Version: November 2000

LatMRG

User's Guide

A Modula-2 software for the theoretical analysis of linear congruential and multiple recursive random number generators

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 Modula-2. 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 searchs 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; lattice structure; spectral test; maximal period

Contents

1 Background and overview

`LatMRG` is a software system implemented as a library of modules written in the Modula-2 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. 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 [29] 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 modules import facilities from the lower-level ones.

The high-level modules (c) are programs in executable form which read their data in files or work interactively with the user. 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 modules (b) provide data types and procedures 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 modules (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 modules (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 modules 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

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

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 [3]. 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 $L_t^* = \{\mathbf{h} \in \mathbb{R}^t : \mathbf{h} \cdot \mathbf{v} \in \mathbb{Z} \text{ for all } \mathbf{v} \in L_t\}$. 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 is 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 (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 $\Lambda = \{\mathbf{v} = \lambda_1\mathbf{v}_1 + \cdots + \lambda_t\mathbf{v}_t : 0 \le \lambda_i \le 1 \text{ for } 1 \le i \le t\}$, 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

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

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 [14, 23, 24, 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 (2),

$$P(z) = \left(z^k - \sum_{i=1}^{k} a_i z^{k-i}\right) \bmod m, \tag{3}$$

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, 23].

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

$$y_n = (b_1 x_n + \cdots + b_k x_{n+k-1}) \bmod m, \tag{4}$$

$$u_n = y_n/m. \tag{5}$$

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

$$\begin{pmatrix} y_n \\ y_{n+1} \\ \vdots \\ y_{n+t-1} \end{pmatrix} = \begin{pmatrix} \mathbf{b}' \\ \mathbf{b}'\mathbf{A} \\ \vdots \\ \mathbf{b}'\mathbf{A}^{t-1} \end{pmatrix} \begin{pmatrix} x_n \\ x_{n+1} \\ \vdots \\ x_{n+k-1} \end{pmatrix} \mod m \stackrel{\text{def}}{=} \mathbf{B}_t \begin{pmatrix} x_n \\ x_{n+1} \\ \vdots \\ x_{n+k-1} \end{pmatrix} \mod m, \tag{6}$$

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 (2) and (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.,

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

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 (2). The last $t - k$ components are thus linear combinations modulo 1, with integer coefficients, of the first k components.

For $1 \leq i \leq k$, 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 $i \leq k$, and $v_{i,j} = (a_1 v_{i,j-1} + \cdots + a_k v_{i,j-k}) \bmod 1$ for $j > k$. For $k+1 \leq i \leq t$, 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, $L_t = \Psi_t + \mathbb{Z}^t = \{\mathbf{v} = \tilde{\mathbf{v}} + \mathbf{z} : \tilde{\mathbf{v}} \in \Psi_t$ and $\mathbf{z} \in \mathbb{Z}^t\}$. The vectors $\mathbf{w}_i = (w_{i,1}, \ldots, w_{i,t})$, $1 \leq i \leq t$, where

$$w_{i,j} = \begin{cases} m & \text{for } j = i \leq k; \\ 0 & \text{for } j \neq i \leq k; \\ v_{j,i} & \text{for } i > k \geq j; \\ 1 & \text{for } j = i > k; \\ 0 & \text{for } k < j \neq i > k, \end{cases}$$

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 (4) and (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 $\mathbf{V}_t'$. The module `MRG` does that. For LCGs in matrix form, bases for L_t and its dual can be constructed as explained in [14, 29].

If one adds a constant b on the right-hand-side of (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 zero vector. Otherwise, the set of t-dimensional vectors produced over any given (sub)cycle (plus the zero 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

$$\psi_t(I) = \{(u_{i_1}, \ldots, u_{i_t}) \mid (x_0, \ldots, x_{n+k-1}) \in \mathbb{Z}_m^k\} \tag{7}$$

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 module `MRG`, 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 [29]. 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, 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.

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

Minkowski proved that there exists lattices with density satisfying $\delta_t \geq \zeta(t)/(2^{t-1}V_t)$ where $\zeta(t) = \sum_{k=1}^{\infty}$ is the Riemann zeta function and $V_t = \pi^{n/2}/(n/2)!$ 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 [3]. This upper bound can be written as

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

where $R(t)$ can be found in Table 1.2 of [3] for $t \leq 24$, and can be approximated with $O(1/t)$ error and approximately 4 decimal digits of precision, for $t \geq 25$, by

$$R(t) = \frac{1}{2}n \log_2\left(\frac{n}{4\pi e}\right) + \frac{3}{2}\log_2(n) - \lg(e/\sqrt{\pi}) + \frac{5.25}{n+2.5}. \tag{8}$$

Table 1 in [28] 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 $1 \leq t \leq 48$. 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 $1 \leq p < \infty$ 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:

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

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

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 $t_1 \geq \cdots \geq t_d \geq d$, consider the worst-case figure of merit

$$M_{t_1,\ldots,t_d} = \min\left[\min_{k+1\leq t\leq t_1} \ell_t/\ell_t^*(m^k),\ \min_{2\leq s\leq k}\ \min_{I\in S(s,t_s)} \ell_I/m,\ \min_{k+1\leq s\leq d}\ \min_{I\in S(s,t_s)} \ell_I/\ell_s^*(m^k)\right], \tag{9}$$

where $S(s,t_s) = \{I = \{i_1,\ldots,i_s\} : 1 = i_1 < \cdots < i_s \leq t_s\}$. This figure of merit makes sure that the lattice is good in projections over t successive dimensions for all $t \leq t_1$, and over non-successive dimensions that are not too far apart. Note that when $s \leq k$, 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 (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, 27, 28]. 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 (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.

1.5.1 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, 14] 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 (9), we define

$$Q_{t_1,\ldots,t_d} = \min \left[\min_{k+1 \le t \le t_1} q_t,\ \min_{2 \le s \le d} \min_{I \in S(s,t_s)} q_t(I) \right]. \tag{10}$$

1.5.2 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., [38]). In the case where $\Psi_t = L_t \cap [0,1)^t$ where L_t is a lattice with dual L_t^*, one has

$$\mathcal{P}_\alpha(\Psi_t) = \sum_{\mathbf{0} \ne \mathbf{w} \in L_t^*} \|\mathbf{w}\|_\pi^{-\alpha}, \tag{11}$$

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

$$\mathcal{P}_{2\alpha}(\Psi_t) = -1 + \frac{1}{n} \sum_{\mathbf{u} \in \Psi_t} \prod_{j=1}^t \left[1 - \frac{(-4\pi^2)^\alpha}{(2\alpha)!} B_{2\alpha}(u_j)\right] \tag{12}$$

where the B_α are the Bernoulli polynomials:

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

and the other polynomials can be found via the identity

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

Hickernell [16] introduced generalizations of $\mathcal{P}_\alpha$, incorporating weights and replacing the simple sum in (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

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

when α is an integer. The identity (12) or (1.5.2) 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 [17] 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 is has been proved (e.g., [38], 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

$$\mathcal{P}_\alpha(\Psi_t) \leq \frac{[(e/t)(2\ln m + t)]^{\alpha t}}{m^\alpha}. \tag{13}$$

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

Alternatively, Hickernell et al. [18], section 4.1, suggest using the figure of merit g_t, where

$$\begin{aligned} g_t^2 &= \frac{n^2}{(3/2)^t - 1} \left(\frac{t-1}{t-1+\log n}\right)^{t-1} \mathcal{P}_2(\Psi_t) \\ &= \frac{n}{(3/2)^t - 1} \left(\frac{t-1}{t-1+\log n}\right)^{t-1} \left[-n + \sum_{\mathbf{u} \in \Psi_t} \prod_{j=1}^t (1 + 3B_{2\alpha}(u_j))\right] \end{aligned} \tag{14}$$

is a normalized version of the inverse of (1.5.2) 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

$$g_t^2 = \gamma_t^{\mathrm{P}}(n) \left[-n + \sum_{\mathbf{u} \in \Psi_t} \prod_{j=1}^t (1 + 3B_{2\alpha}(u_j))\right] \tag{15}$$

where

$$\gamma_t^{\mathrm{P}}(n) = \frac{n}{(3/2)^t - 1}\left(\frac{t-1}{t-1+\log n}\right)^{t-1} \tag{16}$$

is a constant that depends on t and n. As a figure of merit based on $\mathcal{P}_2$, similar to (9), we define

$$G_{t_1,\ldots,t_d} = \min\left[\min_{k+1\le t\le t_1} 1/g_t(\Psi_t),\ \min_{k+1\le s\le d}\ \min_{I\in S(s,t_s)} 1/g_t(\Psi_t(I))\right]. \tag{17}$$

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

$$\mathbf{x}_n = (A_1\mathbf{x}_{n-1} + \cdots + A_k\mathbf{x}_{n-k}) \bmod m \tag{18}$$

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 (18) 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:

$$u_{nw+i} = x_{n,i}/m \qquad \text{for } 0 \le i < w \text{ and } n \ge 0, \tag{19}$$

$$u_n = \frac{1}{m}\left(\sum_{i=1}^{w} b_i x_{n,i} \bmod m\right) = \sum_{i=1}^{w} b_i x_{n,i}/m \bmod 1 \qquad \text{for } n \ge 0, \tag{20}$$

$$u_n = \sum_{i=1}^{w} x_{n,i} m^{-i} \qquad \text{for } n \ge 0, \tag{21}$$

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.)

1.7 Multiply-with-carry

A *Multiply-with-Carry* (MWC) generator [5, 8, 22, 34] is based on the recurrence

$$x_n = (a_1 x_{n-1} + \cdots + a_k x_{n-k} + c_{n-1}) \bmod b, \tag{22}$$

$$c_n = (a_1 x_{n-1} + \cdots + a_k x_{n-k} + c_{n-1}) \operatorname{div} b, \tag{23}$$

$$u_n = x_n/b.$$

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:

$$z_n = a z_{n-1} \bmod m; \qquad w_n = z_n/m. \tag{24}$$

In other words, if

$$w_n = \sum_{i=1}^{\infty} x_{n+i-1} b^{-i} \tag{25}$$

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 (24). This is what the LatMRG package does.

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

$$\ell_t = (1 + a_1^2 + \cdots + a_k^2)^{1/2}. \tag{26}$$

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 (24) 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 component MRGs with relatively prime moduli. 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 `MRG` and in `seekl`).

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, 25]. They can thus be analyzed with the present software.

1.9 Computing a shortest nonzero vector or a reduced basis

The module `REDBAS` 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 module also computes a MRLB 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. The module `REDBAS2` perform basis reductions by taking into account all sources of numerical error during the computations and computing error bounds, thereby yielding "guaranteed error-free" results. Of course, this implies a lot of overhead, which means that the procedures of `REDBAS2` are much slower than those of `REDBAS`. Another approach is used in module `REDBAS3` bypassing all floating point calculations. Again this is slower than `REDBAS`.

1.10 Representation of large numbers

`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 package SENTIERS [30], also written in Modula-2. Of course, these operations 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 modules have been implemented in two versions.

When building a basis or checking maximal period conditions, the modulus and multipliers can be represented either as `LONGINT` (32-bit integers) or `SuperInteger` (arbitrary large integers, from SENTIERS). The modules `MULTLI` and `MULTSI` support these two representations, and provide basic facilities to the other modules for manipulating the multipliers. These are two versions of the generic module `MULT`.

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 `LONGREAL` (64-bit floating-point numbers) or `SuperInteger`. The modules `BASISLR` and `BASISSI` implement basic facilities for these two cases. They are two versions of the generic module `BASIS`.

In summary, `MULT` is for working on the construction of a lattice basis and its dual from the specifications of the generator, lacunary indices, etc., whereas `BASIS` is for working on the basis (reduction, etc.) after it has been constructed.

For the computations that are performed in floating-point arithmetic, one can compute error bounds on everything (using `REDBAS2`), or no error bounds (using `REDBAS`). When performing a search for good generators, for instance, one can first perform all the "screening" computations (involving many generators) without computing the error bounds, and then recompute (verify) with the error bounds only for the retained generator(s).

1.11 Current implementations

The software is currently working under the XDS Modula-2 environment, on SUN computers under Solaris and on PC-type computers under Linux. It can be ported to other machines for which a Modula-2 compiler is available. Its implementation is built upon two other Modula-2 libraries called `mylib` [31] and SENTIERS [30], which must also be ported for `LatMRG` to work on other platforms.

2 Using the programs in executable form

At the high-level end, `LatMRG` provides programs in executable form. These programs read their data in files or work interactively with the user. 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 section.

The program `maxper` checks whether a given generator has maximal period. The program `findmk` can find values of m and k such that $r = (m^k - 1)/(m - 1)$ is prime. The programs `latl` and `lats` perform standard lattice and/or spectral tests for that generator. The only difference between the latter two programs is that in `latl`, the multipliers and basis components are implemented in `LONGINT` and `LONGREAL`, respectively, while in `lats`, they are implemented as `SuperInteger`'s. More comprehensive (and larger) programs perform computer searchs to seek the "best" generators of a given type, according to maximal period and lattice structure criteria. These programs also come in two versions. In `seekl`, the multipliers and basis components are implemented in `LONGINT` and `LONGREAL`, respectively, while in `seeks`, both are implemented as `SuperInteger`'s.

The program `menumrg` is an interactive program which allows the user to build a lattice basis for a given generator, and then manipulate, compare, or reduce lattice bases.

We now give a few concrete examples of data files and results. The timings given are from runs on a SUN SparcStation 20, under SunOS 5.4, using version 4.5.1 of the MCS Modula-2 compiler [19]. These results were obtained with the 'February 1996' version of `LatMRG`.

```
FALSE                          ReadGenFile
1                              J  (Nb. of components)
MRG                            Gener. type
2147483647  1 0                m  (modulus)
1                              k  (order)
742938285                      a  (multiplier)
2  30                          MinDim   MaxDim
BeyerSpectral  Hermite         LatticeInfo
Full                           LatticeType
1  1                           LaGroupSizes   Spacings
NoVerify                       VerifyBB
1000000                        MaxNodesBB
TEX                            ResultForm
```

Figure 1: Example of a data file for the program `latl`, in file `fish31.dat`.

Table 1: Results of latl in file fish31.tex.

t	d_t	q_t	S_t	cumulative cpu (sec)
2	2.31556E-5	0.84858	0.86725	0.02
3	8.02308E-4	0.90348	0.86068	0.03
4	4.52795E-3	0.88522	0.86270	0.04
5	0.01328	0.79661	0.83195	0.05
6	0.02586	0.85416	0.83415	0.06
7	0.05530	0.59317	0.62392	0.07
8	0.06820	0.64472	0.70666	0.09
9	0.10600	0.67859		0.10
10	0.10847	0.73126		0.13
11	0.16903	0.63121		0.15
12	0.24254	0.58925		0.19
13	0.24254	0.58080		0.22
14	0.24254	0.63681		0.26
15	0.24254	0.67276		0.31
16	0.24254	0.76358		0.39
17	0.24254	0.78289		0.48
18	0.25000	0.74582		0.64
19	0.26726	0.75484		1.14
20	0.26726	0.81215		1.71
21	0.26726	0.78067		3.15
22	0.28868	0.88552		5.77
23	0.28868	0.86355		8.37
24	0.30151	0.91232		13.46
25	0.30151	0.86634		26.85
26	0.30151	0.84667		48.06
27	0.30151	0.87238		80.57
28	0.30151	0.88873		126.43
29	0.31623	0.88771		200.71
30	0.31623	0.88201		307.96

2.1 An example with the program latl

Figure 1 gives an example data file for the program latl. The corresponding results appear in Table 1. To call the program and produce these results, type "latl fish31", assuming that the data are in file "fish31.dat". The results will be in file "fish31.tex", which produces Table 1 after going through LaTeX. If TEX was replaced by Terminal in the data file, the results would rather be displayed on the screen. See the description of the program latl 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 second column contains comments describing the meaning of these values. In that example, the Beyer and spectral tests are 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 Beyer quotients q_t and distances d_t between hyperplanes in dimensions 2 to 30 was approximately 308 seconds, i.e., approximately

5 minutes. Computing the Beyer quotients for this example turns out to be much more expensive than computing only the distances between the hyperplanes. For example, to compute only the d_t's (and not the q_t's) in dimensions 2 to 30, it takes less than 3 seconds on the same machine! We also computed d_t and q_t in dimensions 2 to 30 with formal verification using error bounds for all floating point numbers (see the module `REDBAS2`). This took approximately 15 minutes and the results were the same (except for the CPU times) as in Table 1. These times are representative of what happens in general for examples of that size. The program `lats` can also be applied to the same data file and the results will be the same, except for the cpu times. It takes more time to run, but it can accept much larger moduli and multipliers.

```
FALSE                          Read generators from file
1                              J
 MRG                            Gener. type
 32749  1 0                     Modulo m
 2                              Order k
 TRUE                           Maximal period
 FALSE                          Implem. condition
 Decomp                         Factors of m-1
 DecompWrite seek15r.fac        Factors of r
 1                              b1
 180                            c1
 -180                           b2
 -1                             c2
 Exhaust                        Search method: exhaustive search
1 3 8                          NbCat  Dim(0)  Dim(1)
0.0  1.0                       Min and max merit values
2                              Nb of retained generators
Spectral  Hermite              Merit criterion
Spectral                       Information to print
Full                           Lattice type (analyzed)
1  1                           Lacunary ind. group sizes,  spacing
NoVerify                       Verify basis reduction with error bounds.
1000000                        Max nb of nodes in each BB
0.1                            Time limit: 0.1 hour
12345 98765                    S1 S2 : seeds for random number generator
RES                            Results File
```

Figure 2: Example of a data file for the program `seekl`, in file `seek15.dat`.

2.2 Examples with the programs `seekl` and `seeks`

Figure 2 gives an example of a data file for `seekl`. 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 "`f2seek1.fac`", while those of $m - 1$ will not be kept. The results of that program appear in Figure 3 (this is an actual printout, for illustration).

Figure 4 gives another data file example, this time for `seeks`. 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_8. 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 $(4 \times 1 \times 1 \times 1 \times 4)$, that is, a total of 16000 generators (4000 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 5. The program reached the time-limit of 3 hours before examining all the generators it was asked for. It then stopped and reported the best generator it had found so far.

