

*Version: October 21, 2004*

# LatMRG

## User's Guide

**A software package for 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 searches for good (or bad) generators according to different criteria. One can analyse the lattice structure of points formed by successive values in the generator's sequence, or formed by "leapfrog" values. Generators with large moduli and multipliers (e.g. numbers of many hundreds of bits), as well as combined generators, can also be analyzed. Multiply-with-carry generators can also be studied by analyzing their corresponding linear congruential generators.

( Random number generation; linear congruential generators;  
multiple recursive generators; multiply-with-carry; lattice structure; spectral test; maximal period  
)

**Contents**

<b>1</b>	<b>Introduction and overview</b>	<b>1</b>
1.1	Multiple recursive generators . . . . .	1
1.2	Maximal period . . . . .	2
1.3	Lattice structure and spectral test . . . . .	2
1.4	Multiply-with-carry . . . . .	3
1.5	Combined generators . . . . .	4
1.6	Lacunary indices . . . . .	5
1.7	Representation of large numbers . . . . .	5
1.8	Current implementation . . . . .	5
<b>2</b>	<b>Using the programs in executable form</b>	<b>6</b>
2.1	An example with the program <code>lat1</code> . . . . .	7
2.2	Examples with the programs <code>seek1</code> and <code>seeks</code> . . . . .	8
<b>3</b>	<b>Making your own programs</b>	<b>13</b>
3.1	Using the modules of <code>LatMRG</code> . . . . .	13
3.2	Lower-level modules and Changing the representation . . . . .	13
3.3	Modifying the package, recompiling, and relinking . . . . .	13
	<b>APPENDICES</b>	<b>14</b>
<b>A.</b>	<b>Programs in Executable Form</b>	<b>14</b>
	<code>maxper</code> . . . . .	15
	<code>findmk</code> . . . . .	16
	<code>lat1</code> . . . . .	17
	<code>seek1</code> . . . . .	19
	<code>lats</code> . . . . .	27
	<code>latsr</code> . . . . .	27
	<code>seeks</code> . . . . .	27
	<code>seeksr</code> . . . . .	27
	<code>streams</code> . . . . .	28
	<code>menuumrg</code> . . . . .	29
	<code>palpha</code> . . . . .	31
	<code>coeff</code> . . . . .	33

<b>B. Intermediate-Level Modules</b>	<b>34</b>
LATMRG . . . . .	35
LATIO . . . . .	36
MRG . . . . .	44
LATBASIS . . . . .	48
REDBAS . . . . .	50
REDBAS2 . . . . .	53
REDBAS3 . . . . .	54
PRIM . . . . .	55
PALPHA . . . . .	56
<b>C. Lower-Level Modules</b>	<b>60</b>
CONSTANT . . . . .	61
MULT . . . . .	61
BASIS . . . . .	61
CONVERT . . . . .	61
MULTLI . . . . .	62
MULTSI . . . . .	66
BASISLR . . . . .	68
BASISSI . . . . .	72
LILR . . . . .	74
LISI . . . . .	75
SILR . . . . .	76
SISI . . . . .	77
select . . . . .	78

# 1 Introduction and overview

LatMRG is software system implemented as a library of modules written in the Modula-2 language. It provides different tools for examining the theoretical properties of generators based on linear recursions in modular arithmetic. It offers facilities for checking if a generator has maximal period or not, to apply the lattice and spectral tests, and to perform computer searches for “good” generators, according to different criteria.

In this section, we give a quick recall of some definitions and notation, as well as a short outline of what the package does. For more details on the underlying theory and algorithms, see [6, 26] 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 the Section 2. Appendix A gives more formal 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, according, for example, to the size of the modulus  $m$  and to 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 Multiple recursive generators

Consider the linear recurrence

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

where  $m$  and  $k$  are positive integers and each  $a_i$  belongs to the set (or ring)  $\mathbb{Z}_m = \{0, 1, \dots, m-1\}$ . Let  $S$  be the set of  $k$ -dimensional vectors with components in  $\mathbb{Z}_m$ . For  $n \geq 0$ ,  $s_n = (x_n, \dots, x_{n+k-1}) \in S$  is the *state* at step  $n$ . The initial state  $s_0$  is called the *seed*. Let  $u_n = x_n/m \in [0, 1)$  be 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, 20, 21, 31].

## 1.2 Maximal period

Let  $S^*$  be the set obtained by removing the vector 0 from  $S$ . The maximal possible period for the  $s_n$ 's is the cardinality of  $S^*$ , i.e.  $\rho = m^k - 1$ . It is attained if and only if  $m$  is prime and the characteristic polynomial of (1), which is

$$f(x) = \left( x^k - \sum_{i=1}^k a_i x^{k-i} \right) \pmod{m}, \quad (2)$$

is primitive modulo  $m$ . Knuth [17] 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  $p > 2$  [17, 20].

## 1.3 Lattice structure and spectral test

Let  $t$  be any positive integer. A  $t$ -dimensional lattice is a set of the form

$$L = \left\{ \sum_{i=1}^t z_i V_i, \quad \text{each } z_i \text{ integer} \right\}$$

where  $\{V_1, \dots, V_t\}$  is a basis of  $\mathbb{R}^t$ . This basis is said to be a *lattice basis* of  $L$ . If  $m$  is any positive real number, we will say that the basis  $\{W_1, \dots, W_t\}$  satisfying  $V_i' W_j = \delta_{ij} m$  for all  $i, j$  ( $\delta_{ij}$  is the Kronecker's delta), is the  $m$ -dual of the basis  $\{V_1, \dots, V_t\}$ , and that the lattice generated by this  $m$ -dual basis, is the  $m$ -dual to  $L$ . This extension of the usual notion of dual basis, and dual lattice, will allow us, by a suitable choice of  $m$  (in our context it will be the modulus in (1)), to deal uniquely with integer coordinate vectors, which can then be represented exactly on the computer.

Consider the sequence  $x_n$  produced by the recurrence (1), and the set

$$T_t = \{(x_n, \dots, x_{n+t-1}) \mid n \geq 0, s_0 = (x_0, \dots, x_{n+k-1}) \in \mathbb{Z}_m^k\} \quad (3)$$

of all *overlapping*  $t$ -tuples of successive values, starting from all possible initial seeds. Then, the periodic continuation of  $T_t$  with period  $m$ ,

$$L_t = T_t + m\mathbb{Z}^t,$$

forms a lattice with unit cell volume of  $\max(1, m^{t-k})$  (see [9, 14, 15, 17, 21, 25, 30]). Bases for  $L_t$  and its dual can be constructed as explained in [14, 26]. The module MRG does that.

If one adds a constant  $b$  on the right-hand-side of (1), before applying the modulo operation, then the vectors of successive values will all belong to  $L'_t$ , where  $L'_t = L_t + V_{0,t}$  is a shift of  $L_t$  by some constant  $V_{0,t} \in \mathbb{Z}_m^t$ . Such a "shifted lattice" is called a *grid*. Since  $L'_t$  and  $L_t$  have the same structural properties, we do not consider the possible presence of such a constant  $b$  in LatMRG. In other words, we consider only the MLCGs or MRGs.

