Version: January 16, 2012

LatMRG

User's Guide

A software package for theoretical analysis of linear congruential random number generators and integration lattices ******

Pierre L'Ecuyer and Raymond Couture

Département d'Informatique et de Recherche Opérationnelle
Université de Montréal, C.P. 6128, Succ. Centre-Ville, Montréal, Canada, H3C 3J7

Abstract. *LatMRG* is a software toolkit for examining theoretical properties of linear congruential or multiple recursive random number generators. It is implemented as a library of modules written in *C++*. It offers tools to check whether a generator has maximal period or not, to apply the lattice and spectral tests (in large dimensions), and to perform computer searches for good (or bad) generators according to different criteria. One can analyse the lattice structure of points formed by successive values in the generator's sequence, or formed by "leapfrog" values. Generators with large moduli and multipliers (e.g. numbers of many hundreds of bits), as well as combined generators, can also be analyzed. Multiply-with-carry generators can also be studied by analyzing their corresponding linear congruential generators.

(**Keywords:** Random number generation; linear congruential generators; multiple recursive generators; multiply-with-carry generators; lattice structure; spectral test; integration lattices)

Contents

Chapter 1

Background and overview

LatMRG is a software system implemented as a library of classes written in the C++ language. It provides different tools for studying the structure of lattices in the real space and for examining the theoretical properties of random number generators based on linear recurrences in modular arithmetic or the properties of integration lattices (for now, Korobov and rank 1 lattices). It offers facilities for checking if a generator has maximal period or not, for examining its lattice structure (e.g., applying lattice and spectral tests), and for performing computer searches for "good" generators according to different quality criteria. The software can also be used for related applications, such as searching and evaluating lattice rules for quasi-Monte Carlo integration.

In this section, we give a quick recall of some definitions and notation, as well as a short outline of what the package does. For more details on the underlying theory and algorithms, see [6, 30] and other references given there. We classify the modules of *LatMRG* in three groups: (a) low-level, (b) intermediate-level, and (c) high-level. Higher-level programs import facilities from the lower-level ones.

The high-level modules (c) are programs in executable form which read their data from files. They can either analyze a given generator or seek "good" generators according to different criteria. Examples of data files and results are given in Section 2. Appendix A gives specifications of the data file formats and of what the programs do.

The intermediate-level classes (b) provide data types and methods to construct lattice bases for different classes of generators (simple or combined MRGs, lacunary indices, etc.), manipulate such bases, find a shortest vector in a lattice, reduce a basis in the sense of Minkowski, and so on. These tools are used by the upper-level programs (c), but can also be used directly to make programs different than those already provided at level (c), offering thus more flexibility. The lower-level classes (a) implement basic operations on scalars, vectors, matrices, polynomials, and so on. They allow different possible representations for these objects, depending, for example, on the size of the modulus m and the precision we want, as explained in Section 1.6. These lower-level tools are used by the modules of levels (b) and (c). The intermediate and low-level classes are discussed a little further in Section 3, and their specifications are given in appendices B and C.

1.1 Lattices in the real space

The *lattices* considered here are discrete subspaces of the real space $\mathbb{R}^t$, which can be expressed as

$$L_t = \left\{ \mathbf{v} = \sum_{j=1}^{t} z_j \mathbf{v}_j \mid \text{ each } z_j \in \mathbb{Z} \right\}, \tag{1.1}$$

where t is a positive integer, and $\mathbf{v}_1, \ldots, \mathbf{v}_t$ are linearly independent vectors in $\mathbb{R}^t$ which form a *basis* of the lattice. A comprehensive treatment of such lattices can be found in [4]. The matrix $\mathbf{V}$, whose ith line is $\mathbf{v}_i$, is the corresponding *generator matrix* of L_t. A lattice L_t shifted by a constant vector $\mathbf{v}_0 \notin L_t$, i.e., a point set of the form $L'_t = \{\mathbf{v} + \mathbf{v}_0 : \mathbf{v} \in L_t\}$, is called a *grid*, or a *shifted lattice*. The lattices considered in this guide always contain, or are contained in, the integer lattice $\mathbb{Z}^t$, i.e., $\mathbb{Z}^t \subseteq L_t$ or $L_t \subseteq \mathbb{Z}^t$.

The *dual lattice* of L_t is defined as $L_t^* = \{\mathbf{h} \in \mathbb{R}^t \mid \mathbf{h} \cdot \mathbf{v} \in \mathbb{Z} \text{ for all } \mathbf{v} \in L_t\}$. The *dual* of a given basis $\mathbf{v}_1, \ldots, \mathbf{v}_t$ is the set of vectors $\mathbf{w}_1, \ldots, \mathbf{w}_t$ in $\mathbb{R}^t$ such that $\mathbf{v}_i \cdot \mathbf{w}_j = \delta_{ij}$, where $\delta_{ij} = 1$ if $i = j$, and $\delta_{ij} = 0$ otherwise. It forms a basis of the dual lattice. These $\mathbf{w}_j$'s are the columns of the matrix $\mathbf{V}^{-1}$, the inverse of the matrix $\mathbf{V}$. If m is any positive real number, a basis $\{\mathbf{w}_1, \ldots, \mathbf{w}_t\}$ satisfying $\mathbf{v}_i' \mathbf{w}_j = \delta_{ij} m$ for all i, j is called the m-dual of the basis $\{\mathbf{v}_1, \ldots, \mathbf{v}_t\}$. The lattice generated by this m-dual basis is the m-dual to L. This extension of the usual notion of dual basis and dual lattice will allow us, by a suitable choice of m [in our context it will be the modulus in (1.2)], to deal uniquely with integer coordinate vectors, which can be represented exactly on a computer.

The determinant of the matrix $\mathbf{V}$ is equal to the volume of the fundamental parallelepiped $\Lambda = \{\mathbf{v} = \lambda_1 \mathbf{v}_1 + \cdots + \lambda_t \mathbf{v}_t \mid 0 \le \lambda_i \le 1 \text{ for } 1 \le i \le t\}$, and is also the inverse of the average number of points per unit of volume, independently of the choice of basis. It is called the determinant of L_t. The quantity $1/\det(L_t) = 1/\det(\mathbf{V}) = \det(\mathbf{V}^{-1})$ is called the *density* of L_t. When L_t contains $\mathbb{Z}^t$, the density is an integer equal to the cardinality of the point set $L_t \cap [0,1)^t$.

For a given lattice L_t and a subset of coordinates $I = \{i_1, \ldots, i_d\} \subseteq \{1, \ldots, t\}$, denote by $L_t(I)$ the projection of L_t over the d-dimensional subspace determined by the coordinates in I. This projection is also a lattice, whose density divides that of L_t. There are exactly $\det(L_t(I))/\det(L_t)$ points of L_t that are projected onto each point of $L_t(I)$. In group theory language, $L_t(I)$ corresponds to a coset of L_t.

1.2 Multiple recursive generators

Consider the linear recurrence

$$x_n = (a_1 x_{n-1} + \cdots + a_k x_{n-k}) \bmod m. \tag{1.2}$$

where m and k are positive integers and each a_i belongs to the set (or ring) $\mathbb{Z}_m = \{0, 1, \ldots, m-1\}$. For $n \ge 0$, $s_n = (x_n, \ldots, x_{n+k-1}) \in \mathbb{Z}_m^k$ is the *state* at step n. The initial state s_0 is called the *seed*. One can take $u_n = x_n/m \in [0,1)$ as the *output* at step n. This kind of generator is called *multiple recursive* (MRG). When $k = 1$, it gives the well-known multiplicative linear congruential generator (MLCG). MLCGs in *matrix form* can also be expressed as many copies of the same MRG running in parallel. For more details, see [15, 24, 25, 35].

The maximal possible period for the s_n's is the cardinality of $\mathbb{Z}_m^t$ minus 1, i.e. $\rho = m^k - 1$. It is attained if and only if m is prime and the characteristic polynomial of (1.2),

$$P(z) = \left(z^k - \sum_{i=1}^{k} a_i z^{k-i} \right) \bmod m, \tag{1.3}$$

is a primitive polynomial modulo m. Knuth [21] gives necessary and sufficient conditions for that, which are implemented in our package. If $k = 1$ and $m = p^e$, with $e > 1$, then the maximal possible period is 2^{e-2} for $p = 2$ and $(p-1)p^{e-1}$ for $p > 2$ [21, 24].

Instead of taking $u_n = x_n/m$ for the output, one can take a more general linear combination of the components of the state vector, say

$$y_n = (b_1 x_n + \cdots + b_k x_{n+k-1}) \bmod m, \tag{1.4}$$
$$u_n = y_n/m. \tag{1.5}$$

For any integer $t \geq 1$, one has

$$\begin{pmatrix} y_n \\ y_{n+1} \\ \vdots \\ y_{n+t-1} \end{pmatrix} = \begin{pmatrix} \mathbf{b}' \\ \mathbf{b}'\mathbf{A} \\ \vdots \\ \mathbf{b}'\mathbf{A}^{t-1} \end{pmatrix} \begin{pmatrix} x_n \\ x_{n+1} \\ \vdots \\ x_{n+k-1} \end{pmatrix} \bmod m \stackrel{\text{def}}{=} \mathbf{B}_t \begin{pmatrix} x_n \\ x_{n+1} \\ \vdots \\ x_{n+k-1} \end{pmatrix} \bmod m, \tag{1.6}$$

where $\mathbf{b}' = (b_1, \ldots, b_k)$ and

$$A = \begin{pmatrix} 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 1 \\ a_k & a_{k-1} & \ldots & a_1 \end{pmatrix}$$

is the *companion matrix* of the characteristic polynomial $P(z)$. In particular, by taking $t = k$, one sees that the vector $(y_n, \ldots, y_{n+k-1})$ takes all possible values in $\mathbb{Z}_m^k$, when $(x_n, \ldots, x_{n+k-1})$ does so, if and only if the matrix $\mathbf{B}_k$ has full rank. The matrix $\mathbf{B}_t$ can be constructed easily as follows. Put $(x_n, \ldots, x_{n+k-1})' = \mathbf{e}_j$, the jth vector of the canonical basis, with $x_{n+i-1} = \delta_{ij}$, and compute the corresponding column vector $(y_n, \ldots, y_{n+t-1})'$ via (1.2) and (1.4). This vector is the jth column of the matrix $\mathbf{B}_t$.

1.3 Lattice structure and spectral test

Let Ψ_t be the multiset of all the t-dimensional vectors of successive output values of an MRG, from all possible seeds in $\mathbb{Z}_m^k$, i.e.,

$$\Psi_t = \left\{ \mathbf{u}_{0,t} = (x_0/m, \ldots, x_{t-1}/m) \mid (x_0, \ldots, x_{k-1}) \in \mathbb{Z}_m^k \right\}.$$

For $t \leq k$, this set is just $\mathbb{Z}_m^t$ with each element repeated m^{k-t} times. For $t > k$, the first k components of a vector $\mathbf{u}_{0,t} \in \Psi_t$ are arbitrary elements of $\mathbb{Z}_m/m$, but once they are fixed, the remaining $t - k$ components are determined uniquely by the linear recurrence (1.2). The last

$t-k$ components are thus linear combinations modulo 1, with integer coefficients, of the first k components.

For $1 \le i \le k$, let $\mathbf{v}_i = (v_{i,1}, \ldots, v_{i,t})$ be the t-dimensional vector with components $v_{i,j} = \delta_{ij}/m$ for $i \le k$, and $v_{i,j} = (a_1 v_{i,j-1} + \cdots + a_k v_{i,j-k}) \bmod 1$ for $j > k$. For $k+1 \le i \le t$, let $\mathbf{v}_i = \mathbf{e}_i$, the ith unit vector in t dimensions. These vectors are a basis of a lattice L_t that contains $\mathbb{Z}^t$, with unit cell volume of $\max(m^{-t}, m^{-k})$, such that $L_t \cap [0,1)^t = \Psi_t$. In fact, $L_t = \Psi_t + \mathbb{Z}^t = \{\mathbf{v} = \tilde{\mathbf{v}} + \mathbf{z} \mid \tilde{\mathbf{v}} \in \Psi_t$ and $\mathbf{z} \in \mathbb{Z}^t\}$. The vectors $\mathbf{w}_i = (w_{i,1}, \ldots, w_{i,t})$, $1 \le i \le t$, where

$$w_{i,j} = \begin{cases} m & \text{for } j = i \le k; \\ 0 & \text{for } j \ne i \le k; \\ v_{j,i} & \text{for } i > k \ge j; \\ 1 & \text{for } j = i > k; \\ 0 & \text{for } k < j \ne i > k, \end{cases}$$

are linearly independent and satisfy $\mathbf{v}_i \cdot \mathbf{w}_j = \delta_{ij}$. They form the dual basis to $\{\mathbf{v}_1, \ldots, \mathbf{v}_t\}$. The vectors $\mathbf{v}_i$ and $\mathbf{w}_j$ are the lines of the matrices:

$$\mathbf{V}_t = (\mathbf{v}_1 \mathbf{v}_2 \cdots \mathbf{v}_t)' = \begin{pmatrix} 1/m & 0 & \ldots & 0 & v_{1,k+1} & \ldots & v_{1,t} \\ 0 & 1/m & \ldots & 0 & v_{2,k+1} & \ldots & v_{2,t} \\ \vdots & \vdots & \ddots & \vdots & \vdots & & \vdots \\ 0 & 0 & \ldots & 1/m & v_{k,k+1} & \ldots & v_{k,t} \\ 0 & 0 & \ldots & 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 0 & 0 & \ldots & 1 \end{pmatrix}$$

and

$$\mathbf{W}_t = (\mathbf{w}_1 \mathbf{w}_2 \cdots \mathbf{w}_t)' = \begin{pmatrix} m & 0 & \ldots & 0 & 0 & \ldots & 0 \\ 0 & m & \ldots & 0 & 0 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & & \vdots \\ 0 & 0 & \ldots & m & 0 & \ldots & 0 \\ -v_{1,k+1} & -v_{2,k+1} & \ldots & -v_{k,k+1} & 1 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ -v_{1,t} & -v_{2,t} & \ldots & -v_{k,t} & 0 & \ldots & 1 \end{pmatrix},$$

and one has $\mathbf{W}_t' \mathbf{V}_t = I$. In the package *LatMRG*, we store the vectors $m\mathbf{v}_i$ instead of $\mathbf{v}_i$ in the computer, for the components of the former vectors are integer-valued and can thus be represented exactly in the computer.

For the more general case of (1.4) and (1.5), replace the first k lines of $\mathbf{V}_t$ by $\mathbf{B}_t'$. If $\mathbf{B}_k$ is invertible, then $\mathbf{B}_t$ has rank k and the lines $k+1$ to t of $\mathbf{V}$ complete the lattice basis as before. Otherwise, remove the lines in $\mathbf{B}_t'$ which are linearly dependent of others, to obtain a matrix of full rank $k' < k$, and replace them by $k - k'$ vectors of the canonical basis of $\mathbb{R}^t$, divided by m, chosen in a way that the first k lines and k columns of $\mathbf{V}_t$ form an invertible matrix. In both cases, the dual basis is obtained by inverting the matrix $\mathbf{V}_t'$. For LCGs in matrix form, bases for L_t and its dual can be constructed as explained in [15, 30].

If one adds a constant b on the right-hand-side of (1.2), before applying the modulo operation, then the vectors of successive values will all belong to L'_t, where $L'_t = L_t + \mathbf{v}_{0,t}$ is a shift of L_t by some constant $v_{0,t} \in \mathbb{Z}_m^t$, i.e., a *grid*. Since L'_t and L_t have the same structural properties, we simply ignore the presence of such a constant b in *LatMRG*, and consider only homogeneous recurrences.

When m is prime and the MRG has full period $m^k - 1$, then Ψ_t is the set of all t-tuples produced by the generator over its main cycle, plus the $\mathbf{0}$ vector. Otherwise, the set of t-dimensional vectors produced over any given (sub)cycle (plus the $\mathbf{0}$ vector and plus $m\mathbb{Z}^t$) is a strict subset of L_t which in general does not form a lattice. Then, *LatMRG* can analyze the set of all t-tuples produced over the *union* of all subcycles. In some cases, however, the vectors of successive values over one subcycle generate a strict sublattice of L_t, whose intersection with $[0,1)^t$ contains only a fraction of the points of Ψ_t. This is what happens in particular when $k = 1$, m is a power of a prime p, and x_0 is prime to p. The package can take care of the latter case by constructing a basis for the appropriate sublattice.

1.4 Lacunary indices

Instead of forming vectors with successive values like in the above definition of Ψ_t, one can form vectors with values that are some distance apart in the sequence (so-called "leapfrog" values). Let $I = \{i_1, i_2, \cdots, i_t\}$ be a set of fixed integers. Define

$$\psi_t(I) = \left\{(u_{i_1}, \ldots, u_{i_t}) \mid (x_0, \ldots, x_{k-1}) \in \mathbb{Z}_m^k\right\} \tag{1.7}$$

and let $L_t(I) = \psi_t(I) + \mathbb{Z}^t$. If we assume that $0 \le i_1 < i_2 < \cdots < i_t$, this $L_t(I)$ is the projection of the lattice L_{i_t+1} over the t-dimensional subspace determined by the coordinates that belong to I. Using the class `IntLattice`, one can build a basis for L_t and its dual in this more general case, and then perform lattice analysis as usual. Further details and examples are given in [31]. For $(i_1, \ldots, i_t) = (0, \ldots, t-1)$, one has $L_t(I) = L_t$.

To construct the basis in this case, one must compute the vector $(u_{i_1}, \ldots, u_{i_t})$ obtained when the seed $(x_0, \ldots, x_{k-1}) = \mathbf{e}_i$, for each vector $\mathbf{e}_i$ of the canonical basis. The linear transformation from the state $(x_0, \ldots, x_{k-1})$ to the vector $(u_{i_1}, \ldots, u_{i_t})$ is one-to-one for each $t \ge k$ if and only if the transformation applied to the k vectors of the canonical basis gives k linearly independent vectors for $t = k$. For $t < k$, the transformation is onto (surjective) if and only if the transformation gives t linearly independent vectors, that is, if the corresponding matrix has full rank t.

1.5 Figures of Merit

Figures of merit measure the quality of lattices. Here, good quality means that the points cover the space very evenly, i.e., are very uniformly distributed. There are many ways of measuring this uniformity, which give rise to several different figures of merit.

1.5.1 The spectral test

The lattice structure also means that all points of L_t lie in a family of equidistant parallel hyperplanes. Among all such families of hyperplanes that cover all the points, choose the one for which the successive hyperplanes are farthest apart. The distance between these successive hyperplanes is in fact equal to $1/\ell_t$ where ℓ_t is the Euclidean length of the shortest nonzero vector in the *dual* lattice L_t^*. So for a given density of points, we want ℓ_t to be as large as possible. Computing this ℓ_t for an MRG and comparing with the best possible value, given t, m, and k, is known as the *spectral test* in the literature on RNGs [21, 12].

We can view the lattice as a way of packing the space by spheres of radius $\ell_t/2$, with one sphere centered at each lattice point. In the dual lattice, this gives $1/n = m^{-k}$ spheres per unit volume. If we rescale so that the radius of each sphere is 1, we obtain $\delta_t = (\ell_t/2)^t/n$ spheres per unit volume. This number δ_t is called the *center density* of the lattice. For a given value of n, an upper bound on ℓ_t can be obtained in terms of an upper bound on δ_t [one has $\ell_t = 2(n\delta_t)^{1/t}$], and vice-versa. Let δ_t^* be the largest possible value of δ_t for a lattice (i.e., the densest packing by non-overlapping spheres arranged in a lattice). The quantity $\gamma_t = 2(\delta_t^*)^{2/t}$ is called the *Hermite constant* for dimension t [4, 16]. It gives the upper bound $\ell_t^2 \le (\ell_t^*(n))^2 = 2(n\delta_t^*)^{2/t} = \gamma_t n^{2/t}$ for a lattice of density $1/n$. Knowing the Hermite constants, or good approximations of them, is useful because it allows us to normalize ℓ_t to a value between 0 and 1 by taking $\ell_t/\ell_t^*(m^k)$. This is convenient for comparing values for different values of t and m^k. Good values are close to 1 and bad values are close to 0.

The Hermite constants are known exactly only for $t \le 8$, in which case the densest lattice packings are attained by the *laminated* lattices [4]. Conway and Sloane [4, Table 1.2] give the values of δ_t^* for $t \le 8$, and provide lower and upper bounds on δ_t^* for other values of t. The largest value of $\ell_t^2/n^{2/t}$ obtained so far for concrete lattice constructions is a lower bound on γ_t, which we denote by γ_t^{B}. Such values are given in Table 1.2 of [4], page 15, in terms of δ^*. The laminated lattices, which give the lower bound $\ell_t^2/n^{2/t} \ge \gamma_t^{\mathrm{L}} = 4\lambda_t^{-1/t}$, where the constants λ_t are given in [3, Table 6.1, page 158] for $t \le 48$, are the best constructions in dimensions 1 to 29, except for dimensions 10 to 13. (One has $\gamma_t^{\mathrm{L}} = \gamma_t$ for $t \le 8$.)

Minkowski proved that there exists lattices with density satisfying $\delta_t \ge \zeta(t)/(2^{t-1}V_t)$ where $\zeta(t) = \sum_{k=1}^{\infty} k^{-t}$ is the Riemann zeta function and $V_t = \pi^{t/2}/(t/2)!$ is the volume of a t-dimensional sphere of radius 1. This bound provides a lower bound γ_t^{Z} on γ_t.

An upper bound on γ_t is obtained via the bound of Rogers on the density of sphere packings [4]. This upper bound can be written as

$$\gamma_t^{\mathrm{R}} = 4 * 2^{2R(t)/t}$$

where $R(t)$ can be found in Table 1.2 of [4] for $t \le 24$, and can be approximated with $O(1/t)$ error and approximately 4 decimal digits of precision, for $t \ge 25$, by

$$R(t) = \frac{t}{2}\,\log_2\left(\frac{t}{4\pi e}\right) + \frac{3}{2}\log_2(t) - \log_2\left(\frac{e}{\sqrt{\pi}}\right) + \frac{5.25}{t+2.5}. \tag{1.8}$$

Table 1 in [29] gives the ratio $(\gamma_t^{\mathrm{L}}/\gamma_t^{\mathrm{R}})^{1/2}$, of the lower bound over the upper bound on ℓ_t, for $1 \le t \le 48$. This ratio tends to decrease with t, but not monotonously.

Computing the shortest vector in terms of the Euclidean norm is convenient, e.g., for computational reasons, but one can also use another norm instead. For example, one can take the

$\mathcal{L}_p$-norm, defined by $\|\mathbf{v}\|_p = (|v_1|^p + \cdots + |v_t|^p)^{1/p}$ for $1 \le p < \infty$ and $\|\mathbf{v}\|_\infty = \max(|v_1|, \ldots, |v_t|)$ for $p = \infty$. The inverse of the length of the shortest vector is then the $\mathcal{L}_p$-distance between the successive hyperplanes for the family of hyperplanes that are farthest apart among those that cover L_t. For $p = 1$, the length $\ell_t = \|\mathbf{v}\|_1$ of the shortest vector $\mathbf{v}$ (or $\|\mathbf{h}\|_1 - 1$ in some cases, see [21]) is the minimal number of hyperplanes that cover all the points of Ψ_t. The following upper bound on ℓ_t in this case was established by Marsaglia [33] by applying the general convex body theorem of Minkowski:

$$\ell_t \le \ell_t^*(m^k) = (t! m^k)^{1/t} \stackrel{\text{def}}{=} \gamma_t^{\text{M}} m^{k/t}.$$

This upper bound can be used to normalize ℓ_t in this case.

— Upper bound on ℓ_t in general: Minkowski. ?

As a figure of merit, we take the worst-case value of $\ell_t/\ell_t^*(m^k)$ over certain values of t and for selected projections on lower-dimensional subspaces. More specifically, let ℓ_I denote the length of the shortest nonzero vector $\mathbf{v}$ in $L_t^*(I)$, and $\ell_t = \ell_{\{1,\ldots,t\}}$ as before. For arbitrary positive integers $t_1 \ge \cdots \ge t_d \ge d$, consider the worst-case figure of merit

$$M_{t_1,\ldots,t_d} = \min\left[\min_{k+1\le t\le t_1} \ell_t/\ell_t^*(m^k),\ \min_{2\le s\le k}\ \min_{I\in S(s,t_s)} \ell_I/m,\ \min_{k+1\le s\le d}\ \min_{I\in S(s,t_s)} \ell_I/\ell_s^*(m^k)\right], \qquad (1.9)$$

where $S(s,t_s) = \{I = \{i_1,\ldots,i_s\} \mid 1 = i_1 < \cdots < i_s \le t_s\}$. This figure of merit makes sure that the lattice is good in projections over t successive dimensions for all $t \le t_1$, and over non-successive dimensions that are not too far apart. Note that when $s \le k$, the smallest distance between hyperplanes that can be achieved in s dimensions for the MRG is $1/m$, so ℓ_s/m cannot exceed 1, and it is equal to 1 if and only if the linear transformation from the state $(x_0,\ldots,x_{k-1})$ to the output vector $(u_{i_1},\ldots,u_{i_s})$ is surjective (i.e., the corresponding matrix has full rank). For $s < k$, m is typically much smaller than $\ell_s^*(m^k)$, and this is the reason for separating the last two terms in (1.9).

The figure of merit $M_{t_1} = \min_{2\le s\le t_1} \ell_s/\ell_s^*(n)$ (with $d = 1$) has been widely used for ranking and selecting LCGs and MRGs [12, 28, 29]. The quantity $M_{t_1,\ldots,t_d}$ is a worst case over $(t_1 - d) + \sum_{s=2}^{d} \binom{t_s-1}{s-1}$ projections, and this number increases quickly with d unless the t_s are very small. For example, if $d = 4$ and $t_s = t$ for each s, there are 5019 projections for $t = 32$. When too many projections are considered, there are inevitably some that are bad, so the worst-case figure of merit is (practically) always small, and can no longer distinguish between good and mediocre behavior in the most important projections. Moreover, the time to compute $M_{t_1,\ldots,t_d}$ increases with the number of projections. We should therefore consider only the projections deemed important. We suggest using the criterion (1.9) with d equal to 4 or 5, and t_s decreasing with s.

