons and is a fix parameter of the PRNG. The state of this generator at time i is given by $(r_{i-1}, r_{i-2}, r_{i-3}, r_{i-4}, r_{i-5})$. The maximal period of the generator is $m^4 - 1 \approx 2^{155} \approx 4.57 \cdot 10^{46}$.

While a pure multiple recursive generator returns the r_i as pseudo-random numbers directly, a YARN generator "shuffles" the output of the underlying multiple recursive generator by a bijective mapping, see section 4.1.8 for details.

The class `trng::yarn5s` is declared in the header file `trng/yarn5s.hpp` and its public interface is given as follows:

```
namespace trng {

  class yarn5s {
  public:
```

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

```
    typedef long result_type;
    result_type operator()() const;
    static const result_type min;
    static const 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 only one parameter set for a_1, a_2, a_3, a_4 and a_5 and b.

$$a_1 = 2\,068\,619\,238\,, \quad a_2 = 2\,138\,332\,912\,, \quad a_3 = 671\,754\,166\,, \quad a_4 = 1\,442\,240\,992\,, \quad a_5 = 1\,526\,958\,817\,,$$
$$b = 889\,744\,251$$

```
    static const parameter_type trng1;
```

An instance of class `trng::yarn5s` can be instantiated by various constructors as specified for a random number engine. Additionally a non-default parameter set may be chosen.

```
    explicit yarn5s(parameter_type=trng1);
    explicit yarn5s(unsigned long, parameter_type=trng1);
    template<typename gen>
    explicit yarn5s(gen &, parameter_type P=trng1);
```

Class `trng::yarn5s` provides all necessary seeding functions (see Table 3.1) and an additional function that sets $(r_{i-1}, r_{i-2}, r_{i-3}, r_{i-4}, r_{i-5})$.

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

Parallel random number engine requirements:

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

Class `trng::yarn5s` provides also a function that returns a string with the name of the random number engine and an operator `operator()` that makes `trng::yarn5s` applicable to `std::random_shuffle`. In terms of the C++ Standard Template Library `trng::yarn5s` models a "random number generator".

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

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

```
  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 &);
}
```

4.2 Random number distributions

In this section all implemented random number distributions are documented. Each subsection describes 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.

Additionally to the requirements in section 3.2 each random number distribution class provides a member function that calculates the probability distribution function, the cumulative

distribution function and in the case of continuous distributions also the inverse cumulative distribution function. These member functions have the signatures

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

and for discrete random variables

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

The concept of a random number distribution requires two functions, that take a random number engine as argument and generate a random variable with some specific distribution by calling `operator()` of the random number engine. Note, it is not specified how often `operator()` is called. This allows an implementor of a random number distribution to choose between various algorithms [17] 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 one time. Because of this 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 processors.

4.2.1 Uniform distributions

TRNG provides by the classes `uniform01_dist`, `uniform_dist` and `uniform_int_dist` three different kinds of uniform distributions.

Class `uniform01_dist` provides random numbers with distribution function

$$p(x) = \begin{cases} 1 & \text{if } 0 \leq x < 1 \\ 0 & \text{otherwise} \end{cases} .$$

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

```
namespace trng {

  class uniform01_dist {
  public:
    typedef double result_type;
    class param_type;
    explicit uniform01_dist();
    explicit uniform01_dist(const param_type &);
    void reset();
    template<typename R>
    double operator()(R &);
    template<typename R>
    double operator()(R &, const param_type &);
    double min() const;
    double max() const;
    param_type param() const;
```

```
    void param(const param_type &);
    double pdf(double x) const;
    double cdf(double x) const;
    double icdf(double x) const;
  };

  bool operator==(const uniform01_dist::param_type &, const uniform01_dist::param_type &);
  bool operator!=(const uniform01_dist::param_type &, const uniform01_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 uniform01_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> &, uniform01_dist::param_type &);

  bool operator==(const uniform01_dist &, const uniform01_dist &);
  bool operator!=(const uniform01_dist &, const uniform01_dist &);

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

Class `uniform_dist` is a generalization of class `uniform01_dist` and it provides random numbers with distribution function

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

Valid parameters for this distribution are $a, b \in \mathbb{R}$ with $a < b$. If pseudo-random numbers uniformly distributed in $[0,1)$ are desired, class `uniform01_dist` might be faster than `uniform_dist` with parameters $a = 0$ and $b = 1$.

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

```
namespace trng {

  class uniform_dist {
  public:
    typedef double result_type;
    class param_type {
    public:
      double a() const;
      void a(double);
      double b() const;
      void b(double);
      explicit param_type(double a, double b);
    };
    explicit uniform_dist(double a, double b);
    explicit uniform_dist(const param_type &);
    void reset();
    template<typename R>
```

```
    double operator()(R &);
    template<typename R>
    double operator()(R &, const param_type &)
    double min() const;
    double max() const;
    param_type param() const;
    void param(const param_type &);
    double a() const;
    void a(double);
    double b();
    void b(double);
    double pdf(double x) const;
    double cdf(double x) const;
    double icdf(double x) const;
  };

  bool operator==(const uniform_dist::param_type &, const uniform_dist::param_type &);
  bool operator!=(const uniform_dist::param_type &, const uniform_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_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_dist::param_type &);

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

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

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

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

The class `uniform_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:
      int a() const;
      void a(int);
```

```
    int b() const;
    void b(int);
    explicit param_type(int a, int b);
  };
  explicit uniform_int_dist(int a, int b);
  explicit uniform_int_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;
  param_type param() const;
  void param(const param_type &);
  int a() const;
  void a(int);
  int b() const;
  void b(int);
  double pdf(int x) const;
  double cdf(int 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

$$p(x|\mu) = \begin{cases} \frac{1}{\mu} e^{-x/\mu} & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases} \qquad \text{for } x \in \mathbb{R}.$$

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 {

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

  bool operator==(const exponential_dist::param_type &, const exponential_dist::param_type &);
  bool operator!=(const exponential_dist::param_type &, const exponential_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 exponential_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> &, exponential_dist::param_type &);

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

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

4.2.3 Normal distribution

Class `normal_dist` provides 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 {

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

  bool operator==(const normal_dist::param_type &, const normal_dist::param_type &);
  bool operator!=(const normal_dist::param_type &, const normal_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 normal_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> &, normal_dist::param_type &);

  bool operator==(const normal_dist &, const normal_dist &);
```

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

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

4.2.4 Cauchy distribution

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

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

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 {

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

```
  };

  bool operator==(const cauchy_dist::param_type &, const cauchy_dist::param_type &);
  bool operator!=(const cauchy_dist::param_type &, const cauchy_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 cauchy_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> &, cauchy_dist::param_type &);

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

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

4.2.5 Logistic distribution

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

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

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 {

  class logistic_dist {
  public:
    typedef double result_type;
    class param_type {
    public:
      double theta() const;
      void theta(double);
      double eta() const;
      void eta(double);
      explicit param_type(double theta, double eta);
    };
    explicit logistic_dist(double theta, double eta);
    explicit logistic_dist(const param_type &);
    void reset();
    template<typename R>
    double operator()(R &);
    template<typename R>
    double operator()(R &, const param_type &);
```

```
    double min() const;
    double max() const;
    param_type param() const;
    void param(const param_type &);
    double theta() const;
    void theta(double);
    double eta() const;
    void eta(double);
    double pdf(double) const;
    double cdf(double) const;
    double icdf(double) const;
  };

  bool operator==(const logistic_dist::param_type &, const logistic_dist::param_type &);
  bool operator!=(const logistic_dist::param_type &, const logistic_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 logistic_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> &, logistic_dist::param_type &);

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

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

4.2.6 Lognormal distribution

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

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

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 {

  class lognormal_dist {
  public:
    typedef double result_type;
    class param_type {
    public:
```

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

  bool operator==(const lognormal_dist::param_type &, const lognormal_dist::param_type &);
  bool operator!=(const lognormal_dist::param_type &, const lognormal_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 lognormal_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> &, lognormal_dist::param_type &);

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

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

4.2.7 Pareto distribution

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

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

Valid parameters for this distribution are $\gamma, \theta \in \mathbb{R}$ with $\gamma > 0$ and $\theta > 0$. Actually in mathematics literature one can find two different kinds of probability distributions, that are referred as Pareto distribution. Section 4.2.8 introduces another probability distribution that is also sometimes called 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 {

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

  bool operator==(const pareto_dist::param_type &, const pareto_dist::param_type &);
  bool operator!=(const pareto_dist::param_type &, const pareto_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 pareto_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> &, pareto_dist::param_type &);