When  $m$  is prime and the MRG has full period  $m^k - 1$ , then  $T_t$  is the set of all  $t$ -tuples produced by the generator over its main cycle, plus the zero vector. Otherwise, if the generator does not have full period, then 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  $T_t$ . This is what happens in particular when  $k = 1$ ,  $m$  is a power of a prime  $p$ , and  $x_0$  is prime to  $p$ . Our package takes care of the latter case by constructing a basis for the appropriate sublattice.

A lattice basis for which the vectors are “reduced” in some specific sense is called a *Minkowski-reduced lattice basis* (MRLB) (see [1, 2, 14] for more details). 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  $q_t$  as the maximum of the Beyer quotients of all MRLBs. Values of  $q_t$  close to one are said to be more desirable. A figure of merit can be  $Q_T = \min_{k < t \leq T} q_t$  for some large enough  $T$ . The module `REDBAS` permits one to compute a MRLB and the Beyer-quotients for a given lattice. It is based on the main ideas of [1, 10, 14], but with many improvements, as explained in [6].

The lattice structure also means that all points of  $T_t$  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, and let  $D_t$  be the distance between them. This distance is in fact equal to one over the length of the shortest vector in the *dual* lattice to  $L_t$ . `REDBAS` computes that shortest vector through a branch-and-bound algorithm, using bounds based on [1, 10, 14] and other refinements described in [6]. For large dimensions  $t$ , this algorithm is much faster than the one given in [9, 17]. If we rescale the hypercube  $[0, m)^t$  that contains  $T_t$  to  $[0, 1)^t$ , the distance between hyperplanes becomes  $d_t = D_t/m$ . There are theoretical lower bounds  $d_t^*$  on  $d_t$  [3, 13, 17, 24] and one can define the figures of merit  $S_t = d_t^*/d_t$  and  $M_T = \min_{k \leq t \leq T} S_t$ , which lie between 0 and 1. The bounds  $d_t^*$  based on Hermite’s constants, as defined in [17], are known only up to  $t = 8$ . But for larger values of  $t$ , one can define lower bounds  $d_t^*$  based on Rogers’ bounds on the density of sphere packings, or obtain approximations of  $d_t^*$  from the values of  $d_t$  associated with the laminated lattices in dimension  $t$  [3, 24]. Again, one seeks values of  $S_t$  and  $M_T$  near one.

The algorithms for computing a shortest vector or a MRLB are based on branch-and-bound (BB) procedures, for which the bounds are computed through a Choleski decomposition performed in (double precision) floating-point arithmetic. Therefore, numerical roundoff errors occur during these computations and could (eventually) affect the results. In particular, because of wrong bounds in the BB, one may miss a shorter vector and, as a result, 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 “garanteed 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`.

## 1.4 Multiply-with-carry

A *Multiply-with-Carry* (MWC) generator [5, 8, 19, 29] is based in the recurrence

$$x_n = (a_1 x_{n-1} + \cdots + a_k x_{n-k} + c_{n-1}) \bmod b, \quad (4)$$

$$c_n = (a_1 x_{n-1} + \cdots + a_k x_{n-k} + c_{n-1}) \operatorname{div} b, \quad (5)$$

$$u_n = x_n/b.$$

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 two, 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:

$$z_n = az_{n-1} \bmod m; \quad w_n = z_n/m. \quad (6)$$

In other words, if

$$w_n = \sum_{i=1}^{\infty} x_{n+i-1} b^{-i} \quad (7)$$

holds for  $n = 0$ , then it holds for all  $n$ , and consequently  $|u_n - w_n| \leq 1/b$  for all  $n$ . The (approximate) lattice structure of the MWC can therefore be analyzed by analyzing that of the corresponding LCG (6). This is what the LatMRG package does.

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

$$d_t = (1 + a_1^2 + \dots + a_k^2)^{-1/2}. \quad (8)$$

Since  $b$  is a power of two,  $a$  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 (6) has period length  $\rho = (m - 1)/2$ .

## 1.5 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 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 `seek1`).

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 [28, 22]. They can thus be analyzed with the present software.

## 1.6 Lacunary indices

Instead of forming vectors with successive values like in the above definition of  $T_t$ , one can form vectors with values that are some distance apart in the sequence (so-called “leapfrog” values). Let  $i_1, i_2, \dots, i_t$  be a set of fixed integers. Redefine

$$T_t = \{(x_{i_1+n}, \dots, x_{i_t+n}) \mid n \geq 0, s_0 = (x_0, \dots, x_{n+k-1}) \in \mathbb{Z}_m^k\} \quad (9)$$

and let  $L_t = T_t + m\mathbb{Z}^t$ . The definition (3) was a special case of this one, with  $(i_1, \dots, i_t) = (0, \dots, t-1)$ . Using the module `MRG`, one can build a basis for  $L_t$  and its dual in this more general case. Distances between hyperplanes, Beyer quotients, etc., can then be computed as usual.

## 1.7 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` [27], 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. When working on a basis (finding shortest vector, Minkowski reduction, etc.), the vector elements can be represented either as `LONGREAL` (64-bit floating-point numbers) or `SuperInteger`. Modules `BASISLR` and `BASISSI` implement basic facilities for these two cases.

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).

## 1.8 Current implementation

The software is currently working on SUN computers, under Solaris and the MCS `Modula-2` system from `ModulaWare` [16]. 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` and `SENTIERS` [27], which must also be ported for `LatMRG` to work on a different platform.

## 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.

However, there are different libraries and header files that must be used depending on the representation chosen for the multipliers and the basis components by the program `select`. Calling one of the command `lilr`, `silr`, `sisi` in a shell will set the environment variables corresponding to the particular case of `select`. Furthermore, to compile a program (say `maxper`) directly with `xm`, one must also inform XDS of the given context; for example, `xm =m maxper.mod -lilr=sisi`. The `lilr` on the left of `=` is the name of the variable, the `sisi` on the right is its value. (See also the file `Makedoc.tex`.)

The program `maxper` checks whether a given generator has maximal period. The programs `lat1` and `lats` perform standard lattice and/or spectral tests for that generator. The only difference between the latter two programs is that in `lat1`, the multipliers and basis components are implemented in `LONGINT` and `LONGREAL`, respectively, while in `lats`, they are implemented as `SuperInteger`'s. Some more comprehensive (and larger) programs can perform computer searches to seek the “best” generators of a given type, according to maximal period and lattice structure criteria. These programs also come in two versions. In `seek1`, 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 [16]. 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 `lat1`, in file `fish31.dat`.

Table 1: Results of `lat1` 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 `lat1`

Figure 1 gives an example data file for the program `lat1`. The corresponding results appear in Table 1. To call the program and produce these results, type “`lat1 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  $\text{\LaTeX}$ . 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 `lat1` 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 exactly the same 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
NoCond	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 `seek1`, in file `seek15.dat`.

## 2.2 Examples with the programs `seek1` and `seeks`

Figure 2 gives an example of a data file for `seek1`. 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 seek1 in file seek15.res.

```

FALSE                                ReadGenFromFile
1                                    J
MRG                                  TypeGen
2 63 -711                             m
5                                    k
TRUE                                 PerMax
AppFact                              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) and recompile all the modules.

Generally speaking, the proper choice of representation depends on the size of the modulus. 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.

#### 3.3 Modifying the package, recompiling, and relinking

After calling `select`, or after modifying the code of a module, one must recompile and relink. Some systems provide automatic make facilities, which recompile all the modified modules in the correct order and relink appropriately. With *MCS Modula-2*, there is no such automatic facility, but we have written a `Makefile` to facilitate the compiling and linking of programs which use the lower-level modules of LatMRG. Just type `make` and the name of your program, and all the modules needed will be recompiled. See the `Makefile` for more details.

## 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 `lat1`. To run the program, type “`maxper <file>`”, where `<file>` is the name of the data file, without extension.