Instead of considering the shortest nonzero vector in the dual lattice, one can consider the shortest nonzero vector in the primal lattice L_t. Its length represents the distance to the nearest other lattice point from any point of the lattice. A small value means that many points are placed on the same line, at some fixed distance apart.

1.5.2 Minkowski reduced basis

Another way of measuring the quality of a lattice is in terms of the relative lengths of the smallest and largest vectors in a *reduced* basis. A basis can be *reduced* in different senses. One type of

reduced basis considered by this package is a *Minkowski-reduced lattice basis* (MRLB) (see [1, 2, 15] for more details). Roughly, a MRLB is a basis for which the vectors are in some sense the most orthogonal. The ratio of the sizes of the shortest and longest vectors of a MRLB is called its *Beyer-quotient*. In general, a given lattice may have several MRLBs, all with the same length of the shortest vector, but perhaps with different lengths of the longest vector, and thus different Beyer quotients. We define $q_t(I)$ as the maximum of the Beyer quotients of all MRLBs of $L_t(I)$, and denote $q_t(\{1,\ldots,t\})$ by q_t. We prefer values of $q_t(I)$ close to 1. Similar to (1.9), we define

$$Q_{t_1,\ldots,t_d} = \min\left[\min_{k+1\le t\le t_1} q_t,\ \min_{2\le s\le d}\ \min_{I\in S(s,t_s)} q_t(I)\right]. \tag{1.10}$$

Computing q_t is much more time consuming than computing the spectral test.

1.5.3 The $\mathcal{P}_\alpha$ criterion

The quantity $\mathcal{P}_\alpha$ is a measure of non-uniformity (i.e., discrepancy from the uniform distribution, the smaller the better), which has been widely used in the context of quasi-Monte Carlo integration (see, e.g., [40]). In the case where $\Psi_t = L_t \cap [0,1)^t$ where L_t is a lattice with dual L_t^*, one has

$$\mathcal{P}_\alpha(\Psi_t) = \sum_{\mathbf{0}\neq\mathbf{w}\in L_t^*} \|\mathbf{w}\|_\pi^{-\alpha}, \tag{1.11}$$

where $\|\mathbf{w}\|_\pi = \prod_{j=1}^t \max(1,|w_j|)$ for $\mathbf{w} = (w_1,\ldots,w_t)$. For any positive integer α, $\mathcal{P}_{2\alpha}(\Psi)$ can be written equivalently as

$$\mathcal{P}_{2\alpha}(\Psi_t) = -1 + \frac{1}{n}\sum_{\mathbf{u}\in\Psi_t}\prod_{j=1}^t\left[1 - \frac{(-4\pi^2)^\alpha}{(2\alpha)!}B_{2\alpha}(u_j)\right] \tag{1.12}$$

where the B_α are the Bernoulli polynomials:

$$\begin{aligned} B_0(x) &= 1,\\ B_1(x) &= x - 1/2,\\ B_2(x) &= x^2 - x + 1/6,\\ B_3(x) &= x^3 - 3x^2/2 + x/2,\\ B_4(x) &= x^4 - 2x^3 + x^2 - 1/30, \end{aligned}$$

and the other polynomials can be found via the identity

$$\frac{te^{xt}}{e^t - 1} = \sum_{i=0}^\infty \frac{B_i(x)t^i}{i!}.$$

Hickernell [17] introduced generalizations of $\mathcal{P}_\alpha$, incorporating weights and replacing the simple sum in (1.11) by a more general norm. One version of this weighted $\mathcal{P}_\alpha$, where the weight associated to the projection over the coordinates in a set I has the *product-form* $\beta_I = \beta_0\prod_{j\in I}\beta_j$, can be defined by

$$\mathcal{P}_{2\alpha}(\Psi_t) = -\beta_0 + \frac{\beta_0}{n}\sum_{\mathbf{u}\in\Psi_t}\prod_{j=1}^t\left[1 - \frac{(-4\pi^2\beta_j^2)^\alpha}{(2\alpha)!}B_{2\alpha}(u_j)\right] \tag{1.13}$$

when α is an integer. The identity (1.12) or (1.13) gives an algorithm for computing $\mathcal{P}_{2\alpha}(\Psi_t)$ in time $O(nt)$ when α is an integer and Ψ_t is the intersection of a lattice with $[0,1)^t$. Note that the $D_{\mathcal{F},\alpha,p}(P)$ of [18] corresponds to $(\mathcal{P}_{2\alpha}(\Psi_t))^{1/2}$ for $\Psi_t = P$, $p = 2$, and $\beta_j = 1$ for all j. *LatMRG* provides tools for computing $\mathcal{P}_\alpha$ with or without weights.

It has been proved (e.g., [40], Theorem 4.4, page 83) that for any $t \geq 2$, $\alpha > 1$, and prime number $m > e^{\alpha t/(\alpha-1)}$, there exists at least one LCG with modulus m such that

$$\mathcal{P}_\alpha(\Psi_t) \leq \frac{[(e/t)(2\ln m + t)]^{\alpha t}}{m^\alpha}. \tag{1.14}$$

The latter quantity can then be used to normalize $\mathcal{P}_\alpha$.

Alternatively, Hickernell et al. [19], section 4.1, suggest using the figure of merit g_t, where

$$\begin{aligned} g_t^2 &= \frac{n^2}{(3/2)^t - 1}\left(\frac{t-1}{t-1+\log n}\right)^{t-1} \mathcal{P}_2(\Psi_t) \\ &= \frac{n}{(3/2)^t - 1}\left(\frac{t-1}{t-1+\log n}\right)^{t-1} \left[-n + \sum_{\mathbf{u}\in\Psi_t} \prod_{j=1}^{t} (1 + 3B_{2\alpha}(u_j))\right] \end{aligned} \tag{1.15}$$

is a normalized version of the inverse of (1.13) with $\alpha = 1$, $\beta_0 = 1$, and $\beta_j = \pi\sqrt{3/2}$ for $j \geq 1$. This g_t^2 can be rewritten as

$$g_t^2 = \gamma_t^{\mathrm{P}}(n)\left[-n + \sum_{\mathbf{u}\in\Psi_t} \prod_{j=1}^{t} (1 + 3B_{2\alpha}(u_j))\right] \tag{1.16}$$

where

$$\gamma_t^{\mathrm{P}}(n) = \frac{n}{(3/2)^t - 1}\left(\frac{t-1}{t-1+\log n}\right)^{t-1} \tag{1.17}$$

is a constant that depends on t and n. As a figure of merit based on $\mathcal{P}_2$, similar to (1.9), we define

$$G_{t_1,\ldots,t_d} = \min\left[\min_{k+1\leq t\leq t_1} 1/g_t(\Psi_t),\ \min_{k+1\leq s\leq d}\ \min_{I\in S(s,t_s)} 1/g_t(\Psi_t(I))\right]. \tag{1.18}$$

1.6 Matrix multiple recursive generators

MRGs in matrix form, which we denote MMRGs, have been introduced and studied by Niederreiter [36, 37]. The general recurrence has the form

$$\mathbf{x}_n = (A_1\mathbf{x}_{n-1} + \cdots + A_k\mathbf{x}_{n-k}) \bmod m \tag{1.19}$$

where k and m are the order and the modulus as for the MRG, $\mathbf{x}_n = (x_{n,1}, \ldots, x_{n,w})'$ is a w-dimensional vector, and each A_j is a $w \times w$ square matrix, for some positive integer w. The case $w = 1$ corresponds to the usual MRG. The recurrence (1.19) has full period $m^{kw} - 1$ if and only if m is prime and the characteristic polynomial

$$f(x) = \det\left(x^k I - x^{k-1}A_1 - x^{k-2}A_2 - \cdots - A_k\right)$$

is a primitive polynomial modulo m [36].

There are different ways of producing the output. We consider the following 3 cases:

$$u_{nw+i} = x_{n,i}/m \qquad \text{for } 0 \le i < w \text{ and } n \ge 0, \tag{1.20}$$

$$u_n = \frac{1}{m}\left(\sum_{i=1}^{w} b_i x_{n,i} \bmod m\right) = \sum_{i=1}^{w} b_i x_{n,i}/m \bmod 1 \qquad \text{for } n \ge 0, \tag{1.21}$$

$$u_n = \sum_{i=1}^{w} x_{n,i} m^{-i} \qquad \text{for } n \ge 0, \tag{1.22}$$

where $b_1, \ldots, b_w$ are positive integers. Case 3 is that used in [36] and does not give rise to a lattice structure for Ψ_t in the usual sense. For both cases 1 and 2, the set Ψ_t is the intersection of a lattice L_t with the unit hypercube. Case 1 is used in [37], where pseudorandom numbers are generated in vector form, w at a time. It is also explained in [37] how to construct a basis for the lattice L_t when t is a multiple of w. (The generalization to other values of t is trivial.)

— Give equivalence with MRG.

1.7 Multiply-with-carry

A *Multiply-with-Carry* (MWC) generator [5, 8, 23, 34] is based on the recurrence

$$x_n = (a_1 x_{n-1} + \cdots + a_k x_{n-k} + c_{n-1}) \bmod b, \tag{1.23}$$

$$c_n = (a_1 x_{n-1} + \cdots + a_k x_{n-k} + c_{n-1}) \operatorname{div} b, \tag{1.24}$$

$$u_n = x_n/b.$$

where "div" denotes the integer division. The recurrence looks like that of an MRG, except that a *carry* c_n is propagated between the steps.

Assume that b is a power of 2, define $a_0 = -1$,

$$m = \sum_{\ell=0}^{k} a_\ell b^\ell,$$

and let a be the inverse of b in arithmetic modulo m. For simplicity, assume $m > 0$. Then, up to precision $1/b$, the MWC generator is equivalent to the LCG:

$$z_n = a z_{n-1} \bmod m; \qquad w_n = z_n/m. \tag{1.25}$$

In other words, if

$$w_n = \sum_{i=1}^{\infty} x_{n+i-1} b^{-i} \tag{1.26}$$

holds for $n = 0$, then it holds for all n, and consequently $|u_n - w_n| \le 1/b$ for all n. The (approximate) lattice structure of the MWC can therefore be analyzed by analyzing that of the corresponding LCG (1.25). This is what the *LatMRG* package does.

If $a_\ell \geq 0$ for $\ell \geq 1$, then all the recurrent states of the MWC satisfy $0 \leq c_n < a_1 + \cdots + a_k$. In view of this inequality, we want the a_ℓ to be small, so that their sum fits into a computer word (e.g., $a_1 + \cdots + a_k \leq b$). But the coefficients should not be too small either, because in dimension $t = k + 1$, one has (see [8]):

$$\ell_t = (1 + a_1^2 + \cdots + a_k^2)^{1/2}. \tag{1.27}$$

Since b is a power of 2, a is a quadratic residue and so cannot be primitive mod m. Therefore the period length cannot reach $m - 1$ even if m is prime. But if $(m-1)/2$ is odd and 2 is primitive mod m (e.g., if $(m-1)/2$ is prime), then (1.25) has period length $\rho = (m-1)/2$.

1.8 Combined generators

Combining LCGs or MRGs with relatively prime moduli provides a efficient way of implementing linear recurrences based on larger (non-prime) moduli. The combination method that we consider adds, modulo 1, the outputs of the components. The package *LatMRG* permits one to specify a *product* MRG in terms of its component MRGs with relatively prime moduli (see class `MRGLatticeFactory)`. Its modulus is the product of the component moduli and its order is the maximum of the orders of the components. The recurrence governing this product MRG, when taken modulo any one of the component moduli, reduces to the component recurrence. The combined generator can then be studied via this product generator, since one can view the former as embedded in the latter, and since both have the same set of recurrent states (see [7]). Facilities are provided to analyze, for any given MRG, either the lattice L_t generated by all possible initial states, or that generated by the set of recurrent states (see `LatticeType` in module `Const` and in programs `seek*`).

Other types of combinations that have been proposed in the literature are (often, depending on the parameters) closely approximated by combinations of the above types [32, 26]. They can thus be analyzed with the present software.

1.9 Computing a shortest nonzero vector or a reduced basis

The class `Reducer` computes a shortest nonzero vector in a lattice via the branch-and-bound (BB) algorithm proposed by [10], with some additional refinements. For large dimensions t, this algorithm is much faster than the algorithm given in [9, 20]. The class also computes a Minkowski reduced basis via the algorithm of [1], which works by successive applications of the BB procedure for finding a shortest vector. The bounds in the BB procedure are computed through a Choleski decomposition performed in (double precision) floating-point arithmetic. Numerical roundoff errors occur during these computations and could (eventually) affect the results: Because of slightly wrong bounds in the BB, one may miss a shorter vector and, as a result, (conceivably) not obtain a true MRLB at the end of the reduction algorithm. In that case, one may consider redoing the computations with the NTL type `RR`, used to represent arbitrary-precision floating-point numbers (see module `Types`) but giving rise to much slower programs.

1.10 Large numbers, matrices and polynomials

LatMRG can deal with very large moduli and multipliers. There is no limit on size other than the size of the computer memory (and the CPU time). For example, a generator with a modulus of a few hundred bits can be analyzed easily. Operations on large integers are performed using the GNU multi-precision package GMP [14]. GMP is a portable library written in C for arbitrary precision arithmetic on integers, rational numbers, and floating-point numbers. For vectors, matrices of large numbers and polynomials, we use NTL [39]. NTL is a high-performance, portable C++ library providing data structures and algorithms for manipulating arbitrary length integers, and for vectors, matrices, and polynomials over the integers and over finite fields. NTL uses GMP as an underlying package for dealing with large numbers.

Of course, arithmetic operations with these structures are performed in software and are significantly slower than the standard operations supported by hardware. For this reason, most of the basic (low-level) operations required by our higher-level classes have been implemented in two or three versions. When building a basis or checking maximal period conditions, the modulus and multipliers can be represented either as `long`'s (32-bit integers) or `ZZ`'s (arbitrary large integers). After a lattice basis and its dual have been constructed, when working on the basis (finding a shortest vector, Minkowski reduction, etc.), the vector elements can be represented either as `double`'s (64-bit floating-point numbers) or `RR`'s (arbitrary large floating-point numbers).

When performing a search for good generators, for instance, one can first perform all the "screening" computations (involving many generators) using standard type `double`, and then recompute (verify) with the large floating-point numbers `RR` only for the retained generators.

Chapter 2

Using the executable programs

At the high-level end, `LatMRG` provides programs in executable form. These programs read their data in files. Appendix A describes their use in more detail. In this section, we give examples. The user can also tailor his own programs using the lower-level tools offered by the different modules of `LatMRG`. This is discussed in the next chapter.

The program `maxper` checks whether a given generator has maximal period (see the description on page 25). The program `findmk` can find values of m and k such that m and $r = (m^k - 1)/(m - 1)$ are prime numbers (see the description on page 27). The programs `latLLDD`, `latZZDD` and `latZZRR` perform standard lattice or spectral tests for a generator or a lattice (see the description on page 29). The only difference between the latter three programs is that in `latLLDD`, the multipliers and basis components are implemented in `long`'s and `double`'s, respectively; in `latZZDD`, the multipliers and basis components are implemented as large integers `ZZ`'s and `double`'s, respectively; while in `latZZRR`, the multipliers and the basis components are implemented as large integers `ZZ`'s and large floating-point numbers `RR`'s, respectively.

More comprehensive programs perform computer searches to seek the "best" generators of a given type, according to maximal period and lattice structure criteria. These programs also come in three similar versions: `seekLLDD`, `seekZZDD` and `seekZZRR`.

We now give a few concrete examples of data files and results.

2.1 An example with the program `latZZDD`

Figure 2.1 gives an example data file for the program `latZZDD`. The corresponding results appear in Table 2.1. To call the program and produce these results, type "`latZZDD fish31`", assuming that the data are in file "`fish31.dat`". The results will be in file "`fish31.tex`", which produces Table 2.1 after going through LaTeX. If `TEX` was replaced by `Terminal` in the data file, the results would be displayed on the screen. See the description of the program `latZZDD` for more details on how to set up the data files. Note that the first column in the data file gives the data values themselves, while the column on the right contains comments describing the meaning of these values. In that example, the spectral test is applied to the LCG of order 1 with $m = 2^{31} - 1 = 2147483647$ and $a = 742938285$, suggested by Fishman and Moore [13]. The last column indicates the (cumulative) cpu time. The total cpu time to compute all the distances d_t between hyperplanes in dimensions 2 to 30 was approximately 0.37 seconds.

`Spectral BestLat`	Test and Normalizer
`L2NORM`	Norm
`false`	Read generator from file
`1`	J (number of components)
`MRG`	Generator type
`2147483647`	m (modulus)
`1`	k (order)
`NoCond`	Conditions on a
`742938285`	a (multiplier)
`1`	Max dimension of projections
`2 28`	MinDim MaxDim
`true`	Dual lattice
`full`	Lattice Type
`1 1`	Lacunary indices
`10000000`	MaxNodesBB
`true`	Invert flag
`res`	Output Form

Figure 2.1: Example of a data file for the program `latZZDD`, in file `fish31.dat`.

Table 2.1: Results of `latZZDD` for file `fish31.dat`.

t	d_t	S_t	CPU (sec)
2	2.3156E-05	0.86725	0
3	0.00080231	0.86068	0
4	0.0045279	0.8627	0
5	0.01328	0.83195	0
6	0.025863	0.83415	0
7	0.0553	0.62392	0
8	0.068199	0.70666	0
9	0.106	0.61277	0
10	0.10847	0.74947	0
11	0.16903	0.57339	0
12	0.24254	0.4527	0
13	0.24254	0.51436	0.01
14	0.24254	0.56322	0.01
15	0.24254	0.60586	0.02
16	0.24254	0.64004	0.02
17	0.24254	0.68563	0.02
18	0.25	0.70148	0.03
19	0.26726	0.68601	0.04
20	0.26726	0.70891	0.04
21	0.26726	0.73029	0.06
22	0.28868	0.69009	0.09
23	0.28868	0.70131	0.10
24	0.30151	0.67739	0.13
25	0.30151	0.71188	0.16
26	0.30151	0.74118	0.19
27	0.30151	0.76362	0.22
28	0.30151	0.78104	0.26

```
Spectral   Laminated            Test <Normalizer>
FALSE                           Read generators from file
1                               J
 MRG                             Gener. type
 32749  1 0                     Modulo m
 2                               Order k
 TRUE                            Maximal period
 Decomp                          Factors of m-1
 Write seek15r.fac               Factors of r  (File2)
 NoCond                          Implem. condition
 1                               b1
 180                             c1
 -180                            b2
 -1                              c2
 Exhaust                         Search method: exhaustive search
1                               NbCat
3 8                             Dim(0)  Dim(1)
0.2                             Min merit values
1.0                             Max merit values
2                               Nb of retained generators
Full                            Lattice type (analyzed)
1  1                            Lacunary ind: group sizes,  spacing
1000000                         Max num of nodes in each BB
1 m                             Time limit: 1 minute
12345                           S1: seed for the random number generator
RES                             Output in File
```

Figure 2.2: Example of a data file for the program `seekLLDD`, in file `seek15.dat`.

2.2 Examples with the programs `seekLLDD` and `seekZZDD`

Figure 2.2 gives an example of a data file for `seekLLDD`. It will perform an exhaustive search among the 32400 generators of order $k = 2$ with modulus $m = 32749$, $1 \leq a_1 \leq 180$, and $-180 \leq a_2 \leq -1$. The criterion is M_8. The two best generators will be retained, provided their values of M_8 are at least 0.2. The values of $m - 1$ and r will be decomposed by the program. The factors of r will be written in file "`seek15r.fac`", while those of $m - 1$ will not be kept. The results of that program appear in Figure 2.3 (this is an actual printout, for illustration).

Figure 2.4 gives another data file example, this time for `seekZZDD`. It asks for a random search for good generators of order $k = 5$ with modulus $m = 2^{63} - 711$, $1 \leq a_1 \leq 2^{63} - 712$, $a_2 = a_3 = a_4 = 0$, $-2^{63} + 712 \leq a_5 \leq -1$, for which $a_i(m \bmod a_i) < m$ for $i = 1, 5$, and which have maximal period. The criterion is S_{12}. Only the best generator will be retained. The factorization of $m - 1$ will be read in file "`seek63.fac`" and r is prime. For the random search, we will examine 1000 subregions of dimensions $(10 \times 1 \times 1 \times 1 \times 10)$, that is, a total of 100000 generators (10000 values of a_5), if time permits. We give the program a cpu time-limit of 3 hours. A partial view of the results file is given in Figure 2.5.

Note that we are not recommending any of these particular generators. These examples are only to illustrate the capabilities of LatMRG.

```
 SEARCH for good MRGs of order 2
-----------------------------------------
DATA
-----------------------------------------
Component 1
   Modulus m              : 32749
   Order k                : 2
   Factors of m-1         :
   Factors of r           :

   Search method                       : EXHAUST
   Bounds : a1 from : 1
                 to : 180
            a2 from : -180
                 to : -1
   Implementation condition            : NO_COND
   Maximum period required             : true
-----------------------------------------

   Merit criterion                     : M_8
   Seed for RNG                        : 12345
   Max nodes in branch-and-bound       : 1000000
   Lattice Type                        : FULL

RESULTS
-----------------------------------------
    Values of a2 tried                   : 5873
    ...
    Nb. of polynomials to examine        : 5873
    Nb. Generators conserved             : 2
    Total CPU time (after setup)         = 0:0:3.3

 +-----------------------------------------------------------------
 | 2 Generators retained for criterion M_8
 +-----------------------------------------------------------------
 | M                   K               A                       Merit
 +-----------------------------------------------------------------
 | 32749               2               [ 180 -176 ]            0.21911
 +-----------------------------------------------------------------
 | 32749               2               [ 180 -175 ]            0.2185
 +-----------------------------------------------------------------
```

Figure 2.3: Results of program `seekl` in file `seek15.res`.

```
Spectral   BestLat              Test <Normalizer>
FALSE                           Read generators from file
1                               J
 MRG                             Gener. type
 2  63 -711                      Modulo m
 5                               Order k
 TRUE                            Maximal period
 Read  seek63.fac                Factors of m-1 in file
 Prime                           Factors of r
 AppFact                         Implem. condition
 1                               b_1
 2  63  -712                    c_1
 0
 0
 0
 0
 0
 0
 -2  63  -712
 -1
 Random 1000 10 10              Search method: random search
1                               NbCat
6  12                           Dim(0)  Dim(1)
0.0                             Min merit values
1.0                             Max merit values
1                               Nb of retained generators
Full                            Lattice type (analyzed)
1  1                            Lacunary ind. group sizes,  spacing
1000000                         Max nb of nodes in each BB
3 h                             Time limit: 3 hours
12345                           S1: seed for the random number generator
RES                             Output in File
```

Figure 2.4: Example data file for program `seekZZDD`, in file `seek63.dat`.

```
 SEARCH for good MRGs of order 5
-----------------------------------------
DATA
-----------------------------------------
Component 1
   Modulus m              : 9223372036854775097
   Order k                : 5
   Factors of m-1         :
   Factors of r           :

   Search method                       : RANDOM
   Bounds : a1 from : 1
                 to : 9223372036854775096
            a2 from : 0
                 to : 0
            a3 from : 0
                 to : 0
            a4 from : 0
                 to : 0
            a5 from : -9223372036854775096
                 to : -1
   Implementation condition            : APP_FACT
   Maximum period required             : true
-----------------------------------------

   Merit criterion                     : M_8
   Seed for RNG                        : 98765
   Max nodes in branch-and-bound       : 1000000
   Lattice Type                        : FULL

RESULTS
-----------------------------------------
    Values of a5 tried                    : 1469
    ...
    Nb. Generators conserved              : 1
    Total CPU time (after setup)          = 0:4:4.7

 +-------------------------------------------------------------------
 | 1 Generators retained for criterion M_12
 +-------------------------------------------------------------------
 | M                    K      A                                  Merit
 +-------------------------------------------------------------------
 | 9223372036854775097  5      [2649706710257 0 0 0 -1963898772] 0.00032238
 +-------------------------------------------------------------------
```

Figure 2.5: Results of program `seekZZDD` in file `seek63.res`.

Example 2.6 is similar to example 2.4, except that this time, the multipliers must be equal by groups: the first three multipliers must all be equal, and the last two are always equal. We once again do a random search with 1000 regions.

```
Spectral   BestLat              Test <Normalizer>
FALSE                          Read generators from file
1                              J
 MRG                            Generator type
 2  63 -711                     Modulus m
 5                              Order k
 TRUE                           Maximal period
 Read  seek63.fac               Factors of m-1 in file
 Prime                          Factors of r
 Equal 2 3                      Implem. condition
 1                              b_1, b_2, b_3
 2  31  -1                      c_1, c_2, c_3
 -2  31  -1                     b_4, b_5
 -1                             c_4, c_5
 Random 1000 10 10             Search method: random
1                              Num. Categ.
6  12                          Dim_0  Dim_1
0.0                            Min merit values
1.0                            Max merit values
1                              Num. of retained generators
Full                           Lattice type
1  1                           Lacunary ind. group sizes,  spacing
1000000                        Max nb of nodes in each BB
3 h                            Time limit: 3 hours
12345                          S1: seed for the random number generator
RES                            Output in File
```

Figure 2.6: Example data file for program `seekZZDD`, in file `seek63b.dat`.

```
 SEARCH for good MRGs of order 5
-----------------------------------------
DATA
-----------------------------------------
Component 1
   Modulus m              : 9223372036854775097
   Order k                : 5
   Factors of m-1         :
   Factors of r           :

   Search method                       : RANDOM
   Bounds : a1 from : 1
                 to : 2147483647
            a2 from : 1
                 to : 2147483647
            a3 from : 1
                 to : 2147483647
            a4 from : -2147483647
                 to : -1
            a5 from : -2147483647
                 to : -1
   Implementation condition            : EQUAL_COEF
   Maximum period required             : true
-----------------------------------------

   Merit criterion                     : M_8
   Seed for RNG                        : 12345
   Max nodes in branch-and-bound       : 1000000
   Lattice Type                        : FULL

RESULTS
-----------------------------------------
    Values of a5 tried                    : 8955
    ...
    Nb. of polynomials to examine         : 8955
    Nb. Generators conserved              : 1
    Total CPU time (after setup)          = 0:5:19.7
  ------------------------------------------------------------------------
 +--------------------------------------------------------------------
 | 1 Generators retained for criterion M_12
 +--------------------------------------------------------------------
 | M                    K        A                                 Merit
 +--------------------------------------------------------------------
 | 9223372036854775097  5      [2102602298 2102602298 2102602298 -2112400139
-2112400139]          5.7309e-07
 +--------------------------------------------------------------------
```

Figure 2.7: Results of program `seekZZDD` in file `seek63b.res`.