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

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

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

4.2.8 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

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

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

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

```
namespace trng {

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

  bool operator==(const powerlaw_dist::param_type &, const powerlaw_dist::param_type &);
  bool operator!=(const powerlaw_dist::param_type &, const powerlaw_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 powerlaw_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> &, powerlaw_dist::param_type &);

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

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

4.2.9 Tent distribution

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

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

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 {

  class tent_dist {
  public:
    typedef double result_type;
    class param_type {
    public:
      double m() const;
      void m(double);
      double d() const;
      void d(double);
      explicit param_type(double m, double d);
    };
    explicit tent_dist(double m, double d);
    explicit tent_dist(const param_type &);
    void reset();
    template<typename R>
    double operator()(R &);
    template<typename R>
    double operator()(R &, const param_type &);
    double min() const;
    double max() const;
```

```
    param_type param() const;
    void param(const param_type &);
    double m() const;
    void m(double);
    double d() const;
    void d(double);
    double pdf(double) const;
    double cdf(double) const;
    double icdf(double) const;
  };

  bool operator==(const tent_dist::param_type &, const tent_dist::param_type &);
  bool operator!=(const tent_dist::param_type &, const tent_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 tent_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> &, tent_dist::param_type &);

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

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

4.2.10 Weibull distribution

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

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

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

  class weibull_dist {
  public:
    typedef double result_type;
    class param_type {
    public:
```

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

  bool operator==(const weibull_dist::param_type &, const weibull_dist::param_type &);
  bool operator!=(const weibull_dist::param_type &, const weibull_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 weibull_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> &, weibull_dist::param_type &);

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

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

4.2.11 Extreme value distribution

Class `extreme_value_dist` provides random numbers with extreme value distribution with parameters θ and η. The probability distribution function reads

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

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 {

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

  bool operator==(const extreme_value_dist::param_type &,
                  const extreme_value_dist::param_type &);
  bool operator!=(const extreme_value_dist::param_type &,
                  const extreme_value_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 extreme_value_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> &, extreme_value_dist::param_type &);

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

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

4.2.12 Γ-distribution

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

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

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 $\kappa = 1$ 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 {

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

  bool operator==(const gamma_dist::param_type &, const gamma_dist::param_type &);
  bool operator!=(const gamma_dist::param_type &, const gamma_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 gamma_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> &, gamma_dist::param_type &);

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

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

4.2.13 χ^2-distribution

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 {

  class chi_square_dist {
  public:
    typedef double result_type;
    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>
    double operator()(R &);
    template<typename R>
    double operator()(R &, const param_type &);
    double min() const;
    double max() const;
    param_type param() const;
    void param(const param_type &);
    int nu() const;
```

```
    void nu(int);
    double pdf(double) const;
    double cdf(double) const;
    double icdf(double) const;
  };

  bool operator==(const chi_square_dist::param_type &, const chi_square_dist::param_type &);
  bool operator!=(const chi_square_dist::param_type &, const chi_square_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 chi_square_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> &, chi_square_dist::param_type &);

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

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

4.2.14 Bernoulli distribution

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

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

A valid parameter for this distribution is $p \in [0,1]$.

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 &);
    explicit param_type(double p, const T &head, const T &tail);
  };

  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;
  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> &);
}
```

4.2.15 Binomial distribution

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

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

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);
      explicit param_type(double p, int n);
    };

    explicit 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;
    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 &);
}
```

4.2.16 Geometric distribution

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

$$P(x|p) = p(1-p)^x \qquad \text{for } x \in \{0, 1, 2, \dots\}.$$

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;
    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 &);
}
```

4.2.17 Poisson distribution

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

$$P(x|\mu) = \frac{\mathrm{e}^{-\mu}\mu^x}{x!} \qquad \text{for } x \in \{0, 1, 2, \dots\}.$$

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;
    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 &);
}
```

4.2.18 Discrete distribution

Class `discrete_dist` provides random integers with arbitrary discrete distribution. The probability distribution function is given by a set of n positive weights p_i ($i = 0, 1, \ldots, n-1$) and reads

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

The weights p_i have to be passed to the constructor of class `discrete_dist` by some iterator range. 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);
    };
```

```
    template<typename iter>
    explicit 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;
    param_type param() const;
    void param(const param_type &);
    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 &);

}
```

5 Installation

To make the installation procedure portable and comfortable, TRNG utilizes the GNU build system. For a proper installation you will need

- GNU autotools (`autoconf`, `automake`, `libtool`) and
- a recent C++ compiler, that has a good template support and knows `long long` as a build-in data type (e. g. the GNU C++ compiler version 3.0 or newer).

TRNG comes with sample programs, that illustrate how the functionality of TRNG could be used by your applications. Some of these sample programs will use external libraries, namely:

- Boost C++ libraries, [3]
- An implementation of the Message Passing Interface (MPI) standard, various open source implementations can be found at [37, 34].

If you want to compile all sample programs, you will have to install these libraries before. But TRNG itself does not depend on the libraries listed above.

After the tar-ball had been extracted, you have to call the `configure` script. This script tries to find your C++ compiler and generates a set of Makefiles. On most Unix-like boxes, just calling

```
bauke@hal:~/trng-4.0$ ./configure
```

will work fine. The `configure` script can be controled by various options and shell variables. If no options are privided to `configure` TRNG will be installed in the `/usr/local` hierarchy. Call

```
bauke@hal:~/trng-4.0$ ./configure --help
```

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

```
bauke@hal:~/trng-4.0$ export CXX=icc
bauke@hal:~/trng-4.0$ export LDFLAGS=-lstdc++
bauke@hal:~/trng-4.0$ ./configure --prefix=/opt/trng
```

After TRNG had been configured, the library will be compiled and installed by

```
bauke@hal:~/trng-4.0$ make
bauke@hal:~/trng-4.0$ make install
```

The TRNG library can be uninstalled be the `uninstall` target.

```
bauke@hal:~/trng-4.0$ make uninstall
```

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` the user finds some sample programs, these can be compiled by

```
bauke@hal:~/trng-4.0/examples$ make examples
```

6 Examples

6.1 Hello world!

In listing 6.1 we present the simplest nontrivial C++ program that produces psudo-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, here a mean of 6 and a standard deviation of 2 are specifed. Distribution parameters can be changed later, if necessary. In the loop in lines 13 and 14 the random number engine object and the random number distribution object are used to generate 1000 random numbers with normal distribution.

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<1000; ++i)
14     std::cout << normal(R) << '\n';
15   return EXIT_SUCCESS;
16 }
```

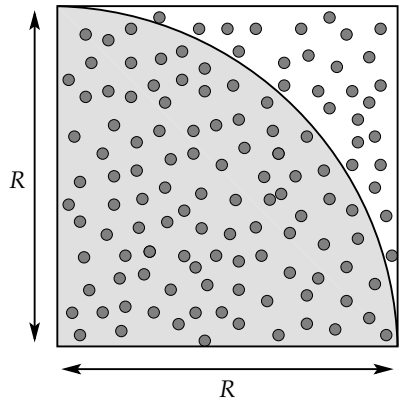

Figure 6.1: The numerical value of π can be estimated by throwing random points into a square.

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 calculate 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 $\sqrt{x^2+y^2} < 1$ the point (x,y) lies within the circle. The program counts the number of points within the circle and at the end of the program the fraction $4 \cdot (\text{points in circle})/(\text{points in square})$ is shown.

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(int argc, char *argv[]) {
7    const long samples=10000001;          // 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     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 designed as a random number generator library for sequential as well as for parallel applications. The library does not depend on any particular communication library, it may be utilized with MPI, OpenMP 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 the Message Passing Interface (MPI) or OpenMP. Here we can not give an introduction to MPI or OpenMP, readers, who are not familiar with parallel programming, may consult [38, 2, 41] 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 is, that the placement of some point in the square does not affect the placement of other points. In other words: throwing N balls into a square is an embarrassingly parallel process. Everything that matters, is the fraction of points in the square that also lie in the circle. Keeping this fact in mind the Monte Carlo calculation of π can be parallelized via block splitting or leapfrog method.

6.2.1 Block splitting

Here we choose the following parallelization strategy. It is based on the block splitting technique, introduced in section 2. A total of N points has to be selected by p processes. We

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

```
#include <iostream>
#include "mpi.h"
#include <trng/yarn2.hpp>
#include <trng/uniform01_dist.hpp>

int main(int argc, char *argv[]) {
  const long samples=10000001;          // total number of points in square
  trng::yarn2 r;                        // random number engine
  MPI::Init(argc, argv);                // initialise MPI environment
  int size=MPI::COMM_WORLD.Get_size();  // get total number of processes
  int rank=MPI::COMM_WORLD.Get_rank();  // get rank of current process
  long in=0l;                           // number of points in circle
  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) {
    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::COMM_WORLD.Reduce(&in, &in_all, 1, MPI::LONG, MPI::SUM, 0);
  if (rank==0)                          // print result
    std::cout << "pi = " << 4.0*in_all/samples << std::endl;
  MPI::Finalize();                      // quit MPI
  return EXIT_SUCCESS;
}
```

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 concrete, 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.