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
 ----
   Modulus  m = 32749
   Order    k = 2
   n_j       = 1
   rho_j = (m_j)^k_j-1 = 1072497000
   Factors of m-1   :  Decomp
   Factors of r     :  DecompWrite seek15r.fac
   Bounds : a1 from : 1
                 to : 180
            a2 from : -180
                 to : -1
   Search method                        : EXHAUST
   Implem. cond.  a_i (m mod a_i) < m : NO
   Maximum period required              : YES

   Merit criterion                      : M_8
   Seeds for RNG                        : 12345,  98765
   Verify Branch-and-bound              : NoVerify
   Maximum nodes in branch-and-bound  : 1000000
   Lattice Type                         : Full

 RESULTS
 -------
   Values of a2 tried                     :        180
   Values of a2 primitive element         :         66
   Polynomials with a2 primitive element :      11880
   Primitive polynomials                  :       4638
   Nb. of polynomials to examine          :      32400
   Nb. Generators conserved               :          2
   Total CPU time (after setup)           : 0:00:13.06
 ------------------------------------------------------
1.
   a_1 =              180
   a_2 =             -176

    t         d_t             q_t           S_t
 ------------------------------------------------------
    3      3.97223E-3                     0.21911
    4      7.13449E-3                     0.65130
    5         0.03511                     0.36151
    6         0.03558                     0.68076
    7         0.09285                     0.41035
    8         0.09285                     0.56613

   Merit =    0.21911  = S_3

2.
   a_1 =              180
   a_2 =             -175

    t         d_t            q_t            S_t
 ------------------------------------------------------
    3      3.98327E-3                     0.21850
    4      7.39990E-3                     0.62794
    5         0.02824                     0.44953
    6         0.03716                     0.65170
    7         0.05717                     0.66649
    8         0.07715                     0.68130

   Merit =    0.21850  = S_3
```

Figure 3: Results of program `seekl` in file `seek15.res`.

```
FALSE                          ReadGenFromFile
1                              J
 MRG                            TypeGen
 2  63  -711                   m
 5                              k
 TRUE                           PerMax
 TRUE                           ImplemCond
 Read  seek63.fac               Factors of m-1
 Prime                          Factors of r
 1                              b_1
 2  63  -712                   c_1
 0                              b_2
 0                              c_2
 0                              b_3
 0                              c_3
 0                              b_4
 0                              c_4
 -2  63  -712                   b_5
 -1                             c_5
 Random 1000 4 4               Search: random, 1000 regions, H=4, Hk=4
1 6 12                         C  Dim(0)  Dim(1)
0.0  1.0                       Minimal and maximal merit values
1                              Nb of retained generators
Spectral       Hermite         Criterion
BeyerSpectral  Hermite         Information to print
Full                           Lattice type (analyzed)
1  1                           Lacunary ind. group sizes,  spacing
NoVerify                       Verify BB with error bounds
1000000                        Max nb nodes in each BB
3.0                            cpu time-limit (hours)
12345 98765                    S1 S2 : seeds for random number generator
RES                            Results File
```

Figure 4: Example data file for program seeks, in file seek63.dat.

```
 SEARCH FOR GOOD MRGs OF ORDER 5
 ---------------------------------------

 :

 RESULTS
 -------
 ****  Program aborted: reached time limit.  ****
     There will be no verification of results.
          time limit :       3.00 hour(s)
   Values of a5 tried                     :        277
   Values of a5 primitive element         :        134
   Polynomials with a5 primitive element  :        536
   Primitive polynomials                  :        103
   Nb. Generators conserved               :          1

   Total cpu time (after setup)           : 3:00:01.76
 -------------------------------------------------------

1.
   a1 = 53663276718
   a2 = 0
   a3 = 0
   a4 = 0
   a5 = -5144244197

    t        d_t          q_t            S_t
 -------------------------------------------------------
    6     1.855E-11                  6.5589E-6
    7     1.855E-11                  1.1384E-3
    8     1.855E-11                    0.05347
    9     2.991E-11
   10     3.360E-10
   11     2.6017E-9
   12     1.2892E-8

   Merit = 6.55894E-6  = S_6
```

Figure 5: Results of program `seeks` in file `seek63.res`.

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 Modula-2 compiler and some knowledge about the Modula-2 language are required. Instead of providing the parameters according to the format of Appendix A, one sets those parameters (or data) by setting the appropriate variables or calling the appropriate procedures from the intermediate or low-level modules. In this section, we introduce briefly the intermediate and low-level modules, which are described in Appendix B and C, 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 for more details on this.

3.1 Using the modules 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 Appendix C.

The intermediate-level module MRG constructs lattice bases for different classes of generators and perform tests on such generators. The modules LATBASIS, REDBAS, REDBAS2, and REDBAS3 offer tools for manipulating lattice bases, finding the shortest vector in a lattice, and for reducing a basis in the sense of Minkowski. Those modules are described in Appendix B. The programs of Appendix A use those intermediate and lower-level modules in their implementation, and so, provide examples of how to use them.

3.2 Lower-level modules and Changing the representation

As discussed in Section 1.6, the multiplier's components can be implemented in the LONGINT or SuperInteger representation, while the basis components can be in the LONGREAL or SuperInteger representation. To select the appropriate representation (in modules MULT and BASIS, respectively), one should use the command select (see Appendix C), set the environment variable LILR, and recompile all the modules.

Generally speaking, the proper choice of representation depends on the size of the modulus m. For example, if the modulus is less than 2^{31}, one can type select LI LR (although it could sometimes happen that vector components get larger than the modulus). For large moduli, type select SI SI. See the description of the command select for more details on its use. After changing the representation, one must recompile the modules. Calling one of the command lilr, silr, sisi in a shell will set the environment variable LILR corresponding to the particular case of select. (See also the file Makedoc.tex.) This is necessary in order to link with the appropriate libraries.

3.3 Modifying the package, recompiling, and relinking

After calling select, or after modifying the code of a module, one must recompile and relink.

APPENDIX A

Programs in Executable Form

maxper

This program verifies whether a given generator has maximal period or not. Integers are represented using the `SuperInteger` type. The modulus m must be prime. 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 or provided by the user in a file. We warn that factoring r can take a *huge* amount of time.

The data for `maxper` must be placed in a file with extension "`.dat`", according to the format displayed in Figure 6. The data fields have the same meaning as for `latl`. To run the program, type "`maxper` ⟨*file*⟩", where ⟨*file*⟩ is the name of the data file, without extension. Currently, only `LCG` and `MRG` are allowed for *TypeGen*.

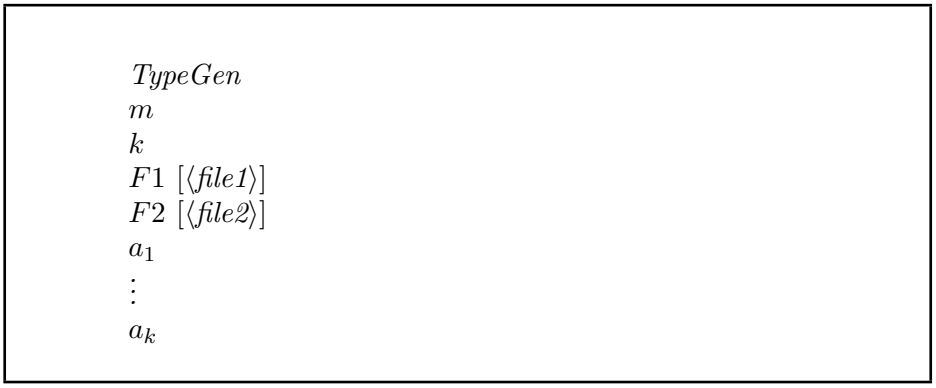

Figure 6: Data file format for `maxper`.

findmk

This program is an interface to the module `PRIM`. It searches and prints the list of all prime integers m such that $2^e + c_1 \leq m \leq 2^e + c_2$ and $r = (m^k - 1)/(m - 1)$ is prime. We may also request that $(m - 1)/2$ be also prime, by setting the boolean variable *Safe* to `TRUE`.

The program reads the data and calls the procedure `FindPrimesmr` in the module `PRIM`, with the corresponding parameters. The data for `findmk` must be placed in a file with extension "`.dat`", in the format of Figure 7. To run the program, type "`findmk` ⟨*file*⟩", where ⟨*file*⟩ is the name of the data file, without extension. The results will be written in ⟨*file*⟩`.res`.

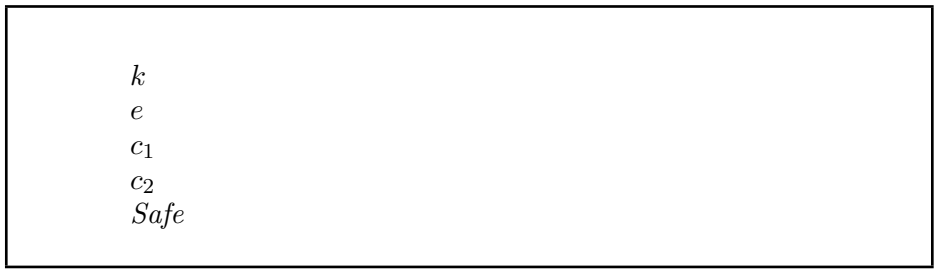

Figure 7: Data file format for `findmk`.

latl

This program applies the spectral and/or lattice test(s) to a given generator. It uses the `LONGINT` representation for the multipliers and modulus (module `MULTLI`) and the `LONGREAL` representation for the bases (module `BASISLR`). The generator analyzed can be a combined generator with J components, and expressed that way in the data file. To run the program, type "`latl` ⟨*file*⟩", where ⟨*file*⟩ is the name of the data file, without extension.

The data must be in a file with extension "`.dat`", according to the format of Figure 8. The data fields have the same meaning as for `seekl`, except that several fields have been removed, a_i replaces b_i and c_i, there is a single category for the dimensions, and the options for *LatticeInfo* and *ResultForm* are different.

$(a_1, \ldots, a_k)$ **:** is the vector of (integer) multipliers. They can be given in format (a) or (b) described in `seekl`.

LatticeInfo ⟨*Norm*⟩ **:** If the value of *LatticeInfo* is `Spectral` or `BeyerSpectral` or `SpectralP`, then the field *Norm* must appear. It indicates which type of normalization is used in the definition of S_t. The allowed values are `(BestLat, Laminated, Rogers, Minkowski)`.

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

`Terminal` indicates that the results will be given 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`".

`TEX` asks the program to produce a file intended for LaTeX, with extension "`.tex`".

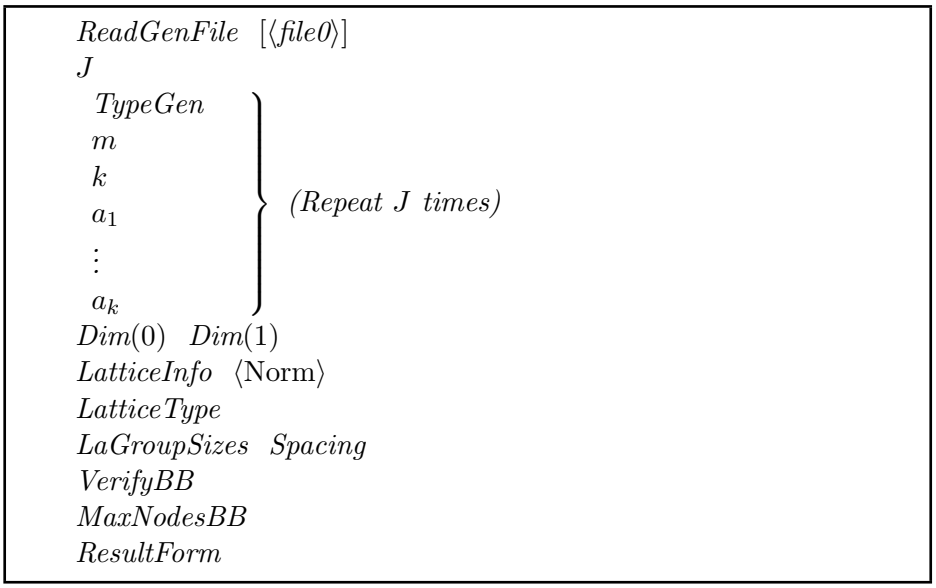

Figure 8: Data file format for latl.

seekl

(To be rewritten....)

This program performs a 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 1.3 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 and 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 1.2) 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). 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 and/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 can be 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.

In the case where the selection criterion is `Spectral` or `SpectralH`, and if $T > 8$, the distances between hyperplanes are computed for dimensions up to T, but the selection of generators is based only on M_8, because the Hermite constants defining d_t^* in that case are known only for $t \leq 8$.

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 data file

The data for `seekl` must be placed in a file with extension "`.dat`", according to the format displayed on Figure 9. 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 "`seekl` ⟨*file*⟩", where ⟨*file*⟩ is the name of the data file, without extension. The results will be in a file with the same name, with extension "`.res`" or "`.gen`" (see *ResultForm*).

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

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

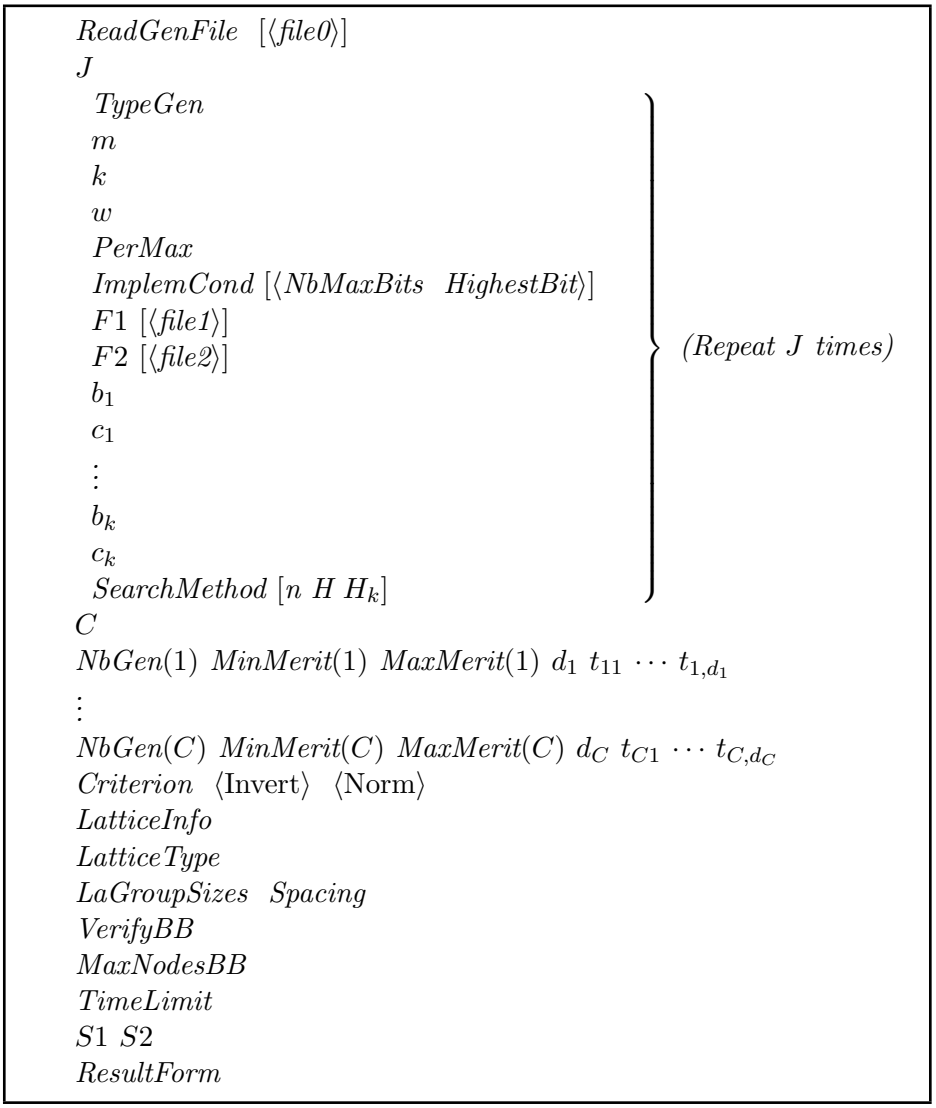

Figure 9: Data file format for `seekl`.

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

For the program `seekl`, all these numbers must fit in a `LONGINT`. For larger numbers, one must use the program `seeks`.

Meaning of the data fields

ReadGenFile **and** ⟨*file0*⟩ **:** `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 looked at are those listed in the file ⟨*file0*⟩`.gen`. This must be a file of type "`.gen`", produced by this program with the `GEN` option for the *ResultForm* 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 in the range [1..`MaxJ`]. When $J > 1$, we look for combined generators.

TypeGen **:** 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, 26]).

`MMRG` means a matrix MRG.

m, k, ⟨w⟩ **:** Modulus, order of the recurrence, and size of the matrix-type coefficients. Must be positive integers, with $k <$ `MMaxDim`. For $m^k >= 2^{31}$, use the program `seeks` instead of `seekl`. The value of w should be specified only for MMRG generators. Otherwise it is assumed to be 1.

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 y z), otherwise *Permax* will be put back to `FALSE`. The software assumes that m is prime, unless $z = 0$ and $y > 1$, 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`.

ImplemCond **and** ⟨*NbPow2 HighestBit*⟩ **:** Can be `NoCond`, `AppFact`, or `SumPow2`. If `NoCond`, then no conditions are imposed on the multipliers a_i. If `AppFact`, then the multipliers must satisfy the "approximate factoring" condition $a_i(m \bmod a_i) < m$ for each i. MRGs are usually easier to implement under this condition [23]. If `SumPow2`, then the positive integers *NbPow2* and *HighestBit* must appear and they indicate that for each i, the multiplier a_i must be the sum of at most *NbPow2* (positive or negative) powers of 2, with the highest power of 2 not exceeding 2^e in absolute value, where $e =$ *HighestBit*. For example, if *SumPow2* = 2 and *HighestBit* = 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.

$F1$ **and** ⟨*file1*⟩ **:** 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). $F1$ indicates how the factors of $m - 1$ are to be found and ⟨*file1*⟩ is a file name. The values allowed for $F1$ are `(Decomp, DecompWrite, Read)`.

`Decomp` means that the program itself will factorize $m - 1$. In this case, the field ⟨*file1*⟩ is not used and can be omitted. To factorize, the program uses the procedure `Factorize` in the module `SUPFACT` of the package SENTIERS [30], with no CPU time limit. It is the responsibility of the user to make sure that the factorization will take a reasonable time.

`DecompWrite` means the same as `Decomp`, except that the program will also write the factors (larger than 1) in the file ⟨*file1*⟩, one factor per line.

`Read` indicates that $m-1$ is already factorized and that the factors will be read from file ⟨*file1*⟩, in the same format. The factors need not be sorted, but must be one per line, and repeated factors must be on successive lines. The factorization must be complete and the program will check if the product of all the factors is really equal to $m-1$.

$F2$ **and** ⟨*file2*⟩ **:** This data line is similar to the previous one, except that it concerns $r = (m^{kw}-1)/(m-1)$ 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 $F2$, so that the set of possible values is `(Decomp, DecompWrite, Read, Prime)`. The module `SUPFACT` in SENTIERS [30] can be used independently to factorize r or to check its primality.

`Prime` indicates that r is prime.

$b = (b_1, \ldots, b_k)$ and $c = (c_1, \ldots, c_k)$: The b_i and c_i are integers such that $-m < b_i \le c_i < m$ for $i = 1, \ldots, k$. They determine the boundary of the (rectangular) area of search. For MMRGs ($w > 1$), each b_i and c_i is replaced by a matrix of size $w \times w$. Each of these matrices is given line by line, w entries on each line of the data file.

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

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

`Random` asks for a random search, performed as described in the first subsection of the description of `seekl`. The integer n gives the number of subregions (clusters) to examine. If $w = 1$, 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 $h_k = \min(H_k, c_k - b_k + 1)$. If $w > 1$, ?????

C**:** The program will retain C lists (or categories) of generators, according to the specifications given below. One must have $1 \le C \le$ `MaxCat`.

$$\begin{array}{l} NbGen(1)\ \ MinMerit(1)\ \ MaxMerit(1)\ d_1\ t_{11}\ \cdots\ t_{1,d_1} \\ \qquad\qquad \vdots \\ NbGen(C)\ \ MinMerit(C)\ \ MaxMerit(C)\ d_C\ t_{C1}\ \cdots\ t_{C,d_C} \end{array}$$

Integers which must satisfy the constraints:

$$\begin{array}{rl} 0 \le NbGen(i) \le \texttt{MaxNbGen}, & \\ 0.0 \le MinMerit(c) \le MaxMerit(c) \le 1.0 & \text{for all } c, \\ MinMerit(c) \le MaxMerit(c-1) & \text{for } c > 1, \\ d_1 \le \cdots \le d_C, & \\ \texttt{BMaxDim} \ge t_{c1} \ge \cdots \ge t_{cd_c} \ge d_c & \text{for all } c, \end{array}$$

where `BMaxDim` is defined in the module `CONFIG`.

There will be $NbGen(c)$ generators in the cth list. The real numbers $MinMerit(c)$ *and* $MaxMerit(c)$ represent the minimal and maximal values of the figure of merit to keep a generator in the cth list. That is, only the generators that satisfy

$$MinMerit(c) \leq \sigma \leq MaxMerit(c) \tag{27}$$

are considered for category c and for the categories $c' > c$. Because of this, $MinMerit(c)$ must be nondecreasing in c, because the lower bound for category c also apply categories $c' > c$. As soon as a generator does not satisfy the constraint (27) in a given dimension for a given category c, it is discarded and no more time is spent to test it for $c' > c$. When looking for good generators, one normally sets $MaxMerit(c)$ to 1.0 for each c.

For category c, the test (Beyer, spectral, ..., depending on the criterion) is performed in dimensions $kw + 1$ to t_{c1} for the vectors of successive values, the pairs (u_1, u_j) are tested for $1 < j \leq t_{c2}$, the triples (u_1, u_j, u_ℓ) are tested for $1 < j < \ell \leq t_{c3}$, and so on. If lacunary indices are specified in the field *LaGroupSizes*, all of this is applied to the vectors obtained *after* taking the lacunary indices, and the test for successive values starts in 2 dimensions instead of $kw + 1$. To *minimize* (instead of maximizing) the criterion within a given category c, place a *negative sign* in front of the value of d_c in the data file. The program will then retain the *worst* instead of the *best* generators with respect to the category c.

Criterion ⟨*Invert*⟩ ⟨*Norm*⟩ **:** Specifies the merit criterion for ranking the generators for each category. The field *Invert* can be either `Invert` or left blank. In the former case, the value of the criterion is replaced by its multiplicative inverse in the results. 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.

`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.

`SpectralP` is similar to `Spectral`, except that it is based on the shortest vector in the *primal* lattice.

`SpectralL1` is similar to `Spectral`, except that the length of the (dual) vectors are measured with the L_1 norm, minus 1. The length of the shortest dual vector is then an upper bound on the minimal number of hyperplanes that cover all the points ([9] and [21], Exercises 3.3.4-15 and 16).

`Palpha` *Alpha* means that the criterion is P_α, with the value of α specified just after the keyword `Palpha`. Specialized (much faster) implementations have been made for $\alpha = 2, 4, 6, 8$.

If *Criterion* is `Spectral` or `SpectralP`, then *Norm* must appear and it indicates which type of normalization is used in the definition of S_t. The admissible values are

`Hermite` means that we use Hermite's bound. In this case, S_t is computed only up to dimension 8 and the criterion is $M_{\min(T,8)}$. This is because for $t > 8$, d_t^* is unknown, so M_t cannot be computed.

`Laminated` means that we use for d_t^* the value of d_t that corresponds to the laminated lattice in dimension t. The criterion is then $M_{\min(T,48)}$ (the programs knows this d_t^* only for $t \leq 48$).

`Rogers` means that d_t^* is obtained from Rogers' bound. The criterion is then $M_{\min(T,48)}$.

If *Criterion* is `Palpha`, the field *Norm* must also appear. The admissible values for `Palpha` are:

`Weights` followed on the next line by positive real numbers $\beta_0, \ldots, \beta_t$ where $t = t_{C,1}$. This means that the criterion will be $\tilde{P}_\alpha$ defined in (28). In this case, α must be even. The β_j's are the weights given to the different dimensions in the weighted version of P_α.

LatticeInfo **:** Indicates which figures of merit we want to compute and print. The possibilities are the same as for the criterion, but we can compute and print more than one. It suffices to write more than one keywords on this line. Note that q_t typically requires much more time to compute than d_t.

`Beyer :` Prints the values of q_t up to dimension T.

`Spectral :` Prints the values of d_t up to dimension T and S_t for $t \leq \min(T, T^*)$, where $T^* = 8$ if *Norm* = `Hermite` and $T^* = 48$ otherwise.

`SpectralP` is similar to `Spectral`, except that it is based on the shortest vector in the *primal* lattice.

`SpectralL1` is similar to `Spectral`, except that the length of the (dual) vectors are measured with the L_1 norm, minus 1. The normalization is based on Minkowski's bound ([12], p. 618).

`Palpha` *Alpha* **:** prints P_α with the value of α specified.

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 [29]. For example, if $p = 2$ and $a \bmod 8 = 5$, then the modulus is divided by 4 as in [11, 20].

LaGroupSizes, Spacing **:** 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.

VerifyBB **:** Indicates whether or not the results of the branch-and-bound procedures will be verified with formal error bounds on the numerical computational errors. The verification (if any) is performed using either `REDBAS2` or `REDBAS3`. When such a verification is not performed, the results are not formally guaranteed, but computations are performed an order of magnitude faster. The possible values are:

`VerifyAll2` **:** Performs the verification for all the generators examined, using `REDBAS2`.

`VerifyRetained2` **:** Performs the verification only for the *NbGen* "best" generators retained, using `REDBAS2`.

`VerifyAll3` **:** Performs the verification for all the generators examined, using `REDBAS3`.

`VerifyRetained3` **:** Performs the verification only for the *NbGen* "best" generators retained, using `REDBAS3`.

`NoVerify` **:** Performs no verification at all.

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.

TimeLimit **:** A real number giving the maximal CPU time, in hours, given to the program for performing its search. When this time-limit expires, 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).

$S1$, $S2$ **:** Those integers are the seeds of the generator used during the search, if necessary; e.g., to generate the α_i's. Before starting the search, the program calls `SetSeed(`$S1, S2$`)` form the module `SUPRAND` of SENTIERS and the module `RAND` of SIMOD. To perform a different random search in a region already studied, just change these seeds. One must have $1 \leq S1 \leq 2147483562$ and $1 \leq S2 \leq 2147483398$.

ResultForm **:** Selects in which form the results will be given. The possible values are `(Terminal, RES, GEN, RESGEN)`. 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.

`RESGEN` means that the results will be in a "`.res`" file *and* the retained generators will be listed in a "`.gen`" file, as with `RES` and `GEN` above.

lats

This is the same program as `latl`, except that multipliers and bases are represented using the type `SuperInteger`. (Modules `MULTSI` and `BASISSI` are used.) Therefore, it can deal with larger values of m and a_i's.

latsr

Same as `latl`, except that the multipliers are represented as `SuperInteger`'s. (Modules `MULTSI` and `BASISLR` are used.)

seeks

Same as `seekl`, except that the multipliers and bases and represented as `SuperInteger`'s. (Modules `MULTSI` and `BASISSI` are used.)

seeksr

Same program as `seekl`, except that multipliers are represented as `SuperInteger`'s. (Modules `MULTSI` and `BASISLR` are used.) It can deal with larger values of m and a_i's than `seekl`, but the computations are less precise than with `seeks`.

streams

This program takes as input a data file in the same format as `lats`, with an additional line at the end that contains 3 positive integers $w_{\min}$, $w_{\max}$, and $s_{\max}$. The program ignores the values of *LaGroupSizes* and *Spacings* in the data file. It partitions the period length of the given generator in *streams* of length 2^w, for $w = w_{\min}, \ldots, w_{\max}$. For each w and for $s = 1, \ldots, s_{\max}$, the program analyzes the lattice structure of the vectors of the form $(u_{i+1}, \ldots, u_{i+s}, u_{i+w+1}, \ldots, u_{i+w+s}, u_{i+2w+1}, \ldots, u_{i+2w+s}, \ldots)$, formed by groups of s successive values, taken w values apart. The analysis is performed in the same way as in `lats`. For each w, the program also prints the worst value of the figure of merit over all values of s. These results can be used to select a specific value of w for splitting the generator's cycle into streams. All the computations are done using the `SI SI` representation.

menumrg

This is an interactive program to build, manipulate, compare, and reduce lattice bases associated with LCGs or MRGs. It uses the "`SI SI`" representation. The main purpose is to examine interactively what happens to a basis when different operations are performed on its vectors. This could be useful for "debugging" or to deal with special classes of bases for which the standardized procedures may not work very well. A variant of this module has been used, for instance, to obtain the numerical results given in [4, 39]. In those cases, the shortest vectors and reduced bases were computed in (partly) "manual" mode.

Two bases `B1` and `B2` and one vector `Ve` are created initially (automatically) and available to the user. All along, `B1` and `B2` will be bases of the same lattice. The program performs different operations, classified according to which of those three objects they involve: (a) the operations which affect only `B1`; (b) those affecting `B1` and `B2`; (c) and those involving `B1` and `Ve`. All those operations are available from a menu (and some submenus), as shown in Figure 10. We now briefly explain the different menu options.