Chapter 3

Making your own programs

`LatMRG` offers more flexibility than just providing a set of executable programs. One can also use the modules provided to write one own's programs. For that, a C++ compiler and some knowledge about the C++ language are required. Instead of providing the parameters according to the format of Appendix A, [1] one sets those parameters (or data) by setting the appropriate variables or creating the appropriate objects. In this section, we introduce briefly the classes and low-level modules, which are described in Appendix B and C, [2] respectively. It is important to recall that the multiplier's and basis components can be implemented with different representations and that one must make sure to select the appropriate representation for the target application. See Section 3.2 [3] for more details on this.

3.1 Using the classes of `LatMRG`

The modules of `LatMRG` (excluding the executable programs) have been classified in two sets: lower-level and intermediate-level. The lower-level modules offer basic facilities for arithmetic operations and conversions, with different representations, for basis vectors and multipliers. They are described in the NTL documentation (see the URL `file:///u/simardr/ntl-5.4/doc/tour.html`).

The intermediate-level classes inheriting from the virtual class `IntLattice` constructs lattice bases for different kinds of generators or point sets. The currently implementing classes are `Rank1Lattice`, `KorobovLattice` and `MRGLattice`. `IntLattice` itself offers tools for manipulating lattice bases, and generally does common operations on bases. The class `Reducer` performs tests on these lattices, such as finding the shortest vector in a lattice, and reducing a basis in the sense of Minkowski. Those classes are described in the following chapters. The programs described in Chapter 2 use those intermediate and lower-level classes in their implementation, and so, provide examples of how to use them.

[1] From Richard: indiquer la bonne section
[2] From Richard: idem
[3] From Richard: idem

3.2 Lower-level modules and Changing the representation

As discussed in section 1.10, the multiplier's components can be implemented in the `long` or `ZZ` representation, while the basis components can be in the `double` or `RR` representation. To select the appropriate representation (see modules `Types` where all the basic possible types are selected), one should compile and link his programs with one of the libraries `liblatLLDD.a`, `liblatZZDD.a` or `liblatZZRR.a`. Generally speaking, the proper choice of representation depends on the size of the modulus m. For example, if m is less than 2^{25}, then the library `liblatLLDD.a` should be appropriate. For most cases, the library `liblatZZDD.a` will be satisfactory.

Chapter 4

Executable programs

The next few modules describe precompiled programs that can be executed directly by following the given instructions.

MaxPeriod

This module contains the main program to determine whether a given generator has maximal period or not. The MRG generator has the form

$$x_n = (a_1 x_{n-1} + \cdots + a_k x_{n-k}) \bmod m.$$

The modulus m must be a prime number. To verify the conditions for maximal period, the factorizations of $m-1$ and $r = (m^k - 1)/(m-1)$ are required. They can be found by the program or provided by the user in a file. The user must be aware that factoring r can take a *huge* amount of time for large integers. Integers are represented using the `ZZ` type from NTL.

The program is called `maxper` and reads the parameters of the generator from a file. The data file must have the extension `.dat`. To run the program, the name of the data file must be given without extension on the command line. For example, if the data file is called `maxp1.dat`, then the program is run by calling

```
maxper maxp1
```

The data file must have the format displayed in Figure 4.1.

# This is a comment line		
GenType		
m		modulus
k		order
Decom1	[*file1*]	factorization of $m-1$
Decor	[*file2*]	factorization of r
a_1		coefficient
$\vdots$		
a_k		coefficient

Figure 4.1: Data file format for `maxper`.

The file must contain the following parameters in that order:

GenType: for now, only `MRG` is allowed for *GenType*.

m: the modulus of congruence of the MRG.

k: the order of the MRG.

Decom1 **and** [*file1*]: refers to the prime number decomposition of $m-1$. *Decom1* indicates how the factors of $m-1$ are to be found, and *file1* is an optional file name. The possible values for *Decom1* are `Decomp, Write, Read, Prime`. The meaning of the values are:

`Decomp`: the program itself will factorize $m-1$. In this case, the field *file1* is unused and can be omitted. To factorize, the program uses the MIRACL software [38]. It is the responsibility of the user to make sure that the factorization will take a reasonable time.

`Write:` means the same as `Decomp`, except that the program will also write the prime factors found in file *file1*.

`Read:` $m-1$ is already factorized and the factors will be read from file *file1*. The prime factors must be given as described in class `IntFactorization` (see page 93 of this guide).

`Prime:` the number is a prime number (cannot occur for $m-1$ but is possible for r).

Decor **and** [*file2*]**:** refers to the prime number decomposition of r. The meaning of the fields is similar to the description above for $m-1$.

$a_1, \ldots, a_k$**:** the coefficients of the MRG must be given each on a separate line.

FindMK

This module contains the main program to search for all prime integers m in a given interval, i.e. such that $2^e + c_1 \leq m \leq 2^e + c_2$ and for which $r = (m^k - 1)/(m - 1)$ is also prime. We may also require that $(m - 1)/2$ be prime by setting the boolean variable *Safe* to `true`. If the boolean *Facto* is `true`, the program will factorize $m - 1$ and write the factors in a file with extension `.fac`. (See class `Primes` on page 43 for the basic method called by the program.)

The program is called `findmk`. The program reads the search parameters from a file. This data file must have the extension `.dat`. To run the program, the name of the data file must be given without extension on the command line, and the results will be written in a file of the same name but with extension `.res`. For example, if the data file is called `find1.dat`, then the program is run by calling

```
findmk find1
```

and the results will be written in file `find1.res`. If factorization of $m - 1$ is required, the factors will be written in file `find1.fac`.

The data file must have the format displayed in Figure 4.2.

# This is a comment line	
k	order
e	
c_1	
c_2	
Safe	a boolean
Facto	a boolean

Figure 4.2: Data file format for `findmk`.

The file must contain the following parameters in that order:

k**:** the order of the MRG.

e**:** the modulus of congruence will be close to 2^e.

c_1**:** gives the lower limit for m as $m \geq 2^e + c_1$.

c_1**:** gives the upper limit for m as $m \leq 2^e + c_2$.

*Safe***:** if `true`, only m such that $(m - 1)/2$ is prime are considered.

*Facto***:** if `true`, the program will factorize $m - 1$ and write the factors in a file with extension `.fac`.

FindMK2

This module contains the main program to search for a fixed number of prime integers m close to a power of 2 (such that $m < 2^e$), and for which $r = (m^k - 1)/(m - 1)$ is also prime. We may also require that $(m - 1)/2$ be prime by setting the boolean variable *Safe* to `true`. (See class `Primes` on page 43 for the basic method called by the program.)

The program is called `findmk2`. The program reads the search parameters from a file. This data file must have the extension `.dat`. To run the program, the name of the data file must be given without extension on the command line, and the results will be written in a file of the same name but with extension `.res`. For example, if the data file is called `find2.dat`, then the program is run by calling

```
findmk2 find2
```

and the results will be written in file `find2.res`.

The data file must have the format displayed in Figure 4.3.

```
   # This is a comment line
k          order
e
s
Safe       a boolean
```

Figure 4.3: Data file format for `findmk2`.

The file must contain the following parameters in that order:

k**:** the order of the MRG.

e**:** the modulus of congruence will be close to 2^e.

s**:** the program will find the first s values of m closest to 2^e and such that $m < 2^e$.

*Safe***:** if `true`, only m such that $(m - 1)/2$ is prime are considered.

LatMain

This is the main program for the lattice tests. Depending on the parameters in the data file, it runs either the spectral, the Beyer, or the P_α tests and compute the figure of merit for the given lattice (see classes `LatTestSpectral`, `LatTestBeyer`, and `LatTestPalpha` for details and the form of the data file). The program reads the name of the data file from the command line. The data file must have the extension `.dat`. To run the program, the name of the data file must be given without extension on the command line. For example, in the `LLDD` case (see below), if the data file is called `lat1.dat`, then one must call

```
latLLDD lat1
```

Depending on the parameters in the data file, the results will be sent on the terminal or in a file with the same name as the data file, but with extension `.res` or `.tex`. In the above example, the results would be in either `lat1.res` or `lat1.tex`.

NEW: It is now possible to apply the tests on several data files in one call. One must use the command line

```
latLLDD file1 file2 ...
```

If the parameter *Output Form* below is set to `Res` in a data file, then the results will be sent to a `.res` file corresponding to each data file name.

NEW: It is also possible to apply the tests on all the data files with extension ``.dat'' in one or more directories in one call. One must use the command line

```
latLLDD dir1 dir2 ...
```

The files without extension ``.dat'' in the directories will be disregarded. If the parameter *Output Form* below is set to `Res` in a data file, then the results will be sent to a `.res` file corresponding to each data file name.

Three different executable programs have been compiled from `LatMain`, depending on the types of `BScal`, `MScal`, `NScal`, `RScal` (the definition of these types is given in module `Types` on page **??**). They are named `latLLDD`, `latZZDD`, and `latZZRR`. The types are as defined in the following table:

	BScal	MScal	NScal	RScal
latLLDD	long	long	double	double
latZZDD	ZZ	ZZ	double	double
latZZRR	ZZ	ZZ	RR	RR

`ZZ` and `RR` are the big integers and the big floating-point numbers defined in NTL. The program `latLLDD` is the fastest but works only when m, the number of points of the lattice, is not too large. The program `latZZRR` is very slow, but works for arbitrary large numbers. `latZZDD` is much faster than `latZZRR`. Thus one should work with `latLLDD` if possible, and if not, with `latZZDD`, and otherwise with `latZZRR` as the last choice.

The data must be in a file with extension `.dat`. Lines whose first non-blank character is a `#` are comments, and are dropped by the reader program. The data fields have the following meaning: For the spectral and Beyer tests, the format must be as given in Figure 4.4. For the Palpha test, the format must be as given in Figure 4.5 below.

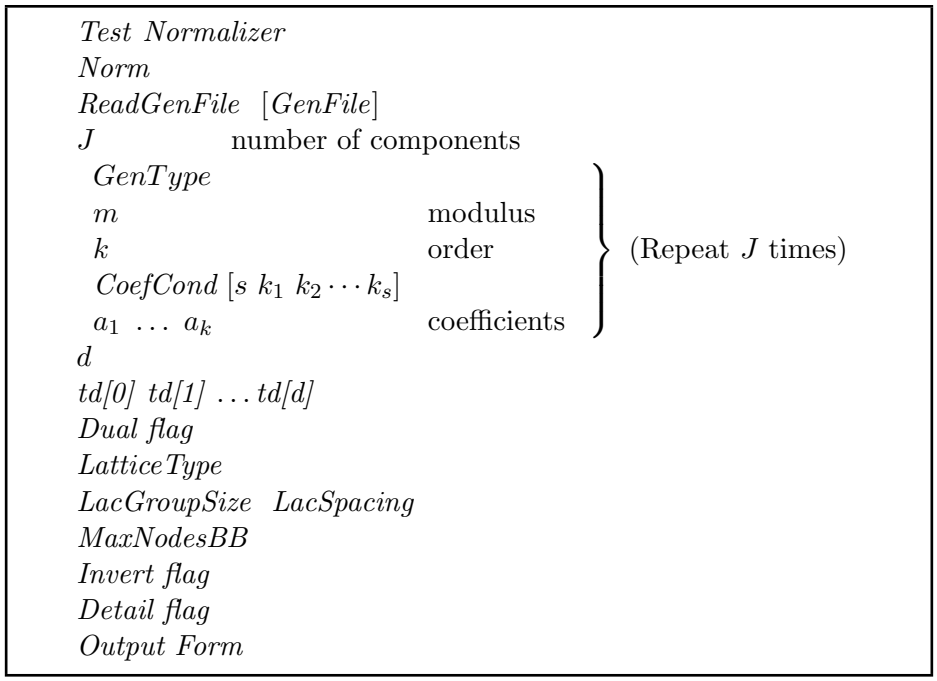

Figure 4.4: Data file format for Beyer and spectral tests.

Test ⟨***Normalizer***⟩**:** The lattice test can be one of `Spectral`, `Beyer` or `Palpha`. The normalizer depends on which lattice test is performed. For the `Spectral` test, it may be one of `BestLat`, `Laminated`, `Rogers`, `Minkowski` or `MinkL1`, otherwise it may be left blank.

Norm**:** To measure the length of vectors. Can be `L1NORM` or `L2NORM`.

ReadGenFile ⟨***Genfile***⟩**:** `boolean` and file name (without extension). When *ReadGenFile* is `false`, the search is made according to the values of the fields below. When `true`, the generators to be tested are those listed in the file ⟨*Genfile*⟩`.gen`.

J**:** Number of components in the combined generator.

GenType**:** Type of generator. For now, the possibilities are:

`MRG` means that this component is an MRG.

`MWC` means that this component is a multiply-with-carry (MWC) generator. Each MWC generator is converted by the program to its corresponding LCG (see, e.g., [8, 27]).

`KOROBOV:` means that this component is a Korobov lattice.

`RANK1:` means that this component is a rank 1 lattice.

m, k**:** Modulus and order of the recurrence. Must be positive integers.

CoefCond ⟨s k_1 $k_2 \cdots k_s$⟩**:** Conditions on the coefficients. The possible cases are:

`NonZero` s k_1 k_2 $\cdots$ k_s**:** all the coefficients a_j are 0, except for s of them: the non-zero coefficients are a_{k_1}, a_{k_2}, ..., a_{k_s}, and their values are given on the following line. For example, for a MRG of order $k = 10$, the line "`NonZero` 4 2 5 8 10" means that the

vector of coefficients is $\mathbf{a} = (0, \alpha, 0, 0, \beta, 0, 0, \gamma, 0, \delta)$ where $\alpha, \beta, \gamma, \delta$ are the s non-zero coefficients given on the following line.

`Equal` s k_1 k_2 $\cdots$ k_s**:** The coefficients are equal by groups. There are s groups: the first group of k_1 coefficients are all equal, the second group of $k_2 - k_1$ coefficients are all equal, and so on until the last group of $(k_s - k_{s-1})$ coefficients. k_j is the vector index of the last element of group j. For example, for a MRG of order $k = 10$, the line "`Equal` 4 2 5 8 10" will give a vector of coefficients of the form $\mathbf{a} = (\alpha, \alpha, \beta, \beta, \beta, \gamma, \gamma, \gamma, \delta, \delta)$ where $\alpha, \beta, \gamma, \delta$ are s coefficients given on the following line.

`NoCond`**:** There is no condition on the coefficients and they are all given on the following line.

$(a_1, a_2, \ldots, a_k)$**:** the vector of (integer) multipliers all given on one line.

***d*:** The number of kinds of projections. The standard case has $d = 1$ and the test will be run for all successive dimensions from *td[0]* = *MinDim* to *td[1]* = *MaxDim*.

***td[0] td[1] ... td[d]*:** The test will be run for all successive dimensions from *td[0]* to *td[1]*, then for all 2-dimensional projections for dimensions up to *td[2]* (if $d \geq 2$), for all 3-dimensional projections for dimensions up to *td[3]* (if $d \geq 3$), ..., and for all d-dimensional projections for dimensions up to *td[d]*. The simplest case has $d = 1$.

***Dual flag*:** `true` if the dual lattice is analyzed; `false` for the primal lattice.

***LatticeType*:** Indicates whether to analyze the lattice generated by all possible states, or a sublattice generated by the set of recurrent states or by a subcycle of the generator. The admissible values are `(Full, Recurrent, Orbit, PrimePower)`.

`Full`**:** The complete lattice, generated by all possible initial states, will be analyzed.

`Recurrent`**:** If the (combined) generator has transient states, then the lattice analyzed will be the sublattice generated by the set of recurrent states.

`Orbit`**:** The grid generated by the (forward) orbit of a state of the (combined) generator is analyzed. This state is specified as follows. On the following J lines, the initial state for each component must be given. This is an integer vector with a number of components equal to the order of the component.

`PrimePower`**:** In the case where some component is an MLCG whose modulus is a power of a prime p, then the states visited over a single orbit (subcycle) of that component generate a sublattice (when $a \equiv 1 \pmod{p}$) or belong to the union of $p - 1$ sublattices (otherwise). If *LatticeType* takes this value, if a component is an MLCG ($k = 1$), and if the modulus of that MLCG is given in the data file in the form (b): $(x\ y\ z)$ with $z = 0$ and x prime, then what is analyzed is one of those sublattices. This is done by dividing the modulus by the appropriate power of p, as described in [30]. For example, if $p = 2$ and $a \bmod 8 = 5$, then the modulus is divided by 4 as in [11, 20].

***LacGroupSize LacSpacing*:** These data fields are positive integers, used to introduce lacunary indices. If the respective values are s and d, then we will analyze the lattice structure of vectors of the form $(u_{i+1}, \ldots, u_{i+s}, u_{i+d+1}, \ldots, u_{i+d+s}, u_{i+2d+1}, \ldots, u_{i+2d+s}, \ldots)$, formed by groups of s successive values, taken d values apart. To analyze vectors of successive values (as usual), take $s = d = 1$ or s larger or equal to *MaxDim*. To analyze lacunary indices that are not evenly spaced, put $s = -t$ where $t =$*MaxDim* and then, on the t lines that follow, give the t lacunary indices $i_1, \ldots, i_t$, which are interpreted as in Section 1.4.

***MaxNodesBB*:** An integer giving the maximum number of nodes to be examined in any given branch-and-bound procedure when computing d_t or q_t. When that value is exceeded, the branch-and-bound is stopped and the generator is rejected. The number of generators rejected for that reason is given in the results. A small value of *MaxNodesBB* will make the program run faster (sometimes much faster), permitting to examine more generators, but will increase the chances of rejecting good generators.

***Invert flag*:** If `true`, the inverse of the length of the shortest vector will be printed in the results, otherwise the length itself is printed.

***Detail flag*:** The default value of the flag is 0. If it is > 0, extra details are printed in the results. If the flag is 1, the shortest vector of the basis is printed. If the flag is 2, all the vectors of the final basis are printed. If the flag is 3, all the vectors of the initial primal and dual bases are printed.

***Output Form*:** Selects on which output the results will be written. The possible values are `(Terminal, RES, TEX)`. Lowercases are also allowed.

`Terminal`: the results will be written on the terminal screen.

`Res`: the results are sent to a file with the same name as the data file, but with extension `.res`.

`Tex`: the results are written in a file intended for LaTeX, with extension `.tex`.

The form of the data file for the Palpha test must be as shown in Fig. 4.5.

# This is a comment line		
`PALPHA`		must always be there
calcType		
`LCG`		only LCG are possible for now
m		number of points
a		multiplier
d		$= 1$ for now
minDim	*maxDim*	two integers
primeM	*verifyM*	two booleans
maxPeriod	*verifyP*	two booleans
alpha		α, an integer
seed		seed of the LCG, an integer
$\beta_0, \beta_1, \ldots, \beta_s$		$s = maxDim$
outputForm		`terminal`, `res` or `tex`

Figure 4.5: Data file format for the P_α test.

Lines whose first non-blank character is a `#` are comments, and are dropped by the reader program. The file must contain the following parameters in that order:

`PALPHA`: must always be there literally. This indicates to the program that the parameters to be read are for the P_α test, instead of the spectral or Beyer tests.

calcType: must be one of `PAL`, `BAL`, or `NORMPAL`. See the description of these cases in the definition of `CalcType` on page **??**.

`LCG`: must always be there literally.

m: the number of points of the point set or the modulus of congruence of the LCG.

a: the multiplier of the LCG or of the Korobov lattice. Restriction: $a \in \{1, 2, \ldots, m-1\}$.

d: The number of kinds of projections. Always $d = 1$ for now.

minDim **and** *maxDim*: the test will be done in all dimensions s such that $minDim \leq s \leq maxDim$.

primeM **and** *verifyM*: if *primeM* is `true`, the program considers that m is a prime number; if `false`, m is assumed not prime. If *verifyM* is `true`, the program will verify that m is effectively prime and reset *primeM* to its correct value. If *verifyM* is `false`, the program will not verify the primality of m; in that case, it is the responsibility of the user to set the right value for *primeM*.

maxPeriod **and** *verifyP*: if *maxPeriod* is `true`, the program considers that the LCG has maximal period; if `false`, the LCG is assumed not to have maximal period. If *verifyP* is `true`, the program will verify that the LCG has maximal period and will reset *maxPeriod* to its correct value. If *verifyP* is `false`, the program will not verify that the LCG has maximal period; in that case, it is the responsibility of the user to set the right value for *maxPeriod*.

alpha: the value of α. Must be one of $\{2, 4, 6, 8\}$.

seed: The starting state of the LCG. Must be one of $\{1, 2, \ldots, m-1\}$.

$\beta_0, \beta_1, \ldots, \beta_s$: $s+1$ real numbers on a line where $s = maxDim$. They are the weights β_j.

outputForm: can take the values `terminal` (the output will be sent on the terminal), `res` (the output will be written in plain text format in a file), or `tex` (the output will be written in LaTeX format in a file). In the last two cases, the name of the output file will always have the same stem as the data file name. For example, if the data file is named `alp1.dat`, then the output file will be called `alp1.res` and `alp1.tex`, respectively.

SeekMain

This is the main program to search for the "best" (or "worst") multiple recursive generators or multiply-with-carry generators of a given form, based on one of the criteria Q_T or M_T defined in Section 1.5. It produces a report listing the retained generators, their properties, and various statistics on the search.

The set of dimensions in which the test is applied can be partitioned into a certain number of intervals, or categories, and one can use a different selection criterion for the generators within each category. One can also impose bounds on the figure of merit within each category. See the data fields for C, *MinMerit* and *MaxMerit* below. For example, one can consider only the generators with $M_8 \geq 0.6$, and among these, retain the list of generators with the *smallest* value of M_{12}. As another example, one can retain the 2 generators with the highest M_8, the 2 generators with the highest M_{16} and the eight generators with the highest M_{32}.

One can search for combined MRGs with J components, or simple MRGs ($J = 1$), or multiply-with-carry (MWC) generators. For a MWC, one simply analyzes the corresponding LCG, which is a special case of an MRG. Therefore, in what follows, we use the term 'MRG component' to denote either a MRG or a MWC. For a simple MRG (or for each component, in case $J > 1$), with given modulus m and order k, the program searches for vectors of multipliers inside the region bounded by the vectors $b = (b_1, \ldots, b_k)$ and $c = (c_1, \ldots, c_k)$ such that $-m < b_i \leq c_i < m$ for each i. The search can be exhaustive in that region, or random. One can search only among maximal period generators (for each component), or not consider the period and examine only the lattice structure. The former (checking maximal period conditions) can be done only if m is prime, or if $k = 1$ and m is a power of a prime. The program can also list the retained generators in a file, in a format more compact than for the result file, and can re-use that file as input to the program, in a later run. This could be useful, for example, if one wishes to perform first a screening over a large region, based on a criterion that does not require expensive computations, and then do a second pass over the retained generators, based on a more stringent criterion, such as looking at the lattice structure in higher dimensions, and/or verifying the results by performing all computations using error bounds.

Method of search

For an exhaustive search for MRGs, all vectors of multipliers of the form $a = (a_1, \ldots, a_k)$ such that $b_i \leq a_i \leq c_i$ for $i = 1, \ldots, k$ will be examined, for a total of $N_v = \prod_{i=1}^{k}(c_i - b_i + 1)$ vectors. This holds for each component. Therefore, if there are J components and $N_{v,j}$ vectors are examined for component j, then a total of $\prod_{j=1}^{J} N_{v,j}$ generators are examined.

For a random search for MRGs, we fix a number of subregions (clusters) we want to examine, and the size h_i of each subregion in dimension i, for $i = 1, \ldots, k$. The program will examine a total of $n \prod_{i=1}^{k} h_i$ vectors of multipliers (for each MRG component) by repeating n times the following: For $i = 1, \ldots, k$, generate α_i randomly, uniformly over the set $\{b_i, \ldots, c_i - h_i + 1\}$; then, examine all the vectors $a = (a_1, \ldots, a_k)$ such that $\alpha_i \leq a_i \leq \alpha_i + h_i - 1$ for each i.

When examining a vector a, the program first checks if the maximal period conditions are satisfied, if this is required. For prime modulus m, the condition (a) (see Section **??**) is verified only once for each distinct value of a_k (which corresponds to $\prod_{i=1}^{k-1} h_i$ different vectors). To verify

the maximal period conditions, the factorizations of $m-1$ and $r = (m^k - 1)/(m-1)$ are required. They can be found by the program, if desired, or provided by the user in a file (see below). All factorizations use the very efficient MIRACL software [38]. We recall that factoring r can take *huge* amounts of time. So, avoid redoing the factorization unnecessarily. For the MRGs, these factorizations are necessary only when m is prime and maximal period is required.

If a is not rejected by the maximal period test, then we move forward to the next MRG component and try all the vectors for that next component (by exhaustive or random search) and examine their combination with the currently examined multipliers for the previous components. For each combined generator, the values of d_t or q_t are computed for dimensions $k+1, \ldots, T$.

The program always keeps lower and upper bounds on the figure of merit (M_T or Q_T), in each dimension, for the generator to be worth considering. The initial values of these bounds are given by the user in the fields *MinMerit* and *MaxMerit*. The lower bound can be 0.0 and the upper bound can be 1.0, which means that there is no effective bounds for some categories if desired.

As soon as a generator has a figure of merit below the lower bound in a given dimension, or above the upper bound for its category (after the computations for all the dimensions in this category have been completed), then this generator is immediately discarded and no further computations are made for it. This can save enormous amounts of time in the case of very large searches up to high dimensions, because with good bounds, few generators will reach the large dimensions.