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.

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

```
1  #include <iostream>
2  #include <omp.h>
3  #include <trng/yarn2.hpp>
4  #include <trng/uniform01_dist.hpp>
5
6  int main(int argc, char *argv[]) {
7    const long samples=10000001;          // total number of points in square
8    long in=0l;                           // number of points in circle
9    // distribute workload over all processes
10 #pragma omp parallel
11   {
12     trng::yarn2 r;                      // random number engine
13     int size=omp_get_num_threads();     // get total number of processes
14     int rank=omp_get_thread_num();      // get rank of current process
15     trng::uniform01_dist u;             // random number distribution
16     r.jump(2*(rank*samples/size));      // jump ahead
17     // throw random points into square
18     for (long i=rank*samples/size; i<(rank+1)*samples/size; ++i) {
19       double x=u(r), y=u(r);            // choose random x– and y–coordinates
20       if (x*x+y*y<=1.0)                 // is point in circle?
21 #pragma omp critical
22         ++in;                           // increase counter
23     }
24   }
25   // print result
26   std::cout << "pi = " << 4.0*in/samples << std::endl;
27   return EXIT_SUCCESS;
28 }
```

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 process local variable. 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 `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.

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 `s`th sub-stream of `p` sub-streams. Sub-streams are numbered from 0 to $\mathtt{p}-1$. Changing line 15 or line 17 in listing 6.3 or listing 6.4 respectively, which reads

Listing 6.5: Parallel Monte Carlo calculation of π using leapfrog and MPI.

```
1  #include <iostream>
2  #include "mpi.h"
3  #include <trng/yarn2.hpp>
4  #include <trng/uniform01_dist.hpp>
5
6  int main(int argc, char *argv[]) {
7    const long samples=10000001;          // total number of points in square
8    trng::yarn2 rx, ry;                   // random number engines for x- and y-coordinates
9    MPI::Init(argc, argv);                // initialize MPI environment
10   int size=MPI::COMM_WORLD.Get_size();  // get total number of processes
11   int rank=MPI::COMM_WORLD.Get_rank();  // get rank of current process
12   // split PRN sequences by leapfrog method
13   rx.split(2, 0);                       // choose sub-stream no. 0 out of 2 streams
14   ry.split(2, 1);                       // choose sub-stream no. 1 out of 2 streams
15   rx.split(size, rank);                 // choose sub-stream no. rank out of size streams
16   ry.split(size, rank);                 // choose sub-stream no. rank out of size streams
17   long in=0l;                           // number of points in circle
18   trng::uniform01_dist u;               // random number distribution
19   // throw random points into square and distribute workload over all processes
20   for (long i=rank; i<samples; i+=size) {
21     double x=u(rx), y=u(ry);            // choose random x- and y-coordinates
22     if (x*x+y*y<=1.0)                   // is point in circle?
23       ++in;                             // increase counter
24   }
25   // calculate sum of all local variables 'in' and storre result in 'in_all' on process 0
26   long in_all;
27   MPI::COMM_WORLD.Reduce(&in, &in_all, 1, MPI::LONG, MPI::SUM, 0);
28   if (rank==0)                          // print result
29     std::cout << "pi = " << 4.0*in_all/samples << std::endl;
30   MPI::Finalize();                      // quit MPI
31   return EXIT_SUCCESS;
32 }
```

Listing 6.6: Parallel Monte Carlo calculation of π using leapfrog and OpenMP.

```
#include <iostream>
#include <omp.h>
#include <trng/yarn2.hpp>
#include <trng/uniform01_dist.hpp>

int main(int argc, char *argv[]) {
  const long samples=10000001;          // total number of points in square
  long in=01;                           // no points in circle
  // distribute workload over all processes
#pragma omp parallel
  {
    trng::yarn2 rx, ry;                 // random number engines for x– and y–coordinates
    int size=omp_get_num_threads();     // get total number of processes
    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) {
      double x=u(rx), y=u(ry);          // choose random x– and y–coordinates
      if (x*x+y*y<=1.0)                 // is point in circle?
#pragma omp critical
        ++in;                           // increase counter
    }
  }
  // print result
  std::cout << "pi = " << 4.0*in/samples << std::endl;
  return EXIT_SUCCESS;
}
```

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

into

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

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 6.5 and listing 6.6 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
```

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 the 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, that 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, that is 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 it mimics the way in which the parallelizable task is split into independent sub-problems. This can always be archived by calling the `split` or the `jump` method of a random number engine object.
- If the number of pseudo-random numbers, that is 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 these advises somewhat more clear, we give a further example. Imagine the simulation of a site percolation process [45] 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 afterwards. 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 to do this, is not to parallelize at all. At least not the analysis of a single realization of occupied sites itself. Usually one is not only interested in the analysis of a single realization of occupied sites 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. Its is quite natural to spread the workload over p processors in such a way, that 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 thats 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 process, but also the pseudo-random numbers, that would be consumed by analyzing the skipped lattices. Listing 6.7 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 [46, 36]) 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 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

Listing 6.7: 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=MPI::COMM_WORLD.Get_size();      // get total number of processes
  int rank=MPI::COMM_WORLD.Get_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;
}
```

process has to skip a certain amount of pseudo-random numbers at the start of the simulation, too.

Listing 6.8 shows the frame for 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 [38, 2] 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 6.8 is, that it does not assume, that the number of sites in any dimension is a multiple of the number of sub-lattices

Listing 6.8: 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_WORLD.Get_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
  bool periods[2] = { false, false };   // no periodic boundary conditions
  // calculate a balanced grid partitioning such that  size = dims[0]*dims[1]
  MPI::Compute_dims(MPI::COMM_WORLD.Get_size(), 2, dims);
  MPI::Cartcomm Comm=MPI::COMM_WORLD.Create_cart(2, dims, periods, true);
  int rank=Comm.Get_rank();             // get rank of current process
  Comm.Get_coords(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
  }
  MPI::Finalize();                      // quit MPI
  return EXIT_SUCCESS;
}
```

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 starts do limited the speedup, that is 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 also with the number of processes. 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 [3]. In this section we show, how to use TRNG in combination with the STL, especially its containers and algorithms and the bind facility of Boost[1]. We assume you are familiar with the concepts of the C++ STL, otherwise we suggest to read [35].

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::`

[1] The bind facility of Boost will be part of future versions of the STL.

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`. The template class `binder_cl` is such a function adapter.

```
template<typename PRN_dist_t, typename PRN_engine_t>
class binder_cl {
  PRN_dist_t &dist;
  PRN_engine_t &engine;
public:
  binder_cl(PRN_dist_t &dist, PRN_engine_t &engine) : dist(dist), engine(engine) {
  }
  typename PRN_dist_t::result_type operator()() {
    return dist(engine);
  }
};
```

It holds a reference to a random number engine and a reference to a random number distribution respectively as private data members. Its call operator calls the call operator of the random number distribution with the random number engine as its argument. With this template class 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(), binder_cl<trng::uniform_int_dist, trng::yarn2>(U, R));
```

The statement

```
binder_cl<trng::uniform_int_dist, trng::yarn2>(U, R)
```

creates a temporary anonymous object of the class `binder_cl<trng::uniform_int_dist, trng::yarn2>`, which is a instantiation of the template class `binder_cl`. Up to now we have not gained very much. Now we can replace an explicit loop by a template function `std::generate`, but the syntax is clumsy and as error-prone as the explicit loop, because the types, that specify the template class `binder_cl` have to be given explicitly.

This is a common obstacle in generic programming in C++ but this can be avoided by a further helper function `make_binder`.

```
template<typename PRN_dist_t, typename PRN_engine_t>
inline
binder_cl<PRN_dist_t,  PRN_engine_t> make_binder(PRN_dist_t &dist, PRN_engine_t &engine) {
  return binder_cl<PRN_dist_t,  PRN_engine_t>(dist, engine);
}
```

With this little helper function the line

```
std::generate(v.begin(), v.end(), binder_cl<trng::uniform_int_dist, trng::yarn2>(U, R));
```

can be simplified to

```
std::generate(v.begin(), v.end(), make_binder(U, R));
```

Adapting function objects to functions and algorithms is a common task in generic programming. The C++ STL is equipped with some adapter functions like `std::bind1st` or `std::bind2nd`, but they are of limited use and from time to time further adapter functions

have to be created, as shown in the preceding paragraphs. The bind facility of the Boost library generalizes the STL function adapters and we do not have to write our own function adapters. Here we can give only a glimpse of the bind facility, everyone how wants to explore the full capabilities of `boost::bind` should read the Boost documentation.