```
Operations on B1 :
  1. Building MRG basis
  2. Reading/Writing
  3. Duality verification
  4. Dualizing
  5. Square lengths
  6. Sorting
  7. Pre-reduction
  8. Shortest vector/Minkowski reduction
  9. Shortest/Minkowski verification

Operations involving B1 and B2 :
  10. Copy B1 in B2
  11. Permute B1 and B2
  12. Equivalence of B1 and B2
  13. Relative coordinates
  14. Lexicographic order


Operations involving B1 and Ve :
  15. Vector construction
  16. Relative coordinates
  17. Insertion of Ve in B1
```

Figure 10: Main menu for the program `menumrg`.

Operations on B1 :

1. *Building MRG basis* **:** Asks the user for a modulus, order, multiplier(s), and dimensions, then constructs the appropriate lattice basis in `B1`.

2. *Reading/Writing* **:** Read/write the basis `B1` from/to a file or the terminal.

3. *Duality verification* **:** Verifies whether the dual basis is "correct" by computing the appropriate scalar products.

4. *Dualizing* **:** Interchange the primal/dual bases.

5. *Square lengths* **:** Compute the squared vector lengths. The values can be displayed either in full (large integers) or in scientific notation.

6. *Sorting* **:** Sorts the vectors of `B1` by increasing length.

7. *Pre-reduction* **:** Performs "pairwise" prereductions T and T^* of [9] in various combinations or orders. Can also ask for a given number of pre-reductions.

8. *Shortest vector/Minkowski reduction* **:** Computes the shortest vector in the lattice, or a MRLB, using the module `REDBAS` (does not compute error bounds).

9. *Shortest/Minkowski verification* **:** Verifies whether the first vector of `B1` is really a shortest vector, or whether the current basis is really a MRLB, by performing the branch-and-bound with error bounds (using either `REDBAS2` or `REDBAS3`). An option is also available to compute the squared distance between each vector of `B1` and the subspace generated by all the other vectors, and to select the vectors for which that distance is smaller than the length of the (current) shortest vector in `B1`.

Operations involving B1 and B2 :

10. *Copy B1 onto B2* **:** Copies the basis `B1` onto `B2`.

11. *Permute B1 and B2* **:** Permutes `B1` and `B2`.

12. *Equivalence of B1 and B2* **:** Tells whether or not `B1` and `B2` generate the same lattice. Assumes that the dual basis is "updated".

13. *Relative coordinates* **:** Gives the relative coordinates of a vector of `B1` relative to the basis `B2`.

14. *Lexicographic order* **:** Displays the squared lengths of the vectors of `B1` and `B2` in lexicographic order. Assumes that the two bases are already sorted.

Operations involving B1 and Ve :

15. *Vector construction* **:** Permits one to either initialize `Ve` to zero; add to `Ve` a multiple of a vector of `B1`; increase the dimension of `Ve` by one (the new coordinate is zero); shift the coordinates of `Ve` to the right by one (provided that the last coordinate is zero); or perform a linear combination of iterates of the latter shifting operations (as in [4]).

16. *Relative coordinates* **:** Computes the coordinates of `Ve` relative to `B1`.

17. *Insertion of Ve in B1* **:** Inserts `Ve` into the basis `B1`, provided that `Ve` belongs to the lattice generated by `B1` and is primitive.

APPENDIX B

Intermediate-Level Modules

LATMRG

The files `LATMRG.h` and `LATMRG.c` exist uniquely to permit one to use LatMRG directly from a **C** program. For this, one must include the directive

```
#include "LATMRG.h"
```

at the beginning of the file and then add

```
LATMRG_BEGIN (argc, argv);
```

as a first instruction in the fonction

```
int main(int argc, char **argv);
```

To use functions from `mylib`, `simod`, and `sentiers`, one must include the header files

```
#include "MYLIB.h"
#include "SIMOD.h"
#include "SENTIERS.h"
```

One can consult the appropriate `*.h` header files in `latmrg/xds` to see the prototypes for the external **C** functions.

```
DEFINITION MODULE ["C"] LATMRG;

PROCEDURE LATMRG_BEGIN();

END LATMRG.
```

palpha

Le $P_\alpha(s)$ est une mesure de la qualité d'intégration d'une règle de treillis. D'une manière générale, une règle de treillis permet d'approximer l'intégrale de f, une fonction, sur le cube unitaire de dimension s. f doit être lisse et périodique par rapport à $x \in \mathbf{R}$.

La formule générale du $P_\alpha(s)$ pour α entier et pair est

$$P_\alpha(s) = -1 + 1/m \sum_{i=0}^{m-2} \prod_{j=0}^{s-1} \Big[1 - \frac{(-1)^{\alpha/2}(2\pi)^\alpha}{\alpha!} B_\alpha(x_{i+j})\Big] \tag{28}$$

où les $B_\alpha(x)$ sont les polynômes de Bernoulli. x_{i+j} est le $i + j^e$ nombre généré et se situe entre 0 et 1. Les premiers polynômes de Bernoulli sont

$$B_0(x) = 1, \qquad B_1(x) = x - 1/2, \qquad B_2(x) = x^3 - x^2 + 1/6,$$
$$B_3(x) = x^3 - (3/2)x^2 + (1/2)x, \qquad B_4(x) = x^4 - 2x^2 + x - 1/30$$

Les $B_\alpha(x)$ se calculent en utilisant l'extension de la série suivante:

$$\frac{te^{xt}}{e^t - 1} = \sum_{s=0}^{\infty} B_s \frac{t^s}{s!}$$

On peut ajouter un poids à chacune des dimensions s, pour des α pairs. La formule du $P_\alpha(s)$ devient:

$$P_{2\alpha}(s) = \beta_0\Big\{ -1 + 1/m \sum_{i=0}^{m-2} \prod_{j=0}^{s-1} \Big[1 - \frac{(-(2\pi\beta_{j+1})^2)^\alpha}{2\alpha!} B_{2\alpha}(x_{i+j})\Big]\Big\}$$

Le $B_\alpha(s)$, à ne pas confondre avec le $B_\alpha(x)$, est une mesure indiquant qu'il existe au moins un $P_\alpha(s) \leq B_\alpha(s)$. D'une manière générale, on peut le voir comme une borne supérieure valide en présence des conditions suivantes: $m > e^{\alpha s/(\alpha-1)}$ et premier, $s \geq 2$ et $\alpha > 1$. Pour plus de détails consulter le théorème 4.4, p.83 du livre de Sloan et Joe. Il se calcule ainsi:

$$B_\alpha(s) = \Big(\frac{e}{s}\Big)^{\alpha s} \frac{(2\ln m + s)^{\alpha s}}{m^\alpha}$$

Le module PALPHA

Module servant de support aux programmes PAl, BAl et Coeff. Ils permettent respectivement de trouver le P_α ou le B_α d'un générateur ou enfin de trouver des bons coefficients pour un générateur en utilisant le P_α comme discriminant.

palpha

Ce programme permet de calculer le P_α pour un générateur et un α donné. Tout dépendant du α la fonction de calcul changera. Les données suivantes doivent être placées dans un fichier avec l'extension `".dat"` selon le format de la figure 1. Les résultats seront placés dans un fichier avec l'extension `".res"`.

balpha

Permet de calculer le B_α d'un générateur. Pour l'instant seulement le cas avec $\alpha = 2$ est implémenté. Le format du fichier est le même que pour le programme PAl. Les données devront être placées dans un fichier avec l'extension `".dat"` et les résultats se trouveront dans le fichier avec le même préfixe, cette fois avec l'extension `".res"`.

Le format du fichier d'entrée pour PAl ou BAl

LCG	*SorteGenerateur*
PAl ou BAl	*Type de calcul*
251	*m modulo*
true	*modulo premier*
33	*a multiplicateur*
2	*mindim*
4	*maxdim*
2	*saut*
2	*alpha*
false	*VerifBooleen*
1	*seed*
1.0 2.0 3.0 1.0 1.0	*Beta(0),...,Beta(MaxDim)*

Figure 1 : Format du fichier pour PAl ou BAl

Type de calcul : Permet de déterminer le type de calcul à effectuer.

Modulo : le modulo du générateur en base 10, soit un nombre premier ou une puissance de 2. Ce nombre doit être inférieur à 3.03E09 ou 2^{31}.

Modulo premier : Booléen indiquant si le modulo utilisé est un nombre premier.

Multiplicateur : le multiplicateur du générateur en base 10. Il doit être inférieur au modulo. Ou un intervalle dans le cas de la recherche de coefficient. La recherche permettra de trouver le meilleur coefficient dans cet intervalle en fonction du P_α.

Mindim : Entier donnant la borne inférieur sur le s (la dimension) dans le calcul à effectuer.

Maxdim : Entier donnant la borne supérieur sur le s (la dimension) dans le calcul à effectuer.

Saut : Entier déterminant le saut à effectuer entre les dimensions.

alpha : Entier pair, en général de 2 à 10.

VerifBooleen : Si `true` alors on vérifira si la période est maximale, mettre à `false` si aucune vérification n'est nécessaire. Dans le cas de la recherche de coefficient, si `true` alors seulement les générateurs à période maximale seront considérés, si `false` alors le P_α de tous les générateurs sera évalué. Si par contre le modulo n'est pas un nombre premier alors peu importe la valeur entrée à *VerifBooleen*, nous allons effectuer une recherche sur l'ensemble des coefficients.

Seed : Entier inférieur au modulo qui servira de germe dans le générateur.

Beta : Nombres réels permettant de donner un poids à chacune des dimensions. Il faut entrer dans le fichier une valeur pour $\beta(0), ..., \beta(MaxDim)$.

LATIO

This module contains several tools for input/output, for performing generator searches, for managing and storing lists of generators, and the like. It contains the actual implementations of the high-level "seek" and "lat" programs. The procedures here use most of the other lower-level and intermediate-level modules. All the `Read...` procedures read the data in the format explained in the documentation of the `seekl` program.

The searches for lists of (good or bad) generators are performed by categories, as explained in the documentation of the `seekl` program. The set of dimensions for which the lattice structure is examined is partitioned into a certain number of intervals (one or more), also called *categories*. We give bounds on the accepted values of the figure of merit for each category and, for any given category c, we consider only those generators that satisfy the bounds for all the categories less or equal to c. Then we retain a list of generators with the best or worst figures of merit for each category.