```

m
k
F1 [<file1>]
F2 [<file2>]
a1
:
ak

```

Figure 6: Data file format for `maxper`.

## findmk

This program is essentially an interface to the module PRIM. It reads the data and calls the procedure `FindPrimesmr` 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`.



```
k  
e  
c1  
c2  
Safe
```

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  $\langle file \rangle$  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.

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

*LatticeInfo*  $\langle Norm \rangle$  : 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 (`Hermite`, `Laminated`, `Rogers`).

*ResultForm* : Selects in which form the results will be given. The possible values are (`Terminal`, `RES`, `TEX`, `TRES`, `TTEX`). 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  $\text{\LaTeX}$ , with extension “`.tex`”.

`TRES`, `TTEX` : same as `RES` and `TEX`, but the results are also written on the terminal screen.

```

ReadGenFile [<file0>]
J
  TypeGen      }
  m            }
  k            } (Repeat J times)
  a1         }
  ⋮           }
  ak         }
Dim(0) Dim(1)
LatticeInfo <Norm>
LatticeType
LaGroupSizes Spacing
VerifyBB
MaxNodesBB
ResultForm

```

Figure 8: Data file format for latl.

## seekl

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 two generators with the highest  $M_8$ , the two 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 an 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  $b = (b_1, \dots, b_k)$  and  $c = (c_1, \dots, c_k)$  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, all vectors of multipliers of the form  $a = (a_1, \dots, a_k)$  such that  $b_i \leq a_i \leq c_i$  for  $i = 1, \dots, k$  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, we fix a number of subregions (clusters) we want to examine, and the size  $h_i$  of each subregion in dimension  $i$ , for  $i = 1, \dots, k$ . 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  $i = 1, \dots, k$ , generate  $\alpha_i$  randomly, uniformly over the set  $\{b_i, \dots, c_i - h_i + 1\}$ ; then, examine all the vectors  $a = (a_1, \dots, a_k)$  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  $k + 1, \dots, T$ .

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 the 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  $\sigma$  is larger than the lower bound for the last category, then we raise this lower bound to  $\sigma$ . 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, but they must be on the same line, separated from data by at least one blank. It is possible also to add some lines after these data lines for other comments, but not before or between these lines.

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

<i>ReadGenFile</i> [ <i>&lt;file0&gt;</i> ]	
<i>J</i>	
<i>TypeGen</i>	}
<i>m</i>	
<i>k</i>	
<i>PerMax</i>	
<i>ImplemCond</i> [ <i>&lt;NbMaxBits HighestBit&gt;</i> ]	
<i>F1</i> [ <i>&lt;file1&gt;</i> ]	
<i>F2</i> [ <i>&lt;file2&gt;</i> ]	
<i>b<sub>1</sub></i>	
<i>c<sub>1</sub></i>	
⋮	
<i>b<sub>k</sub></i>	
<i>c<sub>k</sub></i>	
<i>SearchMethod</i> [ <i>n H H<sub>k</sub></i> ]	
<i>C Dim(0) Dim(1) ⋯ Dim(C)</i>	
<i>MinMerit(1) MaxMerit(1) ⋯ MinMerit(C) MaxMerit(C)</i>	
<i>NbGen(1) ⋯ NbGen(C)</i>	
<i>Criterion</i> <i>&lt;Norm&gt;</i>	
<i>LatticeInfo</i>	
<i>LatticeType</i>	
<i>LaGroupSizes Spacing</i>	
<i>VerifyBB</i>	
<i>MaxNodesBB</i>	
<i>TimeLimit</i>	
<i>S1 S2</i>	
<i>ResultForm</i>	

Figure 9: Data file format for `seek1`.

- 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 `seek1`, 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  $\langle file0 \rangle.gen$ . 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 become meaningless.

**J** : Number of components in the combined generators. Must be an integer in the range [1..9]. When  $J > 1$ , we look for combined generators.

**TypeGen** : Can be **MRG** or **MWC**. If **MRG**, this component is a MRG. If **MWC**, 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, 23]).

**m, k** : Must be positive integers, with  $k < MMaxDim$ . For  $m \geq 2^{31}$ , use the program **seeks** instead of **Seekl**.

**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  $\langle NbMaxBits\ HighestBit \rangle$**  : Can be **NoCond**, **AppFact**, or **MaxBits**. 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 [20]. If **MaxBits**, then the positive integers **NbMaxBits** and **HighestBit** must appear and they indicate that for each  $i$ , the multiplier  $a_i$  must be the sum of at most **NbMaxBits** (positive or negative) powers of two, with the highest power of two not exceeding  $2^e$  in absolute value, where  $e = HighestBit$ . For example, if **MaxBits** = 2 and **HighestBit** = 30, there are  $30 \times 31/2$  possibilities for choosing the two powers of two and 4 possibilities for choosing their signs, yielding 1860 cases where  $a_i$  is obtained from exactly two powers of two. If one adds the 62 cases where  $a_i$  is  $\pm$  a power of two, this gives a total of 1922 possibilities for  $a_i$ . In the case of **MaxBits**, the bounds  $b_j$  and  $c_j$  on the multipliers (see below) are automatically (implicitly) reset to  $b_j = -(2^{e+1} - 1)$  and  $c_j = 2^{e+1} - 1$ .

**F1 and  $\langle file1 \rangle$**  : 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  $\langle file1 \rangle$  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  $\langle file1 \rangle$  is not used and can be omitted. To factorize, the program uses the procedure **Factorize** in the module **SUPFACT** of the package **SENTIERS** [27], 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  $\langle file1 \rangle$ , one factor per line.

**Read** indicates that  $m - 1$  is already factorized and that the factors will be read from file  $\langle file1 \rangle$ , 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  $\langle file2 \rangle$**  : This data line is similar to the previous one, except that it concerns  $r = (m^k - 1)/(m - 1)$  instead of  $m - 1$ . In this case, it is possible that  $r$  be prime when  $m$  is prime (in contrast to  $m - 1$ , which is then even). Therefore, the additional value **Prime** is allowed for **F2**, so that the set of possible values is (**Decomp**, **DecompWrite**, **Read**, **Prime**). The module **SUPFACT** in **SENTIERS** [27] can be used independently to factorize  $r$  or to check its primality.

**Prime** indicates that  $r$  is prime.

$b = (b_1, \dots, b_k)$  and  $c = (c_1, \dots, c_k)$  : The  $b_i$  and  $c_i$  are integers such that  $-m < b_i \leq c_i < m$  for  $i = 1, \dots, k$ . They determine the boundary of the (rectangular) area of search. When **ImplemCond** = **MaxBits**, these values are not used.

**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, while  $H$  determines the size of these subregions, except for the  $k$ th element of the vector  $h$ , where  $H_k$  determines the size. The vector  $h = (h_1, \dots, h_k)$  is computed by the program as follows:  $h_i = \min(H, c_i - b_i + 1)$ ,  $i = 1, \dots, k - 1$ , and  $h_k = \min(H_k, c_k - b_k + 1)$ .

$C$  **Dim(0) Dim(1)  $\dots$  Dim(C)** : Integers such that  $1 \leq C \leq \mathbf{MaxCat}$  and  $2 \leq \mathbf{Dim}(0) \leq \mathbf{Dim}(1) < \dots < \mathbf{Dim}(C) \leq \mathbf{BMaxDim}$ , where **BMaxDim** is the maximal dimension that can be treated by the program (see the modules **CONSTANT** and **BASIS**). The spectral and/or Beyer tests will be performed in dimensions  $\mathbf{MinDim} = \mathbf{Dim}(0)$  to  $\mathbf{MaxDim} = \mathbf{Dim}(C)$ . However, in the case of successive values (no lacunary indices), the test will start in dimension  $\max(\mathbf{Dim}(0), k + 1)$ , where  $k$  is the order of the recurrence defining the MRG. The value of  $C$  is the number of categories (intervals) into which the dimensions are partitioned. These  $C$  categories end at the dimensions  $\mathbf{Dim}(1), \dots, \mathbf{Dim}(C)$ , respectively. To *minimize* (instead of maximizing) the criterion within a given category  $i$ , place a *negative sign* in front of the value of  $\mathbf{Dim}(i)$  in the data file. The program will then retain the *worst* instead of the *best* generators with respect to the category  $i$ .

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

*NbGen(1), ..., NbGen(C)* : Each *NbGen(i)* must be an integer in the range  $[0, \dots, \text{MaxNbGen}]$ . If positive: maximum number of generators to retain in the results for category *i*. The value 0 means that no list of generators is retained for that category.

*<Invert> Criterion <Norm>* : Gives the merit criterion for ranking the generators. The possible values of *Criterion* are (**Beyer**, **Spectral**, **SpectralP**, **SpectralL1**).

**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 [18], Exercises 3.3.4-15 and 16).

If *Criterion* = **Spectral** or **SpectralP**, then *Norm* must appear and it indicates which type of normalization is used in the definition of  $S_t$ . The allowed values are (**Hermite**, **Laminated**, **Rogers**).

**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  $M_{\min(T,48)}$ .

The field *Invert* can be either **Invert** or left blank. In the former case, the values of the criterions  $S_t$  and  $M_T$  are replaced by  $1/S_t$  and  $1/M_T$  in the results.

*LatticeInfo* : Indicates whether we want to compute and print the values of  $q_t$ ,  $d_t$ , or both. The possible values are (**Beyer**, **Spectral**, **BeyerSpectral**, **SpectralP**, **SpectralL1**). Note that the values required for the ranking criterion (previous line) will be computed anyway. Note also that, typically,  $q_t$  requires significantly 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.

**BeyerSpectral** : Prints all of the above.

**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).

*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 [26]. For example, if  $p = 2$  and  $a \bmod 8 = 5$ , then the modulus is divided by 4 as in [11, 17].

*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  $(u_{i+1}, \dots, u_{i+s}, u_{i+d+1}, \dots, u_{i+d+s}, u_{i+2d+1}, \dots, u_{i+2d+s}, \dots)$ , 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  $t = \text{MaxDim}$  and then, on the  $t$  lines that follow, give the  $t$  lacunary indices  $i_1, \dots, i_t$ , 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  $\alpha_i$ 's. Before starting the search, the program calls `SetSeed(S1, S2)` from 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 `lat1`, 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 `lat1`, except that the multipliers are represented as `SuperInteger`'s. (Modules `MULTSI` and `BASISLR` are used.)

## seeks

Same as `seek1`, except that the multipliers and bases are represented as `SuperInteger`'s. (Modules `MULTSI` and `BASISSI` are used.)

## seeksr

Same program as `seek1`, 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 `seek1`, but the computations are less precise than with `seeks`.

## streams

Ce programme prend en entrée un fichier ayant le même format que pour `lats`, sans l'extension `.dat`, avec une ligne additionnelle à la fin contenant 3 entiers positifs `wminStreams`, `wmaxStreams`, et `smaxStreams`. La représentation est `si si`, ce qui permet de traiter de très grands nombres. Pour un générateur donné, `streams` sépare sa période en plusieurs petits générateurs de longueur  $2^w$  où dans le programme  $w$  varie de `wminStreams` à `wmaxStreams`. Il évalue ensuite  $S_T$  et  $Q_T$  pour certains  $t$  et  $s$ .  $t$  correspond à la dimension du vecteur et  $s$  au nombre d'indices à prendre au début de chacun des générateurs jusqu'à ce que nous ayons assez d'éléments dans notre vecteur.  $t$  et  $s$  varient de 1 à `smaxStreams` pour  $Q_T$  et  $S_T$ . Enfin pour chacun des  $w$ , il trouve le minimum  $Q_T$  et  $S_T$  pour tout  $t$  et  $s$ .

## menu<sub>mr</sub>g

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, 32]. In those cases, the shortest vectors and reduced bases were computed in (partly) “manual” mode.

Two bases  $B_1$  and  $B_2$  and one vector  $\mathbf{v}_e$  are created initially (automatically) and available to the user. All along,  $B_1$  and  $B_2$  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  $B_1$ ; (b) those affecting  $B_1$  and  $B_2$ ; (c) and those involving  $B_1$  and  $\mathbf{v}_e$ . 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  $\mathbf{v}_e$  :
15. Vector construction
16. Relative coordinates
17. Insertion of  $\mathbf{v}_e$  in B1

