MATHEMATICS OF COMPUTATION
Volume 00, Number 0, Pages 000–000
S 0025-5718(XX)0000-0

COMPUTATIONALLY EASY, SPECTRALLY GOOD MULTIPLIERS FOR CONGRUENTIAL PSEUDORANDOM NUMBER GENERATORS

GUY STEELE AND SEBASTIANO VIGNA

ABSTRACT. Congruential pseudorandom number generators rely on good *multipliers*, that is, integers that have good performance with respect to the spectral test. We provide lists of multipliers with a good lattice structure up to dimension eight for generators with typical power-of-two moduli, analyzing in detail multipliers close to the square root of the modulus, whose product can be computed quickly.

1. INTRODUCTION

A *multiplicative congruential pseudorandom number generator* (MCG) is a computational process defined by a recurrence of the form

$$x_n = \bigl(ax_{n-1}\bigr) \bmod m,$$

where $m \in \mathbf{Z}$ is the *modulus*, $a \in \mathbf{Z}/m\mathbf{Z}$ is the *multiplier*, and $x_n \in (\mathbf{Z}/m\mathbf{Z}) \setminus \{0\}$ is the *state* of the generator after step n. Such pseudorandom number generators (PRNGs) were introduced by Lehmer [11], and have been extensively studied. By adding a constant $c \in \mathbf{Z}/m\mathbf{Z}$, $c \neq 0$, we obtain a *linear congruential pseudorandom number generator* (LCG), with state $x_n \in \mathbf{Z}/m\mathbf{Z}$:[1]

$$x_n = \bigl(ax_{n-1} + c\bigr) \bmod m.$$

Under suitable conditions on m, a and c, sequences of this kind are periodic and their period is *full*, that is, $m-1$ for MCGs ($c = 0$) and m for LCGs ($c \neq 0$). For MCGs, m must be prime and a must be a *primitive element* of the multiplicative group of residue classes $(\mathbf{Z}/m\mathbf{Z})^\times$ (i.e., its powers must span the whole group). For LCGs, there are simple conditions that must be satisfied [9, §3.2.1.2, Theorem A].

For MCGs, when m is not prime one can look for sequences that have *maximum period*, that is, the longest possible period, given m. We will be interested in moduli that are powers of two, in which case, if $m \geq 8$, the maximum period is $m/4$, and the state must be odd [9, §3.2.1.2, Theorem B].

[1]We remark that these denominations, by now used for half a century, are completely wrong from a mathematical viewpoint. The map $x \mapsto ax$ is indeed a *linear* map, but the map $x \mapsto ax+c$ is an *affine* map [2]: what we call an "MCG" or "MLCG" should called an "LCG" and what we call an "LCG" should be called an "ACG". The mistake originated probably in the interest of Lehmer in (truly) linear maps with prime moduli [11]. Constants were added later to obtain long-period generators with non-prime moduli, but the "linear" name stuck (albeit some authors are using the term "mixed" instead of "linear"). At this point it is unlikely that the now-traditional names will be corrected.

While MCGs and LCGs have some known defects, they can be used in combination with other pseudorandom number generators (PRNGs), or passed through some bijective function that might lessen such defects. Due to their speed and simplicity, as well as a substantial accrued body of mathematical analysis, they have been for a long time the PRNGs of choice in programming languages.

In this paper, we provide lists of multipliers for both MCGs and LCGs, continuing the line of work by L'Ecuyer in his classic paper [10]. The quality of such multipliers is usually assessed by their score in the *spectral test*, described below.

The search for good multipliers is a sampling process from a large space: due to the enormous increase in computational power in the last twenty years, we can now provide multipliers with significantly improved scores. In fact, for multipliers of up to 35 bits we have now explored the sample space exhaustively.

We consider only generators with power-of-two moduli; this choice avoids the expensive modulo operation, because nearly all contemporary hardware supports binary arithmetic that is naturally carried modulo 2^w for some *word size* w. Such generators do have additional known, specific defects (e.g., the periods of the lowest bits are very short, and the flip of a state bit will never propagate to lower bits), but there is a substantial body of literature on how to ameliorate or avoid these defects.

Furthermore, in this paper we pay special attention to *small* multipliers, that is, multipliers close to the square root of the modulus m. For $m = 2^{2w}$, this means multipliers whose size in bits is $w \pm k$ for small k. As is well known, many CPUs with natural word size w can produce with a single instruction, or two instructions, the full $2w$-bit product of two w-bit operands, which makes such multipliers attractive from a computational viewpoint.

Unfortunately, such small multipliers have known additional defects, which have been analyzed by Hörmann and Derflinger [7], who provided experimental evidence of their undesirable behavior using a statistical test based on rejection.

One of the goals of this paper is to deepen their analysis: we first prove theoretically that w-bit multipliers for LCGs with power-of-two modulus 2^{2w} have inherent theoretical defects. Then we show that these defects are ameliorated as we add bits to the multiplier, and we quantify this improvement by defining a new *figure of merit* based on the magnitude on the multiplier. In the end, we provide tables of multipliers of $w + k$ bits, where k is relatively small, with quality closer to that of full $2w$-bit multipliers.

During the search of good multipliers, the authors have accumulated a large database of candidates, which is publicly available for download, in case the reader is interested in looking for multipliers with specific properties. The software used to search for multipliers is available under the [choose license].