```
DEFINITION MODULE LATIO;

FROM SUPINT   IMPORT SuperInteger;
FROM SUPFACT  IMPORT Factors;
FROM CONFIG   IMPORT BDim, OutputType, MaxJ, MaxCat, Indexj, MaxNbGen;
FROM MULT     IMPORT MScal, MVect;
FROM NORM     IMPORT CriterionType;
FROM REDBAS   IMPORT VerifyType;
IMPORT BASISLR;


TYPE
   GenerType   = (MRG, MWC);
```

Type of generator that can be considered.

```
   CondType    = (NoCond, AppFact, MaxBits);
```

Type of condition on the multipliers.

```
   VerifType   = (VerifyAll2, VerifyRetained2,
                  VerifyAll3, VerifyRetained3, NoVerify);
```

Type of verification that can be performed on the results (i.e., type of computational method).

```
   Poly        = POINTER TO InfoPoly;
   InfoPoly    = RECORD
      a        : MVect;
      NextPoly : Poly
      END;
```

Elements to store a linked list of polynomials.

```
   Resultats   = POINTER TO InfoRes;
   InfoRes     = RECORD
      RD2, RS2, RQ2 : BASISLR.BVect;
      DimD2, DimQ2  : BDim
      END;
```

Type of record used as a field of the record type `InfoGen`, to store detailed results for a given generator.

It contains the *squared* values of the d_t, S_t, and q_t, and the maximal dimensions `DimD2` and `DimQ2` for which d_t and q_t, respectively, have been computed.

```
Generateur = POINTER TO InfoGen;
InfoGen    = RECORD
   Maj      : ARRAY Indexj OF MVect;
   a        : MVect;
   DimMerit : BDim;
   Merit    : LONGREAL;
   Res      : Resultats;
   NextGen  : Generateur
   END;
```

Type of record used to store a linked list of generators. Used to store the generators that are kept in each category. The coefficients $a_{j,i}$ of the components are in `Maj` whereas those of the combined generator are in `a`. `DimMerit` is the dimension where the minimum is attained for the figure of merit (for the given category) and `Merit` gives the value of this figure of merit. `Res` is a pointer to the more detailed results and `NextGen` is a pointer to the next generator in the list.

```
ToZone = POINTER TO Zone;
Zone   = RECORD
   No         : INTEGER;
   Binf, Bsup : SuperInteger;
   p          : LONGREAL;
   PetiteDiff : BOOLEAN;
   BsupMsH    : SuperInteger;
   Mq, MBsup  : MScal;
   NextZone   : ToZone;
   END;
```

Type of record used to store the research zones. `No` is the number of the zone, which is in $\{0, 1, 2, 3\}$. `Binf` and `Bsup` are the lower and upper bounds defining the zone. `P` is the fraction of the acceptable values of a lying in this zone (this is used for the random searches). `PetiteDiff` is `TRUE` iff `Bsup - Binf` $\leq$ `H` (or `Hk`), and when it is `TRUE`, `BsupMsH` is equal to `Bsup` $+ 1 -$ `H` (or `Hk`). This is also used for random searches. The variables `Mq` and `MBsup` are used in `ExamAllZones`.

```
ToRegion = POINTER TO Region;
Region   = RECORD
   No         : INTEGER;
   Binf, Bsup : MScal;
   Mq         : MScal
   END;
```

Type of record used to store the regions that are to be searched. `No` is the zone number where this region is (in $\{0, 1, 2, 3\}$). `Binf` and `Bsup` are the lower and upper bounds defining the region. The variable `Mq` is used in `ExamRegion`.

```
PROCEDURE TimeOver () : BOOLEAN;
```

Returns `TRUE` if and only if the cpu time limit has been reached.

```
PROCEDURE CreateGenerateur (VAR G : Generateur);
```

Creates a record for a generator `G` and initializes its fields.

```
PROCEDURE CreateZone (VAR Z : ToZone);
```

Creates a record for a zone `Z` and initializes its fields.

`PROCEDURE CreateRegion (VAR R : ToRegion);`

Creates a record for a region `R` and initializes its fields.

`PROCEDURE LireFact (VAR L : Factors;  NomFch : ARRAY OF CHAR);`

Reads a list of factors in the file `NomFich` and places them in the list `L`. Assumes that the factors in the file are written according to the format used by `Ecrirefact`.

`PROCEDURE EcrireFact (L : Factors;  NomFch : ARRAY OF CHAR);`

Writes the factors of the list `L` to the file `NomFich`. It this file exists, it is rewritten. The factors can be read back later by `LireFact`.

`PROCEDURE Readm (j : Indexj);`

Reads the modulo m_j for the component j. Initializes some constants that depend on it.

`PROCEDURE ReadOnea (aa : SuperInteger);`

Reads one multiplier $a_{j,i}$ or one bound $b_{j,i}$ or $c_{j,i}$ in the format described in `seekl`.

`PROCEDURE ReadReadGen;`

Reads a boolean indicating if the generators to be tested should be taken from a '`.gen`' file, and if true, also reads the name of the file, without the `.gen` extension.

`PROCEDURE ReadTypeGen (j : Indexj);`

Reads the type of generator to be considered, MRG or MWC, for component j.

`PROCEDURE ReadOrder (j : Indexj);`

Reads the order k_j of the recurrence for component j.

`PROCEDURE ReadPerMax (j : Indexj);`

Reads a boolean indicating if component j should have maximal period. The muliplier m_j must have been read in the '3 numbers' format by `readmj`, otherwise this boolean is reset to `FALSE`.

`PROCEDURE ReadImplemCond (j : Indexj);`

Reads the condition that we impose on the coefficients $a_{j,i}$ for the component j.

`PROCEDURE ReadFactmr (j : Indexj);`

Reads what to do about the factorizations of $m_j - 1$ and $r_j = (m_j^{k_j} - 1)/(m_j - 1)$, for the component j.

`PROCEDURE Reada (j : Indexj);`

Reads the k_j multipliers $a_{j,1}, \ldots, a_{j,k_j}$ for component j.

`PROCEDURE Readbc (j : Indexj);`

Reads the $2k_j$ bounds $b_{j,1}, c_{j,1} \ldots, b_{j,k_j}, c_{j,k_j}$ on the multipliers for component j.

`PROCEDURE ReadSearchMethod (j : Indexj);`

Reads the method of search, exhaustive or random, for the polynomials for component j. If the method is `Random`, the it reads also the number of regions and their sizes.

`PROCEDURE ReadMRGComponent (j : Indexj);`

Reads all the parameters related to the component j, by calling the appropriate '`Read...`' procedures above. Also creates the required data structures for this component.

`PROCEDURE ReadMinMaxDim;`

Reads the minimal and maximal numbers of dimensions in which the test is to be applied, the number of categories, and the boundaries of these categories. The format is slightly different for the 'seek' and 'lat' programs.

`PROCEDURE ReadMinMaxMerit;`

Reads the minimum and maximum acceptable values for the figure of merit for each category of dimensions.

`PROCEDURE ReadNbGen;`

Reads the number of generators that we want to retain in each category.

`PROCEDURE ReadCriterion;`

Reads the selection criterion (figure of merit).

`PROCEDURE ReadLatInfo;`

Reads what lattice information we want to be printed in the results.

`PROCEDURE ReadLatType;`

Reads which type of lattice we want to analyze: That generated by the recurrent states only, or that generated by a single orbit, or that generated by the longest cycle of a generator with prime power modulus, or the full lattice generated by the union of all the orbits. In the case of `Orbit`, this procedure also reads the initial state and initializes some additional data structures.

`PROCEDURE ReadLaIndices;`

Reads the indices (lacunary or successive) that are to be considered.

`PROCEDURE ReadVerifyBB;`

Reads the verification level that we want to use.

`PROCEDURE ReadMaxNodesBB;`

Reads the maximum number of nodes that will be allowed in the branch-and-bound tree. When that number is exceeded, the branch-and-bound procedure will stop.

`PROCEDURE ReadTimeLimit;`

Reads the cpu time limit that is allowed for a search. When that limit is exceeded, the search procedure stops and reports what it has found so far.

`PROCEDURE ReadSeeds;`

Reads the seeds that are used for the random number generator of SENTIERS. This generator is used for the random searches.

`PROCEDURE ReadResultForm;`

Reads the form of the output (on the terminal, in a `.gen` file, in a `.res` file, in a `.tex` file).

`PROCEDURE Computem;`

Computes the order and the modulus m for the combined generator, given those of the components. Computes also the period lengths of the components and of the combination.

`PROCEDURE Computea;`

Computes the vector of multipliers a for the combined generators, given the parameters of the components. Assumes that the moduli of the components are all prime. Initializes the vectors `Maj` and `a`.

`PROCEDURE SaisieDonnees;`

Calls the different low-level procedures to read the data from the data file, in the format appropriate for the programs `latl` and `seekl`. Also creates some of the data structures (e.g., `SuperInteger`'s) that are required.

`PROCEDURE ReadGenFile;`

Reads a set of generators from the file '`xxx.gen`', assuming that the data has been read from '`xxx.dat`'.

`PROCEDURE PerMaxPowPrime (j : Indexj) : BOOLEAN;`

Assumes that m_j is a power of a prime p, that $k_j = 1$, and that the recurrence is homogeneous. Returns `TRUE` iff the maximal period conditions are satisfied for that case for component j.

`PROCEDURE Reducemj (VAR a : MVect);`

Reduces m_j and reinitializes the module `MRG` accordingly in the case where the modulus is a power of a prime and where `LatType = PrimePower`. Called by `testGen`.

`PROCEDURE InitZones (j, i : INTEGER);`

Initializes the research zones for the multiplier a_i of the component j. In the case of a random search, this procedure also creates the region for (j, i). Called by `SetupGenSeek`.

`PROCEDURE ChoisirBornes (j : Indexj);`

This procedure is called by `SeekGen` in the case of a random search. It chooses a region at random and initializes its boundaries. This region will then be searched completely.

```
PROCEDURE TestGen
   ( VAR a     : MVect;
     T         : CriterionType;
     VerifyBB  : VerifyType;
     Out       : OutputType
   );
```

Tests the (combined) generator with multiplier (vector) `a`.

`PROCEDURE ConserverGen (C : INTEGER;  tMerit2 : BDim);`

Inserts the most recently tested generator into the list for the categoty `Cat`. The value `tMerit2` indicates the dimension where the worst-case is attained for the figure of merit. This procedure is called by `VerifyCategories`.

`PROCEDURE VerifyCategories (VAR Me2 : BASISLR.BVect);`

Verifies if the most recently tested generator can be included in any of the lists corresponding to the different categories. If so, includes it in the appropriate lists. The vector `Me2` is either `Q2` or `S2`, depending on the figure of merit that is used.

`PROCEDURE ExamAllZones (j, i : INTEGER);`

This procedure is used in the "exhaustive search" case. When $J > 1$, it collects primitive polynomials for the given j. When $J = 1$, it tests all generators in the region. The initial call should be with i equal to the degree of the polynomials for component j. Then, the procedure calls itself recursively for smaller i.

`PROCEDURE ExamRegion (j, i : INTEGER);`

Similar to `ExamAllZones`, but used for the "Random search" case.

`PROCEDURE ExamCombPoly (jj : Indexj);`

Used only when $J > 1$. Examines all combinations of the primitive polynomials found by `ExamRegion` or `ExamAllZones`.

`PROCEDURE ExamCurrenta;`

Tests the generator that corresponds to the current value of the multiplier (vector) `a`, and adds it to the list(s) if it is competitive. Called only by `TestGenFromFile`.

`PROCEDURE TestGenFromFile;`

Examines all the generators from a `.gen` file, which is assumed to be the current input file.

`PROCEDURE CalculNbreGen (j : Indexj);`

Computes the total number of simple generators (i.e., polynomials) examined, for the component `j`, and puts the result in `TotGen [j]`.

`PROCEDURE CalculerResteInfo;`

After the search has been completed, computes all the remaining required information on the retained generators. Computes the d_t or the q_t when required and not already done. Redo the computations with error bounds if requested.

`PROCEDURE PrintErrBounds (T : INTEGER);`

Computes and prints the bounds on the Euclidean distances between the points produced by a L'Ecuyer-style combined MRG and the corresponding points produced by the associated MRG, in dimensions 2 to `T`. This is used only when $J > 1$.

`PROCEDURE EcrisGen (G : Generateur;  Cat : INTEGER);`

Writes the parameters and the results relative to a retained generator `G`, in category `Cat`. Called by `WriteSeekRes`.

`PROCEDURE WriteLatHead;`

Writes a heading for the results of a lattice test. Called by `LatGen`.

`PROCEDURE WriteLatHeadTex;`

Writes a heading for the results of a lattice test in LaTeX form. Called by `LatGen`.

`PROCEDURE WriteDataComponent (j : Indexj);`

Write the statistical data relative to component `j`.

`PROCEDURE WriteStatComponent (j : Indexj);`

Writes the statistics on the searches for primitive (or examined) polynomials for the component `j`. Called by `WriteSeekRes`.

`PROCEDURE WriteTabBest (l : LONGINT;  G : Generateur);`

Prints a one-line summary of the figures of merit in each interval, for best generator retained in category (i.e., interval) number `l`.

`PROCEDURE WriteSeekHead;`

Writes, in the current output file, the heading and the data for a search by the `SeekGen` procedure. Writes also the factorizations in the files if this was requested.

`PROCEDURE WriteSeekRes;`

Writes the results of a search in the current output file, then closes it.

`PROCEDURE WriteSeekGen;`

Writes in a `.gen` file the generators found in a search. If the data was read in `xxx.dat`, then this procedure will open the file `xxx.gen` and write in it.

`PROCEDURE SetupSeekGen;`

Called by `SetUp`. Performs initializations for the searches of generators. Creates the ordered lists to store the 'best' `NbGen [C]` generators in each category `C`. Initializes the bounds on the figures of merit in each dimension. Initializes the polynomial lists (to empty) and the search zones.

`PROCEDURE Setup;`

Obtains and opens data file, reads data and performs initializations. Initializes `REDBAS` and `MRG`. This procedure is called by `SeekGen` and `LatGen`.

`PROCEDURE LatGen;`

Applies the required tests (for "lat" programs) and prints results.

`PROCEDURE SeekGen;`

Performs a seek for "good" generators (implements "seek" programs).

`PROCEDURE CalcLaIndicesStreams (s, w : LONGINT);`

Calculates lacunary indices by groups of s spaced by 2^w

`PROCEDURE LatGenStreams;`

Applies the test for different values of w where a big generator is divided into smaller ones, which are separated by 2^w.

`END LATIO.`

MRG

This module offers tools to analyze simple or combined multiple recursive (linear congruential) generators (MRGs). One must first initialize the module with a given modulus, a given order, and a maximal dimension, by calling `InitMRG`. Each MRG will then be defined by a vector of multipliers `a`, where `a[i]` represents a_i and the MRG is defined by

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

One can then build the lattice basis associated to a vector of multipliers for a given dimension, or apply the spectral test, or the Beyer test, or both simultaneously, for all dimensions up to a specified limit. The multipliers and modulus can be represented as `LONGINT` or `SuperInteger`. This can be changed by the command `select`.

```
DEFINITION MODULE MRG;

FROM MULT     IMPORT MScal, MVect;
FROM BASIS    IMPORT BVect;
FROM CONFIG   IMPORT MDim, BDim, OutputType;
FROM LATBASIS IMPORT Basis;
FROM TESTLAT  IMPORT TestConfig;
FROM NORM     IMPORT CriterionType, NormType, Normaliz;
FROM REDBAS   IMPORT VerifyType;
FROM SUPINT   IMPORT SuperInteger;
FROM SUPFACT  IMPORT Factors;
IMPORT BASISLR, MULTLI, MULTSI;

TYPE
   MRGen =  POINTER TO InfoMRG;
   InfoMRG = RECORD
      Bas   : Basis;
      k     : MDim;                    (* k = Order  of the recurrence *)
      aLI   : MULTLI.MVect;            (* Coefficients of the recurrence *)
      aSI   : MULTSI.MVect;            (*               "                *)
      END;
```

```
   LatticeType = (Full, Recurrent, Orbit, PrimePower);
```

Indicates which type of lattice is analyzed: that generated by all possible states (`Full`), or by the set of recurrent states (`Recurrent`), or by a single forward orbit of the generator (`Orbit`) for the case where it has several cycles, or by a single orbit in the case where some component is an MLCG with modulus that is a power of a prime p (`PrimePower`). In the latter case, one must divide the modulus by an appropriate power of p, as described in [29], before calling a test procedure. When `LatticeType = Orbit`, the initial state must be specified in the array `InSta` (see below). See also the data field *LatticeType* for the program `seekl`.