```

Figure 10: Main menu for the program menu<sub>mr</sub>g.

### Operations on $B_1$ :

1. *Building MRG basis* : Asks the user for a modulus, order, multiplier(s), and dimensions, then constructs the appropriate lattice basis in  $B_1$ .
2. *Reading/Writing* : Read/write the basis  $B_1$  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” prereducations  $T$  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.

## 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  $\alpha$  entier et pair est

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

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

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

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  $\alpha$  pairs. La formule du  $P_\alpha(s)$  devient:

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

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:

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

### Le module PALPHA

Module servant de support aux programmes PAI, BAI et Coeff. Ils permettent respectivement de trouver le  $P_\alpha$  ou le  $B_\alpha$  d'un générateur ou enfin de trouver des bons coefficients pour un générateur en utilisant le  $P_\alpha$  comme discriminant.

### palpha

Ce programme permet de calculer le  $P_\alpha$  pour un générateur et un  $\alpha$  donné. Tout dépendant du  $\alpha$  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  $B_\alpha$  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

<i>LCG</i>	<i>SorteGenerateur</i>
<i>PAL ou BAL</i>	<i>Type de calcul</i>
<i>251</i>	<i>m modulo</i>
<i>true</i>	<i>modulo premier</i>
<i>33</i>	<i>a multiplicateur</i>
<i>2</i>	<i>mindim</i>
<i>4</i>	<i>maxdim</i>
<i>2</i>	<i>saut</i>
<i>2</i>	<i>alpha</i>
<i>false</i>	<i>VerifBooleen</i>
<i>1</i>	<i>seed</i>
<i>1.0 2.0 3.0 1.0 1.0</i>	<i>Beta(0),...,Beta(MaxDim)</i>

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  $P_\alpha$ .

*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 vérifiera 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  $P_\alpha$  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  $\beta(0), \dots, \beta(MaxDim)$ .

## coeff

Ce programme permet de trouver des bons coefficients pour un générateur dont on a donné le modulo et le type. Cette recherche se fait en fonction des  $P_\alpha$ . Il faut donc également préciser un  $\alpha$ .