During execution, only the bounds for the last category can be modified. If the figure of merit for the last category is to be *maximized*, when we have found enough [i.e., $NbGen(C)$] generators with a figure of merit $\geq \sigma$, where σ is larger than the lower bound for the last category, then we raise this lower bound to σ. Similarly, if we minimize in the last category, we can lower the upper bound when we have enough generators beating the bound. In the case where the figure of merit is to be *maximized in all* the categories, then a generator is also discarded as soon as its figure of merit in *any* dimension gets below the lower bound of the last category.

The execution (CPU) time is checked before testing each new generator. When it exceeds the CPU time limit given in the data file, the search is aborted and the partial results are printed.

The executable programs

Three different executable programs have been compiled from `SeekMain`, depending on the types of `BScal`, `MScal`, `NScal`, `RScal` (see module `Types` on page **??** for their definitions): They are named `seekLLDD`, `seekZZDD`, and `seekZZRR`. The types are as defined in the following table:

	BScal	MScal	NScal	RScal
seekLLDD	long	long	double	double
seekZZDD	ZZ	ZZ	double	double
seekZZRR	ZZ	ZZ	RR	RR

`ZZ` and `RR` are the big integers and the big floating-point numbers defined in NTL. The program `seekLLDD` is the fastest but works only when m, the modulus of congruence, is small ($< 2^{25}$). [1] The program `seekZZRR` is very slow, but works for arbitrary large numbers. `seekZZDD` is much faster than `seekZZRR`. Thus one should work with `seekLLDD` if possible, and if not, with `seekZZDD`, and otherwise with `seekZZRR` as the last choice.

[1] From Richard: vérifier cela

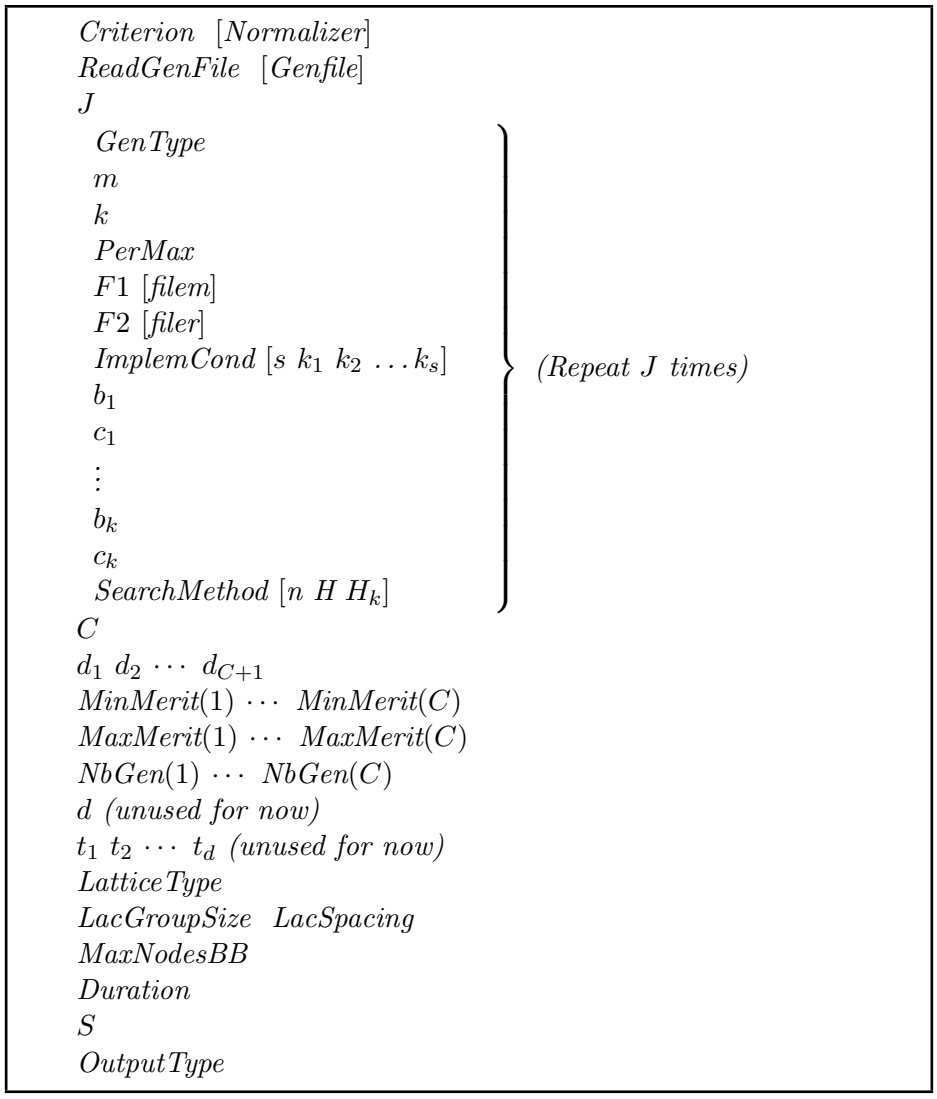

Figure 4.6: Data file format for `seekl`.

The data file

The data for `seek*` must be placed in a file with extension "`.dat`", according to the format displayed on Figure 4.6. The fields in square brackets are optional (depending on the value taken by the first field on the line). The meaning of all data fields is explained below. To run the program, type "`seek*` ⟨*file*⟩", where ⟨*file*⟩ is the name of the data file, without extension, and `seek*` is one of the three seek programs mentionned above. The results will be in a file with the same name, with extension "`.res`" or "`.gen`" (see *OutputType* in module `Const` on page **??**).

Comments may be inserted after data, on the same line, separated from the data by at least one blank. Moreover, any line starting with "#" is considered as a comment and is discarded by the reading program.

The values of m, b_i, and c_i in the data file can be given in one of the two following formats:

a) An integer giving the value directly, in base 10. In this case, there *must* be some other non-numeric text (e.g., a comment) on this data line after the integer.

b) Three integers x, e, and z on the same line, separated by at least one blank. The retained value will be $x^e + z$ if $x \geq 0$, and $-(|x|^e + z)$ if $x < 0$. The value of e must be positive. For example, $(x\ e\ z) = (2\ 5\ -1)$ will give 31, while $(x\ e\ z) = (-2\ 5\ -1)$ will give -31 (not -33).

For the program `seekLLDD`, all these numbers must fit in a `long`. For larger numbers, one must use the programs `seekZZDD` or `seekZZRR`.

Meaning of the data fields

***Criterion*:** Specifies the merit criterion for ranking the generators for each category. The admissible values of *Criterion* are:

`Beyer`: means that the criterion is Q_T. The program will retain the generators with the largest (or smallest) Q_T in each category. In that case, the *Norma* field may be blank and is unread.

`Spectral`: means that the criterion is M_T. The program will retain the generators with the largest values (or smallest) M_T in each category. In that case, the *Norma* field, described in the next item, must appear.

`Palpha`: means that the criterion is P_α. In that case, the fields in the data file must be the same as described for the P_α test on page **??**, except that the multiplier a must be replaced by an interval given as two integers b c on the same line. The rest of this section does not apply to to the P_α case.

***Normalizer*:** If *Criterion* is `Spectral`, then *Normalizer* must appear and it indicates which type of normalization is used in the definition of M_T. The admissible values are

`BestLat`: means that we use for d_t^* the value of d_t that corresponds to the best lattice in dimension t. Only values for $t \leq 48$ are known.

`Laminated`: means that we use for d_t^* the value of d_t that corresponds to the best laminated lattice in dimension t. Only values for $t \leq 48$ are known.

`Rogers`: means that d_t^* is obtained from Rogers' bound.

`Minkowski`: means that we use for d_t^* *Minkowski*'s theoretical bounds on the length of the shortest nonzero vector in a lattice. Only values for $t \leq 48$ have been precomputed.

`MinkL1`: means that length of vectors is computed using the $\mathcal{L}_1$ norm. Here, the length of the shortest nonzero vector gives the minimal number of hyperplanes that cover all the points of the lattice. Only values for $t \leq 48$ have been precomputed.

***ReadGenFile* ⟨*Genfile*⟩:** `boolean` and file name (without extension). When *ReadGenFile* is `FALSE`, the search is made according to the values of the fields below. When it is `TRUE`, the generators to be looked at are those listed in the file ⟨*Genfile*⟩`.gen`. This must be a file of type "`.gen`", produced by this program with the `GEN` option for the *OutputType* data field. In that case, only those generators listed in that file are examined and the vectors b and c below are not used.

***J*:** Number of components in the combined generators. Must be an integer $J > 0$. When $J > 1$, we look for combined generators.

***GenType*:** Can be:

`MRG`: means that this component is an MRG.

`MWC`: means that this component is a multiply-with-carry (MWC) generator. Each MWC generator is converted by the program to its corresponding LCG (see, e.g., [8, 27]).

`KOROBOV`: means that this component is a Korobov lattice.

`RANK1`: means that this component is a rank 1 lattice.

***m, k*:** Modulus and order of the recurrence. Must be positive integers. For $m^k >= 2^{??}$, use the program `seekZZDD` instead of `seekLLDD`.

***Permax*:** `boolean` variable. `TRUE` if maximal period is required, `FALSE` otherwise. When set to `TRUE`, m must be expressed in the data file in the form (b): $(x\ e\ z)$, otherwise *Permax* will be put back to `FALSE`. The software assumes that m is prime, unless $z = 0$ and $e > 1$, in which case it assumes that x is prime. In the latter case, one must have $k = 1$, otherwise *Permax* will be set back to `FALSE`.

***Fm* ⟨*filem*⟩:** This line of data (and also the following one) is used only if maximal period is required and m is assumed to be prime (see the *Permax* field). Otherwise, the program just skips it (but the line must be there). Fm indicates how the factors of $m - 1$ are to be found and ⟨*filem*⟩ is a file name. The values allowed for Fm are `(Decomp, Write, Read)`.

`Decomp`: means that the program itself will factorize $m - 1$. In this case, the field ⟨*filem*⟩ is not used and can be omitted. To factorize, the program uses the MIRACL software [38] (called by method `factorize` in class `IntFactorization` of *LatMRG*) with no CPU time limit. It is the responsibility of the user to make sure that the factorization will take a reasonable amount of time.

`Write`: means the same as `Decomp`, except that the program will also write the prime factors found in file ⟨*filem*⟩, one factor per line, with its multiplicity.

`Read`: indicates that $m - 1$ is already factorized and that the factors will be read from file ⟨*filem*⟩, in the same format. The factors need not be sorted, but must be one per line, each with its multiplicity, as described in method `read` of class `IntFactorization`. The factorization must be complete and the program will check if the product of all the factors is really equal to $m - 1$.

***Fr* ⟨*filer*⟩:** This data line is similar to the previous one, except that it concerns $r = (m^k-1)/(m-1)$ instead of $m - 1$. In this case, it is possible that r be prime when m is prime (in contrast to $m - 1$, which is then even). Therefore, the additional value `Prime` is allowed for Fr, so that the set of possible values is `(Decomp, Write, Read, Prime)`. The software MIRACL can be used independently to factorize r or to check its primality. It is called by the command: "`/u/simardr/miracl/factorm`".

`Prime`: indicates that r is prime.

***ImplemCond* ⟨$s\ k_1\ k_2\ \ldots k_s$⟩:** Imposes different restrictions on the multipliers a_i. Currently, the possible values of *ImplemCond* are the following:

`NoCond`: no condition is imposed on the multipliers a_i.

`AppFact`: the multipliers must satisfy the condition $|a_i|(m \bmod |a_i|) < m$, called "approximate factoring", for each i. MRGs are often easier to implement under this condition [24].

`PowerTwo` s k_1: the positive integers $s = NumPow2$ and $k_1 = HighestBit$ must appear and they indicate that for each i, the multiplier a_i must be the sum of at most s (positive or negative) powers of 2, with the highest power of 2 not exceeding 2^{k_1} in absolute value. For example, if $s = 2$ and $k_1 = 30$, there are $30 \times 31/2$ possibilities for choosing the 2 powers of 2 and 4 possibilities for choosing their signs, yielding 1860 cases where a_i is obtained from exactly 2 powers of 2. If one adds the 62 cases where a_i is $\pm$ a power of 2, this gives a total of 1922 possibilities for a_i. In the case of `PowerTwo`, the bounds b_j and c_j on the multipliers (see below) are automatically reset to $b_j = -(2^{e+1} - 1)$ and $c_j = 2^{e+1} - 1$.

`NonZero` s k_1 k_2 $\cdots$ k_s: (note: k_s must be equal to k) all the multipliers will be equal to 0, except for s of them: the s non-zero multipliers are a_{k_1}, a_{k_2}, $\ldots$, a_{k_s}. For example, for a MRG of order $k = 10$, the line "`NonZero` 4 2 5 8 10" will consider only vector of multipliers of the form $\mathbf{a} = (0, \alpha, 0, 0, \beta, 0, 0, \gamma, 0, \delta)$ where $\alpha, \beta, \gamma, \delta$ are arbitrary integers in their respective interval $[b_i, c_i]$. ($\mathbf{a_0}$ is unused). In that case, only s pairs $[b_i, c_i]$ must appear in the lines for $[b_i, c_i]$ below.

`Equal` s k_1 k_2 $\cdots$ k_s: (note: k_s must be equal to k) all the multipliers must be equal by groups. There are s groups: the first group of k_1 multipliers are all equal, the second group of $k_2 - k_1$ multipliers are all equal, and so on until the last group of $(k - k_{s-1})$ multipliers which are all equal. k_j is the vector index of the last element of group j. For example, for a MRG of order $k = 10$, the line "`Equal` 4 2 5 8 10" will consider only vector of multipliers of the form $\mathbf{a} = (\alpha, \alpha, \beta, \beta, \beta, \gamma, \gamma, \gamma, \delta, \delta)$ where $\alpha, \beta, \gamma, \delta$ are arbitrary integers in their respective interval $[b_i, c_i]$. ($\mathbf{a_0}$ is unused). Elements $a_3 = a_4 = a_5$ are the elements of the second group. In that case, only s pairs $[b_i, c_i]$ must appear in the lines for $[b_i, c_i]$ below.

$b = (b_1, \ldots, b_k)$ and $c = (c_1, \ldots, c_k)$: The b_i and c_i are integers such that $-m < b_i \leq c_i < m$ for $i = 1, \ldots, k$. They determine the boundary of the (rectangular) area of search.

SearchMethod n, H, H_k: *SearchMethod* can be `Exhaust` or `Random`.

`Exhaust`: means that the search will be exhaustive over the whole region determined by b and c. The other parameters of this line are then unused and can be omitted.

`Random`: asks for a random search, performed as described in the first subsection of the description of `seek*`. The integer n gives the number of subregions (clusters) to examine. H determines the size of these subregions, except for the kth element of the vector h, where H_k determines the size. The vector $h = (h_1, \ldots, h_k)$ is computed by the program as follows: $h_i = \min(H, c_i - b_i + 1),\ i = 1, \ldots, k-1$, and $h_k = \min(H_k, c_k - b_k + 1)$.

C: The program will retain $C \geq 1$ lists (or categories) of generators, according to the specifications given below.

Dim(0) ***Dim***(1) $\cdots$ ***Dim***(C): [2] Integers such that $2 \leq Dim(0) \leq Dim(1) < \cdots < Dim(C)$. The tests will be performed in dimensions $MinDim = Dim(0)$ to $MaxDim = Dim(C)$. How-

[2] From Richard: à réviser

ever, in the case of successive values (no lacunary indices), the test will start in dimension $\max(Dim(0), k+1)$, where k is the order of the recurrence defining the MRG. The value of C is the number of categories (intervals) into which the dimensions are partitioned. These C categories *end* at the dimensions $Dim(1), \ldots, Dim(C)$, respectively. To *minimize* (instead of maximizing) the criterion within a given category i, place a *negative sign* in front of the value of $Dim(i)$ in the data file. The program will then retain the *worst* instead of the *best* generators with respect to the category i.

MinMerit$(1) \cdots$ ***MinMerit***(C)**:**

MaxMerit$(1) \cdots$ ***MaxMerit***(C)**:** [3] Real numbers that must satisfy $0.0 \leq MinMerit(i) \leq MaxMerit(i) \leq 1.0$ for each i, and $MinMerit(i) \leq MaxMerit(i-1)$ for $i > 1$. They represent the minimal and maximal values of the figure of merit σ_T (M_T or Q_T) to keep a generator, for each category. That is, only the generators that satisfy $MinMerit(i) \leq \sigma_T \leq MaxMerit(i)$ for $T = Dim(i)$ are considered for the category i and the categories above it. Note that because of this, the values of $MinMerit(i)$ should always be nondecreasing in i, because the lower bounds for all the previous categories also apply to any given category. As soon as a generator does not satisfy this criterion in a given dimension delimiting a category, it is discarded and no more time is spent to test it. When looking for good generators, one normally sets $MaxMerit(i)$ to 1.0 for each i.

NbGen(1)**, ...,** ***NbGen***(C)**:** Each $NbGen(i)$ must be an integer ≥ 0. If positive: maximum number of generators to retain in the results for categoty i. The value 0 means that no list of generators is retained for that category.

***dualF*:** If `TRUE` the test is done for the dual lattice; otherwise for the primal lattice.

***LatticeType*:** Indicates whether to analyze the lattice generated by all possible states, or a sublattice generated by the set of recurrent states or by a subcycle of the generator. The admissible values are `(Full, Recurrent, Orbit, PrimePower)`.

`Full`**:** The complete lattice, generated by all possible initial states, will be analyzed.

`Recurrent`**:** If the (combined) generator has transient states, then the lattice analyzed will be the sublattice generated by the set of recurrent states.

`Orbit`**:** The grid generated by the (forward) orbit of a state of the (combined) generator is analyzed. This state is specified as follows. On the following J lines, the initial state for each component must be given. This is an integer vector with a number of components equal to the order of the component.

`PrimePower`**:** In the case where some component is an MLCG whose modulus is a power of a prime p, then the states visited over a single orbit (subcycle) of that component generate a sublattice (when $a \equiv 1 \bmod p$) or belong to the union of $p-1$ sublattices (otherwise). If *LatticeType* takes this value, if a component is an MLCG ($k = 1$), and if the modulus of that MLCG is given in the data file in the form (b): $(x\ e\ z)$ with $z = 0$ and x prime, then what is analyzed is one of those sublattices. This is done by dividing the modulus by the appropriate power of p, as described in [30]. For example, if $p = 2$ and $a \bmod 8 = 5$, then the modulus is divided by 4 as in [11, 20].

[3] From Richard: à réviser

***LacGroupSize, LacSpacing*:** These data fields are positive integers, used to introduce lacunary indices. If the respective values are s and d, then we will analyze the lattice structure of vectors of the form $(u_{i+1}, \ldots, u_{i+s}, u_{i+d+1}, \ldots, u_{i+d+s}, u_{i+2d+1}, \ldots, u_{i+2d+s}, \ldots)$, formed by groups of s successive values, taken d values apart. To analyze vectors of successive values (as usual), take $s = d = 1$. or s larger or equal to *MaxDim*. To analyze lacunary indices that are not evenly spaced, put $s = -t$ where $t =$*MaxDim* and then, on the t lines that follow, give the t lacunary indices $i_1, \ldots, i_t$, which are to be interpreted as in Section 1.4 .

***MaxNodesBB*:** An integer giving the maximum number of nodes to be examined in any given branch-and-bound procedure when computing d_t or q_t. When that value is exceeded, the branch-and-bound is stopped and the generator is rejected. The number of generators rejected for that reason is given in the results. A small value of *MaxNodesBB* will make the program run faster (sometimes much faster), permitting to examine more generators, but will increase the chances of rejecting good generators.

***Duration c*:** A real number followed by a letter, giving the maximal CPU time given to the program for performing its search. The letter c must be one of *s, m, h,* or d for seconds, minutes, hours, and days, respectively. When the CPU time exceeds this duration, the partial results are printed, with a message. For a random search, the number of subregions printed includes the last region searched (whose search may not be finished).

***S*:** Seed of the generator used for the random search. To perform a different random search in a region already studied, just change the seed. One must have $S > 0$.

***OutputType*:** Selects in which form the results will be given. The possible values are `(Terminal, RES, GEN, TEX)`. Lowercases are also allowed.

`Terminal`: indicates that the results will appear only on the terminal screen.

`RES`: says that the results will be in a file with the same name as the data file, but with extension "`.res`".

`GEN`: says that the retained generators will be listed in a file with the same name as the data file, with extension "`.gen`". This file can then be taken as input to the same program, for example to perform a second pass with a more stringent criterion or to compute higher dimensional lattice "measures" for the retained generators.

`TEX`: means that the results will be in a "`.tex`" file written in LaTeX format.

Chapter 5

Classes

The next few modules describe the *LatMRG* classes and some non-class modules.

Primes

This class provides methods to search for integers m that are prime, for which the integer $r = (m^k - 1)/(m - 1)$ is also prime for a given k, and possibly for which $(m - 1)/2$ is also prime.

```
#include "TypesNTL.h"
#include "IntFactorization.h"
#include <fstream>


namespace LatMRG {

class Primes {
public:
```

`Primes();`

Constructor.

`~Primes();`

Destructor.

```
   void find (int e, int s, bool facto, std::ofstream & fout);
```

Finds the s prime integers $m < 2^e$ that are closest to 2^e. If `facto` is `true`, then $m - 1$ is factorized in its prime factors. The results are printed on stream `fout`.

```
   void find (int k, int e, int s, bool safe, bool facto, std::ofstream & fout);
```

Finds the s integers $m < 2^e$ that are closest to 2^e, such that m and $r = (m^k - 1)/(m - 1)$ are prime. If `safe` is `true`, $(m - 1)/2$ is also required to be prime. The results are printed on stream `fout`. If `facto` is `true`, then $m - 1$ is factorized in its prime factors. If $k = 1$, r is considered to be prime.

```
   void find (int k, int e, long c1, long c2, bool safe, bool facto,
              std::ofstream & fout);
```

Finds all integers m, in $2^e + c_1 \leq m \leq 2^e + c_2$, such that m and $r = (m^k - 1)/(m - 1)$ are prime. If `safe` is `true`, $(m - 1)/2$ is also required to be prime. The results are printed on stream `fout`. If `facto` is `true`, then $m - 1$ is factorized in prime factors. If $k = 1$, r is considered to be prime.

```
private:

   void writeHeader (int k, int e, long c1, long c2, bool safe, bool facto,
                     std::ofstream & fout);
```

Writes the parameters of the find to the stream `fout`.

```
   void writeFooter (std::ofstream & fout);
```

Writes the CPU time of the find to the stream `fout`.

```
   void find (int k, int e, int s, const MScal & S1, const MScal & S2, bool safe,
              bool facto, std::ofstream & fout);
   LatCommon::Chrono timer;
   IntFactorization ifac;
   void nextM (MScal & m)
```

```
};
}
```

LatTestAll

This class is just an auxiliary class that allows launching the spectral, Beyer, or Palpha test on several different generators successively, each one associated with its own data file. One can launch these tests on all the data files (with extension ‘‘.dat’’) in a directory also. The test and generator parameters in the data files must be as described in program `LatMain` (see page 29 of this guide). In fact, the `LatMain` program simply calls the methods of this class.

```
#include "Writer.h"
#include "LatConfig.h"


namespace LatMRG {

class LatTestAll {
public:
```

`int doTest (const char* datafile);`

Reads the parameters of the test and of the generator in input text file `datafile`; then do the test. The data file must always have the extension ‘‘.dat’’, but must be given as argument here *without extension*. For example, if the data file is named `mrg.dat`, then the method must be called as `doTest("mrg")`. Returns 0 if the test completed successfully; returns a negative integer if there was an error.

`int doTestDir (const char* dirname);`

Applies the method `doTest` to all the files with extension ‘‘.dat’’ in directory named `dirname`. Returns 0 if all the tests completed successfully; returns a non-zero integer if there was an error.

`int doTest (LatConfig & config, Writer * writer);`

Runs the test with the test parameters in `config`, and writes all results in `writer`. This method is useful if one wants to do many tests while varying one parameter for each test: one will not have to create a data file for each test; it suffices to modify `config`. Returns 0 if the test completed successfully; returns a negative integer if there was an error.

`LatConfig* readParam (const char* datafile);`

Creates a `LatConfig` and read the parameters of the test into it. The data file must have the extension ‘‘.dat’’, but must be given as argument *without extension*.

`Writer* createWriter (const char* datafile, LatConfig* config);`

Creates a `Writer` from the input file `datafile` (file name must be given without its extension), and the given `config` parameter.

`void clean (LatConfig* config, Writer* writer);`

Frees the memory reserved for `config` and `writer` before the tests.

```
};

}
```

LatticeTest

This class is the base class that lattice tests must implement. These tests are applied on lattices to assess their structural properties and their qualities with respect to different criteria. Included are well-known tests such as the *spectral* test, the *Beyer* test, the P_α test. The corresponding figures of merit for the lattice are the length of the shortest vector in the *primal* or in the *dual* lattice computed with different norms, the Beyer quotient, or the P_α criterion. For the standard spectral test, the figure of merit is based on the length of the shortest non-zero vector in the *dual* lattice, using the $\mathcal{L}_2$ norm to compute the length of vectors, and the inverse of this length gives the maximal distance between successive hyperplanes covering all the points in the *primal* lattice. If one computes the length of the shortest non-zero vector in the *dual* lattice using the $\mathcal{L}_1$ norm, one obtains the minimal number of hyperplanes covering all the points of the *primal* lattice.

```
#include "TypesNTL.h"
#include "UtilLC.h"
#include "Const.h"
#include "Base.h"
#include "IntLattice.h"
#include "Merit.h"
#include "Subject.h"
#include "LatticeTestObserver.h"
#include "Weights.h"
#include "Chrono.h"
#include <string>
#include <list>


namespace LatMRG {

class LatticeTest: public Subject<LatticeTestObserver *> {
public:

   explicit LatticeTest (LatCommon::IntLattice * lattice);
```

Constructor. The test will be applied on `lattice`.

```
   virtual ~LatticeTest ();
```

Destructor.

```
   void setDualFlag (bool dualF);
```

If `dualF` is `true`, the tests will be applied on the *dual* lattice; if it is `false`, the tests will be applied on the *primal* lattice.

```
   bool getDualFlag () const
```

Gets the value of the `m_dualF` flag.

```
   void setInvertFlag (bool invertF);
```

Sets the value of the `m_invertF` flag. If `invertF` is `true`, the inverse of the length of the shortest vector will be printed in the results. Otherwise, the length itself will be printed. By default, the value is set to `false`.