```
VAR
   Bas : Basis;
```

Lattice basis built and used by the procedure `Test`. Can be recovered after calling `Test`.

`Q2, D2, S2 : BASISLR.BVect;`

Those vectors contain the *squares* of the Beyer quotients q_t, of the distances between hyperplanes d_t, and of the values of S_t, respectively, computed in the last call of `Test` or `TestLa`.

`Mins2, Minq2 : BASISLR.BVect;`

Gives the current minimal values of S_t and q_t for a generator to be considered, in each dimension. The module initializes these bounds to zero.

`MaxAllDim : BOOLEAN;`

`TRUE` if we are always looking for generators with the highest figures of merit, in all the dimensions. `FALSE` if we want to *minimize* the figure of merit in some set of dimensions (that is, in some interval or category).

`TestCompleted : BOOLEAN;`

`TRUE` if the last call to `Test` or `TestLa` has been successful, `FALSE` otherwise.

`Lacunary : BOOLEAN;`

When this variable is `TRUE`, the module treats lacunary indices; otherwise it assumes that the vectors are formed with successive indices.

`LatType : LatticeType;`

Indicates which lattice or sublattice is analyzed. See the data field *LatticeType* in `seekl`. However, in the case `PrimePower`, the modulus of each component must be divided appropriately *before* calling `Test` or `TestLa`; these test procedures assume that the modulus has already been ajusted (reduced).

`DualLattice : BOOLEAN;`

When `TRUE`, we seek short vectors in the *dual* lattice (e.g., to compute the distance between adjacent hyperplanes). When `FALSE`, we seek short vectors in the *primal* lattice. The default value is `TRUE`.

`UsedNorm : NormType;`

Which norm type is used to measure the length of the vectors. The default value is `L2Norm`. WARNING: For other norms, the implementation is only partial, and only `Spectral` is admitted for the type of test.

`NormalG : Normaliz;`

To normalize the length of the shortest vectors (See module NORM).

`InvertCrit : BOOLEAN;`

If `TRUE`, the value of the criterion S_t is *inverted* in the results, i.e., the criterion S_t becomes a number larger than 1, the *smaller* the better.

`InSta : ARRAY MDim OF SuperInteger;`

If `LatType = Orbit` then the user must specify a seed which will be stored in `InSta`.

`PROCEDURE InitMRG (m : MScal;  k : MDim;  d : BDim);`

From now on, this module will treat multiple recursive generators of order `k` with modulus `m`. Dimensions of bases cannot exceed `d`. (One can recall this procedure to change `m`, `k`, or `d`). This procedure calls `MULT.MSetm`.

```
PROCEDURE DeleteMRG (d : BDim);
```

Deallocates the memory allocated to the matrices of `SuperIntegers VSI` and `WSI`, and to the vectors of type `BVect`, `xi, LacI` and `q`, at a previous call to `InitMRG`. It also calls the procedure `LATBASIS.DeleteBasis`, because `LATBASIS.InitBasis` is called by `InitMRG`. Must be called only if `InitMRG` has been called previously.

```
PROCEDURE BuildBasis (B : Basis;   VAR a : MVect;   n : BDim);
```

Builds the basis directly in dimension `n`.

```
PROCEDURE IncDimBasis (B : Basis);
```

Increments by 1 the dimension of the basis.

```
PROCEDURE Test
   ( VAR a           : MVect;
     T               : CriterionType;
     FromDim, ToDim  : BDim;
     VerifyBB        : VerifyType;
     Out             : OutputType );
```

Applies the test `T` (Beyer and/or Spectral) to the MRG generator corresponding to `a`, in dimensions `FromDim` to `ToDim`. The initial basis is built automatically, and during the procedure, the variable `Bas` will contain the current basis. `VerifyBB` indicates if the results must be verified in a second pass, with error bounds on the floating-point computations for the branch-and-bound. The parameter `Out` specifies the form of output produced. If a test fails, then the procedure stops and the variable `TestCompleted` is set to `FALSE`.

```
PROCEDURE TestCri
   ( VAR a           : MVect;
     T               : CriterionType;
     FromDim, ToDim  : BDim;
     VerifyBB        : VerifyType;
     Out             : OutputType;
     r               : BDim;
     n               : INTEGER );
```

Does the same thing as `Test`, but two additional parameters `r` and `n` must be specified, which are the rank and the copy factor for copy-rules, and then the basis are modified accordingly. Must be used only if `FromDim` is larger than `r`. Not very useful in the context of LCG's, but it would have been complicated to write this procedure and the one it calls, `CriSpectralTest()`, in a separate module, because they use variables internally declared in `MRG`. The LCG corresponding to the rank 1 rule must have its modulo and multiplier modified before `TestCri` is called. THIS PROCEDURE IS NOT IMPLEMENTED.

```
PROCEDURE BuildLaBasis
   (B : Basis;   VAR a : MVect;   VAR I : BVect;   n, MaxDim : BDim);
```

Builds the (lacunary) basis directly in dimension `n`. One must indicate MaxDim and use IncDimLaBasis only for dimensions smaller than this. The vector `I` must contain the `MaxDim` lacunary indices.

```
PROCEDURE IncDimLaBasis (B : Basis);
```

Increments by 1 the dimension of the basis. The basis must have been initialized by `InitLaBasis` and the dimension of `B` must be less than `MaxDim`.

```
PROCEDURE TestLa
   ( VAR a          : MVect;
     VAR I          : BVect;
     T              : CriterionType;
     FromDim, ToDim : BDim;
     VerifyBB       : VerifyType;
     Out            : OutputType );
```

Similar as `Test`, but with lacunary indices. The vector `I` must contain the `ToDim` lacunary indices.

```
PROCEDURE PerMax (VAR a : MVect;   VAR ListmMS1, Listr : Factors) : BOOLEAN;
```

Checks if the generator corresponding to `a` has maximal period. If `m` is not prime then the procedures stops and `TestCompleted` is set to `FALSE`. `ListmMS1` and `Listr` must contain the factors of `m-1` and `r`, or `NIL` if the factors are unknown. (See `MULT.SetFactors`.)

```
END MRG.
```

TESTLAT

This module offers a procedure to test several projections of the lattice specified by a modulus and a set of generating vectors. Several categories of projections can be considered, with specific projections in each category. All these parameters are specified via `InitTestConfig`, stored in a `TestConfig` structure which is passed to the `TestLattice` procedure.

```
DEFINITION MODULE TESTLAT;

FROM BASIS    IMPORT BVect;
FROM MULT     IMPORT MScal, MVect;
FROM CONFIG   IMPORT MDim, BDim, DimProj, MCoord, Categ, OutputType;
FROM NORM     IMPORT CriterionType, Normaliz, NormType, GammaType;
FROM LATBASIS IMPORT Basis;
FROM REDBAS   IMPORT VerifyType;
IMPORT BASISLR;


TYPE
   CoordSet = SET OF MCoord;

   TestConfig = POINTER TO InfoTestConfig;
   InfoTestConfig = RECORD
      m              : MScal;
```

The modulus.

```
      k              : MDim;
```

Number of generating vectors (usually equal to the order of the linear recurrence).

```
      NbCoord        : MCoord;
```

Total number of coordinates that are considered.

```
      Maxds          : BDim;
      Maxdp          : DimProj;
```

`Maxds` is the maximum dimension of the projections over successive coordinates. `Maxdp` is the maximum dimension of the projections over nonsuccessive coordinates. These are the values used for memory allocation in `InitTestConfig`.

```
      NbCats, C      : Categ;
```

Number of categories (or lists) that are considered. The variable `C` is initially set to `NbCats` by `TestLattice`, and is decreased when the generator is not good enough for the category currently considered.

```
      DimenProj      : ARRAY Categ OF BDim;
      RangeProj      : ARRAY Categ OF ARRAY DimProj OF BDim;
      DimRegLag      : ARRAY Categ OF BDim;
      RangeRegLag    : ARRAY Categ OF ARRAY DimProj OF BDim;
```

These arrays specify which projections are considered for each category. `DimenProj[c]` is the maximum number of dimensions for which the projections over non-successive and non-regular indices are considered. (E.g., if the value is 3, the pairs and triples are considered.) For $j >$

1, `RangeProj[c,j]` indicates the maximal value of $i_j + 1$ that is considered for j-dimensional projections in category c, assuming that $i_1 = 0$. For $j = 1$, it gives the maximum dimension for the successive dimensions, for category c. Non-successive indices that are regularly spaced, i.e., where $i_j = (j-1)s$ for some s, are also considered. For category c, `DimRegLag[c]` gives the maximum value of s and `RangeRegLag[c,s]` gives the maximum value of j (number of dimensions) that is considered for this s.

```
Mins2, Maxs2  : ARRAY Categ OF LONGREAL;
```

For each category c, gives the minimum and maximum value of the worst-case criterion within category c for a generator to be considered for category c and the categories above it. Whenever `ValCrit[c]` is outside the range [`Mins2[c], Maxs2[c]`], the corresponding generator need not be considered any longer for all the categories $c' \geq c$. It may still be considered, however, for the categories $c' < c$.

```
InvertL, InvertS : BOOLEAN;
```

Indicates if we want to print the multiplicative inverses of the length of the shortest vector and/or the figure of merit `S2`. For the spectral test with the Euclidean norm, for instance, if `InvertL = TRUE`, we obtain the distance between hyperplanes instead of the length of the shortest vector. Default values are `FALSE`.

```
Criterion     : CriterionType;
```

Indicates which criterion is used.

```
Normal        : Normaliz;
```

`Normal` specifies the normalization method used to compute `S2`.

```
Verify        : VerifyType;
```

Indicates the level of verification (i.e., `REDBAS2` or `REDBAS3` is used).

```
Output        : OutputType;
```

Indicates what kind of output is written during the test.

```
MasterGen     : ARRAY MDim OF ARRAY MCoord OF MScal;
```

`MasterGen` contains an initial set of k generating vectors for all the `NbCoord` coordinates that are to be considered in the projections. The projections of these vectors, obtained by selecting the appropriate columns in `MasterGen`, together with additional vectors of the form $(\ldots, m, \ldots)$, are used to build the triangular bases in `TBas`.

```
TBas          : Basis;
```

Contains a triangular version of the basis for the `d` coordinates that are currently under consideration. Used by the `TestProjections` procedure.

```
RBas          : ARRAY DimProj OF Basis;
```

`RBas[d]` contains a reduced basis for the `d` coordinates that are currently under consideration. This is used by the `TestProjections` procedure, which goes back and forth in this array while exploring the tree of projections. `RBas[0]` is used for the lattices that correspond to successive coordinates.

`L2, S2 : ARRAY DimProj OF BASISLR.BVect;`

The vector `L2` will contain the lengths of the shortest vectors raised to the power p in case of the spectral test with the $\mathcal{L}_p$ norm. More precisely, `L2[1,t]` is the value for the lattice that corresponds to successive coordinates in t dimensions, whereas `L2[d,t]`, for $d > 1$, is the worst value for all the d-dimensional projections over coordinate sets $I = \{i_1, \ldots, i_d\}$ where $i_1 = 0$ and $i_d = t - 1$. In all cases, `S2[d,t]` will contain the corresponding normalized quantity, also raised to the power p, or the Beyer quotient in case of the Beyer test.

`ValCrit : ARRAY Categ OF LONGREAL;`

`ValCrit[c]` will contain the worst-case value of the figure of merit in `S2[d,t]` over the set of projections that are considered for category `c`.

`OK : BOOLEAN;`

Indicates if the last operation (e.g., call to `TestLattice, SpectralTest`, etc.) has been successful.

`END;`

```
PROCEDURE CreateTestConfig (VAR T : TestConfig);
```

Creates the structure `T`. One must then call `InitTestConfig` to allocate memory for the elements of `MasterGen, TBas, RBas, L2`, and `S2`.

```
PROCEDURE DeleteTestConfig (VAR T : TestConfig);
```

Deletes the structure `T`, after deallocating the memory allocated to its fields by `InitTestConfig`.

```
PROCEDURE InitTestConfig
  ( T : TestConfig;  Norm : NormType;
    m  : MScal;   k  : MDim;      NbCoord : MCoord;
    ds : BDim;    dp : DimProj;   NbCats  : Categ;
    gamma : GammaType;   beta : LONGREAL
  );
```

Initializes the structure `T`. In the parameters, `m` represent the modulus, `k` the number of generating vectors, `NbCoord` the total number of coordinates to be considered, `ds` the maximum dimension of the projections over successive coordinates, `dp` the maximum dimension of the projections over nonsuccessive coordinates, and `NbCats` the number of categories. This procedure creates the elements of `MasterGen[1..k, 1..NbCoord]`, `L2`, and `S2`, creates the bases `TBas` and `RBas[0..d]`, initializes their sizes appropriately, their modulus to m, and their `Norm` field to `Norm`. The parameters `gamma` and `beta` are used to initialize `T^.Normal` by calling `InitNormaliz`.

```
PROCEDURE AddToBasis (T : TestConfig;  d : BDim;  Ind : CoordSet;  n : BDim);
```

Assumes that `MasterGen` has been initialized to a proper set of generating vectors. Adds a vector to the basis `TBas` that corresponds to adding the jth coordinate of the generating vectors, and updates the dual.

```
PROCEDURE RemoveFromBasis (T : TestConfig; d : BDim;  Ind : CoordSet; n : BDim);
```

Removes from the basis `TBas` the vector that corresponds to the jth coordinate, and updates the dual.

```
PROCEDURE TestLattice (T : TestConfig);
```

Applies the tests specified by `T` to the lattice whose set of generating vectors is in its field `MasterGen`.

```
END TESTLAT.
```

REDBAS

This module offers tools to reduce a basis in various ways, and to find a shortest vector in a lattice using pre-reductions and a branch-and-bound (BB) procedure. It also has a procedure to compute a Minkowski-reduced basis (this is much more expensive than finding a shortest vector). One must first call `InitREDBAS` before calling any other procedure in this module.

```
DEFINITION MODULE REDBAS;

IMPORT BASISLR;
FROM LATBASIS IMPORT Basis;
FROM CONFIG   IMPORT BDim;

TYPE
   VerifyType  = (Verify0, Verify2, Verify3);
```

Type of verification performed by the tests: uses `REDBAS`, `REDBAS2`, or `REDBAS3` for `Verify0`, `Verify2`, and `Verify3`, respectively.

```
VAR
   PreRedDieterSV, PreRedLLLSV, PreRedLLLRM : BOOLEAN;
```

These boolean variables indicate which type of pre-reduction is to be performed for `ShortestVector` (SV) and for `ReductMinkowski` (RM). `Dieter` means the pairwise pre-reduction as in the procedure `PreRedDieter`. `LLL` means the LLL reduction of Lenstra, Lenstra, and Lovász.

The variable `PreRedDieterSV` is originally set to `TRUE` and the 2 others are originally set to `FALSE`. These variables are reset automatically depending on the thresholds `MinkLLL, ShortDiet, ShortLLL` as explained below. Note that the LLL reduction is never performed when `SISquares = TRUE`. [1]

```
   ShortDiet : LONGINT;
```

Whenever the number of nodes in the BB tree exceeds the threshold `ShortDiet`, in the `ShortestVector` procedure, `PreRedDieterSV` is automatically set to `TRUE` for the next call; otherwise it is set to `FALSE`. The default value is 1000.

```
   ShortLLL : LONGINT;
```

Whenever the number of nodes in the BB tree exceeds the threshold `ShortLLL`, in the `ShortestVector` procedure, `PreRedLLLSV` is automatically set to `TRUE` for the next call; otherwise it is set to `FALSE`. The default value is 1000.

```
   MinkLLL : LONGINT;
```

Whenever the number of nodes in the BB tree exceeds `MinkLLL` in the `ReductMinkowski` procedure, `PreRedLLLRM` is automatically set to `TRUE` for the next call; otherwise it is set to `FALSE`. The default value is 500 000.

```
   MaxPreRed : LONGINT;
```

Maximum number of transformations in the procedure `PreRedDieter`. After `MaxPreRed` successful transformations have been performed, the prereduction is stopped. The default value is 1 000 000.

[1] From Pierre: Reason?

`MaxNodesBB : LONGINT;`

The maximum number of nodes that we are ready to accept in the branch-and-bound (BB) tree when calling `ShortestVector` or `ReductMinkowski`. When this number is exceeded, the procedure aborts and returns `FALSE`. The default value is 10 000 000.

`PROCEDURE InitREDBAS (MaxDim : BDim);`

Initializes the internal variables of this module with a given upper dimension `MaxDim` for the bases, for memory allocation purposes. This `MaxDim` (saved in the internal variable `CurMaxDim`) should be at least as large as the dimension of any `Basis B` that this module has to work on (i.e., `B^.Dim` can never exceed `CurMaxDim`).

`PROCEDURE DeleteREDBAS;`

Deallocates the memory which was allocated when calling `InitREDBAS` for the last time. Must be called only if `InitREDBAS` has been called previously.

`PROCEDURE GetMaxDim (): BDim;`

Returns the current value of `CurMaxDim`.

`PROCEDURE PairwiseRedPrimal (B : Basis;  i, d : BDim);`

Performs pairwise reductions. This procedure tries to reduce each basis vector with index larger than d, and distinct from i, by adding to it a multiple of the i-th vector. Always use the Euclidean norm.

`PROCEDURE PairwiseRedDual (B : Basis;  i : BDim);`

Performs pairwise reductions, trying to reduce every other vector of the *dual* basis by adding multiples of the i-th vector. That may change the i-th vector in the primal basis. Each such dual reduction is actually performed only if that does not increase the length of vector i in the primal basis. Always use the Euclidean norm.

`PROCEDURE PreRedDieter (B : Basis;  d : BDim);`

Performs the reductions of the preceding two procedures using cyclically all values of i (only for $i > d$ in the latter case) and stops after either `MaxPreRed` successful transformations have been achieved or no further reduction is possible. Always use the Euclidean norm.

`PROCEDURE SetScale (B : Basis);`

This procedure is used when `SISquares = TRUE`, just before doing the Choleski decomposition for the branch-and-bound procedure, in the procedures `ShortestVector` and `ReductMinkowski`. It rescales the square Euclidean lengths of the basis and m-dual basis vectors by factors `EchVV` and `EchWW`, respectively, and m^2 by the product `EchVV * EchWW`. These factors are chosen by the procedure so that the log of the largest squared Euclidean length is not too far from 0. The rescaled values are placed in `VVLR` and `WWLR`, and the rescaled value of m^2 in `mLR2`. The Choleski decomposition and the branch-and-bound procedures use these rescaled values to avoid `LONGREAL` overflow.

`PROCEDURE RedLLL (B : Basis;  fact : LONGREAL;  m : LONGINT;  Max : BDim);`

Performs a LLL basis reduction with coefficient `fact`, which must be smaller than one. If `fact` is closer to one, the basis will be (typically) "more reduced", but that will require (slightly) more work. Always use the Euclidean norm. *Danger: With the current implementation, there could be* `LONGREAL` *overflow when* `SISquares = TRUE` *and there are very long basis vectors.*

```
PROCEDURE CalculCholeski (B : Basis;  VAR DC2 : BASISLR.BVect;
                          VAR C0 : BASISLR.BMatr) : BOOLEAN;
```

Performs a Choleski decomposition of the matrix of scalar products of the basis vectors, i.e., $\mathbf{U}'\mathbf{U} = \mathbf{V}'\mathbf{V}$ where $\mathbf{V}$ is the matrix of the basis vectors and $\mathbf{U}$ is upper triangular. Returns in `C0` the elements of $\mathbf{U}$ that are *strictly above* the diagonal, i.e., $u_{i,j}$ for $j > i$. Returns in `DC2` the squared elements of the diagonal of $\mathbf{U}$. If `SISquares = TRUE`, these elements are rescaled (divided) by `EchVV`. Returns `TRUE` if the decomposition was successful.