The boost equivalent to

```
std::generate(v.begin(), v.end(), make_binder(U, R));
```

reads

```
std::generate(v.begin(), v.end(), boost::bind(U, boost::ref(R)));
```

In this example the function `boost::bind` returns a temporary function object whose call operator requires no arguments. The function `boost::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 `boost::ref` may have unexpected side effects, e. g. the loop

```
for (int i(0); i<10; ++i)
  std::generate(v.begin(), v.end(), boost::bind(U, R));
```

would fill the vector `v` ten times with random numbers, each time with the same set of random numbers. Because `boost::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 6.9 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 function `std::random_shuffle` directly, no function adaption via `boost:bind` is needed.

Listing 6.9: This demo program demonstrates the interplay of TRNG, the C++ STL and the bind facility of Boost.

```
1  #include <cstdlib>
2  #include <iostream>
3  #include <vector>
4  #include <algorithm>
5  #include <trng/config.hpp>
6  #include <trng/yarn2.hpp>
7  #include <trng/uniform_int_dist.hpp>
8  #if defined HAVE_BOOST
9    #include <boost/bind.hpp>
10 #else
11
12   // helper class
13   template<typename PRN_dist_t, typename PRN_engine_t>
14   class binder_cl {
15     PRN_dist_t &dist;
16     PRN_engine_t &engine;
17   public:
18     binder_cl(PRN_dist_t &dist, PRN_engine_t &engine) : dist(dist), engine(engine) {
19     }
20     typename PRN_dist_t::result_type operator()() {
21       return dist(engine);
22     }
```

```
23    };
24
25    // convenience function
26    template<typename PRN_dist_t, typename PRN_engine_t>
27    inline
28    binder_cl<PRN_dist_t,  PRN_engine_t> make_binder(PRN_dist_t &dist, PRN_engine_t &engine) {
29      return binder_cl<PRN_dist_t,  PRN_engine_t>(dist, engine);
30    }
31
32  #endif
33
34
35  // print an iterator range to stdout
36  template<typename iter>
37  void print_range(iter i1, iter i2) {
38    while (i1!=i2) std::cout << (*(i1++)) << '\t';
39    std::cout << "\n\n";
40  }
41
42  int main() {
43    trng::yarn2 R;
44    trng::uniform_int_dist U(0, 100);
45    std::vector<long> v(10);
46
47    std::cout << "random number generation by call operator\n";
48    for (std::vector<long>::size_type i=0; i<v.size(); ++i)
49      v[i]=U(R);
50    print_range(v.begin(), v.end());
51    std::vector<long> w(12);
52  #if defined HAVE_BOOST
53    std::cout << "random number generation by std::generate\n";
54    std::generate(w.begin(), w.end(), boost::bind(U, boost::ref(R)));
55    print_range(w.begin(), w.end());
56    std::cout << "random number generation by std::generate\n";
57    std::generate(w.begin(), w.end(), boost::bind(U, boost::ref(R)));
58    print_range(w.begin(), w.end());
59  #else
60    std::cout << "random number generation by std::generate\n";
61    std::generate(w.begin(), w.end(), make_binder(U, R));
62    print_range(w.begin(), w.end());
63    std::cout << "random number generation by std::generate\n";
64    std::generate(w.begin(), w.end(), make_binder(U, R));
65    print_range(w.begin(), w.end());
66  #endif
67    std::cout << "same sequence as above, but in a random shuffled order\n";
68    std::random_shuffle(w.begin(), w.end(), R);
69    print_range(w.begin(), w.end());
70    return EXIT_SUCCESS;
71  }
```

7 Implementation details and efficiency

Random number engines `trng::mrg`n{s} and `trng::yarn`n{s} 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 [13, 18, 17] 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 can not be stored in a w-bit register. For the special case $a_k < \sqrt{m}$ Schrage [43] 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 [19]. For parallel PRNGs this methods do not apply because the leapfrog method may yield coefficients that violate this condition. Knuth [17, 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 (2.6) 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 todays 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.

7.0.1 Efficient modular reduction

Since the modulo operation in (2.6) 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 [39].

A summand $s = a_k \cdot r_{i-k}$ in (2.6) 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 $2^e - k$ [29]. 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

$$\begin{aligned} s' &= (s \bmod 2^e) + \lfloor s/2^e \rfloor \\ x &= (s' \bmod 2^e) + \lfloor s'/2^e \rfloor\,. \end{aligned} \tag{7.5}$$

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 $k(k+2) \leq m$.

7.0.2 Fast delinearization

YARN generators hide linear structures of LFSR sequences (q) 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 (2.6). Therefore we propose to implement exponentiation by table lookup. If m is a $2e'$-bit number we apply the decomposition

$$\begin{aligned} q_i &= q_{i,1} \cdot 2^{e'} + q_{i,2} \quad \text{with} \\ q_{i,1} &= \lfloor q_i/2^{e'} \rfloor\,, \quad q_{i,0} = q_i \bmod 2^{e'} \end{aligned} \tag{7.6}$$

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 lookups 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.

7.0.3 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 lookup (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 (`boost::mt19937` and `mt19937`) , lagged Fibonacci generators (LFSR sequences over $\mathbb{F}_2$) or RANLUX that can be found in the Boost library [3] or the GNU Scientific Library [10].

Table 7.1: Performance of various random number engines from TRNG and Boost. Test program was compiled and executed on a Pentium IV 3 GHz with an Intel C++ compiler version 9.1 and option -O3.

generator	PRNs per second
TRNG	
`trng::lcg64`	$156 \cdot 10^6$
`trng::mrg2`	$48 \cdot 10^6$
`trng::mrg3`	$42 \cdot 10^6$
`trng::mrg3s`	$25 \cdot 10^6$
`trng::mrg4`	$38 \cdot 10^6$
`trng::mrg5`	$35 \cdot 10^6$
`trng::mrg5s`	$17 \cdot 10^6$
`trng::yarn2`	$26 \cdot 10^6$
`trng::yarn3`	$23 \cdot 10^6$
`trng::yarn3s`	$12 \cdot 10^6$
`trng::yarn4`	$20 \cdot 10^6$
`trng::yarn5`	$22 \cdot 10^6$
`trng::yarn5s`	$9 \cdot 10^6$
Boost	
`boost::minstd_rand`	$77 \cdot 10^6$
`boost::ecuyer1988`	$36 \cdot 10^6$
`boost::kreutzer1986`	$69 \cdot 10^6$
`boost::hellekalek1995`	$2 \cdot 10^6$
`boost::mt11213b`	$74 \cdot 10^6$
`boost::mt19937`	$74 \cdot 10^6$
`boost::lagged_fibonacci607`	$43 \cdot 10^6$
`boost::lagged_fibonacci1279`	$43 \cdot 10^6$
`boost::lagged_fibonacci2281`	$47 \cdot 10^6$
`boost::lagged_fibonacci3217`	$48 \cdot 10^6$
`boost::lagged_fibonacci4423`	$48 \cdot 10^6$
`boost::lagged_fibonacci9689`	$63 \cdot 10^6$
`boost::lagged_fibonacci19937`	$48 \cdot 10^6$
`boost::lagged_fibonacci23209`	$63 \cdot 10^6$
`boost::lagged_fibonacci44497`	$63 \cdot 10^6$
GNU Scientific Library	
`mt19937`	$41 \cdot 10^6$
`r250`	$85 \cdot 10^6$
`gfsr4`	$77 \cdot 10^6$
`ran2`	$27 \cdot 10^6$
`ranlux`	$8 \cdot 10^6$
`ranlux389`	$5 \cdot 10^6$

8 Quality

Sequences of PRNs are sequences of deterministic numbers, that try to mimic true random numbers. But how close are sequences produced by the PRNGs of TRNG to sequences of real random number? This question can be answered (at least partly) by statistical tests and we can apply a battery of tests on this generator. The more tests the 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, 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. The probability distribution is used to judge the 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 should be avoided.

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 [19] and give good results in the spectral test [17], 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. "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 ranom—it is then presumed innocent until proven guilty." [17].

All PRNGs of TRNG and sub-streams of them have been subject to different statistical tests. 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 [30]. The tables 8.1 to 8.14 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. Each test was applied eight times and the tables 8.1 to 8.14 show how often each test has been failed. Note, at a confidence level of 0.1 or 0.9 even a perfect random number generator will "fail" these tests on average in one of ten cases. All statistical tests are implemented by the Random Number

Generator Test Suite (RNGTS) [42][1]. A detailed descriptions of the statistical tests can be found on the RNGTS web site.