`bool getInvertFlag () const`

Gets the value of the `m_invertF` flag.

`void setDetailFlag (int detail);`

When `detail` > 0, this flag indicates to print extra detailed results. Default value: 0.

`int getDetailFlag () const`

Gets the value of the `m_detailF` flag.

`void setMaxAllDimFlag (bool maxAllDimF);`

If `maxAllDimF` is `true`, the merit will be maximized in all dimensions.

`void setMaxNodesBB (long maxNodesBB);`

Sets the maximum number of nodes in the branch-and-bound tree to `maxNodesBB`. Default value: 10^7.

`Merit & getMerit ()`

Gets the results of the last applied test.

`LatCommon::CriterionType getCriterion() const`

Returns the criterion used for the test.

`LatCommon::IntLattice * getLattice () const`

Gets the lattice on which the test is applied.

`int getMinDim () const`

Returns the lowest dimension on which the test is performed.

`int getMaxDim () const`

Returns the highest dimension on which the test is performed.

`void resetFromDim (int order, int & fromDim);`

Ensures that `fromDim` is larger than `order`.

`virtual bool test (int minDim, int maxDim, double minVal[]) = 0;`

Starts the test from dimension `minDim` to dimension `maxDim`. Whenever the normalized value of the merit is smaller than `minVal` for any dimension, the method returns `false` immediately. The method returns `false` if the test was interrupted for any reason before completion, and it returns `true` upon success. The results of the test are kept in `m_merit`.

```
virtual bool test (int minDim, int maxDim, double minVal[],
                   const double* weights);
```

Similar to `test` above, but with the weights `weights`.

`protected:`

`LatCommon::Chrono timer;`

Timer which measures CPU time as test goes on.

`Merit m_merit;`

Contains the results of the last test.

`LatCommon::CriterionType m_criter;`

The criterion used to evaluate the figure of merit.

`bool m_dualF;`

If `true`, the test is applied on the dual lattice, otherwise on the primal lattice.

`bool m_invertF;`

If `true`, the inverse of the length of the shortest vector is printed in the results. Otherwise, the length is printed.

`int m_detailF;`

When `m_detailF` > 0, this flag indicates to print extra detailed results. Default value: 0.

`bool m_maxAllDimFlag;`

If `true`, the merit is to be maximized in all dimensions.

`long m_maxNodesBB;`

The maximum number of nodes in the branch-and-bound tree.

`LatCommon::IntLattice* m_lat;`

The lattice on which the test is applied.

`int m_fromDim, m_toDim;`

The dimension parameters.

`typedef std::list<LatticeTestObserver*> ob_list;`

The list that contains the observers for the lattice test.

`void dispatchBaseUpdate (LatCommon::Base &);`

Dispatches a `baseUpdate` signal to all observers.

`void dispatchBaseUpdate (LatCommon::Base & V, int i);`

Dispatches a `baseUpdate(V, i)` signal to all observers. Only base vector i will be sent.

`void dispatchResultUpdate (double[], int);`

Dispatches a `resultUpdate` signal to all observers.

`void dispatchTestInit (const std::string &, std::string[], int);`

Dispatches a `testInit` signal to all observers.

`void dispatchTestCompleted ();`

Dispatches a `testCompleted` signal to all observers.

```
   void dispatchTestFailed (int);
```

Dispatches a `testFailed` signal to all observers.

```
};

}
```

LatTestSpectral

This class implements the *spectral* test. It implements the abstract class `LatticeTest`. The figure of merit for this test is the length of the shortest vector in the *primal* or in the *dual* lattice computed with different norms. For the standard spectral test, the figure of merit is based on the length of the shortest non-zero vector in the *dual* lattice, using the $\mathcal{L}_2$ norm to compute the length of vectors, and the inverse of this length gives the maximal distance between successive hyperplanes covering all the points in the *primal* lattice. If one computes the length of the shortest non-zero vector in the *dual* lattice using the $\mathcal{L}_1$ norm instead, one obtains the minimal number of hyperplanes covering all the points of the *primal* lattice.

The main program is obtained by compiling the `LatMain.cc` file (see the description of module `LatMain`, on page 29, for how to run a program).

```
#include "IntLattice.h"
#include "LatticeTest.h"
#include "Reducer.h"
#include "Normalizer.h"
#include "Weights.h"
#include "Const.h"


namespace LatMRG {

class LatTestSpectral : public LatticeTest {
public:

   LatTestSpectral (const LatCommon::Normalizer * normal,
                    LatCommon::IntLattice * lat);
```

Constructor. The *spectral* test will be applied to the lattice `lat` using normalizer `normal` to normalize the figure of merit.

```
   ~LatTestSpectral ();
```

Destructor.

```
   bool test (int fromDim, int toDim, double minVal[]);
```

Applies the spectral test for dimensions varying from `fromDim` to `toDim`. Whenever the normalized value of the merit is smaller than `minVal` for any dimension, the method returns `false` immediately. The method returns `false` if the test was interrupted for any reason before completion, and it returns `true` upon success. The results of the last test are kept in `m_merit`.

```
   bool test (int fromDim, int toDim, double minVal[], const double* weights);
```

Similar to `test` above, but with the weights `weights`. `weights` is the array of the weights of all projections defined as follows:

`weights[0]` is the weight of projections [1, ... , `fromDim`];
`weights[1]` is the weight of projections [1, ... , `fromDim` + 1];
⋮
`weights`[`toDim - fromDim`] is the weight of projections [1, ... , `toDim`].

If `weights` = 0, it means unit weight for all projections.

`void setLowerBoundL2 (double S2);`

Sets the lower bound on the square length of the shortest vector in each dimension, based on the spectral value `S2`.

`void setLowerBoundL2 (double S2, const double* weights);`

Similar to `setLowerBoundL2` above, but with the weights `weights`.

`const LatCommon::Normalizer* getNormalizer()`

Returns the normalizer used in this test.

```
private:
```

`const LatCommon::Normalizer* m_normalizer;`

The normalizer used to normalize the figure of merit.

`NVect m_boundL2;`

The lower bound on the square length of the shortest vector in each dimension. As soon as a vector of length smaller than this bound is found, the search for the shortest vector in this lattice is stopped and the lattice is rejected.

`void initLowerBoundL2 (int dim1, int dim2);`

Initializes the constants `m_S2toL2` below, necessary to compute the lower bounds in `setLowerBoundL2`, for all dimensions d such that `dim1` $\leq d \leq$ `dim2`. This function must be called only after the lattice has been built (after a call to `buildBasis` or more specifically `initStates`, since the constants depend on the initialization in `initStates`).

`double *m_S2toL2;`

These precomputed constants allows the calculation of the square length of a lattice vector ℓ_2 from a value of the merit S_2 for each dimension i, i.e. $\ell_2[i] = S_2[i]$*`m_S2toL2[`i`]`.

`void prepAndDisp (int dim);`

Prepares and dispatches the results for dimension `dim` to all observers attached to this test.

`void init ();`

Sends the initialization message to all observers attached to this test.

```
};

}
```

LatTestBeyer

This class implements the *Beyer* test. It implements the abstract class `LatticeTest`. The figure of merit for this test is the Beyer quotient. The main program is obtained by compiling the `LatMain.cc` file (see the description of module `LatMain`, on page 29, for how to run a program). [1]

```
#include "IntLattice.h"
#include "LatticeTest.h"


namespace LatMRG {

class LatTestBeyer : public LatticeTest {
public:

   LatTestBeyer (LatCommon::IntLattice * lat);
```

Constructor. The *Beyer* test will be applied on lattice `lat`.

```
   ~LatTestBeyer ()
```

Destructor.

```
   bool test (int fromDim, int toDim, double minVal[]);
```

Applies the *Beyer* test for dimensions varying from `fromDim` to `toDim`. Whenever the normalized value of the merit is smaller than `minVal` for any dimension, the method returns `false` immediately. The method returns `false` if the test was interrupted for any reason before completion, and it returns `true` upon success. The results of the last test are kept in `merit`.

```
private:

   void prepAndDisp (int dim);
```

Prepares and dispatches the results for dimension `dim` to all observers attached to this test.

```
   void init ();
```

Sends the initialization message to all observers attached to this test.

```
};

}
```

[1] From Richard: Dixit Pierre: il est douteux que cette classe devrait exister: ce devrait être une méthode de Lattice ou quelque chose du genre. Idem pour les autres `LatTest*`

TestProjections

Implements methods used to calculate the worst-case figure of merit, defined as follows:

$$M_{t_1,\ldots,t_d} = \min\left[\min_{k+1\le t\le t_1}\frac{\ell_t}{\ell_t^*(m^k)},\ \min_{2\le s\le k}\ \min_{I\in S(s,t_s)}\frac{\ell_I}{m},\ \min_{k+1\le s\le d}\ \min_{I\in S(s,t_s)}\frac{\ell_I}{\ell_s^*(m^k)}\right],$$

where $S(s,t_s) = \{I = \{i_1,\ldots,i_s\} \mid 1 = i_1 < \cdots < i_s \le t_s\}$. In other words, this figure of merit applies the chosen test over the s successive dimensions for all $s \le t_1$, and over the r nonsuccessive dimensions of the lattice of dimension t_r for all $2 \le r \le d$. For *dimension stationary* lattices, for example Korobov lattices, only the sets of dimensions whose first coordinate is 1 need to be considered.

Here is an example of the spectral test with different projections for a Korobov lattice with $m = 1021$ and $a = 333$, when we consider the dual lattice with normalization BESTLAT. The figure of merit is $M_{10,8,6,5}$ and the resulting minimal merit is 0.440728, for projection 1,6. The length is the length of the shortest vector of the lattice with the $\mathcal{L}_2$ norm.

projections	length	merit
2	22.2036	0.64666
3	7	0.619324
4	3.87298	0.576145
5	3.74166	0.760239
6	2	0.488395
7	2	0.552276
8	2	0.594822
9	2	0.654906
10	2	0.697213
1,3	25.4951	0.742522
1,4	20.6155	0.600409
1,5	30.6105	0.891502
1,6	15.1327	0.440728
1,7	16.6433	0.484722
1,8	32.3883	0.943279
1,2,4	7.07107	0.625612
1,2,5	8.12404	0.718773
1,2,6	9.48683	0.839346
1,3,4	9.43398	0.83467
1,3,5	9.69536	0.857795
1,3,6	8.12404	0.718773
1,4,5	7.54983	0.66797
1,4,6	8.12404	0.718773
1,5,6	6.78233	0.600066
1,2,3,5	5.74456	0.854561
1,2,4,5	3.87298	0.576145
1,3,4,5	5.47723	0.814792

```
#include "IntLattice.h"
#include "LatticeTest.h"
#include "Weights.h"
#include "ProjIterator.h"
#include "CoordinateSets.h"
#include "Writer.h"


namespace LatMRG {

class TestProjections {
public:
```

`TestProjections (LatCommon::IntLattice *master, LatCommon::IntLattice *lattice, LatticeTest *test, int td[], int d);`

`master` is the original lattice for which we want to calculate the M merit as defined above. `lattice` is the working lattice used for intermediate calculations; all the projections from `master` are stored in `lattice` before calling the test. `test` is the lattice test to be applied on the different projections. `d` is the last element of array `td`, which gives the maximal dimensions for the different projections. `td[0]` and `td[1]` gives the minimal and the maximal dimensions for the test applied on successive dimensions. For example, if `d` = 3 and `td` = [2, 32, 16, 12], then the test will be applied on all successive dimensions $2 \leq t \leq 32$, on all possible 2-dimensional projections for dimensions ≤ 16, and on all possible 3-dimensional projections for dimensions ≤ 12.

`~TestProjections ();`

Destructor.

`void setOutput (Writer * rw);`

Sends the output to `rw`. If this method is not called, output will be sent to standard output.

`void setDualFlag (bool flag);`

If `flag` is `true`, the tests will be applied on the *dual* lattice; if it is `false`, the tests will be applied on the *primal* lattice.

`void setInvertFlag (bool flag);`

Sets the value of the `m_invertF` flag. If `invertF` is `true`, the inverse of the length of the shortest vector will be printed in the results. Otherwise, the length itself will be printed.

`void setPrintF (bool flag);`

If flag is `true`, the value of the merit will be printed for all projections, otherwise not. This flag should be set `false` in the case of the `seek*` programs, and `true` otherwise.

`void build (const LatCommon::Coordinates & proj);`

Builds the basis (and dual basis) of the projection `proj` for this lattice. The result is placed in the work lattice. The basis is triangularized to form a proper basis. For example, if $d = 2$ and `indices` = $[1, 4]$, then a 2-dimensional basis is built using coordinates 1 and 4 of the master basis.

`double run (bool stationary, bool last, double minVal[]);`

Calculates the $M_{t_1,\ldots,t_d}$ merit by running all the tests for the different projections. If set `true`, `stationary` means that the lattice is known to be *dimension stationary*. If set `true`, the flag `last` means that only the projections which include the last dimension of the lattice will be considered.

This is good for performing incremental searches for good lattices. `minVal` is the minimal value of the normalized merit in each dimension for a lattice to be considered. This method returns the worst value of the merit over all projections.

```
double run (bool stationary, bool last, double minVal[],
            const LatCommon::Weights & weights);
```

As method `run` above, but with the weights `weights`.

```
int calcNumProjections (bool stationary, bool last);
```

Calculates the number of projections, given the parameters of this object. The flag `stationary` must be set `true` if the lattice is *dimension stationary*. If set `true`, the flag `last` means that only the projections which include the last dimension of the lattice will be considered.

```
int getNumProjections ()
```

Returns the number of projections considered for the last call to `run`. If the test was interrupted because the lattice was rejected as uninteresting, the number returned will be less than the total number of projections.

```
protected:
```

```
double run (ProjIterator & projit, double minVal[],
            const LatCommon::Weights & weights);
```

Run only for projections given by the iterator `projit`.

```
LatCommon::IntLattice* m_master;
```

Lattice on which the $M_{t_1,\ldots,t_d}$ merit will be calculated.

```
LatCommon::IntLattice* m_lattice;
```

Working lattice which is used to test the different projections.

```
LatticeTest* m_test;
```

The lattice test used to calculate the merit.

```
double* m_weightsTemp;
```

Weights.

```
bool m_dualF;
```

If `true`, the test is applied on the dual lattice, otherwise on the primal lattice.

```
bool m_invertF;
```

If `true`, the inverse length of the shortest vector is printed for all projections, otherwise the length is printed.

```
bool m_printF;
```

If `true`, the value of the merits is printed for all projections, otherwise not.

```
   bool m_racF;
```

If `true`, the square root of the values of the merit is printed after the test; otherwise the values themselves are printed.

```
   int *m_td;
```

The maximal dimensions for each kind of projections. `m_td[0]` is the minimal dimension for successive dimensions. `m_td[1]` is the maximal dimension for successive dimensions (1-dimensional projections), `m_td[2]` is the maximal dimension for 2-dimensional projections, `m_td[3]` is the maximal dimension for 3-dimensional projections, and so on. The last element, `m_td[m_d]`, is the maximal dimension for `m_d`-dimensional projections.

```
   int m_d;
```

The number of kinds of projections. Also the number of elements of array `m_td` is `m_d + 1`.

```
   int m_numproj;
```

The number of projections.

```
private:

   Writer *m_writer;
```

Output will be written on `m_writer`. By default, it is written on standard output.

```
   bool m_wrFlag;
};

}
```

ProjIterator

This abstract class is the basis for different kinds of projection iterators used to walk through sets of projections.

```
#include "Weights.h"
#include "CoordinateSets.h"


namespace LatMRG {

class ProjIterator {
public:

   virtual ~ProjIterator()
```

Destructor.

```
   virtual void reset() = 0;
```

Resets the iterator to the first projection.

```
   virtual const LatCommon::Coordinates * operator->() const

   virtual const LatCommon::Coordinates & operator*() const = 0;
```

Returns the current projection.

```
   virtual ProjIterator & operator++() = 0;
```

Advances to the next projection.

```
   bool atEnd() const

   virtual operator bool() const = 0;
```

Returns `true` if the current projection is valid, or `false` if all projections have already been visited.

```
};

}
```

ProjIteratorDefault

This projection iterator walks through all projections of a given range of orders and coordinate indices.

```
#include "ProjIterator.h"


namespace LatMRG {

class ProjIteratorDefault : public ProjIterator {
public:

   ProjIteratorDefault (unsigned int minCoord, unsigned int maxCoord,
                        unsigned int minOrder, unsigned int maxOrder);
```

Constructor. Creates a projection iterator for all projections of orders `minOrder` to `maxOrder` with coordinate indices ranging from `minCoord` to `maxCoord`.

```
   ProjIteratorDefault (unsigned int minCoord, unsigned int maxCoord,
                        unsigned int minOrder, unsigned int maxOrder,
                        bool forceMinCoord, bool forceMaxCoord);
```

Constructor. Creates a projection iterator for all projections of orders `minOrder` to `maxOrder` with coordinate indices ranging from `minCoord` to `maxCoord`. If `forceMinCoord` is `true`, coordinate `minCoord` is always present. If `forceMaxCoord` is `true`, coordinate `maxCoord` is always present.

```
   virtual void reset();
```

Resets the iterator to the first projection of minimal order.

```
   virtual void resetAtOrder (unsigned int order);
```

Resets the iterator to the first projection of order `order`.

```
   virtual const LatCommon::Coordinates & operator*() const;
```

Returns the current projection.

```
   virtual ProjIterator & operator++();
```

Advance to the next projection.

```
   virtual operator bool() const
```

Returns `true` if the current projection is valid, or `false` if all projections have already been visited.

```
protected:

   unsigned int m_order;

   unsigned int m_minCoord;
   unsigned int m_maxCoord;

   unsigned int m_minOrder;
   unsigned int m_maxOrder;

   bool m_atEnd;

   bool m_forceMinCoord;     // force first coordinate to the minimum index
   bool m_forceMaxCoord;     // force last coordinate to the maximum index

   LatCommon::Coordinates m_projection;
};

}
```

ProjIteratorSuccCoords

This projection iterator walks through all projections composed of successive coordinate indices.

```
#include "ProjIteratorDefault.h"


namespace LatMRG {

class ProjIteratorSuccCoords : public ProjIteratorDefault {
public:

   ProjIteratorSuccCoords (unsigned int minCoord, unsigned int maxCoord,
                           unsigned int minOrder, unsigned int maxOrder);
```

Constructor. Creates a projection iterator for all projections of orders `minOrder` to `maxOrder` with coordinate indices ranging from `minCoord` to `maxCoord`.

```
   ProjIteratorSuccCoords (unsigned int minCoord, unsigned int maxCoord,
                           unsigned int minOrder, unsigned int maxOrder,
                           bool forceMinCoord, bool forceMaxCoord);
```

Constructor. Creates a projection iterator for all projections of orders `minOrder` to `maxOrder` with coordinate indices ranging from `minCoord` to `maxCoord`. If `forceMinCoord` is `true`, coordinate `minCoord` is always present. If `forceMaxCoord` is `true`, coordinate `maxCoord` is always present.

```
   virtual void resetAtOrder (unsigned int order);
```

Resets the iterator to the first projection of order `order`.

```
   virtual ProjIterator & operator++();
```

Advance to the next projection.

```
};

}
```

ProjIteratorNonSuccCoords

This projection iterator walks through all projections composed of non-successive coordinate indices.

```
#include "ProjIteratorDefault.h"


namespace LatMRG {

class ProjIteratorNonSuccCoords : public ProjIteratorDefault {
public:

   ProjIteratorNonSuccCoords (unsigned int minCoord, unsigned int maxCoord,
                              unsigned int minOrder, unsigned int maxOrder);
```

Constructor. Creates a projection iterator for all projections of orders `minOrder` to `maxOrder` with coordinate indices ranging from `minCoord` to `maxCoord`.

```
   ProjIteratorNonSuccCoords (unsigned int minCoord, unsigned int maxCoord,
                              unsigned int minOrder, unsigned int maxOrder,
                              bool forceMinCoord, bool forceMaxCoord);
```

Constructor. Creates a projection iterator for all projections of orders `minOrder` to `maxOrder` with coordinate indices ranging from `minCoord` to `maxCoord`. If `forceMinCoord` is `true`, coordinate `minCoord` is always present. If `forceMaxCoord` is `true`, coordinate `maxCoord` is always present.

```
   virtual void resetAtOrder (unsigned int order);
```

Resets the iterator to the first projection of order `order`.

```
   virtual ProjIterator & operator++();
```

Advances to the next projection.

```
protected:

   static bool successive (const LatCommon::Coordinates & indices);
};

}
```

MRGLattice

This class implements lattice basis built from multiple recursive linear congruential generators (MRGs). One must first call the constructor with a given congruence modulus m, a given order k for the recurrence, and a maximal dimension for the basis. One must then build the lattice basis associated to a vector of multipliers for a given dimension. Each MRG is defined by a vector of multipliers A, where $A[i]$ represents a_i. This MRG satisfies the recurrence

$$x_n = (a_1 x_{n-1} + \cdots + a_k x_{n-k}) \bmod m.$$

```
#include "TypesNTL.h"
#include "Const.h"
#include "Lacunary.h"
#include "MRGComponent.h"
#include "Modulus.h"
#include "Lacunary.h"
#include "IntLattice.h"
#include <string>


namespace LatMRG {

class MRGLattice: public LatCommon::IntLattice {
public:

   MRGLattice (const MScal & m, const MVect & a, int maxDim, int k,
               LatCommon::LatticeType latt,
               LatCommon::NormType norm = LatCommon::L2NORM);
```

Constructor with modulus of congruence m, order of the recurrence k, multipliers a, maximal dimension `MaxDim`, and lattice type `Latt`. Vectors and (square) matrices of the basis have maximal dimension `maxDim`, and the indices of vectors and matrices vary from dimension 1 to `maxDim`. The norm to be used for the basis vectors is `norm`.

```
   MRGLattice (const MScal & m, const MVect & a, int maxDim, int k, BVect & lac,
               LatCommon::LatticeType latt,
               LatCommon::NormType norm = LatCommon::L2NORM);
```

As in the constructor above but the basis is built for the lacunary indices `lac`.

```
   MRGLattice (const MRGLattice & Lat);
```

Copy constructor. The maximal dimension of the created basis is set equal to `Lat`'s current dimension.

```
   MRGLattice & operator= (const MRGLattice & Lat);
```

Assigns `Lat` to this object. The maximal dimension of this basis is set equal to `Lat`'s current dimension.

```
   ~MRGLattice();
```

Destructor.

```
   void kill();
```

Cleans and releases memory used by this object.

```
virtual void buildBasis (int d);
```

Builds the basis in dimension d.

```
virtual void incDim();
```

Increments the dimension of the basis by 1 by calling either `incDimBasis` or `incDimLaBasis`.

```
bool isLacunary() const
```

Returns `true` for the case of lacunary indices, returns `false` for non-lacunary indices.

```
BScal & getLac (int j);
```

Returns the j-th lacunary index.

```
virtual void setLac (const Lacunary & lat);
```

Sets the lacunary indices for this lattice to `lat`.

```
virtual MScal getRho() const
virtual MScal getLossRho() const
virtual void setRho (const MScal & val)
virtual void setLossRho (const MScal & val)
```

Sets and gets the values of `m_rho` and `m_lossRho`.

```
virtual const MVect & getCoef() const
```

Returns a non-mutable copy of the multipliers (coefficients) of the MRG.

```
std::string toStringCoef() const;
```

Returns the vector of multipliers A as a string.

```
std::vector<MRGComponent *> comp;
```

The components of the lattice when it is built out of more than one component. When there is only one component, it is unused as the parameters are the same as above.

```
protected:
```

```
void initStates ();
```

Initializes a square matrix of order k. This initial matrix contains a system of generators for the given group of states.

```
void init();
```

Initializes some of the local variables.

```
void initOrbit();
```

Initializes this object when the lattice type is `ORBIT`.

```
void insertion (BMat & Sta);

void lemme2 (BMat & Sta);

void trace (char* msg, int d);
```
For debugging purposes.

```
virtual void incDimBasis ();
```
Increments the basis by 1 in case of non-lacunary indices.

```
void incDimLaBasis (int);
```
Increments the basis by 1 in case of lacunary indices.

```
void buildNaBasis (int d);
```
Builds the basis of the MRG recurrence in case of non-lacunary indices.

```
void buildLaBasis (int d);
```
Builds the basis of the MRG recurrence in case of lacunary indices.

```
MScal m_lossRho;
MScal m_rho;
```
Used for the calculation of a combined MRG.

```
MVect m_aCoef;
```
The coefficients of the recurrence.

```
LatCommon::LatticeType m_latType;
```
Indicates which lattice or sublattice is analyzed.

```
bool m_lacunaryFlag;
```
Is `true` in the case of lacunary indices, `false` otherwise.

```
Lacunary m_lac;
```
Contains the lacunary indices when `LacunaryFlag` is `true`, otherwise is undefined.

```
MScal m_t4, m_t5, m_t6, m_t7, m_t8, m_e;
MVect m_xi;
```
Work variables.

```
BMat m_sta, m_wSI;
```
♣ To be completed.

```
bool *m_ip;
```
When the flag `m_ip[i]` is `true`, the i-th diagonal element of matrix `m_sta` is non-zero (modulo m) and divides m. Otherwise (when `m_ip[i]` is `false`), the i-th line of matrix `m_sta` is identically 0.

```
};

}
```

MRGLatticeLac

This class implements lattice bases built from multiple recursive linear congruential generators (see class `MRGLattice`) using *lacunary indices*.

```
#include "TypesNTL.h"
#include "Const.h"
#include "Lacunary.h"
#include "MRGLattice.h"


namespace LatMRG {

class MRGLatticeLac:   public MRGLattice {
public:

   MRGLatticeLac (const MScal & m, const MVect & A, int maxDim, int k,
                  BVect & lac, LatCommon::LatticeType latt,
                  LatCommon::NormType norm = LatCommon::L2NORM);
```

Constructor with modulus of congruence m, order of the recurrence k, multipliers A, maximal dimension `maxDim`, and lattice type `latt`. Vector and matrix indices vary from 1 to `maxDim`. The length of the basis vectors is computed with `norm`. The bases are built using the *lacunary indices* `lac`.

```
   virtual ~MRGLatticeLac();
```

Destructor.

```
   void buildBasis (int d);
```

Builds the basis of the MRG recurrence in dimension d using the lacunary indices.

```
   void incDim()
```

Increases the dimension of the basis by 1.

```
   BScal & getLac (int j);
```

Returns the j-th lacunary index.

```
   void setLac (const Lacunary & lat);
```

Sets the lacunary indices for this lattice to `lat`.

```
protected:

   void initStates();

   void incDimBasis (int);
```

Increases the dimension of the basis by 1.

```
   Lacunary m_lac;
```

The lacunary indices.

```
};

}
```

Modulus

This class keeps parameters closely associated with a modulus of congruence. Using it, it will not be necessary to recalculate the square roots of large integers, which are used repeatedly in searches for good generators.

```
#include "TypesNTL.h"


namespace LatMRG {

class Modulus {
public:

   Modulus ();
   Modulus (const MScal & m);
```

Constructor with modulus of congruence m.

```
   Modulus (long b, long e, long c);
```

Constructor with value $m = b^e + c$. Restrictions: $b > 1$ and $e > 0$.

```
   virtual ~Modulus ();
```

Destructor.

```
   void init (const MScal & m);
```

Initializes with value m. Computes `mRac` and `mRacNeg`.

```
   void init (long b, long e, long c);
```

Initializes with value $m = b^e + c$. Restrictions: $b > 1$ and $e > 0$. Computes `mRac` and `mRacNeg`.

```
   void reduceM (const MScal & a);
```

Reduces the modulus m and sets the variable `mRed` to the reduced modulus. The modulus must have the form $m = p^e$. The multiplier of the LCG is a.

```
   bool perMaxPowPrime (const MScal & a);
```

Assumes that m is a power of a prime $p = b$, the order $k = 1$, and the recurrence is homogeneous. Returns `true` iff the maximal period conditions are satisfied.

```
   MScal m;
```

Value m of the modulus.

```
   MScal mRed;
```

Reduced value of the modulus. Computed by `reduceM`.

```
   bool primeF;
```

This flag is `true` when m is prime, otherwise `false`.

```
   bool threeF;
```

When this flag is `true`, the value of m is built out of the three numbers b, e and c as described below; otherwise, the flag is set `false`.

```
   long b;
   long e;
   long c;
```

When `threeF` is `true`, then m is given in the form $m = b^e + c$; otherwise, b, e and c are undefined.

```
   MScal mRac;
```

$\sqrt{\lfloor m \rfloor}$.

```
   MScal mRacNeg;
```

$-\sqrt{\lfloor m \rfloor}$.

```
private:

   MScal bm1;
```

The constant $b - 1$.

```
   MScal b2;
```

The constant b^2.

```
   MScal Y, Eight, Four;
```

Work variables.

```
};

}
```

MRGComponent

This class is used to implement a MRG component in a combined MRG. It exists in order to avoid creating numerous relatively heavy `MRGLattice` objects to represent MRG components. Each MRG component is defined by a modulus m, an order k and a vector of multipliers a, where $a[i]$ represents a_i. This MRG satisfies the recurrence

$$x_n = (a_1 x_{n-1} + \cdots + a_k x_{n-k}) \bmod m$$

```
#include "TypesNTL.h"
#include "Const.h"
#include "IntFactorization.h"
#include "Modulus.h"
#include <string>


namespace LatMRG {

class MRGComponent {
public:
```

`MRGComponent (const MScal & m, const MVect & a, int k);`

Constructor with modulus m, vector a and order k.