```
PROCEDURE TransformStage3 (B : Basis;  VAR z : BASISLR.VectLI;  VAR k : BDim);
```

Procedure used in `ReductMinkowski` to perform a transformation of stage 3 described in [1]. Also used in `ShortestVector`. Assumes that $\sum_{i=1}^{t} z_i V_i$ is a short vector that will enter the basis. Tries to reduce some vectors by looking for indices $i < j$ such that $|z_j| > 1$ and $q = \lfloor z_i/z_j \rfloor \neq 0$, and adding qV_i to V_j when this happens. Returns in k the last index j such that $|z_j| = 1$.

```
PROCEDURE ShortestVector (B : Basis) : BOOLEAN;
```

Computes the shortest vector in lattice with basis `B`, using branch-and-bound. If `MaxNodesBB` is exceeded during *one* of the branch-and-bounds, the procedure aborts and returns `FALSE`. Otherwise, it returns `TRUE`.

```
PROCEDURE ReductMinkowski (B : Basis;  d : BDim) : BOOLEAN;
```

Reduces `B` into a Minkowski reduced basis, with respect to the Euclidean norm, assuming that the first d vectors are already reduced and sorted. If `MaxNodesBB` is exceeded during *one* of the branch-and-bounds, the procedure aborts and returns `FALSE`. Otherwise, it returns `TRUE`, and the basis is reduced and is sorted by vector lengths (the shortest vector is `V[1]` and the longest is `V[Dim]`). This procedure does not care about numerical imprecision due to the (64-bit) floating-point representation. In this sense, the results are not 100% reliable.

```
END REDBAS.
```

REDBAS2

This module offers tools to reduce a basis in the sense of Minkowski and find a shortest vector in a lattice. The difference between this and `REDBAS` is that here, bounds on the numerical errors are computed to make sure that the results are formally correct.

```
DEFINITION MODULE REDBAS2;

FROM CONFIG   IMPORT BDim;
FROM LATBASIS IMPORT Basis;

PROCEDURE InitREDBAS2 (d : BDim);
```

Initializes the module. The value of `d` should be at least as large as the dimension of the largest basis to be considered by the procedures below.

```
PROCEDURE VerifMinkowski (B : Basis; d : BDim) : BOOLEAN;
```

Performs a formal verification and returns `TRUE` if the basis `B` is really reduced. Assumes that the first `d` vectors are really reduced and verifies only the remaining ones.

```
PROCEDURE ReductVerifMinkowski (B : Basis) : BOOLEAN;
```

Performs a reduction as in `ReductMinkowski`, then verifies it formally, and if the basis is not really reduced, completes the reduction. This is usually much more costly than `ReductMinkowski`, but the result is perfectly reliable.

```
PROCEDURE VerifShortest (B : Basis) : BOOLEAN;
```

Performs a formal verification and returns `TRUE` if the first vector of basis `B` is the shortest vector in the lattice generated by `B`.

```
PROCEDURE ShortestVectorVerif (B : Basis) : BOOLEAN;
```

Computes the shortest vector as in `ShortestVector`, performs a formal verification, and if the vector was not really the shortest due to numerical (floating-point) imprecisions, computes the shortest one. Returns `TRUE` if the procedure succeeds. This is usually much more costly than `ShortestVector`, but the result is perfectly reliable.

```
END REDBAS2.
```

REDBAS3

This module offers tools similar to those of REDBAS2, except that the "provably correct" results are obtained by a different method than in REDBAS2, bypassing all floating-point calculations, as explained in [6].

```
DEFINITION MODULE REDBAS3;

FROM CONFIG   IMPORT BDim;
FROM LATBASIS IMPORT Basis;
FROM BASISLR  IMPORT VectLI;

PROCEDURE VerifMinkowski (B : Basis;  d : BDim) : BOOLEAN;
```

Performs a formal verification and returns TRUE if the basis B is really reduced.

```
PROCEDURE VerifShortest (B : Basis; d1, d2 : BDim; VAR z : VectLI) : BOOLEAN;
```

Performs a formal verification and returns TRUE if the first vector of basis B is the shortest vector in the lattice generated by the first d1 vectors of B. The parameter d2 is a threshold, which should be between 1 and the dimension of the lattice, and may affect the performance. To compute the relevant coefficients (expressed as determinants in the primal or dual basis), we use the primal basis up to dimension d2, and then the dual basis. When the procedure returns FALSE, then the coefficients of a shorter vector with respect to the basis are returned in z.

```
END REDBAS3.
```

LATBASIS

This module offers tools to manipulate lattice bases (see Section 1.3). Each lattice is represented by a basis V and its dual W. It is sometimes possible, as in the case with lattices associated with LCGs or MRGs, to multiply a lattice (and its dual) by a constant factor in such a way that they are included in $\mathbb{Z}^t$, allowing exact representation of basis vector coordinates. The duality relation will now read $V_i \cdot W_j = m\delta_{ij}$ for some integer constant m.

The squared lengths of the basis vectors and that of their duals are stored as `LONGREAL` in `VVLR` and `WWLR` respectively. However, if they are too large to fit in a `LONGREAL`, even approximately, then `SISquares` is set to `TRUE` and they are stored as `SuperInteger` in `VVSI` and `WWSI`.

A variable of type `Basis` is created by calling `CreateBasis`. The squared vector lengths `VVLR` and `WWLR` (or `VVSI` and `WWSI`) can then be updated by calling `UpdateVVWW`. But it is not necessary to update them before calling any of the procedures of the modules `REDBAS` and `REDBAS2` (including `ReductMinkowski` and `ShortestVector`).

```
DEFINITION MODULE LATBASIS;

FROM SUPINT  IMPORT SuperInteger;

IMPORT BASISLR, BASISSI;
FROM BASIS    IMPORT BMatr, BScal;
FROM CONFIG   IMPORT BDim;
FROM NORM     IMPORT NormType;
FROM MULT     IMPORT MScal;

TYPE
   Basis = POINTER TO InfoBasis;
   InfoBasis = RECORD
      ResDim     : BDim;             (* Dimension of allocated memory.      *)
      Dim        : BDim;             (* Actual dimension.                   *)
      mLR        : LONGREAL;         (* Modulus m.                          *)
      mSI        : SuperInteger;     (* Modulus m.                          *)
      mLR2       : LONGREAL;         (* m^2, rescaled if SISquares.         *)
      mSI2       : SuperInteger;     (* Value of m^2 when SISquares.        *)
      V, W       : BMatr;            (* Basis and dual basis vectors.       *)
      Norm       : NormType;
      Lmin, Lmin2  : LONGREAL;       (* Length and squared length of shor-  *)
                                     (* test basis vector, using Norm.      *)
                                     (* Different meaning in REDBB.         *)
      SISquares    : BOOLEAN;
      VVLR, WWLR   : BASISLR.BVect;  (* Squared Euclidean vector lengths,   *)
                                     (* if not SISquares.                   *)
      EchVV, EchWW : SuperInteger;   (* Scaling factors when SISquares.     *)
      VVSI, WWSI : BASISSI.BVect;    (* Rescaled squared Euclidean vector   *)
                                     (* lengths, if SISquares.              *)
      XX : ARRAY BDim OF BOOLEAN;    (* Used for ReductMinkowski only.      *)
                                     (* Ask R. Couture for details.         *)
      END;
```

A lattice basis. `ResDim` is the dimension of the arrays allocated for the basis (the maximum value that `Dim` can take). The memory for the `BVect` and `BMatr` elements is allocated for dimensions 0 to `ResDim` if `ResDim` > 0, whereas no memory is allocated if `ResDim` $= 0$.

`Dim` is the current dimension of the basis, `V` contains the basis, and `W` the m-dual basis. The integer constant m used in the duality relation is stored as a `LONGREAL` in `mLR`, and as a `SuperInteger` in `mSI`.

`Norm` is the type of norm used to measure the vector lengths. `Lmin` and `Lmin2` are used to store the length and the squared length of the shortest vector found to date, using this norm. (Usually, only one of these 2 variables is used at a time; see `REDBAS`.)

The boolean `SISquares` is set to `TRUE` by `InitBasis` when m exceeds approximately 2^{480}, and to `FALSE` otherwise.

When `SISquares = FALSE`, `mLR2` contains the value of m^2, `VVLR` and `WWLR` *may* memorize the squared Euclidean lengths of the vectors in V and in W. They are not always updated, and when they are not, they are set to negative values.

When `SISquares = TRUE`, `mSI2` contains m^2, `mLR2` contains m^2 divided (rescaled) by `EchVV * EchWW`, the squared Euclidean vector lengths (again, not always updated), are stored in `VVSI` and `WWSI`, and `Lmin2` is rescaled by `EchVV`. Also, the squared vector lengths rescaled by the factors `EchVV` and `EchWW`, are put in `VVLR` and `WWLR`, respectively, by `SetScale`. The scaling factors `EchVV` and `EchWW` are computed by the procedure `SetScale` and set to values such that the log of the squared Euclidean length of the longest vector is not too far from 0. The aim of this rescaling is to avoid `LONGREAL` overflow.

```
PROCEDURE CreateBasis (VAR B : Basis);
```

Creates the data structure `B` for a basis. `SISquares` is initialized to `FALSE`, `Norm` is initialized to `L2Norm`, and `ResDim` is set to 0. One must then call `InitBasis` to reserve the memory for the `BVect` and `BMatr` elements.

```
PROCEDURE InitBasis (B : Basis;  m : MScal;  Nor : NormType;  d : BDim);
```

Initializes the modulus and the norm for the basis `B`. Reserves the memory for the `BVect` and `BMatr` fields of `B`, for indices 0 to d if $d > 0$. Here, `d` is the new value of `ResDim`. If `d < B^.ResDim`, then the memory already reserved which is no longer needed is deallocated.

```
PROCEDURE DeleteBasis (B : Basis);
```

Does the same as `InitBasis (B, B^.m, B^.Norm, 0)`, and then the opposite of `CreateBasis (B)`.

```
PROCEDURE WriteBasis (B : Basis);
```

Writes content of `B` in current output file.

```
PROCEDURE ReadBasis (VAR B : Basis);
```

Reads a basis from current input file. The basis must have been written to the file by `WriteBasis`.

```
PROCEDURE CopyBasis (B1, B2 : Basis);
```

Copies `B1` into `B2`.

```
PROCEDURE Permute (B : Basis;  i, j : BDim);
```

Exchanges vectors i and j in the basis and its dual.

```
PROCEDURE SortBasis (B : Basis;  d : BDim);
```

Sorts the basis vectors (with index from $d + 1$ to `B^.Dim`) by increasing length. The dual vectors are permuted accordingly. Assumes that `VVLR` and `WWLR` (or `VVSI` and `WWSI`, if `SISquares`) are up to date.

```
PROCEDURE Dualize (B : Basis);
```

Exchanges `V, VVLR, VVSI, EchVV` and `W, WWLR, WWSI, EchWW`.

```
PROCEDURE UpdateVVWW (B : Basis;  d : BDim);
```

Updates `VVLR` and `WWLR` (or `VVSI` and `WWSI` if `SISquares`) as the square Euclidean lengths of the vectors in `V` and `W` (for vectors with index from $d+1$ to `B^.Dim`).

```
PROCEDURE EraseVVWW (B : Basis);
```

Sets the vectors `VVLR` and `WWLR` (or `VVSI` and `WWSI` if `SISquares`) to negative values.

```
PROCEDURE CheckDuality (B : Basis): BOOLEAN;
```

Check that `Basis B` satisfies the duality relation $V[i]\cdot W[j] = m\,\delta_{ij}$. If so, returns `TRUE`, otherwise returns `FALSE`.

```
PROCEDURE BaseEquivalence (B1, B2 : Basis) : BOOLEAN;
```

Check that `Basis B1` and `B2` are equivalent. If so, returns `TRUE`, otherwise returns `FALSE`.

```
PROCEDURE RandomLattice (B : Basis;   m : BScal;   d : BDim);
```

Constructs a *random* basis for an integral lattice, chosen with the uniform distribution over the *finite* set of all lattices whose fundamental volume is m^{d-1} and which contains $m\mathbb{Z}^d$ as a sublattice, for the specified values of m and d.

```
END LATBASIS.
```

PRIM

This module provides some procedures to search for integers m that are prime, for which $r = (m^k - 1)/(m - 1)$ is also prime for certain values of k, and perhaps for which $(m - 1)/2$ is also prime. The values of r are always represented as `SuperInteger`'s.

```
DEFINITION MODULE PRIM;

FROM SUPINT   IMPORT SuperInteger;
FROM SUPFACT  IMPORT Factors;


PROCEDURE Trier (VAR L : Factors);
```

Sorts the factors in the list `L` in *decreasing* order.

```
PROCEDURE SFindPrimer (k, e : LONGINT;  S1, S2 : SuperInteger;
                       Safe : BOOLEAN);
```

Finds and prints all integers m such that $S_1 \le m \le S_2$ and such that both m and $r = (m^k - 1)/(m - 1)$ are prime. If `Safe = TRUE`, it is also required that $(m - 1)/2$ be prime. If $k = 1$, then r is considered prime. The retained m are printed in decreasing order, in the form $m = 2^e \pm x$ and also as decimal integers. The factors of $m - 1$ are also printed.

```
PROCEDURE FindPrimer (k, m0, m1 : LONGINT;  Safe : BOOLEAN);
```

Similar to `SFindPrimemr`, but for values of m less than 2^{31}.

```
PROCEDURE FindPrimesmr
      (k, e, c1, c2 : LONGINT;  Safe : BOOLEAN;   F : ARRAY OF CHAR);
```

Calls `SFindPrimer` for $S_1 = 2^e + c_1$ and $S_2 = 2^e + c_2$, creates the file `F` and prints the results in it.

```
END PRIM.
```

APPENDIX C

Lower-Level Modules

NORM

This module provides tools to define normalization constants for the shortest vectors.

```
DEFINITION MODULE NORM;

IMPORT BASISLR;
FROM CONFIG IMPORT BDim, MDim;
FROM MULT   IMPORT MScal;

TYPE
   CriterionType = (Spectral, SpectralP, SpectralL1, Beyer, BeyerSpectral,
                    Palpha);
```

Types of merit criteria for lattices. `Spectral` means that the criterion is based on the shortest vector in the *dual* lattice. In this case, one must specify which approximation is used for γ_t (see `GammaType` below). `SpectralP` is similar to `Spectral`, except that it is based on the shortest vector in the *primal* lattice. `SpectralL1` is similar to `Spectral`, except that the length of the (dual) vectors are measured with the L_1 norm, minus 1. The length of the shortest dual vector is then an upper bound on the minimal number of hyperplanes that cover all the points ([9] and [21], Exercises 3.3.4-15 and 16). `Beyer` means that the criterion is the Beyer quotient q_t. `BeyerSpectral` means that both `Beyer` and `Spectral` apply simultaneously. `Palpha` means that the criterion is P_α. In this case, one must specify a value of α and weights may also be used (see the module `PALPHA`).

```
   NormType = (L1Norm, L2Norm, SupNorm);
```

Type of norm used to measure the vector lengths. For $V = (v_1, \ldots, v_t)$, the L_1 norm is $\|V\| = |v_1| + \cdots + |v_t|$, the L_2 (or Euclidean) norm is $\|V\| = (v_1^2 + \cdots + v_t^2)^{1/2}$, and the sup norm is $\|V\| = \max(|v_1|, \ldots, |v_t|)$.

```
   GammaType = (BestLat, Laminated, Minkowski, Rogers, MinkL1);
```

Types of bound on γ_t which can be used to normalize the length of the shortest vector. The normalized value is the figure of merit used as a selection criterion. `BestLat, Laminated, Minkowski, Rogers` correspond respectively to the bounds γ_t^{L}, γ_t^{B}, γ_t^{Z}, and γ_t^{R}, for the Euclidean norm. `MinkL1` must be used for the $\mathcal{L}_1$ norm and corresponds to $\gamma_t^{\mathrm{M}} = (t!)^{1/t}$.

```
   Normaliz = POINTER TO InfoNormaliz;
   InfoNormaliz = RECORD
      Name  : ARRAY [0..31] OF CHAR;
      Norm  : NormType;
      GType : GammaType;
      Beta  : LONGREAL;
      Maxt  : BDim;
      Gamma : BASISLR.BVect;
      Cst   : BASISLR.BVect;
      END;
```

Data structure used to store a vector of values that are used to normalize the lengths ℓ_t of the shortest vectors. The array `Gamma` contains values of $\tilde{\gamma}_t$ that are approximations of γ_t which appears in the bounds given in Section 1. The normalized figure of merit will be $\ell_t/\bar{\ell}_t$, where

$$\bar{\ell}_t = \begin{cases} \tilde{\gamma}_t m \beta^t & \text{if } t \le k; \\ \tilde{\gamma}_t m^{k/t} \beta^t & \text{if } t > k, \end{cases} \tag{29}$$

where β is an additional multiplicative factor stored in the variable `Beta` (often equal to 1). The vector `Cst` contains the values of $\bar{\ell}_t^2$ in case of the Euclidean norm, and $\bar{\ell}_t$ otherwise.

The structure must be created by the procedure `CreateNormaliz`, initialized by `InitNormaliz`, and deleted by `DeleteNormaliz`. The field `Name` is a character string that describes the type of $\tilde{\gamma}_t$. `Norm` is the type of norm that is used to measure the vector lengths, `GType` indicates the type of normalization constants $\tilde{\gamma}_t$, `Beta` is the weighting factor used in (29), `Maxt` is the maximum index for which the arrays have been initialized, i.e., the largest value of t that can be considered.

```
PROCEDURE CreateNormaliz (VAR G : Normaliz);
```

Creates the structure `G` without initializing anything in it.