### Le format du fichier d'entrée pour Coeff

<i>LCG</i>	<i>SorteGenerateur</i>
<i>Coeff</i>	<i>Type de calcul</i>
<i>251</i>	<i>m modulo</i>
<i>true</i>	<i>modulo premier</i>
<i>3 250</i>	<i>intervalle pour le multiplicateur a</i>
<i>2</i>	<i>mindim</i>
<i>4</i>	<i>maxdim</i>
<i>2</i>	<i>saut</i>
<i>2</i>	<i>alpha</i>
<i>false</i>	<i>VerifBooleen</i>
<i>1</i>	<i>seed</i>
<i>1.0 2.0 3.0 1.0 1.0</i>	<i>Beta(0),...,Beta(MaxDim)</i>

Figure 1 : Format du fichier pour Coeff

Tous les champs sont identiques au cas du PAI ou BAI sauf pour le cinquième qui indique plutôt l'intervalle à l'intérieur duquel chercher le meilleur coefficient.

## APPENDIX B

### Intermediate-Level Modules

## LATMRG

Les fichiers `LATMRG.h` et `LATMRG.c` existent uniquement pour permettre à un utilisateur de programmer directement en `C`. Dans ce cas, il doit inclure la directive

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

au début de son fichier, et ajouter

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

comme première instruction dans sa fonction

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

S'il utilise des fonctions de `mylib`, de `simod` ou de `sentiers`, il devra inclure les fichiers d'en-tête

```
#include "MYLIB.h"
```

```
#include "SIMOD.h"
```

```
#include "SENTIERS.h"
```

Il pourra consulter les fichiers `*.h` appropriés de `latmrg/xds` pour voir les prototypes des fonctions externes `C`.

---

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

```
PROCEDURE LATMRG_BEGIN();
```

```
END LATMRG.
```

## 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 `seek1` program.

The searches for lists of (good or bad) generators are performed by categories, as explained in the documentation of the `seek1` 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 CONSTANT IMPORT BDim, BMaxDim;
FROM MULT     IMPORT MScal, MDim, MMaxDim, MVect;
FROM MRG      IMPORT TestType, VerifyType, OutputType;
IMPORT BASISLR;
```

CONST

```
MaxJ      = 5;
```

Maximum number of components  $J$  for a combined generator.

```
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.

TYPE

```
Critere   = [Beyer..SpectralR];
```

Type for the selection criterion.

```
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).

```
Indexj      = [1..MaxJ];
```

Index of a component  $j$ .

```
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. If 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 seek1.

```
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 multiplier  $m_j$  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  $a_{j,1}, \dots, a_{j,k_j}$  for component  $j$ .

PROCEDURE Readbc (j : Indexj);

Reads the  $2k_j$  bounds  $b_{j,1}, c_{j,1}, \dots, b_{j,k_j}, c_{j,k_j}$  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 `lat1` and `seek1`. 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         : TestType;
  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 category `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  $\text{\LaTeX}$  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

$$x_n := (a_1x_{n-1} + \cdots + a_kx_{n-k}) \bmod m.$$

One can then build the lattice basis associated to a vector `a` 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 MDim, MScal, MVect;
FROM BASIS     IMPORT BVect;
FROM CONSTANT  IMPORT BDim;
FROM LATBASIS  IMPORT Basis, NormType;
FROM SUPINT    IMPORT SuperInteger;
FROM SUPFACT   IMPORT Factors;
IMPORT BASISLR;
```

TYPE

```
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 [26], 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 `seek1`.

```
TestType = (Beyer, Spectral, SpectralL1, SpectralH,
            SpectralL, SpectralR, BeyerSpectral);
```

Type of test to be performed: Beyer test, spectral test, or both. For the spectral test, there are different types, indicating different ways of normalizing the distance  $d_t$  to obtain the figure of merit. One can normalize using Hermite's bound (`SpectralH`), or the value that corresponds to the laminated lattice (`SpectralL`), or Rogers bound (`SpectralR`). `Spectral` is equivalent to `SpectralH`. `BeyerSpectral` means that both `Beyer` and `Spectral` apply simultaneously. `SpectralL1` is the spectral test with the  $L_1$  norm.

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

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

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

Kind of output to be produced. `None` means that nothing will be written to the current output (the results can be recovered from `Bas`, `Q2`, `D2`, `S2`). `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 L<sup>A</sup>T<sub>E</sub>X format. TCurOut and TCurOutTex are similar, except that the results are also echoed to the terminal screen.

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 **seek1**. 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 adjusted (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.

**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              : TestType;
  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              : TestType;
  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.

```
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         : TestType;
  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 procedure 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.
```

## 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  $V_i \cdot W_j = \delta_{ij}m$  for some integer constant  $m$ . We offer two possible representations for the vector coordinates: `LONGREAL` or `SuperInteger`. The latter is a type defined in a package named `SENTIERS` which offers tools for manipulating arbitrary large integers. The `SuperInteger` representation should be used when the coordinates of the (dual) basis vectors, are too large to fit a `LONGREAL` without loss of precision. But computations are then much slower. The program `select` decides which one is used. All computations will be performed using appropriate procedures, depending on the representation.

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 and do not fit a `LONGREAL`, even approximately, they are then stored as `SuperInteger` in `VVSI` and `WWSI`. This is done when `SISquares` is set to `TRUE`.

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;
```

```
IMPORT BASISLR, BASISSI;
FROM BASIS    IMPORT BScal, BVect, BMatr;
FROM CONSTANT IMPORT BDim;
FROM MULT    IMPORT MScal;
```

```
TYPE
```

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

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

```
  Basis = POINTER TO InfoBasis;
  InfoBasis = RECORD
    Dim      : BDim;                (* Actual dimension.      *)
    mLR      : LONGREAL;            (* Modulus                 *)
    mSI      : BASISSI.BScal;
    Norm     : NormType;
    V, W    : BMatr;                (* Basis and dual basis vectors. *)
    VVLR, WWLR : BASISLR.BVect;    (* Squares of vector lengths.  *)
    VVSI, WWSI : BASISSI.BVect;    (* Same, in SuperInteger.     *)
    SISquares : BOOLEAN;
  END;
```

A lattice basis. `Dim` is the dimension, `V` contains the basis, and `W` the 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. *VVLR* and *WWLR* *may* memorize the lengths of the vectors in *V* and in *W* (the squared length, in the case of the Euclidean norm). They are not always updated, and when they are not, they are set to negative values. If *SISquares* is *TRUE*, the norm is assumed to be Euclidean, and the square lengths are in *VVSI* and *WWSI* instead (also not always updated).

PROCEDURE CreateBasis (VAR B : Basis);

Creates the data structure B for a basis. *SISquares* is initialized to *FALSE*. The *Norm* is initialized to *L2Norm*.

PROCEDURE InitBasis (B : Basis; d1, d2 : BDim);

Reserves the memory for the *BScal*, *BVect* and *BWatr* fields of B, for indices *d1* to *d2*. Here, *d2* is new the size of *V* or *W*, and the value of *d1* can be either 0 or one unit more than the value of *d2* from a previous call to *InitBasis*.

PROCEDURE DeleteBasis (B : Basis; d1, d2 : BDim);

Deallocates the memory allocated for *V*, *W*, *VVSI* and *WWSI*, at a previous call to *InitBasis*. Must be called only if *InitBasis* has been called previously.