Table 8.1: Test results for random number engine `trng::lcg64`.

	confidence level			
test	0.05	0.1	0.9	0.95
KS-Uniformity-Test	1 of 8 failed	2 of 8 failed	1 of 8 failed	1 of 8 failed
Chi-Square-Uniformity-Test	1 of 8 failed	2 of 8 failed	1 of 8 failed	0 of 8 failed
Gap-Test	0 of 8 failed	0 of 8 failed	2 of 8 failed	1 of 8 failed
n-Block-Test	0 of 8 failed	0 of 8 failed	2 of 8 failed	0 of 8 failed
Ising-Model-Test (energy)	0 of 8 failed	1 of 8 failed	1 of 8 failed	0 of 8 failed
Ising-Model-Test (specific heat)	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed
CouponCollector-Test	0 of 8 failed	1 of 8 failed	4 of 8 failed	2 of 8 failed
Permutation-Test	0 of 8 failed	0 of 8 failed	1 of 8 failed	0 of 8 failed
Poker-Test	0 of 8 failed	1 of 8 failed	2 of 8 failed	1 of 8 failed
Maximum-of-t Test	0 of 8 failed	0 of 8 failed	5 of 8 failed	3 of 8 failed
Serial correlation Test	0 of 8 failed	0 of 8 failed	2 of 8 failed	2 of 8 failed
Random-Walk Test	2 of 8 failed	2 of 8 failed	0 of 8 failed	0 of 8 failed
Serial Test	0 of 8 failed	0 of 8 failed	3 of 8 failed	0 of 8 failed
Collision-Test (Hash-Test)	1 of 8 failed	3 of 8 failed	0 of 8 failed	0 of 8 failed
Squeeze-Test	3 of 8 failed	3 of 8 failed	3 of 8 failed	1 of 8 failed
Birthday-Spacing-Test	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed
Binary-Rank Test (K-S)	0 of 8 failed	1 of 8 failed	0 of 8 failed	0 of 8 failed
Minimum-Distance Test	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed
Craps-Test	1 of 8 failed	2 of 8 failed	1 of 8 failed	0 of 8 failed

[1] We had to apply some minor modifications to RNGTS in order to adapt this test suite to TRNG.

Table 8.2: Test results for random number engine `trng::mrg2`.

	confidence level			
test	0.05	0.1	0.9	0.95
KS-Uniformity-Test	2 of 8 failed	3 of 8 failed	3 of 8 failed	1 of 8 failed
Chi-Square-Uniformity-Test	0 of 8 failed	1 of 8 failed	0 of 8 failed	0 of 8 failed
Gap-Test	0 of 8 failed	0 of 8 failed	1 of 8 failed	1 of 8 failed
n-Block-Test	1 of 8 failed	2 of 8 failed	1 of 8 failed	1 of 8 failed
Ising-Model-Test (energy)	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed
Ising-Model-Test (specific heat)	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed
CouponCollector-Test	1 of 8 failed	1 of 8 failed	0 of 8 failed	0 of 8 failed
Permutation-Test	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed
Poker-Test	0 of 8 failed	1 of 8 failed	1 of 8 failed	0 of 8 failed
Maximum-of-t Test	0 of 8 failed	1 of 8 failed	3 of 8 failed	3 of 8 failed
Serial correlation Test	1 of 8 failed	1 of 8 failed	0 of 8 failed	0 of 8 failed
Random-Walk Test	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed
Serial Test	0 of 8 failed	0 of 8 failed	1 of 8 failed	1 of 8 failed
Collision-Test (Hash-Test)	0 of 8 failed	1 of 8 failed	0 of 8 failed	0 of 8 failed
Squeeze-Test	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed
Birthday-Spacing-Test	0 of 8 failed	0 of 8 failed	1 of 8 failed	0 of 8 failed
Binary-Rank Test (K-S)	2 of 8 failed	4 of 8 failed	0 of 8 failed	0 of 8 failed
Minimum-Distance Test	1 of 8 failed	2 of 8 failed	4 of 8 failed	3 of 8 failed
Craps-Test	0 of 8 failed	0 of 8 failed	1 of 8 failed	0 of 8 failed

Table 8.3: Test results for random number engine `trng::mrg3`.

	confidence level			
test	0.05	0.1	0.9	0.95
KS-Uniformity-Test	1 of 8 failed	1 of 8 failed	5 of 8 failed	3 of 8 failed
Chi-Square-Uniformity-Test	0 of 8 failed	0 of 8 failed	1 of 8 failed	1 of 8 failed
Gap-Test	0 of 8 failed	0 of 8 failed	2 of 8 failed	1 of 8 failed
n-Block-Test	0 of 8 failed	0 of 8 failed	3 of 8 failed	3 of 8 failed
Ising-Model-Test (energy)	0 of 8 failed	1 of 8 failed	1 of 8 failed	0 of 8 failed
Ising-Model-Test (specific heat)	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed
CouponCollector-Test	0 of 8 failed	0 of 8 failed	1 of 8 failed	1 of 8 failed
Permutation-Test	0 of 8 failed	0 of 8 failed	1 of 8 failed	1 of 8 failed
Poker-Test	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed
Maximum-of-t Test	1 of 8 failed	1 of 8 failed	2 of 8 failed	2 of 8 failed
Serial correlation Test	1 of 8 failed	1 of 8 failed	0 of 8 failed	0 of 8 failed
Random-Walk Test	0 of 8 failed	0 of 8 failed	1 of 8 failed	0 of 8 failed
Serial Test	1 of 8 failed	1 of 8 failed	0 of 8 failed	0 of 8 failed
Collision-Test (Hash-Test)	0 of 8 failed	0 of 8 failed	2 of 8 failed	1 of 8 failed
Squeeze-Test	2 of 8 failed	2 of 8 failed	1 of 8 failed	1 of 8 failed
Birthday-Spacing-Test	3 of 8 failed	3 of 8 failed	0 of 8 failed	0 of 8 failed
Binary-Rank Test (K-S)	2 of 8 failed	3 of 8 failed	0 of 8 failed	0 of 8 failed
Minimum-Distance Test	1 of 8 failed	1 of 8 failed	2 of 8 failed	2 of 8 failed
Craps-Test	0 of 8 failed	0 of 8 failed	1 of 8 failed	1 of 8 failed

Table 8.4: Test results for random number engine `trng::mrg3s`.

	confidence level			
test	0.05	0.1	0.9	0.95
KS-Uniformity-Test	0 of 8 failed	1 of 8 failed	3 of 8 failed	2 of 8 failed
Chi-Square-Uniformity-Test	1 of 8 failed	2 of 8 failed	0 of 8 failed	0 of 8 failed
Gap-Test	2 of 8 failed	3 of 8 failed	1 of 8 failed	0 of 8 failed
n-Block-Test	1 of 8 failed	1 of 8 failed	2 of 8 failed	2 of 8 failed
Ising-Model-Test (energy)	0 of 8 failed	2 of 8 failed	1 of 8 failed	0 of 8 failed
Ising-Model-Test (specific heat)	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed
CouponCollector-Test	0 of 8 failed	1 of 8 failed	0 of 8 failed	0 of 8 failed
Permutation-Test	0 of 8 failed	1 of 8 failed	1 of 8 failed	0 of 8 failed
Poker-Test	2 of 8 failed	2 of 8 failed	1 of 8 failed	0 of 8 failed
Maximum-of-t Test	0 of 8 failed	0 of 8 failed	3 of 8 failed	1 of 8 failed
Serial correlation Test	1 of 8 failed	1 of 8 failed	1 of 8 failed	1 of 8 failed
Random-Walk Test	1 of 8 failed	1 of 8 failed	2 of 8 failed	1 of 8 failed
Serial Test	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed
Collision-Test (Hash-Test)	1 of 8 failed	1 of 8 failed	1 of 8 failed	0 of 8 failed
Squeeze-Test	1 of 8 failed	1 of 8 failed	3 of 8 failed	3 of 8 failed
Birthday-Spacing-Test	0 of 8 failed	1 of 8 failed	0 of 8 failed	0 of 8 failed
Binary-Rank Test (K-S)	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed
Minimum-Distance Test	0 of 8 failed	0 of 8 failed	3 of 8 failed	2 of 8 failed
Craps-Test	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed

Table 8.5: Test results for random number engine `trng::mrg4`.