```
PROCEDURE InitNormaliz
   (   G  : Normaliz;
    Name  : ARRAY OF CHAR;
    Norm  : NormType;
    GType : GammaType;
    Beta  : LONGREAL;
    Maxt  : BDim;
       m  : MScal;
       k  : MDim
   );
```

Initializes the structure `G` so that `Gamma[t]` contains the approximation of γ_t that corresponds to `T`, for $1 \leq t \leq$ `Maxt`. `Cst[t]` will contain the value of $\bar{\ell}_t^2$ in case of the Euclidean norm (`Norm = L2Norm`), and $\bar{\ell}_t$ otherwise, where $\bar{\ell}_t$ is defined in (29), $\tilde{\gamma}_t$ is determined by `GType`, and β is the value of `Beta`.

Normally, this procedure is called with $\beta = 1$. Taking $\beta < 1$ inflates the figure of merit by $(1/\beta)^t$, thus weakening the requirements for large t in a worst-case figure of merit such as (9).

The constant `Maxt` cannot exceed the maximum value of t for which the approximation of γ_t has been pre-computed and is available (this depends on the type `GType` of $\tilde{\gamma}_t$).

```
PROCEDURE DeleteNormaliz (VAR G : Normaliz);
```

Deletes the structure `G`.

```
PROCEDURE WriteNormaliz (G : Normaliz);
```

Writes the information contained in `G`.

```
END NORM.
```

CONFIG

Here, we define constants, elementary types, and configuration variables that are used throughout the package.

```
DEFINITION MODULE CONFIG;

CONST
   MMaxDim = 16;
```

Maximum number of dimensions of arrays and matrices in module MULT. Maximal order of a multiple recursive generator.

```
   BMaxDim = 48;
```

Maximum number of dimensions for a lattice basis (module BASIS).

```
   MaxCoord = 64;
```

Maximum total number of coordinates that can be considered when constructing the generating vectors in module TESTLAT.

```
   MaxDimProj = 8;
```

Maximum number of dimensions for the projections over non-successive indices, that can be considered by the module TESTLAT.

```
   MaxCat = 5;
```

Maximum number of categories into which the set of dimensions can be partitioned.

```
   MaxNbGen = 256;
```

Maximum number of generators that can be stored in a list (in module SEARCH).

```
   MaxJ = 5;
```

Maximum number of components J for a combined generator.

```
TYPE
   MDim = [0..MMaxDim];
```

A dimension or coordinate in module MULT.

```
   BDim = [0..BMaxDim];
```

A dimension or coordinate for a lattice basis.

```
   MCoord = [0..MaxCoord];
```

A coordinate number that can be considered when constructing the generating vectors, and for the projections.

```
   DimProj = [0..MaxDimProj];
```

Dimension of a projection.

`Categ = [0..MaxCat];`

A category of generators.

`Indexj = [1..MaxJ];`

Index of a component j of a combined generator.

`OutputType = (None, CurOut, CurOutTex);`

Kind of output to be produced. `None` means that nothing is written to the current output. `CurOut` and `CurOutTex` mean that the results will be written to the current default output. The difference is that for the latter, they are written in LaTeX format.

`END CONFIG.`

MULT

This can be either `MULTLI` or `MULTSI`, depending on the chosen representation. One can copy one of these two in `MULT` (i.e. copy the "`.def`" and the "`.mod`" for one of them) and change the module name (in the code) to `MULT`. This can also be done automatically using the command `select`. This module provides tools for manipulating multipliers before and during the construction of lattice basis (e.g., checking if the period is maximal, etc.).

BASIS

This can be either `BASISLR` or `BASISSI`, depending on the chosen representation. One can copy one of these two in `BASIS` (i.e. copy the "`.def`" and the "`.mod`" for one of them) and change the module name (in the code) to `BASIS`. This can also be done automatically using the command `select`. This module provides tools for manipulating the vectors of a lattice basis (reduction, etc.) after its construction.

CONVERT

This can be either `LILR`, `SILR`, `SISI` or `LISI`, depending the chosen representation. One can copy one of these four in `CONVERT` (i.e. copy the "`.def`" and the "`.mod`" for one of them) and change the module name (in the code) to `CONVERT`. This can also be done automatically using the command `select`.

MULTLI

This module offers basic tools to manipulate scalars, vectors and matrices used as multipliers for linear congruential (or multiple recursive) generators. They are represented using the `LONGINT` type. Before calling any of the procedures with suffix `Modm`, one must first call `MSetm` to fix the value of the modulus `m`. `MaxPeriod` and `PrimElem` can be used to verify the maximal period conditions for prime moduli given in Knuth [20]. Before calling these procedures, one must first call `SetFactors` to give or compute the decomposition of $m-1$ and r.

```
DEFINITION MODULE MULTLI;

FROM SUPINT  IMPORT SuperInteger;
FROM SUPFACT IMPORT Factors;
FROM CONFIG  IMPORT MDim;

TYPE
   MScal   = LONGINT;
   MVect   = ARRAY MDim OF MScal;
   MMatr   = ARRAY MDim OF MVect;

VAR
   MZero, MOne, MTwo : MScal;            (* Constants 0, 1, and 2.       *)

PROCEDURE MCreateScal (VAR S : MScal);
```

Creates `S`. (Is there for compatibility with `MULTSI`. Does nothing in the case of `MULTLI`.)

```
PROCEDURE MDeleteScal (VAR S : MScal);
```

Deallocates the memory allocated to `S` by a previous call to `MCreateScal`.

```
PROCEDURE MCreateVect (VAR V : MVect; k1, k2 : MDim);
```

Creates entries of `V`, using `MCreateScal`, for indices `k1` to `k2` inclusive. If `k2` is smaller than `k1` then does nothing.

```
PROCEDURE MDeleteVect (VAR V : MVect; k1, k2 : MDim);
```

Deallocates the memory that was allocated for the creation of V, of type `MVect`, at a previous call to `MCreateVect` with indices from `k1` to `k2` inclusive.

```
PROCEDURE MCreateMatr (VAR M : MMatr; k1, k2 : MDim);
```

Creates the elements of `M`, as in `MCreateVect`.

```
PROCEDURE MDeleteMatr (VAR M : MMatr; k1, k2 : MDim);
```

Deallocates the memory that was allocated for the creation of M, of type `MMatr`, at a previous call to `MCreateMatr` with parameters `k1` and `k2`.

```
PROCEDURE MReadScal (VAR S : MScal);
```

Reads `S` from current intput file.

```
PROCEDURE MWriteScal (S : MScal);
```

Writes `S` to current output file.

```
PROCEDURE MWriteScalBin (S : MScal);
```

Writes `S` to current output file in binary form.

```
PROCEDURE MWriteVect (VAR V : MVect;  n : MDim);
```

Writes content of `V` in current output file.

```
PROCEDURE MScalToLI (S : MScal) : LONGINT;
```

Returns the `LONGINT` representation of `S`.

```
PROCEDURE MScalToLR (S : MScal) : LONGREAL;
```

Returns the `LONGREAL` representation of `S`.

```
PROCEDURE MScalToSup (S : MScal;  T : SuperInteger);
```

Converts `S` to the `SuperInteger T` (which must have been created before).

```
PROCEDURE SupToMScal (T : SuperInteger;  VAR S : MScal);
```

Converts `T` to the `MScal S` (which must have been created before).

```
PROCEDURE LIToMScal (I : LONGINT;  VAR S : MScal);
```

Converts `I` to the `MScal S`.

```
PROCEDURE MAffect (S : MScal;  VAR T : MScal);
```

Gives to `T` the value of `S`.

```
PROCEDURE MEqual (S, T : MScal): BOOLEAN;
```

Returns `TRUE` if `S` is equal to `T`, `FALSE` otherwise.

```
PROCEDURE MGreater (S1, S2 : MScal): BOOLEAN;
```

Returns `TRUE` if `S1` is greater than `S2`, `FALSE` otherwise.

```
PROCEDURE MSmaller (S1, S2 : MScal): BOOLEAN;
```

Returns `TRUE` if `S1` is smaller than `S2`, `FALSE` otherwise.

```
PROCEDURE MChangeSign (VAR S : MScal);
```

Changes the sign of S.

```
PROCEDURE MInc (VAR S : MScal; I : LONGINT);
```

Adds I to S. The result is in S.

```
PROCEDURE MAdd (S1, S2 : MScal;   VAR T : MScal);
```

Adds `S1` to `S2` and puts the results in `T`.

```
PROCEDURE MSubtract (S1, S2 : MScal;   VAR T : MScal);
```

Subtracts `S2` from `S1` and puts the results in `T`.

`PROCEDURE MAffectIdentMatr (VAR M : MMatr;  n : MDim);`

Initializes `M[1..n,1..n]` to the identity matrix.

`PROCEDURE MCopyVect (VAR U, V : MVect;  n : MDim);`

Copies vector `U`, of dimension `n`, into vector `V`.

`PROCEDURE MCopyMatr (VAR M, N : MMatr;  n : MDim);`

Copies matrix `M`, of dimension `n`×`n`, into matrix `N`.

`PROCEDURE MModifVect (VAR U,V : MVect;  R : MScal; n : MDim);`

Adds `R` times the vector `V` to the vector `U`.

`PROCEDURE MPowerLI (S : MScal;  i : LONGINT;  VAR R : MScal);`

Returns S^i in `R`.

`PROCEDURE MMultiply (T, S : MScal;  VAR R : MScal);`

Returns the product of `T` by `S` in `R`.

`PROCEDURE MQuotient (T, S : MScal;  VAR Q : MScal);`

Returns the (truncated) quotient of `T` by `S` in `Q`.

`PROCEDURE MModulo (T, S : MScal;  VAR R : MScal);`

Returns `T` modulo `S` in `R`.

`PROCEDURE MSetm (m : MScal);`

Sets the value of the modulus `m` which will be used from now on in the procedures below.

`PROCEDURE MGetm (VAR m : MScal);`

Returns in `m` the value of the current modulus.

`PROCEDURE MProdModm (S, T : MScal;  VAR R : MScal);`

Returns in `R` the product of `S` by `T` modulo `m`. Assumes that `S` and `T` are between `-m` and `m`.

`PROCEDURE MProdScalModm (VAR U, V : MVect;  VAR Q, R : MScal;  n : MDim);`

Returns in `R` the scalar product of `U` by `V` modulo `m`, using components `1` to `n` of these vectors. `Q` contains the quotient.

`PROCEDURE MModifVectModm (VAR U, V : MVect;  R : MScal;  n : MDim);`

Adds `R` times the vector `V` to the vector `U`.

`PROCEDURE MPowerPolyMod (VAR A : MVect;  n : MDim;  q : MScal;  VAR R : MVect);`

Returns in `R` the coefficients of the polynomial $x^q \bmod f(x)$ (mod m), where $f(x) = x^n - A_1 x^{n-1} - \cdots - A_n$.

`PROCEDURE SetFactors (VAR ListmMS1, Listr : Factors;  k : MDim);`

Sets the factors of $m-1$ and r [$= (m^k-1)/(m-1)$]. If `ListmMS1` is `NIL`, then the procedure will calculate and return the factors of $m-1$. If not, then the procedure will take the list of factors (see `SUPFACT` of

`SENTIERS`) and verify if they are factors of $m-1$. The program will stop if the product does not give $m-1$. The same things are done with `Listr` and the factors of r.

`PROCEDURE PrimElem (VAR A : MVect) : BOOLEAN;`

Verifies a first condition for $f(x)$ to be primitive modulo m, namely that $(-1)^{k-1}a_k$ is a primitive element modulo m. Must have called `SetFactors` before.

`PROCEDURE MaxPeriod (VAR A : MVect) : BOOLEAN;`

Verifies the remaining conditions for $f(x)$ to be primitive modulo m, assuming that `PrimElem` has returned `TRUE`.

`END MULTLI.`

MULTSI

Same as MULTLI, except that the values are represented using the SuperInteger type, and that MCreateScal, MCreateVect, and MCreateMatr are required to create the SuperIntegers. The initial memory space (in words) allocated for the digits of the SuperIntegers thus created is 2 to the power MSizeSI (currently 4 words). (See Create in SUPINT).

```
DEFINITION MODULE MULTSI;

FROM SUPINT  IMPORT SuperInteger;
FROM SUPFACT IMPORT Factors;
FROM CONFIG  IMPORT MMaxDim, MDim;

TYPE
   MScal   = SuperInteger;
   MVect   = ARRAY MDim OF MScal;
   MMatr   = ARRAY MDim OF MVect;


VAR
   MZero, MOne, MTwo : MScal;          (* These should be constants      *)


PROCEDURE MCreateScal (VAR S : MScal);
PROCEDURE MDeleteScal (VAR S : MScal);
PROCEDURE MCreateVect (VAR V : MVect; k1, k2 : MDim);
PROCEDURE MDeleteVect (VAR V : MVect; k1, k2 : MDim);
PROCEDURE MCreateMatr (VAR M : MMatr; k1, k2 : MDim);
PROCEDURE MDeleteMatr (VAR M : MMatr; k1, k2 : MDim);
PROCEDURE MReadScal (VAR S : MScal);
PROCEDURE MEqual (S, T : MScal): BOOLEAN;
PROCEDURE MGreater (S1, S2 : MScal): BOOLEAN;
PROCEDURE MSmaller (S1, S2 : MScal): BOOLEAN;
PROCEDURE MChangeSign (S : MScal);
PROCEDURE MInc (VAR S : MScal; I : LONGINT);
PROCEDURE MAdd (S1, S2 : MScal; VAR S3 : MScal);
PROCEDURE MSubtract (S1, S2 : MScal;  VAR S3 : MScal);
PROCEDURE MWriteScal (S : MScal);
PROCEDURE MWriteScalBin (S : MScal);
PROCEDURE MWriteVect (VAR V : MVect;  n : MDim);
PROCEDURE MScalToLI (S : MScal) : LONGINT;
PROCEDURE MScalToLR (S : MScal) : LONGREAL;
PROCEDURE MScalToSup (S: MScal; T : SuperInteger);
PROCEDURE LIToMScal (I : LONGINT;  VAR S : MScal);
PROCEDURE MAffect (S : MScal;  VAR T : MScal);
PROCEDURE MAffectIdentMatr (VAR M : MMatr;  n : MDim);
PROCEDURE MCopyVect (VAR U, V : MVect;  n : MDim);
PROCEDURE MCopyMatr (VAR M, N : MMatr; n : MDim);
PROCEDURE MModifVect (VAR U,V : MVect; R : MScal; n : MDim);
PROCEDURE MPowerLI (S : MScal;  i : LONGINT;  VAR R : MScal);
PROCEDURE MMultiply (T, S : MScal;   VAR  R : MScal);
PROCEDURE MQuotient (T, S : MScal;   VAR  Q : MScal);
PROCEDURE MModulo (T, S : MScal;   VAR  R : MScal);
PROCEDURE MSetm (m : MScal);
```

```
PROCEDURE MGetm (VAR m : MScal);
PROCEDURE MProdModm (S, T : MScal;  VAR R : MScal);
PROCEDURE MProdScalModm (VAR U, V : MVect;  VAR a, R : MScal;   n : MDim);
PROCEDURE MModifVectModm (VAR U,V : MVect; R : MScal; n : MDim);
PROCEDURE MPowerPolyMod (VAR A : MVect; n : MDim; q : MScal; VAR R : MVect);
PROCEDURE SetFactors (VAR ListmMS1, Listr : Factors; k : MDim);
PROCEDURE PrimElem (VAR A : MVect) : BOOLEAN;
PROCEDURE MaxPeriod (VAR A : MVect) : BOOLEAN;
END MULTSI.
```

BASISLR

This module offers basic tools to manipulate scalars, vectors, and matrices that are used to represent lattice bases, using the LONGREAL type. It is used by the modules LATBASIS, REDBAS, etc., when this representation type is used. See that module for more details.

```
DEFINITION MODULE BASISLR;

FROM SUPINT   IMPORT SuperInteger;
FROM CONFIG   IMPORT BMaxDim, BDim;

TYPE
   BScal   = LONGREAL;
   BVect   = ARRAY BDim OF BScal;
   BMatr   = ARRAY BDim OF BVect;
   VectLI  = ARRAY BDim OF LONGINT;

VAR
   BZero, BOne, BMinusOne : BScal;    (* 0, 1, and -1.                    *)

PROCEDURE BCreateScal (VAR S : BScal);

PROCEDURE BDeleteScal (VAR S : BScal);

PROCEDURE BCreateVect (VAR V : BVect; d1, d2 : BDim);

PROCEDURE BDeleteVect (VAR V : BVect; d1, d2 : BDim);

PROCEDURE BCreateMatr (VAR M : BMatr; d1, d2 : BDim);

PROCEDURE BDeleteMatr (VAR M : BMatr; d1, d2 : BDim);
```

These 6 procedures do nothing. There are there only for compatibility with the corresponding procedures in BASISSI.

```
PROCEDURE BReadScal (VAR S : BScal);

PROCEDURE BWriteScal (S : BScal;   nb : CARDINAL);
```

Writes content of S in current output file. Uses a total of at least nb positions.

```
PROCEDURE BWriteVect (VAR V : BVect;  n : BDim;  nb : CARDINAL);
```

Writes content of V for indices [1..n] in current output file. Uses a total of at least nb positions for each number.

```
PROCEDURE BWriteMatr (VAR M : BMatr;  n : BDim; nb : CARDINAL);
```

Writes content of M for indices [1..n][1..n] in current output file. Uses a total of at least nb positions for each number.

```
PROCEDURE BProdScalVect (s : BScal;  VAR V : BVect;  d : BDim);
```

Multiplies vector V by scalar s.

```
PROCEDURE BProdScal (VAR U, V : BVect;  n : BDim;  VAR S : BScal);
```

Returns in S the scalar product of U by V, using components from 1 to n.

```
PROCEDURE BProdScalLR (VAR U, V : BVect;  n : BDim) : LONGREAL;
```

Returns the scalar product of `U` by `V`, using components from 1 to `n`.

```
PROCEDURE BProdScalSI (VAR U, V : BVect; n : BDim; Res : SuperInteger);
```

Returns in `S` the scalar product of `U` by `V`, using components from 1 to `n`. `U` and `V` are rounded and converted into `SuperInteger` representation before the computation of the scalar product.

```
PROCEDURE BL1NormLR (VAR V : BVect;  n : BDim) : LONGREAL;
```

Returns the L_1 norm of `V`, using components from 1 to `n`.

```
PROCEDURE BL2NormLR (VAR V : BVect;  n : BDim) : LONGREAL;
```

Returns the L_2 norm of `V`, using components from 1 to `n`.

```
PROCEDURE BL1NormSI (VAR V : BVect;  n : BDim;  S1 : SuperInteger);
```

Returns in `S1` the L_1 norm of `V`, using components from 1 to `n`.

```
PROCEDURE BSupNormLR (VAR V : BVect;  n : BDim) : LONGREAL;
```

Returns the sup norm of `V`, using components from 1 to `n`.