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 vector by increasing length. The dual vectors are permuted accordingly. Also updates *VVLR* (or *VVSI*).

PROCEDURE Dualize (B : Basis);

Exchanges *V*, *VVLR*, *VVSI* and *W*, *WWLR*, *WWSI*.

PROCEDURE UpdateVVWW (B : Basis);

Updates *VVLR* and *WWLR* (or *VVSI* and *WWSI*) according to the present values of *V* and *W*.

END LATBASIS.

## REDBAS

This module offers tools to reduce a basis in the sense of Minkowski and find a shortest vector in a lattice. See [6] for more details.

---

DEFINITION MODULE REDBAS;

```
IMPORT BASISLR, BASISSI;
FROM LATBASIS IMPORT Basis;
FROM CONSTANT IMPORT BDim;
```

TYPE

```
  Critere2Type = (Hermite, Rogers, Minkowski, Laminated, MinkL1);
```

Type of bound used to normalize the length of the shortest vector. The normalized value is the figure of merit used as a selection criterion. `MinkL1` must be used for the  $L_1$  norm; the others are for the Euclidean norm.

```
  PreReductionSV = (Dieter, LLL, DieterLLL);
```

Types of pre-reduction for `ShortestVector`. `Dieter` means the pairwise pre-reduction as in the procedure `PreRedDieter` below. `LLL` means the LLL reduction of Lenstra, Lenstra, and Lovász. `DieterLLL` means both.

```
  PreReductionRM = (NoLLL, WithLLL);
```

Types of pre-reductions for `ReductMinkowski`: with and without LLL. Note that the LLL reduction is not performed if `SISquares = TRUE`.

```
  VectLI = ARRAY BDim OF LONGINT;
```

VAR

```
  Gamma      : BASISLR.BVect;
```

If `Critere2` is `Hermite`, then this array contains the Hermite constants  $\gamma_t$  (for  $2 \leq t \leq 8$ ). The distances  $d_t$  between hyperplanes in the  $t$ -dimensional unit hypercube always satisfy  $d_t \geq d_t^* = \gamma_t^{-1} m^{-k/t}$ . If `Critere2` is `Laminated` or `Rogers`, then the vectors contains the appropriate constants so that  $d_t^*$  is computed in terms of the laminated lattice or in terms of Rogers' bound, respectively (see [24] for more details on this). With `Critere2 = MinkL1`, `Gamma[t]` contains  $t!$ .

```
  PreRedSV    : PreReductionSV;
```

```
  PreRedRM    : PreReductionRM;
```

These two variables indicate which pre-reduction is to be performed for `ShortestVector` (SV) and for `ReductMinkowski` (RM).

```
  MaxNodesBB : LONGINT;
```

The maximum number of nodes that we are ready to accept in the branch-and-bound (BB) tree.

```
  MinkLLL, ShortDietLLL, ShortLLL : LONGINT;
```

These are threshold constants. `PreRedRM` is originally set to `NoLLL`. Whenever the number of nodes in the BB tree exceeds `MinkLLL` in the `ReductMinkowski` procedure, `PreRedRM` is automatically set to `WithLLL`. Whenever the number of nodes in the BB tree exceeds `ShortDietLLL`, in the

ShortestVector procedure, PreRedSV is automatically set to DieterLLL. If the number of nodes in the BB tree exceeds ShortLLL but does not exceed ShortDietLLL, then PreRedSV is set to LLL.

Critere2 : Critere2Type;

Critere2 tells us which figure of merit is currently used by this module, for the Euclidean norm.

MaxDimS2 : BDim;

The maximal dimension for which  $d_t^*$  is defined.

PROCEDURE InitREDBAS (d2 : BDim; ttype : Critere2Type);

Initializes the module with a given upper dimension d2 for the bases, for memory allocation purposes. This d2 should be at least as large as the dimension of any Basis this module has to work on.

PROCEDURE DeleteREDBAS (d2 : BDim);

Deallocates the memory allocated to the vector of SuperIntegers Sv, to the matrices of type BMatr VTemp and WTemp and to the BVect ZeroVect, during a call to the procedure InitREDBAS. Must be called only if InitREDBAS has been called previously.

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. This reduction should not be used when SISquares is set to TRUE.

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

Performs pairwise reductions, trying to reduce every basis vector with index larger than d, and distinct from i, by adding multiples of the i-th vector.

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.

PROCEDURE PreRedDieter (B : Basis; n : LONGINT; 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 the nth successful transformation or after no further reduction is possible.

PROCEDURE CalculCholeski

(B : Basis;  
 VAR DC2 : BASISLR.BVect;  
 VAR CO : BASISLR.BMatr);

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

Reduces B into a Minkowski reduced basis. The first d vectors are assumed reduced. MaxNodesBB is the maximal number of nodes that we are willing to examine during each branch-and-bound. When this value is exceeded for one of the branch-and-bounds, the procedure aborts, returns FALSE, and the basis

## 52 REDBAS

might not be reduced. Otherwise, it returns `TRUE`. In the latter case, the basis is sorted by vector lengths. (The shortest vector is `V[1]`). This procedure does not care about numerical imprecision, and therefore, the results are not 100% reliable.

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

Computes the shortest vector in lattice with basis `B`, using branch-and-bound. This procedure does not care about numerical imprecision, and therefore, the results are not 100% reliable.

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

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

```
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 [6].

---

```
DEFINITION MODULE REDBAS2;
```

```
FROM CONSTANT 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 CONSTANT IMPORT BDim;
FROM LATBASIS IMPORT Basis;
FROM REDBAS  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.
```

## PRIM

This procedure 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  $S_1 \leq m \leq S_2$  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 **SFindPrimer**, 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.
```

## PALPHA

Ce module offre les fonctions utilisées dans les programmes `coeff` et `palpha`

---

```

DEFINITION MODULE PALPHA;

FROM TABLES IMPORT TabLR;

CONST
  MaxDim = 32;                (* Nb. max. de dimensions. *)

TYPE
  CaractGenerateur = RECORD
    TypeGen : ARRAY[1..32] OF CHAR;
    M : INTEGER;
    Prime : BOOLEAN;
    A : INTEGER;
    BorneInfA, BorneSupA : LONGINT;
    TypeCalcul : ARRAY[1..32] OF CHAR;
    Min, Max, Saut : INTEGER;
    Alpha : INTEGER;
    PeriodeMax : BOOLEAN;
    Seed : LONGINT;
    TabBeta : TabLR;
  END;

```

Record qui garde toutes les caractéristiques associées au générateur et au type de calcul demandé.

```
  Gener01 = PROCEDURE () : LONGREAL;
```

```
PROCEDURE Strip (r : LONGINT) : LONGREAL;
```

Provient du module `unif01`.

```
PROCEDURE StripLI (dr : LONGREAL; r : LONGINT) : LONGINT;
```

Provient du module `unif01`.

```
PROCEDURE SetTitre (Titre : ARRAY OF CHAR);
```

Provient du module `unif01`.

```
PROCEDURE SetLCG ( m, a, c, s : LONGINT );
```

Provient du module `ulcg`.

```
PROCEDURE SmallLCG () : LONGREAL;
```

Provient du module `ulcg`.

```
PROCEDURE MediumLCG () : LONGREAL;
```

Provient du module `ulcg`.

```
PROCEDURE MediumMLCG () : LONGREAL;
```

Provient du module `ulcg`.

```
PROCEDURE LargeLCG () : LONGREAL;
```

Provient du module `ulcg`.