	confidence level			
test	0.05	0.1	0.9	0.95
KS-Uniformity-Test	0 of 8 failed	1 of 8 failed	2 of 8 failed	1 of 8 failed
Chi-Square-Uniformity-Test	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed
Gap-Test	0 of 8 failed	0 of 8 failed	1 of 8 failed	0 of 8 failed
n-Block-Test	0 of 8 failed	1 of 8 failed	4 of 8 failed	2 of 8 failed
Ising-Model-Test (energy)	1 of 8 failed	2 of 8 failed	0 of 8 failed	0 of 8 failed
Ising-Model-Test (specific heat)	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed
CouponCollector-Test	1 of 8 failed	1 of 8 failed	2 of 8 failed	0 of 8 failed
Permutation-Test	2 of 8 failed	2 of 8 failed	0 of 8 failed	0 of 8 failed
Poker-Test	2 of 8 failed	2 of 8 failed	0 of 8 failed	0 of 8 failed
Maximum-of-t Test	0 of 8 failed	0 of 8 failed	4 of 8 failed	1 of 8 failed
Serial correlation Test	1 of 8 failed	1 of 8 failed	0 of 8 failed	0 of 8 failed
Random-Walk Test	1 of 8 failed	1 of 8 failed	0 of 8 failed	0 of 8 failed
Serial Test	1 of 8 failed	1 of 8 failed	3 of 8 failed	3 of 8 failed
Collision-Test (Hash-Test)	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed
Squeeze-Test	3 of 8 failed	3 of 8 failed	0 of 8 failed	0 of 8 failed
Birthday-Spacing-Test	3 of 8 failed	3 of 8 failed	0 of 8 failed	0 of 8 failed
Binary-Rank Test (K-S)	1 of 8 failed	1 of 8 failed	0 of 8 failed	0 of 8 failed
Minimum-Distance Test	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed
Craps-Test	0 of 8 failed	3 of 8 failed	1 of 8 failed	1 of 8 failed

Table 8.6: Test results for random number engine `trng::mrg5`.

	confidence level			
test	0.05	0.1	0.9	0.95
KS-Uniformity-Test	2 of 8 failed	2 of 8 failed	1 of 8 failed	0 of 8 failed
Chi-Square-Uniformity-Test	0 of 8 failed	0 of 8 failed	1 of 8 failed	1 of 8 failed
Gap-Test	0 of 8 failed	0 of 8 failed	1 of 8 failed	0 of 8 failed
n-Block-Test	2 of 8 failed	3 of 8 failed	0 of 8 failed	0 of 8 failed
Ising-Model-Test (energy)	0 of 8 failed	1 of 8 failed	1 of 8 failed	0 of 8 failed
Ising-Model-Test (specific heat)	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed
CouponCollector-Test	0 of 8 failed	0 of 8 failed	2 of 8 failed	2 of 8 failed
Permutation-Test	1 of 8 failed	1 of 8 failed	2 of 8 failed	1 of 8 failed
Poker-Test	1 of 8 failed	3 of 8 failed	0 of 8 failed	0 of 8 failed
Maximum-of-t Test	0 of 8 failed	0 of 8 failed	3 of 8 failed	0 of 8 failed
Serial correlation Test	0 of 8 failed	0 of 8 failed	2 of 8 failed	1 of 8 failed
Random-Walk Test	1 of 8 failed	1 of 8 failed	1 of 8 failed	0 of 8 failed
Serial Test	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed
Collision-Test (Hash-Test)	0 of 8 failed	0 of 8 failed	1 of 8 failed	1 of 8 failed
Squeeze-Test	0 of 8 failed	0 of 8 failed	1 of 8 failed	1 of 8 failed
Birthday-Spacing-Test	1 of 8 failed	1 of 8 failed	0 of 8 failed	0 of 8 failed
Binary-Rank Test (K-S)	1 of 8 failed	1 of 8 failed	0 of 8 failed	0 of 8 failed
Minimum-Distance Test	0 of 8 failed	2 of 8 failed	2 of 8 failed	0 of 8 failed
Craps-Test	0 of 8 failed	0 of 8 failed	2 of 8 failed	2 of 8 failed

Table 8.7: Test results for random number engine `trng::mrg5s`.

	confidence level			
test	0.05	0.1	0.9	0.95
KS-Uniformity-Test	1 of 8 failed	2 of 8 failed	2 of 8 failed	1 of 8 failed
Chi-Square-Uniformity-Test	0 of 8 failed	0 of 8 failed	1 of 8 failed	0 of 8 failed
Gap-Test	1 of 8 failed	1 of 8 failed	1 of 8 failed	1 of 8 failed
n-Block-Test	1 of 8 failed	1 of 8 failed	0 of 8 failed	0 of 8 failed
Ising-Model-Test (energy)	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed
Ising-Model-Test (specific heat)	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed
CouponCollector-Test	0 of 8 failed	0 of 8 failed	3 of 8 failed	1 of 8 failed
Permutation-Test	0 of 8 failed	2 of 8 failed	0 of 8 failed	0 of 8 failed
Poker-Test	0 of 8 failed	1 of 8 failed	0 of 8 failed	0 of 8 failed
Maximum-of-t Test	0 of 8 failed	0 of 8 failed	1 of 8 failed	0 of 8 failed
Serial correlation Test	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed
Random-Walk Test	0 of 8 failed	1 of 8 failed	0 of 8 failed	0 of 8 failed
Serial Test	0 of 8 failed	1 of 8 failed	0 of 8 failed	0 of 8 failed
Collision-Test (Hash-Test)	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed
Squeeze-Test	0 of 8 failed	0 of 8 failed	1 of 8 failed	1 of 8 failed
Birthday-Spacing-Test	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed
Binary-Rank Test (K-S)	4 of 8 failed	4 of 8 failed	0 of 8 failed	0 of 8 failed
Minimum-Distance Test	0 of 8 failed	1 of 8 failed	1 of 8 failed	1 of 8 failed
Craps-Test	1 of 8 failed	2 of 8 failed	0 of 8 failed	0 of 8 failed

Table 8.8: Test results for random number engine `trng::yarn2`.

	confidence level			
test	0.05	0.1	0.9	0.95
KS-Uniformity-Test	0 of 8 failed	1 of 8 failed	2 of 8 failed	1 of 8 failed
Chi-Square-Uniformity-Test	0 of 8 failed	0 of 8 failed	2 of 8 failed	2 of 8 failed
Gap-Test	0 of 8 failed	1 of 8 failed	0 of 8 failed	0 of 8 failed
n-Block-Test	0 of 8 failed	1 of 8 failed	1 of 8 failed	1 of 8 failed
Ising-Model-Test (energy)	1 of 8 failed	1 of 8 failed	1 of 8 failed	1 of 8 failed
Ising-Model-Test (specific heat)	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed
CouponCollector-Test	0 of 8 failed	0 of 8 failed	1 of 8 failed	0 of 8 failed
Permutation-Test	0 of 8 failed	1 of 8 failed	0 of 8 failed	0 of 8 failed
Poker-Test	0 of 8 failed	1 of 8 failed	0 of 8 failed	0 of 8 failed
Maximum-of-t Test	0 of 8 failed	0 of 8 failed	1 of 8 failed	0 of 8 failed
Serial correlation Test	0 of 8 failed	0 of 8 failed	2 of 8 failed	1 of 8 failed
Random-Walk Test	0 of 8 failed	1 of 8 failed	2 of 8 failed	1 of 8 failed
Serial Test	0 of 8 failed	1 of 8 failed	1 of 8 failed	0 of 8 failed
Collision-Test (Hash-Test)	1 of 8 failed	2 of 8 failed	2 of 8 failed	1 of 8 failed
Squeeze-Test	0 of 8 failed	0 of 8 failed	3 of 8 failed	1 of 8 failed
Birthday-Spacing-Test	1 of 8 failed	1 of 8 failed	0 of 8 failed	0 of 8 failed
Binary-Rank Test (K-S)	0 of 8 failed	1 of 8 failed	0 of 8 failed	0 of 8 failed
Minimum-Distance Test	1 of 8 failed	2 of 8 failed	1 of 8 failed	1 of 8 failed
Craps-Test	0 of 8 failed	0 of 8 failed	2 of 8 failed	2 of 8 failed

Table 8.9: Test results for random number engine `trng::yarn3`.