`MRGComponent (long b, long e, long c, const MVect & a, int k);`

Constructor with modulus $m = b^e + c$, vector a and order k.

```
MRGComponent (const MScal & m, int k, LatCommon::DecompType decom1,
              const char *filem1,     LatCommon::DecompType decor,
              const char *filer);
```

Constructor with modulus m and order k. Arguments `decom1` and `decor` refer to the prime factor decomposition of $m-1$ and $r = (m^k-1)/(m-1)$, respectively. If `decor` equals `DECOMP`, the constructor will factorize r. If `decor` equals `DECOMP_WRITE`, the constructor will factorize r and write the prime factors to file `filer`. If `decor` equals `DECOMP_READ`, the constructor will read the factors of r from file `filer`. If `decor` equals `DECOMP_PRIME`, r is assumed to be prime. Similar considerations apply to `decom1` and `filem1` with respect to $m-1$.

```
MRGComponent (Modulus & modul, int k, LatCommon::DecompType decom1,
              const char *filem1,     LatCommon::DecompType decor,
              const char *filer);
```

Constructor similar to the above, except that the modulus of congruence m is inside the object `modul`.

`~MRGComponent();`

Destructor.

`MRGComponent (const MRGComponent & comp);`

Copy constructor;

`MRGComponent & operator= (const MRGComponent & comp);`

Assignment operator.

`void setA (const MVect & A);`

Sets the multipliers of the recurrence to A.

`bool maxPeriod (const MVect & A);`

Returns `true` if coefficients A give a MRG with maximal period; returns `false` otherwise.

`bool maxPeriod23 (const MVect & A);`

Returns `true` if coefficients A give a MRG with maximal period; returns `false` otherwise. This method supposes that condition 1 is `true` and tests only conditions 2 and 3. See method `isPrimitive` of class `PolyPE` on page 89 of this guide.

`IntFactorization ifm1;`

The prime factor decomposition of $m - 1$.

`IntFactorization ifr;`

The prime factor decomposition of $r = (m^k - 1)/(m - 1)$, where k is the order of the recurrence.

`Modulus module;`

The modulus m of the recurrence.

`MScal getM()`

Returns the value of the modulus m of the recurrence.

`int k;`

The order k of the recurrence.

`MVect a;`

The multipliers a_i of the recurrence, $i = 1, \ldots, k$.

`MScal rho;`

The length of the period ρ for this MRG. For now, the recurrence is assumed to correspond to a primitive polynomial and `rho` is calculated as

$$\rho = m^k - 1$$

This value is calculated by `MRGLatticeFactory` and stored here for simplicity.

`MScal nj;`

Value needed for the calculation of the multipliers of a combined MRG. It is defined by

$$n_j = (m/m_j)^{-1} \bmod m_j \qquad \text{for } j = 1, \ldots, J,$$

where $n_j =$ `nj`, m_j is this object's modulus `m`, m is the calculated modulus for the combined MRG (see class `MRGLatticeFactory`), and $(m/m_j)^{-1} \bmod m_j$ is the inverse of m/m_j modulo m_j. This value is calculated by `MRGLatticeFactory` and stored here for simplicity.

`MVect orbitSeed;`

Contains the starting state of the component for the case when the lattice type is `ORBIT`. It is made of k numbers.

```
   std::string toString ();
```

Returns this object as a string.

```
private:

   void init (const MScal & m, int k, LatCommon::DecompType decom1,
              const char *filem1,     LatCommon::DecompType decor,
              const char *filer);
```

Does the same as the constructor above with similar arguments.

```
};

}
```

MRGComponentFactory

This class is used to create `MRGComponent`s from other types of recurrences [for instance, multiply-with-carry (MWC)].

```
#include "MRGComponent.h"
#include "TypesNTL.h"
#include "UtilLM.h"


namespace LatMRG {

class MRGComponentFactory {
public:

   static MRGComponent * fromMWC (const MScal & b, const MVect & a, int k);
```

Creates a `MRGComponent` from an multiply-with-carry type of recurrence. The MWC recurrence has the form

$$\begin{aligned} x_n &= (a_1 x_{n-1} + \cdots + a_k x_{n-k} + c_{n-1}) d \bmod b, \\ c_n &= \lfloor (a_0 x_n + a_1 x_{n-1} + \cdots + a_k x_{n-k} + c_{n-1}) / b \rfloor \end{aligned}$$

where b is a positive integer, $a_0, \ldots, a_k$ are arbitrary integers such that a_0 is relatively prime to b, and d is the multiplicative inverse of $-a_0$ modulo b. The MRG derived from such a MWC is defined by

$$m = \sum_{i=0}^{k} a_i b^i$$

where a is the inverse of b in arithmetic modulo m.

```
};

}
```

MRGLatticeFactory

This class is used to create `MRGLattice`'s from other types of recurrences than a MRG or from a combination of several MRG lattices.

```
#include "Const.h"
#include "TypesNTL.h"
#include "MRGLattice.h"
#include "MRGComponent.h"


namespace LatMRG {

class MRGLatticeFactory {
public:

   static MRGLattice * fromCombMRG (MRGComponent **comp, int J, int maxDim,
                                    BVect * Lac, LatCommon::LatticeType lat,
                                    LatCommon::NormType norm);
```

Creates a `MRGLattice` from the combination of J `MRGComponent`'s, with maximal dimension `maxDim`, lacunary indices `Lac`, lattice type `lat` and vector norm `norm`. `Lac` can be `NULL` if no lacunary indices are to be used. The combined MRG is calculated as described in [26].

```
   static MRGLattice * fromMWC (const MVect & a, const MScal & b, int maxDim,
                                int k, BVect *Lac, LatCommon::LatticeType lat,
                                LatCommon::NormType norm);
```

Creates a `MRGLattice` from a multiply-with-carry (MWC) recurrence, with maximal dimension `maxDim`, lacunary indices `Lac`, lattice type `lat` and vector norm `norm`. The MWC recurrence is

$$\begin{aligned} x_n &= (a_1 x_{n-1} + \cdots + a_k x_{n-k} + c_{n-1})\, d \bmod b, \\ c_n &= \lfloor (a_0 x_n + a_1 x_{n-1} + \cdots + a_k x_{n-k} + c_{n-1})/b \rfloor \end{aligned}$$

where b is a positive integer, $a_0, \ldots, a_k$ are arbitrary integers such that a_0 is relatively prime to b, and d is the multiplicative inverse of $-a_0$ mod b. The MRG derived from such a MWC is defined by $m = \sum_{l=0}^{k} a_l b^l$ and a is the inverse of b in arithmetic modulo m.

```
   static MRGLattice * fromMWC (const MVect & a, const MScal & b, int maxDim,
                                int k, LatCommon::LatticeType lat,
                                LatCommon::NormType norm);
```

Same as above, but with no lacunary indices.

```
};

}
```

Subject

This template class is used as a base class for any class that needs to have observers attached or detached from it. It was created to avoid recoding the attaching and detaching code of observers more than once.

```
#include <list>


namespace LatMRG {

template <typename T>
class Subject {
public:
```

`Subject()`

Constructor.

`virtual ~Subject();`

Destructor.

`void attach (T observer);`

Attaches the observer `observer` to this object.

`void detach (T observer);`

Detaches the observer `observer` to this object.

```
protected:
```

`typedef std::list<T> list_ob;`

`list_ob m_observers;`

The list that contains the observers.

```
};

}
```

LatticeTestObserver

Interface that classes must implement in order to receive information and results from a lattice test. See `LatticeTest`.

```
#include "Base.h"
#include <string>


namespace LatMRG {

class LatticeTestObserver {
public:

   virtual ~LatticeTestObserver() {}
```

Destructor.

```
   virtual void baseUpdate (LatCommon::Base & base) = 0;
```

Called when the base is incremented in a lattice test. `base` is a copy of the base used in the test.

```
   virtual void baseUpdate (LatCommon::Base & V, int i) = 0;
```

Called when the base is incremented in a lattice test. `V[i]` is a copy of basis vector i used in the test.

```
   virtual void resultUpdate (double results[], int n) = 0;
```

Called when new results have been calculated for one dimension. The results are placed in the array `results` of size `n`.

```
   virtual void testInit (const std::string & test, std::string headers[],
                          int n) = 0;
```

Called when a test is initiated. `test` contains the name of the test being performed, `headers` is the array of names of the values calculated by this test and `n` is the size of the array `headers`.

```
   virtual void testCompleted() = 0;
```

Called when the test has terminated successfully.

```
   virtual void testFailed (int dim) = 0;
```

Called when the test has terminated but failed. `dim` indicates in which dimension the test failed.

```
};

}
```

Merit

This class is used to keep the values of test results for a lattice. The values of the figure of merit in each dimension for a given lattice are kept in vectors. For reasons of efficiency, the values kept in merit objects may not always be the merit itself but instead a simple function of the merit. Only at the end of the tests will the real merit be printed. For the spectral test, in the case of the $\mathcal{L}_2$ norm, the square of the length of the shortest non-zero vector in the lattice is kept in the merit object. However, in the case of the $\mathcal{L}_1$ norm, the length of the shortest non-zero vector in the lattice is kept in the merit object.

As an example, the real normalized merit for the spectral test (the shortest vector normalized with the constants for laminated lattices in the case of the $\mathcal{L}_2$ norm) for the LCG with $m = 2^{31} - 1$ and $a = 16807$, in dimensions up to 10, is given in the following table:

	$\mathcal{L}_2$ norm		$\mathcal{L}_1$ norm	
dim	Dual	Primal	Dual	Primal
2	0.33751	0.33751	0.25647	0.25647
3	0.44118	0.54043	0.32637	0.51556
4	0.57519	0.61619	0.57143	0.55061
5	0.73612	0.61872	0.67539	0.58963
6	0.64541	0.5889	0.58879	0.52362
7	0.57112	0.60365	0.5	0.45958
8	0.60961	0.44125	0.50909	0.35768
9	0.57732	0.65822	0.46667	0.58086
10	0.65033	0.53741	0.5	0.48685

```
#include <vector>
#include <string>
#include "Const.h"


namespace LatMRG {

class Merit {
public:

   Merit (int maxDim);
```

Constructor. This object may contain figures of merit for dimensions up to `maxDim`.

```
   Merit (const Merit & mer);
```

Copy Constructor.

```
   Merit & operator= (const Merit & mer);
```

Assignment operator.

```
   ~Merit ();
```

Destructor.

`int getDim () const`

Returns the effective dimension of the vectors.

`double & getMerit (int j)`

Returns the *unnormalized* value of the merit in dimension j.

`double getNormVal (int j) const`

Returns the *normalized* value of the merit in dimension j.

`double & operator[] (int j)`

Returns the *normalized* value of the merit in dimension j. (Same as `getNormVal`.)

`void set (double x);`

Sets the value of the merit to x for all dimensions.

`double getST (int from, int T, int & dimWorst);`

Returns the smallest *normalized* value of the merit in the dimension range `from` to `T` (inclusive). As a side effect, the dimension where the smallest value occurs is returned in `dimWorst`.

`double getST (int from, int T)`

Returns the smallest *normalized* value of the merit in the dimension range `from` to `T` (inclusive). As a side effect, the dimension where the smallest value occurs is set in the private variable `m_dimWorst`.

`int getDimWorst () const`

Returns the dimension of the worst (the smallest usually) figure of merit.

`void setWorstMerit (double x)`

Sets the worst figure of merit to x.

`double getWorstMerit () const`

Returns the worst figure of merit. It must have been calculated before.

`std::string toString (int from, int T, bool rac, bool invert) const;`

Returns both the *unnormalized* and the *normalized* values of the merit from dimension `from` to `T` as a string. If `rac` is `true`, the square root of the values of the merit is returned; otherwise, the values themselves are returned. If `invert` is `true`, the inverses of the length of the shortest vector are returned; otherwise, the lengths are returned.

`private:`

`std::vector<double> m_val;`

The *unnormalized* values of the figure of merit for each dimension.

`std::vector<double> m_normVal;`

The *normalized* values of the figure of merit. The corresponding values of `m_val` are normalized so that `m_normVal` usually takes values in [0, 1].

```
   double m_worstMerit;
```

The worst value of the normalized figure of merit for this object.

```
   int m_dimWorst;
```

The dimension where the worst value of the merit occurs.

```
};

}
```

Coefficient

Classes inheriting from this base class implement different constraints that can be set on the coefficients (or multipliers) a_i of a recursive generator of order k, of the form

$$x_n = (a_1 x_{n-1} + a_2 x_{n-2} + \cdots + a_k x_{n-k}) \bmod m.$$

This base class itself applies no constraint on the coefficients.

```
#include "TypesNTL.h"


namespace LatMRG {

class Coefficient {
public:
```

`Coefficient()`

Constructor.

`virtual ~Coefficient()`

Destructor. Does nothing for now.

`virtual void set (const MScal & q, MVect & A, int & i)`

Sets the next coefficient `A[i]` $= a_i = q$ to be tested. Index i is not changed for this base class, but may be changed for subclasses depending on the constraints applied on the a_i's.

```
};

}
```

CoefEqual

This class chooses the coefficients a_i of a recursive generator of order k such that all the coefficients are equal by groups. There are s groups: the first group of k_1 coefficients are all equal, the second group of $k_2 - k_1$ coefficients are all equal, and so on until the last group of $(k - k_{s-1})$ coefficients which are all equal. Thus the recursion is of the form

$$\begin{aligned} x_n &= \alpha_1(x_{n-1} + \cdots + x_{n-k_1}) + \alpha_2(x_{(n-1-k_1)} + \cdots + x_{n-k_2}) + \\ &\qquad \cdots + \alpha_n(x_{(n-1-k_{s-1})} + \cdots + x_{n-k}) \bmod m. \end{aligned}$$

For example, for a MRG of order $k = 7$, one may consider only recurrences of the form

$$x_n = \alpha(x_{n-1} + x_{n-2}) + \beta(x_{n-3} + x_{n-4} + x_{n-5}) + \gamma(x_{n-6} + x_{n-7}) \bmod m,$$

where α, β, γ are arbitrary integers smaller than m. An extreme case is when all coefficients are the same, as in

$$x_n = \alpha(x_{n-1} + x_{n-2} + x_{n-3} + x_{n-4} + x_{n-5} + x_{n-6} + x_{n-7}) \bmod m.$$

```
#include "Coefficient.h"


namespace LatMRG {

class CoefEqual: public Coefficient {
public:

   CoefEqual (int *I, int s);
```

Constructor. The vector `I` (with $s + 1$ elements), contains the indices $k_0, k_1, k_2, \ldots, k_s$ of the last member of each group of equal coefficients, starting with `I[0]` $= 0$ and ending with `I[s]` $= k$. For example, for a recurrence of order $k = 7$ of the form

$$x_n = \alpha(x_{n-1} + x_{n-2}) + \beta(x_{n-3} + x_{n-4} + x_{n-5}) + \gamma(x_{n-6} + x_{n-7}) \bmod m,$$

and thus with $s = 3$ groups of equal coefficients, `I` must be $(0, 2, 5, 7)$.

```
   ~CoefEqual();
```

Destructor.

```
   void set (const MScal & q, MVect & A, int & i);
```

Sets the *r-th* group of equal coefficients `A[j]` $= a_j$ to the value q, for $j = i, i-1, \ldots, (k_{r-1} + 1)$. The input value of $i = k_r$, the index of the upper member of the *r-th* group. On return, the value of i is reset to $i = k_{r-1} + 1$, the index of the lowest member of the *r-th* group.

```
private:

   int *m_I;
```

Contains the indices as defined in the constructor. It is a pointer to the vector `I` defined elsewhere.

```
   int m_s;
```

The number of different groups of unequal coefficients (it is equal to the value of s in the constructor).

```
};
}
```

CoefZero

This class chooses the coefficients a_i of a recursive generator of order k such that all the coefficients are zero except for a select few. There are s non-zero coefficients: they have indices $k_1, k_2, \ldots, k_s$. Thus the recursion is of the form

$$x_n = \alpha_1 x_{n-k_1} + \alpha_2 x_{n-k_2} + \cdots + \alpha_s x_{n-k} \bmod m.$$

For example, for a MRG of order $k = 101$, one may consider only recurrences with three non-zero coefficients

$$x_n = (\alpha x_{n-3} + \beta x_{n-51} + \gamma x_{n-101}) \bmod m,$$

where α, β, γ are arbitrary integers smaller than m. This allows for a simpler description of the input files and a faster search for such generators than examining separately all zero coefficients.

```
#include "Coefficient.h"


namespace LatMRG {

class CoefZero: public Coefficient {
public:

   CoefZero (int *I, int s);
```

Constructor. The vector `I` (with $s + 1$ elements), contains the indices $k_1, k_2, \ldots, k_s$ of the non-zero coefficients, starting with `I[1]` $= k_1$ and ending with `I[s]` $= k$. For example, for a recurrence of order $k = 7$ of the form

$$x_n = (\alpha x_{n-3} + \beta x_{n-51} + \gamma x_{n-101}) \bmod m,$$

and thus with $s = 3$ non-zero coefficients, `I` must be $(0, 3, 51, 101)$.

```
   ~CoefZero();
```

Destructor.

```
   void set (const MScal & q, MVect & A, int & i);
```

Sets the *r-th* non-zero coefficient `A[i]` $= a_i$ to the value q, for $i = k_r$. On return, the value of i will be reset to $i = k_{r-1} + 1$, where $i = k_{r-1}$ is the index of the next non-zero coefficient.

```
private:

   int *m_I;
```

Contains the indices as defined in the constructor. It is a pointer to the vector `I` defined elsewhere.

```
   int m_s;
```

The number of non-zero coefficients (it is equal to the value of s in the constructor).

```
};

}
```

CoefApproxFact

This class chooses the coefficients a_i of a recursive generator of order k such that the coefficients satisfy the condition $|a_i|(m \bmod |a_i|) < m$, called "approximate factoring", for each i. MRGs are often easier to implement under this condition [24].

```
#include "Coefficient.h"
#include "Zone.h"


namespace LatMRG {

class CoefApproxFact: public Coefficient {
public:

   CoefApproxFact (Zone *Z, MScal & m);
```

Constructor. m is the modulus of congruence of the generator.

```
   ~CoefApproxFact();
```

Destructor.

```
   void set (const MScal & q, MVect & A, int & i);
```

Sets $A[i]$ as a function of q, depending on the zone and m. i is unchanged.

```
private:

   Zone *m_Z;
```

Contains the zone as defined in the constructor. It is a pointer to a zone defined elsewhere.

```
   MScal m_m;
```

The modulus of congruence m as defined in the constructor.

```
};

}
```

Zone

This class implements search zones in parameter space for the coefficients of the recurrences defining generators or lattices.

```
#include "SeekConfig.h"
#include "Random.h"


namespace LatMRG {

class Zone {
public:
```

`enum ZoneType { ZONE1, ZONE2, ZONE3, NZONES };`

Possible zone number. `ZONE1` is the case where $b < -\sqrt{m}$, `ZONE3` is the case where $b > \sqrt{m}$, and `ZONE2` is the case where $-\sqrt{m} \leq b \leq \sqrt{m}$.

`Zone ();`

Constructor.

`~Zone ();`

Destructor.

`void init (const Component & comp, int s, int i);`

Initializes the research zones for the multiplier a_i of `comp` which is the s-th component of the combined generator. In the case of a random search, creates also the region for this multiplier.

```
void calcInfBound (const MScal & b, const MScal & c, const Component & comp,
                   int z, MScal & inf);
```

Computes the lower bound `inf` of the intersection of the zone with the search interval $[b, c]$. If `z` is equal to `ZONE1` or `ZONE3`, the computed bound is such that $\lfloor m/q \rfloor = c$ (approximately). If `z = ZONE2`, the computed bound is b. In any case, the lower bound of the zone is always $\leq b$.

```
void calcSupBound (const MScal & b, const MScal & c, const Component & comp,
                   int z, MScal & sup);
```

Computes the upper bound `sup` of the intersection of the zone with the search interval $[b, c]$. If `z` is equal to `ZONE1` or `ZONE3`, the computed bound is such that $\lfloor m/q \rfloor = b$ (approximately). If `z = ZONE2`, the computed bound is $\min\{c, \sqrt{m}\}$.

`MScal & getInf()`

Returns the lower boundary of this region.

`MScal & getSup()`

Returns the upper boundary of this region.

`MScal & getSupMsH()`

`double getFrac()`

Returns the value of `frac`.

`bool smallF;`

Is `true` if and only if `sup - inf` $\leq$ `H` (or `Hk`).

`ZoneType getNo ()`

Returns the zone number for this region.

`void chooseBoundaries (const Component & comp, Zone *zone);`

Selects a random region and initializes its boundaries. The program will search this region exhaustively.

`static void initFrontieres (const SeekConfig & config);`

Sets the values of the upper boundaries in the three zones based on m_j for each of the J components generators. Also sets the seed for the random number generator used in the random search.

`static MMat Frontiere;`

Upper boundary of each zone.

`static const bool DivQ[NZONES];`

Is `true` if at upper boundary of each zone.

`Zone *nextZone;`

List of zones.

`std::string toString();`

Returns this zone as a string.

`protected:`

`static LatCommon::Random ran;`

The random number generator used in the search.

`ZoneType No;`

The zone number where this region is found.

`MScal inf;`

Lower boundary defining the region.

`MScal sup;`

Upper boundary defining the region.

`double frac;`

The fraction of the acceptable values of the multiplier a lying in this zone.

`MScal supMsH;`

When `small` is `true`, `supMsH` is equal to `sup` $+ 1 -$ `H` (or `sup` $+ 1 -$ `Hk`).

```
   MScal T1, T2;
```
      Work variables.
```
};
}
```

Lacunary

This class implements sets of lacunary indices.

```
#include "TypesNTL.h"
#include "UtilLM.h"
#include <string>


namespace LatMRG {

class Lacunary {
public:
```

`Lacunary (const BVect & C, int t)`

Constructor for a set of t lacunary indices given in $C[j]$, for $j = 1, 2, \ldots, t$.

`explicit Lacunary (int t = 0)`

Constructor for a set of t lacunary indices. The lacunary indices can be read later by `ReadLac`.

`~Lacunary ()`

Destructor.

`BScal & operator[] (int i)`

Returns the lacunary index `m_lac[j]`.

`BScal & getLac (int i)`

Same as above.

```
static void setLac (int toDim, int lacGroupSize, NTL::ZZ & lacSpacing,
                    BVect & Lac);
```

Sets the lacunary indices in `Lac` up to dimension `toDim` by groups of `lacGroupSize`, each group spaced `lacSpacing` apart. `Lac` is assumed to have memory for at least `toDim` + 1 elements.