PROCEDURE MultModLI (a, s, c, m : LONGINT) : LONGINT;

Provient du module unif01.

PROCEDURE WrLCG ();

Provient du module ulcg.

PROCEDURE ReadTypeGen(VAR TypeG : ARRAY OF CHAR);

Lit le type du générateur et le place dans la variable TypeG.

PROCEDURE ReadTypeCalcul(VAR TypeC : ARRAY OF CHAR);

Lit le type de calcul à effectuer et le place dans la variable TypeC.

PROCEDURE ReadM(VAR M : INTEGER);

Lit le modulo de la récurrence et le place dans la variable M.

PROCEDURE ReadBorneA(VAR InfA, SupA : LONGINT);

Lit l'intervalle à l'intérieur duquel le meilleur coefficient doit être trouvé.

PROCEDURE ReadA(VAR A : INTEGER);

Lit le coefficient de la récurrence et le place dans la variable A.

PROCEDURE ReadMinMax(VAR Min, Max, Saut : INTEGER);

Lit la plage de dimensions à parcourir et la place dans les variables Min et Max.

PROCEDURE ReadAlpha(VAR Alpha : INTEGER);

Lit le  $\alpha$  déterminant quel  $P_\alpha$  calculer et le place dans la variable Alpha.

PROCEDURE ReadPeriodeMax(VAR PeriodeMax : BOOLEAN);

Lit le booléen indiquant si la période doit nécessairement être maximale et le place dans la variable PeriodeMax.

PROCEDURE ReadSeed(VAR Seed : INTEGER);

Lit le germe qui initialisera la récurrence et le place dans la variable Seed.

PROCEDURE ReadBeta(VAR Gen: CaractGenerateur);

Lit les  $\beta_j, j = 0, 1, \dots, s$  associés à chaque dimension.

PROCEDURE SaisieDonnees(VAR Gen: CaractGenerateur);

Lit les caractéristiques du générateur et le genre de calcul demandé. Utilise les procédures définies précédemment. Place les renseignements dans la variable Gen.

PROCEDURE CalculPALpha2(Gen: CaractGenerateur; S: INTEGER; PtrTabUij : TabLR ): LONGREAL;

Calcule le  $P_2(S)$  pour le générateur contenu dans Gen où  $S$  est la dimension. La période doit être maximale. Voici le calcul effectué.

$$P_2(s) = \beta_0 \left( -1 + (1/m) \sum_{i=0}^{m-2} \prod_{j=0}^{s-1} (1 + 2\pi^2 \beta_{j+1}^2 (1/6 + u_{i+j} (u_{i+j} - 1))) \right. \\ \left. + (1/m) \prod_{j=0}^{s-1} (1 + 2\pi^2 \beta_{j+1}^2 / 6) \right)$$

$u_{i+j}$  est le nombre entre 0 et 1 produit par le générateur à la  $i + j^e$  étape.

$m$  est le modulo

PROCEDURE PALpha2PerNonMax(Gen: CaractGenerateur; S: INTEGER): LONGREAL;

Calcule le  $P_2(S)$  pour le générateur contenu dans Gen où  $S$  est la dimension, la période n'est pas nécessairement maximale. On s'assure de prendre tous les points en utilisant les multiples, de 0 à

$m - 1$ , du vecteur suivant  $(1, a, a^2, a^3, \dots, a^{s-1}) \bmod m/m$ . Ce vecteur correspond à  $(u_i, u_{i+1}, \dots, u_{i+s-1})$  si le générateur est à période maximale. Les deux méthodes de calcul donnent donc la même réponse, la deuxième nous assure de ne pas entrer dans un cycle lorsque la période n'est pas maximale.

PROCEDURE CalculPALpha4(Gen: CaractGénérateur; S: INTEGER; PtrTabUij : TabLR): LONGREAL;

Pareil comme pour CalculPALpha2 mais la fonction mathématique est:

$$P_4(s) = \beta_0 \left( -1 + (1/m) \sum_{i=0}^{m-2} \prod_{j=0}^{s-1} (1 - 2\pi^4 \beta_{j+1}^4 / 3 (u_{i+j}^4 - 2u_{i+j}^2 + u_{i+j} - 1/30)) \right. \\ \left. + (1/m) \prod_{j=0}^{s-1} (1 + 2\pi^4 \beta_{j+1}^4 / 90) \right)$$

PROCEDURE PALpha4PerNonMax(Gen: CaractGénérateur; S: INTEGER): LONGREAL;

Calcule également le  $P_4(s)$  mais pour des générateurs à période non maximale.

PROCEDURE CalculPALpha6(Gen: CaractGénérateur; S: INTEGER; PtrTabUij : TabLR): LONGREAL;

Tout comme CalculPALpha2 avec la fonction suivante:

$$P_6(s) = \beta_0 \left( -1 + (1/m) \sum_{i=0}^{m-2} \prod_{j=0}^{s-1} (1 + 2\pi^6 \beta_{j+1}^6 / 45 (u_{i+j}^6 - 3u_{i+j}^5 + 2.5u_{i+j}^4 - 0.5u_{i+j}^2 + 1/42)) \right. \\ \left. + (1/m) \prod_{j=0}^{s-1} (1 + 2\pi^6 \beta_{j+1}^6 / 945) \right)$$

PROCEDURE PALpha6PerNonMax(Gen: CaractGénérateur; S: INTEGER): LONGREAL;

Calcule également le  $P_6(s)$  mais pour des générateurs à période non maximale.

PROCEDURE CalculPALpha8(Gen: CaractGénérateur; S: INTEGER; PtrTabUij : TabLR): LONGREAL;

$$P_8(s) = \beta_0 \left( -1 + (1/m) \sum_{i=0}^{m-2} \prod_{j=0}^{s-1} (1 - 2\pi^8 \beta_{j+1}^8 / 215 (u_{i+j}^8 - 4u_{i+j}^7 + (14/3)u_{i+j}^6 - (7/3)u_{i+j}^4 + (2/3)u_{i+j}^2 - 1/30)) \right. \\ \left. + (1/m) \prod_{j=0}^{s-1} (1 + 2\pi^8 \beta_{j+1}^8 / 9450) \right)$$

PROCEDURE PALpha8PerNonMax(Gen: CaractGénérateur; S: INTEGER): LONGREAL;

Calcule également le  $P_8(s)$  mais pour des générateurs à période non maximale.

PROCEDURE CalculBALpha2(Gen: CaractGénérateur; S: INTEGER): LONGREAL;

Calcule le  $B_\alpha$  qui indique qu'il existe au moins un  $a$  pour lequel  $P_\alpha \leq B_\alpha$ .

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

PROCEDURE PeriodeMax(Gen : CaractGénérateur): BOOLEAN;

Vérifie si la période du générateur est maximale et inscrit le résultat dans le fichier.

PROCEDURE EcritDonnees(Gen : CaractGénérateur);

Réécrit en en-tête dans le fichier de résultat les données lus dans le fichier d'entrée.

PROCEDURE EcrireEntete;

Écrit dans le fichier de sortie l'entête d'un fichier latex, ce qui permettra de compiler le fichier pour avoir une version .dvi.