	confidence level			
test	0.05	0.1	0.9	0.95
KS-Uniformity-Test	1 of 8 failed	3 of 8 failed	2 of 8 failed	2 of 8 failed
Chi-Square-Uniformity-Test	0 of 8 failed	0 of 8 failed	1 of 8 failed	1 of 8 failed
Gap-Test	0 of 8 failed	0 of 8 failed	1 of 8 failed	0 of 8 failed
n-Block-Test	0 of 8 failed	0 of 8 failed	4 of 8 failed	2 of 8 failed
Ising-Model-Test (energy)	1 of 8 failed	1 of 8 failed	1 of 8 failed	0 of 8 failed
Ising-Model-Test (specific heat)	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed
CouponCollector-Test	0 of 8 failed	1 of 8 failed	3 of 8 failed	2 of 8 failed
Permutation-Test	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed
Poker-Test	0 of 8 failed	0 of 8 failed	1 of 8 failed	1 of 8 failed
Maximum-of-t Test	0 of 8 failed	0 of 8 failed	2 of 8 failed	2 of 8 failed
Serial correlation Test	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed
Random-Walk Test	0 of 8 failed	0 of 8 failed	1 of 8 failed	0 of 8 failed
Serial Test	0 of 8 failed	1 of 8 failed	0 of 8 failed	0 of 8 failed
Collision-Test (Hash-Test)	0 of 8 failed	1 of 8 failed	0 of 8 failed	0 of 8 failed
Squeeze-Test	0 of 8 failed	1 of 8 failed	2 of 8 failed	1 of 8 failed
Birthday-Spacing-Test	0 of 8 failed	1 of 8 failed	1 of 8 failed	0 of 8 failed
Binary-Rank Test (K-S)	2 of 8 failed	3 of 8 failed	0 of 8 failed	0 of 8 failed
Minimum-Distance Test	2 of 8 failed	3 of 8 failed	1 of 8 failed	0 of 8 failed
Craps-Test	0 of 8 failed	1 of 8 failed	2 of 8 failed	1 of 8 failed

Table 8.10: Test results for random number engine `trng::yarn3s`.

	confidence level			
test	0.05	0.1	0.9	0.95
KS-Uniformity-Test	1 of 8 failed	4 of 8 failed	1 of 8 failed	0 of 8 failed
Chi-Square-Uniformity-Test	0 of 8 failed	1 of 8 failed	0 of 8 failed	0 of 8 failed
Gap-Test	0 of 8 failed	0 of 8 failed	1 of 8 failed	0 of 8 failed
n-Block-Test	0 of 8 failed	0 of 8 failed	4 of 8 failed	1 of 8 failed
Ising-Model-Test (energy)	0 of 8 failed	1 of 8 failed	0 of 8 failed	0 of 8 failed
Ising-Model-Test (specific heat)	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed
CouponCollector-Test	0 of 8 failed	0 of 8 failed	1 of 8 failed	0 of 8 failed
Permutation-Test	0 of 8 failed	0 of 8 failed	1 of 8 failed	1 of 8 failed
Poker-Test	1 of 8 failed	1 of 8 failed	1 of 8 failed	1 of 8 failed
Maximum-of-t Test	0 of 8 failed	0 of 8 failed	5 of 8 failed	3 of 8 failed
Serial correlation Test	0 of 8 failed	0 of 8 failed	1 of 8 failed	1 of 8 failed
Random-Walk Test	0 of 8 failed	1 of 8 failed	1 of 8 failed	0 of 8 failed
Serial Test	2 of 8 failed	4 of 8 failed	0 of 8 failed	0 of 8 failed
Collision-Test (Hash-Test)	1 of 8 failed	1 of 8 failed	1 of 8 failed	0 of 8 failed
Squeeze-Test	0 of 8 failed	1 of 8 failed	1 of 8 failed	1 of 8 failed
Birthday-Spacing-Test	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed
Binary-Rank Test (K-S)	2 of 8 failed	2 of 8 failed	0 of 8 failed	0 of 8 failed
Minimum-Distance Test	0 of 8 failed	2 of 8 failed	1 of 8 failed	0 of 8 failed
Craps-Test	0 of 8 failed	0 of 8 failed	1 of 8 failed	1 of 8 failed

Table 8.11: Test results for random number engine `trng::yarn4`.

	confidence level			
test	0.05	0.1	0.9	0.95
KS-Uniformity-Test	2 of 8 failed	3 of 8 failed	1 of 8 failed	0 of 8 failed
Chi-Square-Uniformity-Test	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed
Gap-Test	1 of 8 failed	1 of 8 failed	1 of 8 failed	0 of 8 failed
n-Block-Test	0 of 8 failed	0 of 8 failed	2 of 8 failed	2 of 8 failed
Ising-Model-Test (energy)	1 of 8 failed	2 of 8 failed	1 of 8 failed	1 of 8 failed
Ising-Model-Test (specific heat)	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed
CouponCollector-Test	0 of 8 failed	0 of 8 failed	2 of 8 failed	2 of 8 failed
Permutation-Test	1 of 8 failed	1 of 8 failed	2 of 8 failed	1 of 8 failed
Poker-Test	0 of 8 failed	0 of 8 failed	4 of 8 failed	1 of 8 failed
Maximum-of-t Test	1 of 8 failed	1 of 8 failed	1 of 8 failed	0 of 8 failed
Serial correlation Test	0 of 8 failed	0 of 8 failed	2 of 8 failed	2 of 8 failed
Random-Walk Test	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed
Serial Test	1 of 8 failed	1 of 8 failed	1 of 8 failed	1 of 8 failed
Collision-Test (Hash-Test)	0 of 8 failed	0 of 8 failed	2 of 8 failed	1 of 8 failed
Squeeze-Test	1 of 8 failed	1 of 8 failed	2 of 8 failed	2 of 8 failed
Birthday-Spacing-Test	0 of 8 failed	1 of 8 failed	2 of 8 failed	1 of 8 failed
Binary-Rank Test (K-S)	1 of 8 failed	2 of 8 failed	0 of 8 failed	0 of 8 failed
Minimum-Distance Test	1 of 8 failed	1 of 8 failed	0 of 8 failed	0 of 8 failed
Craps-Test	1 of 8 failed	2 of 8 failed	1 of 8 failed	1 of 8 failed

Table 8.12: Test results for random number engine `trng::yarn5`.

	confidence level			
test	0.05	0.1	0.9	0.95
KS-Uniformity-Test	0 of 8 failed	0 of 8 failed	2 of 8 failed	1 of 8 failed
Chi-Square-Uniformity-Test	0 of 8 failed	0 of 8 failed	1 of 8 failed	0 of 8 failed
Gap-Test	0 of 8 failed	0 of 8 failed	3 of 8 failed	1 of 8 failed
n-Block-Test	0 of 8 failed	1 of 8 failed	0 of 8 failed	0 of 8 failed
Ising-Model-Test (energy)	1 of 8 failed	1 of 8 failed	0 of 8 failed	0 of 8 failed
Ising-Model-Test (specific heat)	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed
CouponCollector-Test	0 of 8 failed	1 of 8 failed	0 of 8 failed	0 of 8 failed
Permutation-Test	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed
Poker-Test	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed
Maximum-of-t Test	0 of 8 failed	0 of 8 failed	4 of 8 failed	3 of 8 failed
Serial correlation Test	0 of 8 failed	0 of 8 failed	1 of 8 failed	0 of 8 failed
Random-Walk Test	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed
Serial Test	0 of 8 failed	1 of 8 failed	1 of 8 failed	1 of 8 failed
Collision-Test (Hash-Test)	0 of 8 failed	1 of 8 failed	0 of 8 failed	0 of 8 failed
Squeeze-Test	0 of 8 failed	0 of 8 failed	2 of 8 failed	1 of 8 failed
Birthday-Spacing-Test	0 of 8 failed	0 of 8 failed	2 of 8 failed	0 of 8 failed
Binary-Rank Test (K-S)	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed
Minimum-Distance Test	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed
Craps-Test	0 of 8 failed	0 of 8 failed	2 of 8 failed	2 of 8 failed

Table 8.13: Test results for random number engine `trng::yarn5s`.

	confidence level			
test	0.05	0.1	0.9	0.95
KS-Uniformity-Test	0 of 8 failed	0 of 8 failed	2 of 8 failed	2 of 8 failed
Chi-Square-Uniformity-Test	0 of 8 failed	0 of 8 failed	2 of 8 failed	1 of 8 failed
Gap-Test	0 of 8 failed	0 of 8 failed	3 of 8 failed	2 of 8 failed
n-Block-Test	1 of 8 failed	2 of 8 failed	0 of 8 failed	0 of 8 failed
Ising-Model-Test (energy)	0 of 8 failed	1 of 8 failed	1 of 8 failed	0 of 8 failed
Ising-Model-Test (specific heat)	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed
CouponCollector-Test	1 of 8 failed	1 of 8 failed	2 of 8 failed	2 of 8 failed
Permutation-Test	1 of 8 failed	2 of 8 failed	1 of 8 failed	1 of 8 failed
Poker-Test	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed
Maximum-of-t Test	0 of 8 failed	0 of 8 failed	2 of 8 failed	2 of 8 failed
Serial correlation Test	1 of 8 failed	3 of 8 failed	1 of 8 failed	0 of 8 failed
Random-Walk Test	0 of 8 failed	1 of 8 failed	0 of 8 failed	0 of 8 failed
Serial Test	0 of 8 failed	0 of 8 failed	1 of 8 failed	0 of 8 failed
Collision-Test (Hash-Test)	0 of 8 failed	1 of 8 failed	1 of 8 failed	1 of 8 failed
Squeeze-Test	1 of 8 failed	1 of 8 failed	4 of 8 failed	2 of 8 failed
Birthday-Spacing-Test	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed
Binary-Rank Test (K-S)	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed
Minimum-Distance Test	0 of 8 failed	1 of 8 failed	1 of 8 failed	0 of 8 failed
Craps-Test	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed

Table 8.14: Test results for random number engine `boost::mt19937` (Mersenne Twister generator).

	confidence level			
test	0.05	0.1	0.9	0.95
KS-Uniformity-Test	2 of 8 failed	2 of 8 failed	3 of 8 failed	2 of 8 failed
Chi-Square-Uniformity-Test	1 of 8 failed	2 of 8 failed	0 of 8 failed	0 of 8 failed
Gap-Test	1 of 8 failed	2 of 8 failed	2 of 8 failed	0 of 8 failed
n-Block-Test	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed
Ising-Model-Test (energy)	1 of 8 failed	1 of 8 failed	1 of 8 failed	1 of 8 failed
Ising-Model-Test (specific heat)	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed
CouponCollector-Test	0 of 8 failed	1 of 8 failed	2 of 8 failed	2 of 8 failed
Permutation-Test	0 of 8 failed	0 of 8 failed	0 of 8 failed	0 of 8 failed
Poker-Test	1 of 8 failed	1 of 8 failed	0 of 8 failed	0 of 8 failed
Maximum-of-t Test	1 of 8 failed	1 of 8 failed	1 of 8 failed	1 of 8 failed
Serial correlation Test	0 of 8 failed	0 of 8 failed	2 of 8 failed	1 of 8 failed
Random-Walk Test	1 of 8 failed	1 of 8 failed	0 of 8 failed	0 of 8 failed
Serial Test	0 of 8 failed	1 of 8 failed	1 of 8 failed	1 of 8 failed
Collision-Test (Hash-Test)	4 of 8 failed	5 of 8 failed	4 of 8 failed	3 of 8 failed
Squeeze-Test	2 of 8 failed	2 of 8 failed	1 of 8 failed	0 of 8 failed
Birthday-Spacing-Test	2 of 8 failed	2 of 8 failed	1 of 8 failed	1 of 8 failed
Binary-Rank Test (K-S)	1 of 8 failed	1 of 8 failed	2 of 8 failed	1 of 8 failed
Minimum-Distance Test	0 of 8 failed	2 of 8 failed	0 of 8 failed	0 of 8 failed
Craps-Test	0 of 8 failed	2 of 8 failed	0 of 8 failed	0 of 8 failed

9 FAQ

What license or licenses are you using for TRNG? TRNG is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License in version 2 as published by the Free Software Foundation.

Why is TRNG written in C++? C++ provides a lot of advanced features like inline functions and static polymorphism via templates that makes it possible to implement a fast, portable and easy to use library of PRNGs. Other languages, like FORTRAN or C, do no implement these (or comparable) features, are significantly slower, like Java or scripting languages, or not supported by fewer platforms, like C#.

How can I use TRNG in my FORTRAN programs? No you cannot. TRNG is implemented by 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? No you cannot. Here the same statements apply like in the last question. Fortunately 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, `unistd.h` 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. Now you have to compile your "C" program by a C++ compiler, but it can benefit the high quality parallel PRNGs of TRNG.

Where can I report bugs or make a feature request. Send bugs reports and a feature requests to heiko.bauke@physik.uni-magdeburg.de.

Bibliography

[1] Heiko Bauke and Stephan Mertens. Pseudo random coins show more heads than tails. *Journal of Statistical Physics*, 114(3):1149–1169, 2004.

[2] Heiko Bauke and Stephan Mertens. *Cluster Computing*. Springer, 2005.

[3] Boost C++ libraries. http://www.boost.org.

[4] 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.

[5] Aaldert Compagner. Definitions of randomness. *American Journal of Physics*, 59(8):700–705, August 1991.

[6] Aaldert Compagner. The hierarchy of correlations in random binary sequences. *Journal of Statistical Physics*, 63:883–896, 1991.

[7] 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.

[8] 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.

[9] Jay Fillmore and Morris Marx. Linear recursive sequences. *SIAM Review*, 10(3):342–353, 1968.

[10] GNU Scientific Library. http://www.gnu.org/software/gsl/.

[11] S. W. Golomb. *Shift Register Sequences*. Aegan Park Press, Laguna Hills, CA, revised edition, 1982.

[12] Peter Grassberger. On correlations in "good" random number generators. *Physics Letters A*, 181(1):43–46, 1993.

[13] A. Grube. Mehrfach rekursiv-erzeugte Pseudo-Zufallszahlen. *Zeitschrift für angewandte Mathematik und Mechanik*, 53:T223–T225, 1973.

[14] Dieter Jungnickel. *Finite Fields: Structure and Arithmetics*. Bibliographisches Institut, 1993.

[15] Scott Kirkpatrick and Erich P. Stoll. A very fast shift-register sequence random number generator. *Journal of Computational Physics*, 40(2):517–526, 1981.

[16] Donald E. Knuth. *The Art of Computer Programming*, volume 2: Seminumerical Algorithms. Addison Wesley Professional, 1st edition, 1969.

[17] Donald E. Knuth. *The Art of Computer Programming*, volume 2: Seminumerical Algorithms. Addison Wesley Professional, 3rd edition, 1998.

[18] Pierre L'Ecuyer. Random numbers for simulation. *Communications of the ACM*, 33(10):85–97, 1990.

[19] Pierre L'Ecuyer. A search for good multiple recursive random number generators. *ACM Transactions on Modeling and Computer Simulation*, 3(2):87–98, 1993.

[20] Pierre L'Ecuyer. Tables of linear congruential generators of different sizes and good lattice structure. *Mathematics of Computation*, 68:249–260, 1999.

[21] 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.

[22] Pierre L'Ecuyer. Random number generation. In James E. Gentle, Wolfgang Härdle, and Yuichi Mori, editors, *Handbook of Computational Statistics*. Springer, 2004.

[23] 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.

[24] 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.

[25] Rudolf Lidl and Harald Niederreiter. *Introduction to finite fields and their applications*. Cambridge University Press, 2nd edition, 1994.

[26] Rudolf Lidl and Harald Niederreiter. *Finite Fields*, volume 20 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, 2nd edition, 1997.

[27] George Marsaglia. Random numbers fall mainly in the planes. *Proceedings of the National Academy of Sciences*, 61:25–28, 1968.

[28] Michael Mascagni. Parallel linear congruential generators with prime moduli. *Prallel Computing*, 24(5–6):923–936, 1998.

[29] Michael Mascagni and Hongmei Chi. Parallel linear congruential generators with Sophie-Germain moduli. *Parallel Computing*, 30(11):1217–1231, 2004.

[30] 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.

[31] A. De Matteis and S. Pagnutti. A class of parallel random number generators. *Parallel Computing*, 13(2):193–198, 1990.

[32] A. De Matteis and S. Pagnutti. Long-range correlations in linear and non-linear random number generators. *Parallel Computing*, 14(2):207–210, 1990.

[33] Stephan Mertens and Heiko Bauke. Entropy of pseudo-random-number generators. *Physical Review E*, 69:055702–1–055702–4, 2004.

[34] MPICH2. http://www-unix.mcs.anl.gov/mpi/mpich.

[35] 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.

[36] M. E. J. Newman and G. T. Barkema. *Monte Carlo Methods in Statistical Physics*. Oxford University Press, 1999.

[37] Open MPI. http://www.open-mpi.org.

[38] Peter Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann Publishers Inc, 1996.

[39] 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.

[40] Ora E. Percus and Malvin H. Kalos. Random number generators for MIMD parallel processors. *Journal of Parallel and Distributed Computing*, 6:477–497, 1989.

[41] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, 2003.

[42] Random number generator test suite. http://www.comp-phys.org:16080/rngts/.

[43] Linus Schrage. A more portable fortran random number generator. *ACM Transactions on Mathematical Software*, 5(2):132–138, 1979.

[44] 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.

[45] Dietrich Stauffer and Ammon Aharony. *Introduction to Percolation Theory*. Taylor & Francis Ltd, 2nd edition, 1994.

[46] Robert H. Swendsen and Jian-Sheng Wang. Nonuniversal critical dynamics in monte carlo simulations. *Physical Review Letters*, 58:86–88, 1987.

[47] Zhe-Xian Wan. *Lectures on Finite Fields and Galois Rings*. World Scientific, 2003.

[48] Neal Zierler. Linear recurring sequences. *J. Soc. Indust. Appl. Math.*, 7(1):31–48, 1959.

[49] Robert M. Ziff. Four-tap shift-register-sequence random-number generators. *Computers in Physics*, 12(4), 1998.