`int getSize () const`

Returns the size of the lacunary set (the number of lacunary indices).

`bool calcIndicesStreams (int s, int w, int & minDim, int maxDim, int order);`

Computes lacunary indices by groups of s, spaced apart by 2^w. If $w = 0$, this is the case of non-lacunary indices. If `MinDim` is smaller than `Order`, it is reset to `Order` + 1. Returns `true` in the lacunary case, and `false` otherwise.

`std::string toString () const;`

Returns this object as a string.

```
private:
```

`int m_dim;`

The dimension (or size) of `m_lac`.

```
   BVect m_lac;
```

The set of lacunary indices is `m_lac[j]` for $j = 1, 2, \ldots,$ `m_dim`.

```
};
}
```

PolyPE

This class implements polynomials $P(x)$ in $\mathbb{Z}_m[X]$ defined as

$$P(x) = c_0 + c_1 x^1 + c_2 x^2 + \cdots + c_n x^n \tag{5.1}$$

with degree n and integer coefficients c_i in $\mathbb{Z}_m$. The arithmetic operations on objects of this class are done modulo m and modulo a polynomial $f(x)$ of degree k. Thus all polynomials will be reduced modulo $f(x)$. In *LatMRG*, the modulus polynomial $f(x)$ is usually written in the form

$$f(x) = x^k - a_1 x^{k-1} - \cdots - a_{k-1} x - a_k, \tag{5.2}$$

and is associated with the recurrence

$$x_n = (a_1 x_{n-1} + a_2 x_{n-2} + \cdots + a_k x_{n-k}) \bmod m. \tag{5.3}$$

The two functions `setM` and `setF` *must* be called to initialize the modulus m and the modulus polynomial $f(x)$ before doing any arithmetic operations on `PolyPE` objects, otherwise the results are unpredictable.

Type `MScal` is used to represent polynomial coefficients. It may be chosen as `long` for $m < 2^{50}$ (on 64-bit machines), or as the big integer type `ZZ` otherwise. The possible associated types `MVect` are `long*` and `vec_ZZ`. Type `PolE` for the polynomials may be chosen as `zz_pE` when $m < 2^{50}$, or it may be set to `ZZ_pE` which is implemented with the big integer type `ZZ_p`.

```
#include "TypesNTL.h"
#include "IntFactorization.h"
#include "IntPrimitivity.h"
#include <string>


namespace LatMRG {

class PolyPE : public PolE {
public:

   static void setM (const MScal & m);

      Initializes the modulus m for this class. This must be called before doing any operations on PolyPE
      objects, otherwise the results are unpredictable.

   static const MScal & getM ()

      Returns a read-only reference to m.

   static void setF (const MVectP & C);

      Initializes the modulus polynomial f(x) = c_0 + c_1 x + c_2 x^2 + ... + c_k x^k of degree k for this class from
      the coefficients c_i = C[i] of vector C of dimension k + 1. This function must be called before doing
      any arithmetic operations on PolyPE objects, otherwise the results are unpredictable.

   static void setF (const MVect & C);

      Same as above, but instead from a MVect.
```

`static const PolX & getF ()`

Returns the modulus polynomial $f(x)$.

`static long getK ()`

Returns the degree of the modulus $f(x)$.

`static void reverse (MVect & c, long k, int kind);`

Given a vector $C = [c_0, c_1, \ldots, c_{k-1}, c_k]$, this function reorders the components of C in the form $C = [c_k, c_{k-1}, \ldots, c_1, c_0]$ for `kind = 1`, and in the form $C = [-c_k, -c_{k-1}, \ldots, -c_1, 1]$ for `kind = 2`. For other values of `kind`, it has no effect.

`PolyPE ();`

Minimal constructor: this object is set to the **0** polynomial.

`const PolX & getVal ()`

`void setVal (long j);`

`void setVal (const MVect & C);`

Initializes this object to C.

`void setVal (std::string & str);`

Initializes this object to the polynomial in `str`.

`void powerMod (const MScal & j);`

Sets $v = x^j \bmod f(x) \pmod m$.

`void toVector (MVect & c);`

Returns the coefficients of this polynomial as a vector C of k components, where k is the degree of the modulus $f(x)$.

`bool isPrimitive (const IntPrimitivity & fm, const IntFactorization & fr);`

Returns `true` if the modulus $f(x)$ is a primitive polynomial modulo m. For this to be true, assuming that $f(x)$ has the form (5.2) above, the three following conditions must be satisfied:

1. $[(-1)^{k+1}a_k]^{(m-1)/q} \bmod m \neq 1$ for each prime factor q of $m-1$;
2. $x^r \bmod (f(x), m) \;=\; (-1)^{k+1}a_k \bmod m$;
3. $x^{r/q} \bmod (f(x), m)$ has positive degree for each prime factor q of r, with $1 < q < r$;

where $r = (m^k - 1)/(m-1)$. The factorizations of $m-1$ and r must be in `fm` and `fr` respectively. Condition 1 is the same as saying that $(-1)^{k+1}a_k$ is a primitive root of m. Condition 3 is automatically satisfied when r is prime.

`bool isPrimitive (const IntFactorization & r);`

Given the factorization of r, this method returns `true` if conditions 2 and 3 above are satisfied by the modulus $f(x)$. It does not check condition 1, assuming it to be `true`.

```
   std::string toString () const;
```

Returns this object as a string.

```
private:

   static MScal m_m;
```

Modulus of congruence.

```
   static long m_k;
```

Degree of the modulus polynomial f.

```
};

}
```

IntPrimitivity

This class deals with primitive roots and primitive elements modulo an integer. Let a, e and p be integers, with p a prime number. Assume also that a and $m = p^e$ are relatively prime. The smallest positive integer $\lambda(m)$ for which $a^\lambda = 1 \pmod{m}$ is called the order of a modulo m. Any a which has the maximum possible order for a given m is called a *primitive root* modulo m. For the following important cases, the value of the order for given m is [22]:

$$\begin{aligned} \lambda(2^e) &= 2^{e-2}, \qquad e > 2 \\ \lambda(p^e) &= p^{e-1}(p-1), \qquad p > 2. \end{aligned}$$

```
#include <stdexcept>
#include "TypesNTL.h"
#include "IntFactorization.h"


namespace LatMRG {

class IntPrimitivity {
public:

   IntPrimitivity ();

   IntPrimitivity (const IntFactorization & f, const MScal & p, long e = 1);
```

Constructor fixing the modulus of congruence as $m = p^e$. The argument f must contain the prime factor decomposition of $p - 1$ and its inverse factors.

```
   bool isPrimitiveElement (const MScal & a) const throw(std::range_error);
```

Returns `true` if a is a primitive element modulo p^e. This method uses the prime factor decomposition of $p - 1$ and its inverse factors.

```
   bool isPrimitiveElement (const MVect &V, int k) const throw(std::range_error);
```

Returns `true` if $(-1)^{k+1}V[k]$ is a primitive element modulo p^e. This method uses the prime factor decomposition of $p - 1$ and its inverse factors.

```
   void setpe (const MScal & p, long e);
```

Sets the value of p, e and $m = p^e$.

```
   MScal getP ()
```

Gets the value of p.

```
   long getE ()
```

Gets the value of e.

```
   void setF (const IntFactorization & f)
```

Sets the value of f.

`IntFactorization getF ()`

Gets the value of f.

`std::string toString () const;`

Returns this object as a string.

`private:`

`MScal m_p;`

Prime number p.

`long m_e;`

Exponent used to compute the modulus $m = p^e$.

`MScal m_m;`

The modulus $m = p^e$.

`IntFactorization m_f;`

Factorization of $p - 1$.

`};`

`}`

IntFactorization

Given any natural integer n, there is a unique decomposition in prime factors of the form

$$n = p_1^{\nu_1} p_2^{\nu_2} \cdots p_s^{\nu_s}$$

where p_i is a prime integer with ν_i its multiplicity, and where the factors are sorted in increasing order. In the case of very large integers, it may not be possible to find all the prime factors within a reasonable amount of time. In that case, a similar decomposition to the above may be used with some of the factors composite.

The class `IntFactorization` implements the decomposition of integers in factors, preferably prime (see class `IntFactor`). It contains functions to factorize an integer in prime factors, to sort and print the list of its factors. Integers are factorized by calling the MIRACL software [38], which uses many different methods in succession to effect the factorization.

```
#include <vector>
#include <list>
#include <string>
#include <stdexcept>

#include "TypesNTL.h"
#include "Const.h"
#include "IntFactor.h"


namespace LatMRG {

class IntFactorization {
public:
```

`explicit IntFactorization (const MScal & x);`

Integer x is the number whose "prime" factors decomposition is kept in this object.

`explicit IntFactorization (const char *fname = 0);`

Integer `number` and its prime factors will be read from file `fname` by calling method `read` below. If no argument is given, `number` is initialized to 0.

`~IntFactorization ();`

Destructor.

`IntFactorization (const IntFactorization & f);`

Copy constructor.

`IntFactorization & operator= (const IntFactorization & f);`

Assignment operator.

`void clear ();`

Empties the lists of primes factors and set this number to 0.

`void read (const char *f) throw (std::invalid_argument);`

Reads the list of (possibly) prime factors of a number from file `f`. The first line contains the number itself. The following lines contain one factor per line: the factor (first field) with its multiplicity (second field) and its status (third field). The status field is written as `P` if the factor is known to be prime, `Q` if the factor is probably prime, `C` if the factor is composite, and `U` if its status is unknown or unimportant. For example, given the number $120 = 2^3 * 3 * 5$, then the file must be

```
120
2    3    P
3    1    P
5    1    P
```

```
void addFactor (const MScal & p, int mult = 1,
                LatCommon::PrimeType st = LatCommon::UNKNOWN);
```

Adds factor p with multiplicity `mult` and prime status `st` to this object.

`void unique ();`

Replaces repeated equal factors in `factorList` by one factor with its multiplicity. Also sorts the factors.

`void factorize ();`

Tries to find all the prime factors of this number.

`bool checkProduct () const;`

Checks that the number is equal to the product of its factors. Returns `true` if it is, otherwise `false`.

`void calcInvFactors ();`

Given the list of prime factors p of `number`, fills the list of inverse factors with the values `number`$/p$.

`MScal getNumber () const`

Returns the value of this number.

`const std::list<IntFactor> & getFactorList () const`

Returns a non-mutable list of the factors.

`const std::vector<MScal> & getInvFactorList () const`

Returns a non-mutable list of the inverse factors.

`void setNumber (const MScal & x)`

Sets the value of this number to x.

`LatCommon::PrimeType getStatus () const`

Returns the status of this number.

`void setStatus (LatCommon::PrimeType s)`

Sets the status of this number to s.

```
   std::string toString () const;
```

Returns the list of (possibly) prime factors of this object in the same format as described in method `read` above.

```
private:

   MScal m_number;
```

The number whose "prime" factor decomposition is kept in this object.

```
   LatCommon::PrimeType m_status;
```

The status of this number, i.e. whether it is prime, composite, ...

```
   std::list<IntFactor> m_factorList;
```

The list of the "prime" factors in the decomposition of `number`.

```
   std::vector<MScal> m_invFactorList;
```

Given the list of prime factors p of `number`, `invFactorList` contains all the sorted values `number`$/p$ (indexing starts at 0). However, one must have called the function `calcInvFactors` beforehand. For example, if `number` $= 24$, its prime factors decomposition is $24 = 2^3 \cdot 3$, and `invfactorList` $= [8, 12]$.

```
   class CompFactor {
   public:
      bool operator() (const IntFactor & f1, const IntFactor & f2) {
         return f1.getFactor() < f2.getFactor(); }
   };
```

Nested class used to sort the prime factors of a number in increasing order. Returns `true` if the factor of `f1` is smaller than the factor of `f2`; otherwise returns `false`.

```
   void sort () { CompFactor comp; m_factorList.sort (comp); }
```

Sorts the list of factors in increasing order, the smallest factor first.

```
};

}
```

IntFactor

The objects of this class are the "prime" factors in the decomposition of a positive integer. The class contains functions to determine whether a number is prime, probably prime or composite. The related class `IntFactorization` contains the list of "prime" factors. (This class should be a private nested class inside `IntFactorization`.)

```
#include <string>
#include "TypesNTL.h"
#include "Const.h"


namespace LatMRG {

class IntFactor {
public:

   IntFactor (const MScal & x, int mult = 1,
              LatCommon::PrimeType stat = LatCommon::UNKNOWN)
```

Constructor for a factor x, with multiplicity `mult` and status `stat`.

```
   MScal getFactor () const
```

Returns the value of `factor`.

```
   void setFactor (const MScal & x)
```

Sets the value of `factor` to x.

```
   int getMultiplicity () const
```

Returns the multiplicity of this object.

```
   void setMultiplicity (int m)
```

Sets the multiplicity of this object to m.

```
   LatCommon::PrimeType getStatus () const
```

Returns the status of this object.

```
   void setStatus (LatCommon::PrimeType s)
```

Sets the status of this object to s.

```
   static LatCommon::PrimeType isPrime (const MScal & y, long k);
```

Tests whether y is prime. First tests whether y is divisible by all small primes p ($p < 2^{16}$) that are kept in file `prime.dat`. Then applies the Miller-Rabin probability test with k trials.

```
   LatCommon::PrimeType isPrime (long k)
```

Tests whether this factor is prime. Similar to `isPrime` above.

```
   static std::string toString (LatCommon::PrimeType stat);
```

Transforms status `stat` in an easily readable string and returns it.

```
   std::string toString () const;
```

Returns this object as a string.

```
private:

   MScal m_factor;
```

The factor.

```
   int m_multiplicity;
```

The multiplicity of this factor.

```
   LatCommon::PrimeType m_status;
```

The status of this factor, i.e. whether it is prime, composite, ...

```
   static LatCommon::PrimeType isProbPrime (const MScal & y, long k);
```

Applies the Miller-Rabin probability test with k trials to y.

```
};

}
```

LatConfig

This class is used to save the configuration of a lattice test. It is used to keep all the parameters read in the data file and passed to different methods for the spectral, Beyer or P_α tests.

```
#include "TypesNTL.h"
#include "Const.h"
#include "MRGComponent.h"
#include "NTL/ZZ.h"


namespace LatMRG {

class LatConfig {
public:
```

`LatConfig();`

Constructor.

`~LatConfig();`

Destructor.

`void kill();`

Frees the memory used by this object.

`void write();`

For debugging: writes the configuration on the console.

`void setJ (int j);`

Reinitializes this object and allocates enough memory for j MRGs.

`bool readGenFile;`

This flag is set `true` if the generator is to be read from a file, otherwise it is set `false`.

`std::string fileName;`

If `readGenFile` is `true`, the name of the file from which to read the generator.

`int J;`

The number of MRG components.

`LatCommon::GenType *genType;`

The array of generator types for each MRG component. See module `Const` for more details.

`MRGComponent **comp;`

The array of MRG components which describe the combined generator.

```
//   int fromDim;
```

The minimal dimension for which the test will be performed. ********** REMPLACÉ PAR `td[0]`. À ÉLIMINER.

`//  int toDim;`

The maximal dimension for which the test will be performed. ********** REMPLACÉ PAR `td[1]`. À ÉLIMINER.

`int d;`

The number of categories of projections (see `td` below). The classical case corresponds to $d = 1$, for which the chosen test is done on all successive dimensions from `td[0] = fromDim`, up to and including `td[1] = toDim`.

`int *td;`

Array containing the maximal dimensions for the projections in each category. `td[1]` is the maximal dimension for successive 1-dimensional projections, `td[2]` is the maximal dimension for 2-dimensional projections, `td[3]` is the maximal dimension for 3-dimensional projections, and so on. The value of `td[0]` is the minimal dimension for the case of successive dimensions.

`LatCommon::CriterionType criter;`

The criterion for which the test will be performed. See module `Const` for the possible criterion types.

`LatCommon::NormaType norma;`

The bound used for the normalization in the definition of S_t. Only applicable for the spectral test.

`LatCommon::CalcType calcPalpha;`

The type of the calculation in the P_α test.

`int alpha;`

The value of α for the P_α test.

`std::vector<double> Beta;`

The values of B_i, $i = 0, 1, \ldots,$`toDim` for the P_α test.

`int seed;`

The seed for the generator in the P_α test.

`bool dualF;`

This flag is set `true` if the test is to be applied on the dual lattice. If it is `false`, the test is applied on the primal lattice.

`bool invertF;`

If `invertF` is `true`, the inverse of the length of the shortest vector will be printed in the results. Otherwise, the length itself will be printed.

`int detailF;`

This flag indicates to print more detailed results if `detailF` > 0. Default value: 0.

`LatCommon::NormType norm;`

Norm used to measure the length of vectors. See module `Const` for a definition of the possible norms.

```
   LatCommon::LatticeType latType;
```

Indicates the type of lattice used in the test. See `Const` for a definition of the possible lattice types.

```
   bool lacunary;
```

This flag is set `true` if the test is applied for lacunary indices. If it is `false`, the test is applied for successive indices.

```
   int lacGroupSize;
   NTL::ZZ lacSpacing;
```

Used for lacunary indices. If the respective values are s and d, then the program will analyze the lattice structure of vectors formed by groups of s successive values, taken d values apart, i.e. groups of the form $(u_{i+1}, \ldots, u_{i+s}, u_{i+d+1}, \ldots, u_{i+d+s}, u_{i+2d+1}, \ldots, u_{i+2d+s}, \ldots)$.

```
   BVect Lac;
```

The lacunary indices, either read explicitly or computed from `lacGroupSize` and `lacSpacing`.

```
   bool primeM;
```

Is `true` when the modulus of congruence m is a prime number, is `false` otherwise.

```
   bool verifyM;
```

If `true`, the program will verify that the modulus m is a prime number. If `false`, will not verify it.

```
   bool maxPeriod;
```

Is `true` when the generator has maximal period, is `false` otherwise.

```
   bool verifyP;
```

If `true`, the program will verify that the generator has maximal period. If `false`, will not verify it.

```
   long maxNodesBB;
```

The maximum number of nodes to be examined in any given branch-and-bound procedure when computing d_t or q_t.

```
   LatCommon::OutputType outputType;
```

File format used to store the results. See `Const` for a definition of the possible output types.

```
};

}
```

SeekConfig

♣ Compléter la description de la classe

À EXAMINER: il pourrrait y avoir des avantages à inclure une variable `LatConfig` à l'intérieur de `SeekConfig`. Plusieurs variables n'auraient pas à être répétées dans `SeekConfig`, et la recherche appelle les tests avec des paramètres de `LatConfig`, parfois explicitement, ce qui cause des problèmes (cas PALPHA). Cela pourrait simplifier le code.

```
#include <string>
#include <vector>

#include "TypesNTL.h"
#include "Const.h"
#include "Modulus.h"
#include "UtilLM.h"


namespace LatMRG {

struct Component {
   LatCommon::GenType genType;        // Generator type: MRG, MWC
   Modulus modulus;        // Modulus m
   int k;                  // Generator order
   bool PerMax;            // True if maximal period is required, else false
   LatCommon::ImplemCond implemCond;
   int NumBits;
   int HighestBit;
   int ncoef;
   int *Icoef;
   LatCommon::DecompType F1;
   std::string file1;
   LatCommon::DecompType F2;
   std::string file2;
   LatCommon::SearchMethod searchMethod;
   int numReg, H, Hk;
   MVect b, c;             // intervals where to search the coefs
   bool ApproxTotGen;
};


class SeekConfig {
public:

   SeekConfig();
   ~SeekConfig();
   void write();
   MScal* getMs() { return Ms; }
   int* getKs()  { return Ks; }
   int getMaxK();
   int getMaxTd();
   bool readGenFile; // read generator from file
   std::string fileName;  // file name
   int J;            // nombre de generateurs dans la combinason
```

```
   std::vector<Component> compon;
   MScal* Ms;
   MVect* As;
   int* Ks;
   int C;
   double* minMerit;
   double* maxMerit; // C values
   int* numGen; // C values

   LatCommon::CriterionType criter;
   LatCommon::NormaType normaType;   // Criterion <Norm>

   int d;
```

The number of series of projections (see `td` below). The classical case corresponds to $d = 1$, for which the chosen test is done on all successive dimensions from `td[0] = fromDim`, up to and including `td[1] = toDim`.

```
   int *td;
```

Array containing the maximal dimensions for the projections in each category. `td[1]` is the maximal dimension for successive 1-dimensional projections, `td[2]` is the maximal dimension for 2-dimensional projections, `td[3]` is the maximal dimension for 3-dimensional projections, and so on. However, the value of `td[0]` is the minimal dimension for the case of successive dimensions.

```
   int getMaxDim()
   LatCommon::LatticeType latType;

   bool dualF;
```

If this flag is `true`, the test is to be applied on the dual lattice. If it is `false`, the test is to be applied on the primal lattice.

```
   bool invertF;
```

If `invertF` is `true`, the inverse of the length of the shortest vector will be printed in the results. Otherwise, the length itself will be printed.

```
   int alpha;
```

The value of α for the P_α test.

```
   int lacGroupSize;
   int lacSpacing;
   long maxNodesBB;
   double duration;
   long seed;  // seed of the random number generator
   LatCommon::OutputType outputType;
};

}
```

ParamReader

Utility class used to read basic parameter fields in a configuration file. Lines whose first non-blank character is a # are considered as comments and discarded.

```
#include "NTL/ZZ.h"
#include "TypesNTL.h"
#include "UtilLM.h"
#include "Const.h"
#include "MRGComponent.h"
#include <string>
#include <vector>


namespace LatMRG {

class ParamReader {
public:
```

`ParamReader();`

Constructor.

`ParamReader (std::string fileName);`

Constructor. Opens the file `fileName`.

`~ParamReader();`

Destructor.

`void getLines();`

Reads all the lines from the file and stores them into this object's buffer. Lines whose first non-blank character is a # are considered as comments and discarded. Empty lines are also discarded.

`void getToken (std::string & field, unsigned int ln, unsigned int pos);`

Puts into `field` the `pos`-th string token from line `ln`.

`int tokenize (std::vector<std::string> & tokens, unsigned int ln);`

Splits line `ln` from the file into several string tokens. Separator characters are defined in function `IsDelim`. Tokens are stored in vector `tokens`.

`void readString (std::string & field, unsigned int ln, unsigned int pos);`

Reads a string from the `pos`-th token of the `ln`-th line into `field`.

`void readBool (bool & field, unsigned int ln, unsigned int pos);`

Reads a boolean from the `pos`-th token of the `ln`-th line into `field`.

`void readChar (char & field, unsigned int ln, unsigned int pos);`

Reads a character from the `pos`-th token of the `ln`-th line into `field`.