PROCEDURE EcrirePAlpha(PAlpha:LONGREAL;Dim:INTEGER;Gen:CaractGénérateur);

Écrit dans le fichier de résultat le calcul du  $P_\alpha$ .

PROCEDURE InitTab(Tab : TabLR; Dim : INTEGER);

Initialise un tableau de dimensions données avec des zéros.

PROCEDURE Moyenne(Tab : TabLR; Debut, Fin : INTEGER): LONGREAL;

Calcule la moyenne des éléments d'un tableau.

PROCEDURE EcartType(Tab : TabLR; Debut, Fin : INTEGER) : LONGREAL;

Calcule l'écart-type des éléments d'un tableau.

PROCEDURE SortTab(Tab : TabLR; Dim : INTEGER);

Tri le contenu d'un tableau en utilisant un bubblesort.

PROCEDURE MakeInterv(TabPA1 : TabLR; Dim : INTEGER);

Sépare le contenu d'un tableau trié en intervalle. Il est possible d'utiliser l'écart-type et la moyenne pour fixer une borne permettant de rejeter des valeurs qui peuvent paraître suspectes. Ces intervalles sont ensuite inscrits dans un fichier afin de produire des graphiques avec gnuplot.

PROCEDURE PAlpha2s

```
(Gen      : CaractGénérateur;
 S1, S2   : INTEGER;
 VAR P2   : ARRAY OF LONGREAL
 );
```

Calcule  $P_2$  en dimensions S1 a S2, en supposant que les poids sont tous égaux à 1 et que les points sont produits par un MLCG de période maximale. Les valeurs sont retournées dans P2[S1..S2]. Le tableau P2 doit bien sûr commencer à 0.

PROCEDURE PAlpha2PerNonMaxs

```
(Gen      : CaractGénérateur;
 S1, S2   : INTEGER;
 VAR P2   : ARRAY OF LONGREAL
 );
```

Calcule  $P_2$  en dimensions S1 a S2, en supposant que les poids sont 1. Les valeurs sont retournées dans P2[S1..S2]. Le tableau P2 doit bien sûr commencer à 0.

PROCEDURE CalculPAlphaTest(Gen: CaractGénérateur; S: INTEGER ): LONGREAL;

Calcule le  $P_2(s)$  mais de manière non efficace. Sert seulement à s'assurer que l'optimisation trouvée est valide.

END PALPHA.

## APPENDIX C

### Lower-Level Modules

## CONSTANT

Here, we define constants and types which are used throughout the package.

---

```
DEFINITION MODULE CONSTANT;

CONST
  BMaxDim = 50;                                (* Max. dimension for a lattice. *)

TYPE
  BDim = [0..BMaxDim];

END CONSTANT.
```

## 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 “.tex” for one of them) and change the module name (in the code) to MULT. This can also be done automatically using the command `select`.

## 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 “.tex” for one of them) and change the module name (in the code) to BASIS. This can also be done automatically using the command `select`.

## 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 “.tex” 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 [17]. 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;
```

```
CONST
```

```
  MMaxDim = 20;
```

Dimension of arrays and matrices. Maximal order of a generator.

```
TYPE
```

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

```
  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  $S$  at a previous call to `MDeleteScal`.

```
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 input 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 R and S are between -m and m.

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

Returns in R the scalar product of U by V modulo m, using components 1 to n of these vectors. a 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  $x^q \bmod f(x) \pmod m$ , where  $f(x) = x^n - A_1x^{n-1} - \dots - A_n$ .

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;
```

```
CONST
```

```
  MMaxDim = 20;
  MSizeSI = 2;                                (* Used in MCreateScal      *)
```

```
TYPE
```

```
  MDim      = [0..MMaxDim];
  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 PrimeElem (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 CONSTANT IMPORT BMaxDim, BDim;
```

```
TYPE
```

```
  BScal   = LONGREAL;
  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 `S` and initializes it to zero.

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

Deallocates the memory allocated to `S` at a previous call to `BDeleteScal`.

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

Creates entries of `V`, using `BCreateScal`, for indices `d1` to `d2` inclusive. If `d2` is smaller than `d1` then does nothing.

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

Deallocates the memory that was allocated for the creation of `V`, of type `BVect`, at a previous call to `BCreateVect` with indices from `d1` to `d2` inclusive.

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

Creates the elements of `M`, using `BCreateScal`.

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

Deallocates the memory that was allocated for the creation of `M`, of type `BMatr`, at a previous call to `BCreateMatr` with parameters `d1` and `d2`.

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

Returns 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 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.

```
END BASISLR.
```

## BASISSI

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

---

```
DEFINITION MODULE BASISSI;
```

```
IMPORT BASISLR;
FROM SUPINT   IMPORT SuperInteger;
FROM CONSTANT 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 entries for indices `d1` to `d2` inclusive. If `d2` is smaller than `d1` then does nothing.

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

Deallocates the memory that was allocated for the creation of `V`, of type `BVect`, at a previous call to `BCreateVect` with indices from `d1` to `d2` inclusive.

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

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

Deallocates the memory that was allocated for the creation of `M`, of type `BMatr`, at a previous call to `BCreateMatr` with parameters `d1` and `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 BSupNormLR (VAR V : BVect; n : BDim) : LONGREAL;
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 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 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 LIToBScal (i : LONGINT; VAR S : BScal);
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, 1988.
- [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  $2^\beta$ : 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, 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] A. Grube. Mehrfach rekursiv-erzeugte Pseudo-Zufallszahlen. *Zeitschrift für angewandte Mathematik und Mechanik*, 53:T223–T225, 1973.
- [16] D. Jäger. *MCS Modula-2 Cross System, User's Guide*. La Chanenche, France, 1992.
- [17] D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, Reading, Mass., second edition, 1981.
- [18] D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, Reading, Mass., third edition, 1998.
- [19] C. Koç. Recurring-with-carry sequences. *Journal of Applied Probability*, 32:966–971, 1995.
- [20] P. L'Ecuyer. Random numbers for simulation. *Communications of the ACM*, 33(10):85–97, 1990.
- [21] P. L'Ecuyer. Uniform random number generation. *Annals of Operations Research*, 53:77–120, 1994.
- [22] P. L'Ecuyer. Combined multiple recursive random number generators. *Operations Research*, 44(5):816–822, 1996.
- [23] P. L'Ecuyer. Random number generation. In Jerry Banks, editor, *Handbook of Simulation*, pages 93–137. Wiley, 1998. chapter 4.
- [24] P. L'Ecuyer. Tables of linear congruential generators of different sizes and good lattice structure. *Mathematics of Computation*, 68(225):249–260, 1999.
- [25] P. L'Ecuyer, F. Blouin, and R. Couture. A search for good multiple recursive random number generators. *ACM Transactions on Modeling and Computer Simulation*, 3(2):87–98, 1993.
- [26] 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.
- [27] 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, Ste-Foy (Que.), Canada, 1988.
- [28] P. L'Ecuyer and S. Tezuka. Structural properties for two classes of combined random number generators. *Mathematics of Computation*, 57(196):735–746, 1991.
- [29] G. Marsaglia. Yet another rng. Posted to the electronic billboard `sci.stat.math`, August 1, 1994.
- [30] George 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.
- [31] H. Niederreiter. A pseudorandom vector generator based on finite field arithmetic. *Mathematica Japonica*, 31:759–774, 1986.

- [32] 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, 1994.