```
PROCEDURE BSupNormSI (VAR V : BVect;  n : BDim;  S1 : SuperInteger);
```

Returns in `S1` the sup norm of `V`, using components from 1 to `n`.

```
PROCEDURE BModifVectLR (VAR U, V : BVect;  c : LONGREAL;  n : BDim);
```

Adds `c` times the vector `V` to the vector `U`. Assumes that both vectors have dimension `n`.

```
PROCEDURE BModifVectLI (VAR U, V : BVect;  c : LONGINT;  n : BDim);
```

Adds `c` times the vector `V` to the vector `U`. Assumes that both vectors have dimension `n`.

```
PROCEDURE BModifVect (VAR U, V : BVect;  c : BScal;  n : BDim);
```

Adds `c` times the vector `V` to the vector `U`. Assumes that both vectors have dimension `n`.

```
PROCEDURE BChangeSign (VAR S : BScal);
```

Changes the sign of S.

```
PROCEDURE BChangeSignVect (VAR V : BVect;  n : BDim);
```

Changes the sign of elements `1` to `n` of vector `V`.

```
PROCEDURE BAffect (S : BScal;  VAR T : BScal);
```

Gives to `T` the value of `S`.

```
PROCEDURE BExchange (VAR S, T : BScal);
```

```
PROCEDURE BEqual (S, T : BScal): BOOLEAN;
```

Returns `TRUE` if `S` is equal to `T`, `FALSE` otherwise.

`PROCEDURE BSmaller (S, T : BScal): BOOLEAN;`

Returns `TRUE` if `S` is smaller than `T`, `FALSE` otherwise.

`PROCEDURE BGreater (S, T : BScal): BOOLEAN;`

Returns `TRUE` if `S` is greater than `T`, `FALSE` otherwise.

`PROCEDURE BAbsSizeOrd (S, T : BScal): INTEGER;`

Returns -1 (resp. 0, +1) according as `S` $>$ (resp. $=, <$) `T`.

`PROCEDURE BAbs (S : BScal;  VAR T : BScal);`

Returns in `T` the absolute value of `S`.

`PROCEDURE BSetNeg (VAR S : BScal);`

Changes the sign of `S` if it is not negative.

`PROCEDURE BInc (VAR S : BScal; i : LONGINT);`

Adds `i` to `S`.

`PROCEDURE BAdd (S, T : BScal;  VAR Res : BScal);`

Returns in `Res` the sum of `S` and `T`.

`PROCEDURE BMultiply (S, T : BScal;  VAR Res : BScal);`

Returns in `Res` the product of `S` by `T`.

`PROCEDURE BDivideRound (S, T : BScal; VAR Res : BScal);`

Returns in `Res` the result of the division of `S` by `T`. The result is rounded. For results just in the middle of two integers, the value returned by the function is the integer the nearest of zero. Ex.: If $S = 5.0$ and $T = 10.0$, then the returned value is 0. If $S = -5.0$ and $T = 10.0$, then the returned value is also 0.

`PROCEDURE BCopyVect (VAR U, V : BVect;  n : BDim);`

Copies the vector `U`, of dimension `n`, into vector `V`.

`PROCEDURE BCopyMatr (VAR M1, M2 : BMatr;  n : BDim);`

Copies matrix `M1`, of dimension `n`, into matrix `M2`.

`PROCEDURE BConvLR (S : BScal): LONGREAL;`

Returns the `LONGREAL` representation of `S`. Is there only for compatibility with `BASISSI`.

`PROCEDURE BConvSI (R : BScal;  S : SuperInteger);`

Returns in `S` the `SuperInteger` representation of `R`.

`PROCEDURE LIToBScal (i : LONGINT; VAR S : BScal);`

Converts i to `S`

`PROCEDURE BAffectIdentMatr (VAR M : BMatr;  n : BDim);`

Initializes `M[1..n,1..n]` to the identity matrix.

```
PROCEDURE BTrMatr (M : BMatr;   VAR Mtr : BMatr;   d : BDim);
```

Computes the transposed matrix of `M`, and returns the result in `Mtr`.

```
PROCEDURE BCanonicalVect (VAR V : BVect;  i : BDim);
```

Produces the vector with all coordinates equal to zero with the exception of the i-th coordinate which is equal to one.

```
PROCEDURE BRandomScal (VAR s : BScal; m : BScal);
```

Produces a random integer in the interval $[0, m)$, with a uniform distribution.

```
PROCEDURE GCD2vect (VAR V : VectLI;   k, n : BDim) : LONGINT;
```

Computes and returns the greatest common divisor of the vectors `V[k],...,V[n]`.

```
PROCEDURE BEuclide (A, B : BScal;  VAR C, D, E, F, G : BScal);
```

For given A and B, returns C, D, E, F, G such that:

$$\begin{aligned} CA + DB &= G = \gcd(A, B) \\ EA + FB &= 0, \end{aligned}$$

where gcd is the greatest common denominator.

```
END BASISLR.
```

BASISSI

Same as BASISLR, except that the values are represented using the SuperInteger type.

```
DEFINITION MODULE BASISSI;

FROM SUPINT   IMPORT SuperInteger;
FROM CONFIG   IMPORT BDim;

CONST
   BSizeSI = 2;                         (* Used in BCreateScal.           *)
TYPE
   BScal   = SuperInteger;
   BVect   = ARRAY BDim OF BScal;
   BMatr   = ARRAY BDim OF BVect;

VAR
   BZero, BOne, BMinusOne : BScal;      (* 0, 1, and -1                   *)

PROCEDURE BCreateScal (VAR S : BScal);
```

Creates the SuperInteger S. The initial memory space (in words) allocated for the digits of S is 2 to the power BSizeSI (currently 4 words). (See Create in SUPINT).

```
PROCEDURE BDeleteScal (VAR S : BScal);
```

Deallocates the memory allocated for the SuperInteger S, at a previous call to BCreateScal.

```
PROCEDURE BCreateVect (VAR V : BVect; d1, d2 : BDim);
```

Creates the SuperInteger entries for indices d1 to d2 inclusive (i.e., extends V[0..d1-1] to V[0..d2]). If d2 is smaller than d1 then does nothing.

```
PROCEDURE BDeleteVect (VAR V : BVect; d1, d2 : BDim);
```

Deallocates the memory allocated to the SuperInteger elements of V[d1..d2].

```
PROCEDURE BCreateMatr (VAR M : BMatr; d1, d2 : BDim);
```

Creates the SuperInteger entries for indices d1 to d2 inclusive; i.e., extends the square matrix M[0..d1-1, 0..d1-1] to M[0..d2, 0..d2]. If $d1 > 0$, creates the missing entries in the d1 vectors already there, in addition to creating the d2 - d1 + 1 missing vectors.

```
PROCEDURE BDeleteMatr (VAR M : BMatr; d1, d2 : BDim);
```

Does exactly the opposite of BCreateMatr (M, d1, d2).

```
PROCEDURE BReadScal (VAR S : BScal);

PROCEDURE BWriteScal (S : BScal;   nb : CARDINAL);
```

Writes the contents of S to the current output. Uses nb digits per line.

```
PROCEDURE BWriteVect (VAR V : BVect;  n : BDim;  nb : CARDINAL);
```

Writes contents of V to current output. Uses nb digits per line for each number.

```
PROCEDURE BWriteMatr (VAR M : BMatr;  n : BDim;  nb : CARDINAL);
```

Writes contents of `M` to current output. Uses nb digits per line for each number.

```
PROCEDURE BProdScalVect (s : BScal;  VAR V : BVect;  d : BDim);
PROCEDURE BProdScal (VAR U, V : BVect;  n : BDim;  VAR S : BScal);
PROCEDURE BProdScalLR (VAR U, V : BVect;  n : BDim) : LONGREAL;
PROCEDURE BProdScalSI (VAR U, V : BVect; n : BDim; Res : SuperInteger);
PROCEDURE BL1NormLR (VAR V : BVect;  n : BDim) : LONGREAL;
PROCEDURE BL1NormSI (VAR V : BVect;  n : BDim;  S1 : SuperInteger);
PROCEDURE BL2NormLR (VAR V : BVect;  n : BDim) : LONGREAL;
PROCEDURE BSupNormLR (VAR V : BVect;  n : BDim) : LONGREAL;
PROCEDURE BSupNormSI (VAR V : BVect;  n : BDim;  S1 : SuperInteger);
PROCEDURE BModifVectLR (VAR U, V : BVect;  c : LONGREAL;  n : BDim);
PROCEDURE BModifVectLI (VAR U, V : BVect;  c : LONGINT;  n : BDim);
PROCEDURE BModifVect (VAR U, V : BVect;  c : BScal;  n : BDim);
PROCEDURE BChangeSign (VAR S : BScal);
PROCEDURE BChangeSignVect (VAR V : BVect;  n : BDim);
PROCEDURE BAffect (S : BScal;  VAR T : BScal);
PROCEDURE BExchange (VAR S, T : BScal);
PROCEDURE BEqual (S, T : BScal): BOOLEAN;
PROCEDURE BSmaller (S, T : BScal): BOOLEAN;
PROCEDURE BGreater (S, T : BScal): BOOLEAN;
PROCEDURE BAbsSizeOrd (S, T : BScal): INTEGER;
PROCEDURE BAbs (S : BScal; VAR T : BScal);
PROCEDURE BSetNeg (VAR S : BScal);
PROCEDURE BInc (VAR S : BScal; i : LONGINT);
PROCEDURE BAdd (S, T : BScal; VAR Res : BScal);
PROCEDURE BMultiply (S, T : BScal; VAR Res : BScal);
PROCEDURE BDivideRound (S, T : BScal; VAR Res : BScal);
PROCEDURE BCopyVect (VAR U, V : BVect;  n : BDim);
PROCEDURE BCopyMatr (VAR M1, M2 : BMatr;  n : BDim);
PROCEDURE BConvLR (S : BScal): LONGREAL;
PROCEDURE BConvSI (R : BScal; S : SuperInteger);
PROCEDURE LIToBScal (i : LONGINT; VAR S : BScal);
PROCEDURE BAffectIdentMatr (VAR M : BMatr;  n : BDim);
PROCEDURE BTrMatr (M : BMatr; VAR Mtr : BMatr; d : BDim);
PROCEDURE BCanonicalVect (VAR V : BVect; i : BDim);
PROCEDURE BRandomScal (VAR s : BScal; m : BScal);

PROCEDURE BEuclide (A, B : BScal;  VAR C, D, E, F, G : BScal);
```

For given A and B, returns C, D, E, F, G such that:

$$\begin{aligned} CA + DB &= G = \gcd(A, B) \\ EA + FB &= 0. \end{aligned}$$

where gcd is the greatest common denominator.

```
END BASISSI.
```

LILR

Procedures to convert between LONGINT (multipliers) and LONGREAL (elements of basis vectors). This is one of the four versions of the module CONVERT. The other versions are LISI, SILR, SISI. Which one is used depends on the current representation, as selected by the command select.

```
DEFINITION MODULE LILR;

FROM MULTLI  IMPORT MScal;              (* LONGINT.                    *)
FROM BASISLR IMPORT BScal;              (* LONGREAL.                   *)
FROM SUPINT  IMPORT SuperInteger;

PROCEDURE MScalToBScal (S : MScal;  VAR R : BScal);
```

Converts S to R (from LONGINT to LONGREAL).

```
PROCEDURE BScalToMScal (S : BScal;  VAR R : MScal);
```

Converts S to R (from LONGREAL to LONGINT).

```
PROCEDURE SupToBScal (S : SuperInteger;  VAR R : BScal);
```

Converts S to R (from SuperInteger to LONGREAL).

```
PROCEDURE SupToMScal (S : SuperInteger;  VAR R : MScal);
```

Converts S to R (from SuperInteger to LONGINT).

```
PROCEDURE BScalToSup (S : BScal;  R : SuperInteger);
```

Converts S to R (from LONGREAL to SuperInteger).

```
PROCEDURE MScalToSup (S : MScal;  R : SuperInteger);
```

Converts S to R (from LONGINT to SuperInteger).

```
END LILR.
```

LISI

Procedures to convert between `LONGINT` (multipliers) and `SuperInteger` (elements of basis vectors). Similar to `LILR`.

```
DEFINITION MODULE LISI;

FROM MULTLI  IMPORT MScal;              (* LONGINT.                    *)
FROM BASISSI IMPORT BScal;              (* SuperInteger.               *)
FROM SUPINT  IMPORT SuperInteger;

PROCEDURE MScalToBScal (S : MScal;  VAR R : BScal);
```

Converts `S` to `R` (from `LONGINT` to `SuperInteger`).

```
PROCEDURE BScalToMScal (S : BScal;  VAR R : MScal);
```

Converts `S` to `R` (from `SuperInteger` to `LONGINT`).

```
PROCEDURE SupToBScal (S : SuperInteger;  VAR R : BScal);
```

Copies `S` to `R`.

```
PROCEDURE SupToMScal (S : SuperInteger;  VAR R : MScal);
```

Converts `S` to `R` (from SuperInteger to `LONGINT`).

```
PROCEDURE BScalToSup (S : BScal;  R : SuperInteger);
```

Copies `S` to `R`.

```
PROCEDURE MScalToSup (S : MScal;  R : SuperInteger);
```

Converts `S` to `R` (from `LONGINT` to SuperInteger).

```
END LISI.
```

SILR

Procedures to convert between `SuperInteger` (multipliers) and `LONGREAL` (elements of basis vectors). Similar to `LILR`.

```
DEFINITION MODULE SILR;

FROM SUPINT  IMPORT SuperInteger;
FROM MULTSI  IMPORT MScal;              (* SuperInteger.                  *)
FROM BASISLR IMPORT BScal;              (* LONGREAL.                      *)

PROCEDURE MScalToBScal (S : MScal;   VAR R : BScal);
```

Converts `S` to `R` (from `SuperInteger` to `LONGREAL`).

```
PROCEDURE BScalToMScal (S : BScal;   VAR R : MScal);
```

Converts `S` to `R` (from `LONGREAL` to `SuperInteger`).

```
PROCEDURE SupToBScal (S : SuperInteger;   VAR R : BScal);
```

Converts `S` to `R` (from SuperInteger to `LONGREAL`).

```
PROCEDURE SupToMScal (S : SuperInteger;   VAR R : MScal);
```

Copies `S` to `R`.

```
PROCEDURE BScalToSup (S : BScal;   R : SuperInteger);
```

Converts `S` to `R` (from `LONGREAL` to SuperInteger).

```
PROCEDURE MScalToSup (S : MScal;   R : SuperInteger);
```

Copies `S` to `R`.

```
END SILR.
```

SISI

Procedures to convert between `SuperInteger` (multipliers) and `SuperInteger` (elements of basis vectors). Simply copies the value (equivalent to `Affect` in `SUPINT`).

```
DEFINITION MODULE SISI;

FROM MULTSI  IMPORT MScal;             (* SuperInteger.                  *)
FROM BASISSI IMPORT BScal;             (* SuperInteger.                  *)
FROM SUPINT  IMPORT SuperInteger;

PROCEDURE MScalToBScal (S : MScal;  VAR R : BScal);
```

Copies `S` to `R`.

```
PROCEDURE BScalToMScal (S : BScal;  VAR R : MScal);
```

Copies `S` to `R`.

```
PROCEDURE SupToBScal (S : SuperInteger; VAR R : BScal);
```

Copies `S` to `R`.

```
PROCEDURE SupToMScal (S : SuperInteger; VAR R : MScal);
```

Copies `S` to `R`.

```
PROCEDURE BScalToSup (S : BScal; R : SuperInteger);
```

Copies `S` to `R`.

```
PROCEDURE MScalToSup (S : MScal; R : SuperInteger);
```

Copies `S` to `R`.

```
END SISI.
```

select

This command will change automatically the representations of the moduli and multipliers in the modules `MULT`, `BASIS` and `CONVERT`. To run this program, type "`select`" xx yy, where xx is `LI` or `SI`, and yy is `LR` or `SI`. Parameters xx and yy determine the types used for multipliers and for the moduli, respectively. The program copies `MULT`xx in `MULT`, `BASIS`yy in `BASIS`, and $xxyy$ in `CONVERT`. The module names are also changed appropriately.

Acknowledgements

This work has been supported by NSERC-Canada grant # ODGP0110050 and FCAR-Québec grant # 93ER1654 to the first author. François Blouin, Anna Bragina, Ajmal Chaumun, Marco Jacques, François Paradis, and Josée Turgeon have participated in the implementation of this software package, since 1988. We thank L. Afflerbach and H. Grothe for helpful discussions in the early stage of the development, in 1989.

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, 3rd edition, 1999.

[4] R. Couture and P. L'Ecuyer. On the lattice structure of certain linear congruential sequences related to AWC/SWB generators. *Mathematics of Computation*, 62(206):798–808, 1994.

[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] H. Grothe. *Matrixgeneratoren zur Erzeugung Gleichverteilter Pseudozufallsvektoren.* Dissertation (thesis), Tech. Hochschule Darmstadt, Germany, 1988.

[15] P. M. Gruber and C. G. Lekkerkerker. *Geometry of Numbers.* North-Holland, Amsterdam, 1987.

[16] 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.

[17] 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.

[18] 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.

[19] D. Jäger. *MCS Modula-2 Cross System, User's Guide.* La Chanenche, France, 1992.

[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] C. Koç. Recurring-with-carry sequences. *Journal of Applied Probability*, 32:966–971, 1995.

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

[24] P. L'Ecuyer. Uniform random number generation. *Annals of Operations Research*, 53:77–120, 1994.

[25] P. L'Ecuyer. Combined multiple recursive random number generators. *Operations Research*, 44(5):816–822, 1996.

[26] P. L'Ecuyer. Random number generation. In Jerry Banks, editor, *Handbook of Simulation*, pages 93–137. Wiley, 1998. chapter 4.

[27] P. L'Ecuyer. Good parameters and implementations for combined multiple recursive random number generators. *Operations Research*, 47(1):159–164, 1999.

[28] P. L'Ecuyer. Tables of linear congruential generators of different sizes and good lattice structure. *Mathematics of Computation*, 68(225):249–260, 1999.

[29] 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.

[30] P. L'Ecuyer, G. Perron, and F. Blouin. Sentiers: Un logiciel modula-2 pour l'arithmétique sur les grands entiers. Technical Report DIUL-RT-8802, Computer Science Department, Laval University, 1988.

[31] P. L'Ecuyer and R. Simard. *MyLib: A Small Library of Basic Utilities in ANSI C*, 2001. Software user's guide.

[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] I. H. Sloan and S. Joe. *Lattice Methods for Multiple Integration*. Clarendon Press, Oxford, 1994.

[39] S. Tezuka, P. L'Ecuyer, and R. Couture. On the add-with-carry and subtract-with-borrow random number generators. *ACM Transactions of Modeling and Computer Simulation*, 3(4):315–331, 1993.