`void readInt (int & field, unsigned int ln, unsigned int pos);`

Reads an integer from the `pos`-th token of the `ln`-th line into `field`.

```
void readLong (long & field, unsigned int ln, unsigned int pos);
```

Reads a long from the `pos`-th token of the `ln`-th line into `field`.

```
void readZZ (NTL::ZZ & field, unsigned int ln, int pos);
```

Reads a large integer from the `pos`-th token of the `ln`-th line into `field`.

```
void readDouble (double & field, unsigned int ln, unsigned int pos);
```

Reads a double from the `pos`-th token of the `ln`-th line into `field`.

```
void readGenType (LatCommon::GenType & field, unsigned int ln,
                  unsigned int pos);
```

Reads a `GenType` from the `pos`-th token of the `ln`-th line into `field`.

```
void readNumber3 (MScal & r, long & b, long & e, long & c,
                  unsigned int ln, unsigned int pos);
```

Reads b, e and c, starting at the `pos`-th token of the `ln`-th line and uses them to define r. The numbers in the data file may be given in one of the two following formats:

- A single integer giving the value of $r = b$ directly on a line. In that case, one sets $e = c = 0$.
- Three integers b, e, c on the same line, separated by at least one blank. The r value will be set as $r = b^e + c$ if $b > 0$, and $r = -(|b|^e + c)$ if $b < 0$. One must have $e \geq 0$. For example, $(b, e, c) = (2, 5, -1)$ will give $r = 31$, while $(b, e, c) = (-2, 5, -1)$ will give $r = -31$.

```
void readBScal (BScal & field, unsigned int ln, int pos);
```

Reads a `BScal` from the `pos`-th token of the `ln`-th line into `field`.

```
void readMScal (MScal & field, unsigned int ln, unsigned int pos);
```

Reads a `MScal` from the `pos`-th token of the `ln`-th line into `field`.

```
void readRScal (RScal & field, unsigned int ln, unsigned int pos);
```

Reads a `RScal` from the `pos`-th token of the `ln`-th line into `field`.

```
void readMVect (MVect & field, unsigned int & ln, unsigned int pos,
                unsigned int num, int j);
```

Reads `num` tokens (from the `pos`-th token of the `ln`-th line) into `field`, starting at index j of `field`.

```
void readIntVect (int* field, unsigned int ln, unsigned int pos,
                  unsigned int num, int j);
```

Reads `num` tokens (from the `pos`-th token of the `ln`-th line) into `field`, starting at index j of `field`.

```
void readDoubleVect (double* field, unsigned int ln, unsigned int pos,
                     unsigned int num, int j);
```

Reads `num` tokens (from the `pos`-th token of the `ln`-th line) into `field`, starting at index j of `field`.

```
void readInterval (MVect & B, MVect & C, unsigned int & ln, int k);
```

Reads `2k MScal` tokens into vectors `B` and `C`, starting at the `ln`-th line. These represent a box $[B_i, C_i]$, $i = 1, 2, \ldots, k$. The B_i, C_i must be given in the order $B_1, C_1, B_2, C_2, \ldots, B_k, C_k$, each on a line of its own. Each coefficient may be given in the form described in `readNumber3` above.

```
void readCriterionType (LatCommon::CriterionType & field, unsigned int ln,
                        unsigned int pos);
```

Reads a criterion from the `pos`-th token of the `ln`-th line into `field`.

```
void readNormType (LatCommon::NormType & field, unsigned int ln,
                   unsigned int pos);
```

Reads a norm from the `pos`-th token of the `ln`-th line into `field`.

```
void readNormaType (LatCommon::NormaType & field, unsigned int ln,
                    unsigned int pos);
```

Reads a type of normalization from the `pos`-th token of the `ln`-th line into `field`.

```
void readCalcType (LatCommon::CalcType & field, unsigned int line,
                   unsigned int pos);
```

Reads the type of calculation to do (`PAL` or `BAL`) for the `PALPHA` test from the `pos`-th token of the `ln`-th line into `field`.

```
void readDecompType (LatCommon::DecompType & field, unsigned int line,
                     unsigned int pos);
```

Reads the decomposition type from the `pos`-th token of the `ln`-th line into `field`.

```
void readLatticeType (LatCommon::LatticeType & field, unsigned int ln,
                      unsigned int pos);
```

Reads a lattice type from the `pos`-th token of the `ln`-th line into `field`.

```
void readOrbit (int J, MRGComponent **comp, unsigned int & ln);
```

Reads the initial state for an orbit of a combined generator with J components `comp`, starting at the `ln`-th line. Each line must have a set of k_j numbers which are the seeds of the orbit for each component, where k_j is the order of the j-th component. On exiting this method, the line number is reset to `ln` + J.

```
bool checkPrimePower (LatCommon::LatticeType lat, long e, long c, int k);
```

In the case where `lat = PRIMEPOWER`, checks that the modulus m is given in the form $m = b^e$, (that is $e > 0$ and $c = 0$. Checks also that the order $k = 1$. If these conditions are not satisfied, stops the program.

```
void readLacunary (int k, int toDim, unsigned int & ln, bool & lacunary,
                   int & lacGroupSize, NTL::ZZ & lacSpacing, BVect & Lac);
```

Reads the fields, starting at the `ln`-th line, for a generator of order k and for a test in dimension up to `toDim`, to determine the lacunary indices, which are positive integers that will be read into `Lac`. The vector of lacunary indices `Lac` is created here with the appropriate dimension. If the first two values read are $s =$ `lacGroupSize` and $d =$ `lacSpacing` and if $s > 0$, then what will be analyzed is the lattice structure of vectors of the form $(u_{i+1}, \ldots, u_{i+s}, u_{i+d+1}, \ldots, u_{i+d+s}, u_{i+2d+1}, \ldots, u_{i+2d+s}, \ldots)$, formed by groups of s successive values, taken d values apart. To analyze lacunary indices that are not evenly spaced, put $s = -t$ where $t =$ `toDim` and on the t lines that follow, give the t lacunary indices $i_1, \ldots, i_t$, which are to be interpreted as in Section 1.4. In all these cases, the `lacunary` flag is set `true`. To analyze vectors of successive values (the non-lacunary case), take $s = d = 1$ or $s \geq$ `toDim`. In this case, the `lacunary` flag is set `false`.

```
   void readOutputType (LatCommon::OutputType & field, unsigned int ln,
                        unsigned int pos);
```

Reads an output form from the `pos`-th token of the `ln`-th line into `field`.

```
   void readImplemCond (LatCommon::ImplemCond & field, unsigned int ln,
                        unsigned int pos);
```

Reads an implementation condition from the `pos`-th token of the `ln`-th line into `field`.

```
   void readSearchMethod (LatCommon::SearchMethod & field, unsigned int ln,
                          unsigned int pos);
```

Reads a search method from the `pos`-th token of the `ln`-th line into `field`.

```
   bool checkBound (const MScal & m, const MVect & A, int k);
```

Checks that the components of A satisfy $-m < A_i < m$, for $i = 1, 2, \ldots, k$.

```
private:

   std::vector<std::string> m_lines;
```

Internal line buffer.

```
   std::string m_fileName;
```

The path of the opened file.

```
   bool isDelim (char c);
```

Checks if the character `c` is to be considered as a token separator or not.

```
   void init() {}
```

Does nothing for now.

```
};

}
```

ParamReaderLat

This class is used to read a configuration file for the executable programs `lat*`, created from the `LatMain` main program. The format of the configuration file is described in this guide for the program `LatMain` on page 29.

```
#include "ParamReader.h"
#include "LatConfig.h"

#include <string>


namespace LatMRG {

class ParamReaderLat : public ParamReader {
public:

   ParamReaderLat (std::string fileName);
```

Constructor. Opens configuration file `fileName`.

```
   ~ParamReaderLat();
```

Destructor.

```
   void read (LatConfig & config);
```

Reads the configuration file into `config` for the Beyer and the spectral tests.

```
};

}
```

ParamReaderSeek

This class is used to read a configuration file for the executable programs `seek*`, created from the `SeekMain` main program. The format of the configuration file is described in this guide for the program `SeekMain` on page 34.

```
#include "ParamReader.h"
#include "SeekConfig.h"

#include <string>


namespace LatMRG {

class ParamReaderSeek : public ParamReader {
public:

   ParamReaderSeek();
```

Default Constructor.

```
   ParamReaderSeek (std::string fname);
```

Constructor. `fname` is the name of the configuration file.

```
   ~ParamReaderSeek();
```

Destructor.

```
   void init();
```

Variables initialization.

```
   void read (SeekConfig & config);
```

Reads the configuration parameters and puts them in an instance of `SeekConfig`.

```
};

}
```

ReportHeader

This is an abstract class that must implemented to print a header in `ReportLat` or `ReportSeek`.

```
#include "Writer.h"


namespace LatMRG {

class ReportHeader {
public:

   ReportHeader (Writer *writer)
```

Constructor. See the module `Writer` for more information.

```
   virtual ~ReportHeader()
```

Destructor.

```
   virtual void printHeader() = 0;
```

Writes the report header using the `Writer` passed to the constructor.

```
protected:

   Writer* m_writer;
```

The `Writer` used to write the report header.

```
};

}
```

ReportHeaderLat

This class is an implementation of the `ReportHeader` abstract class for the programs `lat*`. It prints the configuration of the test launched, the MRG or MWC components, and the combined MRG if applicable.

```
#include "ReportHeader.h"
#include "LatConfig.h"
#include "MRGLattice.h"
#include "Writer.h"


namespace LatMRG {

class ReportHeaderLat : public ReportHeader {
public:

   ReportHeaderLat (Writer *writer, LatConfig *config,
                    MRGLattice *lattice);
```

Constructor. `writer` is the writing engine used to write the report header. `config` is the configuration of the lattice test to be performed, which is populated from an instance of `ParamReaderLat`. `lattice` is the final MRG lattice on which the lattice test will be performed. It can be the result of a combination of MRG components, a MWC transformed into a MRG, and so on.

```
   void printHeader();
```

Does the actual writing of the report header with the `Writer` passed to the constructor.

```
private:

   LatConfig* m_config;
```

Pointer to the configuration of the lattice test.

```
   MRGLattice* m_lattice;
```

Pointer to the final MRG lattice on which the lattice test will be performed.

```
};

}
```

ReportFooter

This is an abstract class that must be implemented to print a footer in `ReportLat` or `ReportSeek`.

```
#include "Writer.h"


namespace LatMRG {

class ReportFooter {
public:

   ReportFooter (Writer *writer)
```

Constructor. See the module `Writer` for more information.

```
   virtual ~ReportFooter()
```

Destructor.

```
   virtual void printFooter() = 0;
```

Writes the report footer using the `Writer` passed to the constructor.

```
protected:

   Writer* m_writer;
```

The `Writer` to be used to write the report footer.

```
};

}
```

ReportFooterLat

This class is an implementation of the `ReportFooter` abstract class for the program `lat*`.

```
#include "ReportFooter.h"
#include "Writer.h"
#include "LatticeTest.h"


namespace LatMRG {

class ReportFooterLat : public ReportFooter {
public:

   ReportFooterLat (Writer *, LatticeTest * test = 0);
```

Constructor. `writer` is the writing engine used to write the report footer. `test` is the lattice test thas was performed and for which the results are to be written.

```
   void setLatticeTest (LatticeTest * test)
```

`test` is the lattice test thas was performed and for which the results are to be written.

```
   void printFooter();
```

Defined in abstract class `ReportFooter`.

```
private:

   LatticeTest * m_test;
```

Pointer to the final lattice test which was performed.

```
};

}
```

ReportLat

This class formats and prints the actual report for a lattice test for the program `lat*`. It implements the interface `LatticeTestObserver` to be able to receive information and results from the lattice test.

```
#include "LatticeTestObserver.h"
#include "ReportHeader.h"
#include "ReportFooter.h"
#include "DoubleFormatter.h"
#include "FormatterImpl.h"
#include "Writer.h"
#include "LatConfig.h"

#include <string>


namespace LatMRG {

class ReportLat : public LatticeTestObserver {
public:

   ReportLat (Writer* writer, LatConfig* config, ReportHeader* header,
              ReportFooter* footer);
```

Constructor.

```
   ~ReportLat();
```

Destructor.

```
   void printHeader();
```

Prints the header using the `ReportHeader` passed to the constructor.

```
   void printFooter();
```

Prints the footer using the `ReportFooter` passed to the constructor.

```
   void printTable();
```

Prints the table of results obtained from the successive calls of `resultUpdate`. If more than one tests are performed, the results will be concatenated in the same table.

```
   void baseUpdate (LatCommon::Base &);
```

Defined in interface `LatticeTestObserver`. Prints the base directly in the report.

```
   void baseUpdate (LatCommon::Base & V, int i);
```

Defined in interface `LatticeTestObserver`. Prints basis vector $V[i]$ directly in the report.

```
   void resultUpdate (double[], int);
```

Defined in interface `LatticeTestObserver`. The results are stacked in the internal table and will be printed upon a call to `printTable`.

`void testInit (const std::string &, std::string[], int);`

Defined in interface `LatticeTestObserver`. The columns in the internal table are set up to be able to receive results from calls to `resultUpdate`.

`void testCompleted();`

Defined in interface `LatticeTestObserver`. Indicates a successful test.

`void testFailed (int);`

Defined in interface `LatticeTestObserver`. Indicates a failed test. An error message is printed in the report.

`Writer * getWriter()`

Returns the writing engine used in this class

```
private:
```

`Writer * m_writer;`

Writing engine used to print the report.

`ReportHeader * m_header;`

Report header used to print the report.

`ReportFooter * m_footer;`

Report footer used to print the report.

`int m_base_col;`

Indicates the index of the first column in the results table to insert results for the current test.

`int m_next_base;`

Indicates the index of the first column in the results table to insert results for the next test.

`Table m_results;`

The internal table that contains the results of lattice test(s).

`LatConfig * m_config;`

The configuration of the lattice test(s).

`DoubleFormatter m_dFormat;`

Formatter used to format the results in the table when writing the report.

`FormatterImpl<int> m_iFormat;`

Formatter used to format the dimensions (first column) in the table when writing the report.

```
};

}
```

Writer

This is the abstract class that does the writing of basic elements (`string`'s, `int`'s, `double`'s, etc.) into a file or into an `ostream`. Derived classes must be implemented to write in different formats, for instance text or LaTeX.

```
#include "TypesNTL.h"
#include "Table.h"

#include <iostream>
#include <string>


namespace LatMRG {

class Writer {
public:
```

`Writer (const char* fileName);`

Constructor. Opens a `Writer` to write in the file `filename`.

`Writer (std::ostream* stream);`

Constructor. Opens a `Writer` to write directly in an `ostream`.

`virtual ~Writer();`

Destructor.

`virtual void beginTabbedSection() = 0;`

Begins a tabbed section. In a tabbed section, every element is aligned at a tab start.

`virtual void endTabbedSection() = 0;`

Ends a tabbed section.

`virtual void addTab() = 0;`

Advances the tab start once.

`virtual void removeTab() = 0;`

Moves back the tab start once.

`virtual void clearTab() = 0;`

Resets the tab start to the beginning of the line.

`virtual void newLine() = 0;`

Starts a new line. If in a tabbed section, the new line starts at the current tab start position.

`virtual void newParagraph() = 0;`

Starts a new paragraph. Ends automaticaly a tabbed section.

```
   virtual void writeBool (const bool & value);
```

Writes a `bool`.

```
   virtual void writeInt (const int & value);
```

Writes an `int`.

```
   virtual void writeString (const std::string & value);
```

Writes a `string`.

```
   virtual void writeDouble (const double & value);
```

Writes a `double`.

```
   virtual void writeMScal (const MScal & value);
```

Writes a `MScal`.

```
   virtual void writeMathString (const std::string) = 0;
```

Writes a `string` in a mathematical format using LaTeX notation.

```
   virtual void writeStandOutMathString (const std::string) = 0;
```

Writes a `string` just as in an `equation` environment in LaTeX.

```
   virtual void writeTable (Table &, const std::string alignment) = 0;
```

Writes a formatted table. A column horizontal alignment can be modified using the string `alignment`. The first character of the string is the alignment of the first column, the second character is the alignment of the second column and so on. Three alignments are possible for a column: `r` for right alignment, `c` for center alignment, `l` for left alignment. If nothing is given, or if the string length is smaller than the number of columns of the table, then left alignment is used by default for the rest of the columns.

```
protected:

   std::ostream* m_stream;
```

The stream in which the writings are done.

```
private:

   bool m_clean;
```

Set to `true` if this object needs to clean `_stream` upon destruction.

```
};

}
```

WriterRes

This class implements the `Writer` abstract class to write basic elements in plain text format.

```
#include "Writer.h"
#include "Table.h"

#include <iostream>
#include <fstream>
#include <string>


namespace LatMRG {

class WriterRes : public Writer {
public:
```

`WriterRes (const char* fileName, unsigned int margins = 5);`

Constructor. Opens the writer to write in file `fileName`. `margins` is the number of white spaces that will be used as a margin inside a table's cell when printing a table.

`WriterRes (std::ostream* stream, unsigned int margins = 5);`

Same as above, except that the writer is opened to write directly into `stream`.

`void beginTabbedSection()`

Defined in `Writer`.

`void endTabbedSection();`

Defined in `Writer`.

`void addTab();`

Defined in `Writer`.

`void removeTab();`

Defined in `Writer`.

`void clearTab();`

Defined in `Writer`.

`void newLine();`

Defined in `Writer`.

`void newParagraph();`

Defined in `Writer`.

`void writeMathString (const std::string);`

Defined in `Writer`.

`void writeStandOutMathString (const std::string);`

Defined in `Writer`.

```
   void writeTable (Table & data, const std::string pos);
```

Defined in `Writer`.

```
private:

   unsigned int m_margins;
```

The number of white space used as a margin in a table's cell when printing.

```
   std::string m_prefix;
```

Used to remember the state of the tabs and white spaces needed to align correctly a line, for instance when in a tabbed section.

```
   bool m_inTabbed;
```

Set to `true` if the writer is in a tabbed section, otherwise it is `false`.

```
};

}
```

Formatter

This class is an interface that must implemented to format values in a `TableColumn` which composes a `Table`.

```
#include <string>


namespace LatMRG {

class Formatter {
public:

   virtual ~Formatter()
```

Destructor. Does nothing for now.

```
   virtual std::string format (void *value) = 0;
```

Method that must implemented to format `value` into a string.

```
};

}
```

FormatterImpl

This class is a template that implements the interface `Formatter`.

```
#include "Formatter.h"
#include <string>
#include <sstream>


namespace LatMRG {

template <typename T>
class FormatterImpl: public Formatter {
public:

   std::string format (void* value);
```

Formats `value`, which is supposed to be of type `T`, into a string.

```
};

}
```

DoubleFormatter

This class is an implementation of the `Formatter` interface which formats a `double` into a string.

```
#include "Formatter.h"
#include <string>


namespace LatMRG {

class DoubleFormatter : public Formatter {
public:

   DoubleFormatter (int precision);
```

Constructor. This object is initialized to format `double`'s with `precision` decimals.

```
   std::string format (void* value);
```

Formats the double `value` into a string. The argument `value` is assumed to be a pointer to a `double`.

```
private:

   int m_precision;
```

Precision used by the formatter.

```
};

}
```

TableColumn

This abstract class is the representation of a table column for class `Table`. A column is made of a string header and a variable number of values. Values in one column must be of the same type. A `Formatter` must be provided to format the values in the column into a string.

```
#include "Formatter.h"
#include <string>


namespace LatMRG {

class TableColumn {
public:
```

`TableColumn (Formatter *f, std::string h)`

Constructor. `f` is used to transform the column's values into `string`'s. `h` is the header of this column.

`virtual ~TableColumn()`

Destructor.

`virtual std::string getFormattedValue (int pos) = 0;`

Returns the formatted value of cell `pos` using the formatter passed to the constructor. If `pos` is negative, the column's header is returned.

`virtual void* getValue (unsigned int pos) = 0;`

Returns the void pointer of the value of cell at position `pos`.

`virtual void setValue (void* value, unsigned int pos) = 0;`

Sets the cell at position `pos` to `value`.

`virtual void insertValue (void* value, unsigned int pos) = 0;`

Inserts `value` at position `pos` in the column.

`virtual void addValue (void* value) = 0;`

Appends `value` at the end of the column.

`virtual void removeValue (unsigned int pos) = 0;`

Removes the cell at position `pos`.

`virtual int size() = 0;`

Returns the effective size of the column.

`virtual std::string getHeader() const`

Returns this column's header.

`virtual void setHeader (std::string s)`

Sets this column's header to `s`.

```
protected:
```

```
   std::string m_header;
```

This column's header.

```
   Formatter* m_format;
```

The formatter used to format the values of this column.

```
};
```

```
}
```

TableColumnImpl

This class implements the `TableColumn` interface. A `TableColumnImpl` can hold any type of data, given that the `Formatter` passed to its constructor correctly formats the same type of data.

```
#include "TableColumn.h"
#include "Formatter.h"

#include <stdexcept>
#include <string>
#include <vector>


namespace LatMRG {

template <typename T>
class TableColumnImpl : public TableColumn {
public:
```

`TableColumnImpl (Formatter *format, std::string header);`

Constructor. `format` must be a formatter for type `T` and `header` is the name of the column that will be printed.

`TableColumnImpl (Formatter *format, std::string header, unsigned int size);`

Same as above, except that the vector of data will be initialized at size `size`.

`~TableColumnImpl()`

Destructor.

`std::string getFormattedValue (int pos);`

Defined in `TableColumn`.

`void * getValue (unsigned int pos);`

Defined in `TableColumn`.

`void setValue (void *value, unsigned int pos);`

Defined in `TableColumn`.

`void insertValue (void *value, unsigned int pos);`

Defined in `TableColumn`.

`void addValue (void *value);`

Defined in `TableColumn`.

`void removeValue (unsigned int pos);`

Defined in `TableColumn`.

`int size();`

Defined in `TableColumn`.

```
protected:
```

```
   typedef std::vector<T> vect;
```

Type definition for the vector of values in the column.

```
   vect m_data;
```

Internal vector which contains the values of the column.

```
};
}
```

Table

This class implements a table of values. It is made of an arbitrary number of `TableColumn`'s, each one containing one type of data. The table can be printed in several formats using a class derived from `Writer`.

```
#include "TableColumn.h"
#include <vector>


namespace LatMRG {

class Table {
public:
```

`Table();`

Constructor.

`Table (int size);`

Same as above, except that enough memory will be reserved for `size TableColumn`'s.

`virtual ~Table();`

Destructor.

`void add (TableColumn * column);`

Appends a `TableColumn` at the end of the table.

`void insert (TableColumn * column, unsigned int pos);`

Inserts a `TableColumn` in the table at position `pos`.

`void remove (unsigned int pos);`

Removes the column in the table at position `pos`.

`void sort (unsigned int pos);`

NOT YET IMPLEMENTED.

`int getHeight() const;`

Returns the height of the highest column in the table.

`int size() const`

Return the numbers of columns in the table.

`TableColumn * operator[] (int pos);`

Returns the column at position `pos`.

```
protected:
```

`std::vector<TableColumn *> m_columns;`

The vector containing the columns for the table.

```
   std::vector<TableColumn *>::iterator m_iter;
```

The iterator for the column's vector.

```
};

}
```

References

[1] L. Afflerbach and H. Grothe. Calculation of Minkowski-reduced lattice bases. *Computing*, 35:269–276, 1985.

[2] L. Afflerbach and H. Grothe. The lattice structure of pseudo-random vectors generated by matrix generators. *Journal of Computational and Applied Mathematics*, 23:127–131, 1988.

[3] J. H. Conway and N. J. A. Sloane. *Sphere Packings, Lattices and Groups.* Grundlehren der Mathematischen Wissenschaften 290. Springer-Verlag, New York, 1988.

[4] J. H. Conway and N. J. A. Sloane. *Sphere Packings, Lattices and Groups.* Grundlehren der Mathematischen Wissenschaften 290. Springer-Verlag, New York, 3rd edition, 1999.

[5] R. Couture and P. L'Ecuyer. Linear recurrences with carry as random number generators. In *Proceedings of the 1995 Winter Simulation Conference*, pages 263–267, 1995.

[6] R. Couture and P. L'Ecuyer. Computation of a shortest vector and Minkowski-reduced bases in a lattice. In preparation, 1996.

[7] R. Couture and P. L'Ecuyer. Orbits and lattices for linear random number generators with composite moduli. *Mathematics of Computation*, 65(213):189–201, 1996.

[8] R. Couture and P. L'Ecuyer. Distribution properties of multiply-with-carry random number generators. *Mathematics of Computation*, 66(218):591–607, 1997.

[9] U. Dieter. How to calculate shortest vectors in a lattice. *Mathematics of Computation*, 29(131):827–833, 1975.

[10] U. Fincke and M. Pohst. Improved methods for calculating vectors of short length in a lattice, including a complexity analysis. *Mathematics of Computation*, 44:463–471, 1985.

[11] G. S. Fishman. Multiplicative congruential random number generators with modulus 2^{β}: An exhaustive analysis for $\beta = 32$ and a partial analysis for $\beta = 48$. *Mathematics of Computation*, 54(189):331–344, Jan 1990.

[12] G. S. Fishman. *Monte Carlo: Concepts, Algorithms, and Applications.* Springer Series in Operations Research. Springer-Verlag, New York, NY, 1996.

[13] G. S. Fishman and L. S. Moore III. An exhaustive analysis of multiplicative congruential random number generators with modulus $2^{31} - 1$. *SIAM Journal on Scientific and Statistical Computing*, 7(1):24–45, 1986.

[14] Free Software Foundation, Inc. *GMP: The GNU Multiple Precision Arithmetic Library*, 2006. Available at `http://gmplib.org/`.

[15] H. Grothe. *Matrixgeneratoren zur Erzeugung Gleichverteilter Pseudozufallsvektoren.* Dissertation (thesis), Tech. Hochschule Darmstadt, Germany, 1988.

[16] P. M. Gruber and C. G. Lekkerkerker. *Geometry of Numbers.* North-Holland, Amsterdam, 1987.

[17] F. J. Hickernell. Lattice rules: How well do they measure up? In P. Hellekalek and G. Larcher, editors, *Random and Quasi-Random Point Sets*, volume 138 of *Lecture Notes in Statistics*, pages 109–166. Springer-Verlag, New York, 1998.

[18] F. J. Hickernell. What affects the accuracy of quasi-Monte Carlo quadrature? In H. Niederreiter and J. Spanier, editors, *Monte Carlo and Quasi-Monte Carlo Methods 1998*, pages 16–55, Berlin, 2000. Springer-Verlag.

[19] F. J. Hickernell, H. S. Hong, P. L'Ecuyer, and C. Lemieux. Extensible lattice sequences for quasi-Monte Carlo quadrature. *SIAM Journal on Scientific Computing*, 22(3):1117–1138, 2001.

[20] D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms.* Addison-Wesley, Reading, MA, second edition, 1981.

[21] D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms.* Addison-Wesley, Reading, MA, third edition, 1998.

[22] D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms.* Addison-Wesley, Reading, MA, third edition, 1998.

[23] C. Koç. Recurring-with-carry sequences. *Journal of Applied Probability*, 32:966–971, 1995.

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

[25] P. L'Ecuyer. Uniform random number generation. *Annals of Operations Research*, 53:77–120, 1994.

[26] P. L'Ecuyer. Combined multiple recursive random number generators. *Operations Research*, 44(5):816–822, 1996.

[27] P. L'Ecuyer. Random number generation. In Jerry Banks, editor, *Handbook of Simulation*, pages 93–137. Wiley, 1998. chapter 4.

[28] P. L'Ecuyer. Good parameters and implementations for combined multiple recursive random number generators. *Operations Research*, 47(1):159–164, 1999.

[29] P. L'Ecuyer. Tables of linear congruential generators of different sizes and good lattice structure. *Mathematics of Computation*, 68(225):249–260, 1999.

[30] P. L'Ecuyer and R. Couture. An implementation of the lattice and spectral tests for multiple recursive linear random number generators. *INFORMS Journal on Computing*, 9(2):206–217, 1997.

[31] P. L'Ecuyer and R. Couture. An implementation of the lattice and spectral tests for multiple recursive linear random number generators. *INFORMS Journal on Computing*, 9(2):206–217, 1997.

[32] P. L'Ecuyer and S. Tezuka. Structural properties for two classes of combined random number generators. *Mathematics of Computation*, 57(196):735–746, 1991.

[33] G. Marsaglia. Random numbers fall mainly in the planes. *Proceedings of the National Academy of Sciences of the United States of America*, 60:25–28, 1968.

[34] G. Marsaglia. Yet another rng. Posted to the electronic billboard `sci.stat.math`, August 1, 1994.

[35] H. Niederreiter. A pseudorandom vector generator based on finite field arithmetic. *Mathematica Japonica*, 31:759–774, 1986.

[36] H. Niederreiter. The multiple-recursive matrix method for pseudorandom number generation. *Finite Fields and their Applications*, 1:3–30, 1995.

[37] H. Niederreiter. Pseudorandom vector generation by the multiple-recursive matrix method. *Mathematics of Computation*, 64(209):279–294, 1995.

[38] Michael Scott. *MIRACL—A Multiprecision Integer and Rational Arithmetic C/C++ Library.* Shamus Software Ltd, Dublin, Ireland, 2003. Available at `http://www.shamus.ie/`.

[39] V. Shoup. *NTL: A Library for doing Number Theory.* Courant Institute, New York University, New York, NY, 2005. Available at `http://shoup.net/ntl/`.

[40] I. H. Sloan and S. Joe. *Lattice Methods for Multiple Integration.* Clarendon Press, Oxford, 1994.