2. Spectral figures of merit

For every integer $d \geq 2$, the *dimension*, we can consider the set of d-dimensional points in the unit cube

$$\Lambda_d = \left\{ \left(\frac{x}{m}, \frac{f(x)}{m}, \frac{f^2(x)}{m}, \dots, \frac{f^{d-1}(x)}{m} \right) \,\middle|\, x \in \mathbf{Z}/m\mathbf{Z} \right\},$$

where

$$f(x) = (ax + c) \bmod m$$

is the next-state map of a full-period generator. This set is the intersection of a d-dimensional *lattice* with the unit cube [9, §3.3.4.A]. Thus, all points in Λ_d lie on a family of equidistant, parallel *hyperplanes*; in fact, there are many such families.

The *spectral test* examines the family with the largest distance between adjacent hyperplanes: the smaller this *largest interplane distance* is, the more evenly the generator fills the unit d-dimensional cube. Using this idea, the *figure of merit* for dimension d of an MCG or LCG is defined as

$$f_d(m, a) = \frac{\nu_d}{\gamma_d^{1/2}\sqrt[d]{m}},$$

where $1/\nu_d$ is the largest distance between adjacent hyperplanes found by considering all possible families of hyperplanes covering Λ_d. We will usually imply the dependency on the choice of m and a.

The definition of f_d also relies on the *Hermite constant* γ_d for dimension d. For $2 \le d \le 8$, the Hermite constant has these values:

$$\gamma_2 = (4/3)^{1/2}, \gamma_3 = 2^{1/3}, \gamma_4 = 2^{1/2}, \gamma_5 = 2^{3/5}, \gamma_6 = (64/3)^{1/6}, \gamma_7 = 4^{3/7}, \gamma_8 = 2.$$

For all higher dimensions except $d = 24$ only upper and lower bounds are known. Note that $1/\bigl(\gamma_d^{1/2}\sqrt[d]{m}\bigr)$ is the smallest possible such largest interplane distance [9, §3.3.4.E, equation (40)]; it follows that $0 < f_d \le 1$.

The reason for expressing the largest interplane distance in the form of a reciprocal $1/\nu_d$ is that ν_d is the *length of the shortest vector in the dual lattice* Λ_d^*. The dual lattice consists of all vectors whose scalar product with every vector of the original lattice is an integer. In particular, it has the following basis [9, §3.3.4.C]:

$$\begin{gathered}(m, 0, 0, 0, \ldots, 0, 0)\\ (-a, 1, 0, 0, \ldots, 0, 0)\\ (-a^2, 0, 1, 0, \ldots, 0, 0)\\ \vdots \qquad \vdots\\ (-a^{d-2}, 0, 0, 0, \ldots, 1, 0)\\ (-a^{d-1}, 0, 0, 0, \ldots, 0, 1)\end{gathered}$$

That is, Λ_d^* is formed by taking all possible linear combinations of the vectors above with integer coefficients. Note that the constant c of an LCG has no role in the structure of Λ_d and Λ_d^*, and that we are under a full-period assumption.

The dual lattice is somewhat easier to work with, as its points have all integer coordinates; moreover, as we mentioned, if we call ν_d the length of its shortest vector, the maximum distance between parallel hyperplanes covering Λ_d is $1/\nu_d$ (and, indeed, this is how the figure of merit f_d is computed).

3. Computationally easy multipliers

Multipliers smaller than $\sqrt{m}$ have been advocated, in particular when the modulus is a power of two, say $m = 2^{2w}$, because they do not require a full $2w$-bit multiplication: writing $x_{\text{\_}}$ and $x^{\text{-}}$ for the w lowest and highest bits, respectively, of a $2w$-bit value x (that is, $x_{\text{\_}} = x \bmod 2^w$ and $x^{\text{-}} = \lfloor x/2^w \rfloor$), we have

$$(ax) \bmod 2^{2w} = \bigl(ax_{\text{\_}} + a \cdot 2^w x^{\text{-}}\bigr) \bmod 2^{2w} = \bigl(ax_{\text{\_}} + 2^w \cdot ax^{\text{-}}\bigr) \bmod 2^{2w}.$$

The first multiplication, ax_{-}, has a $2w$-bit operand a and a w-bit operand x_{-}, and in general the result may be $2w$ bits wide; but the second multiplication, ax^{-}, can be performed by an instruction that takes two w-bit operands and produces only a w-bit result that is only the low w bits of the full product, because the modulo operation effectively discards the high w bits of that product. Moreover, if the multiplier $a = 2^w a^{-} + ax^{-}$ has a high part that is small (say, $a^{-} < 256$) or of a special form (for example, $a^{-} = j2^n$ where j is 1, 3, 5, or 9), then the first multiplication may also be computed using a faster method. Contemporary optimizing compilers know how to exploit such special cases, perhaps by using a small immediate operand rather than loading the entire multiplier into a register, or perhaps by using shift instructions and/or such instructions as `lea` (Load Effective Address), which in the Intel 64-bit architecture may be used to compute $x + jy$ on two 64-bit operands x and y for $j = 2$, 4, or 8 [8, p. 3-554]. And even if the compiler produces the same code for, say, a multiplier that is $(3/2)w$ bits wide as for a multiplier that is $2w$ bits wide, some hardware architectures may notice the smaller multiplier on the fly and handle it in a faster way.

Multiplication by a constant a of size w, that is, of the form a_{-} (in other words, $a^{-} = 0$), is especially simple:

$$a_{-}x \bmod 2^{2w} = \left(a_{-}x_{-} + 2^w a_{-}x^{-}\right) \bmod 2^{2w}$$

and notice that the addition can be performed as a w-bit addition of the low w bits of $a_{-}x^{-}$ into the high half of $a_{-}x_{-}$.

In comparison, multiplication by a constant a of size $w+1$, that is, of the form $2^w + a_{-}$ (in other words, $a^{-} = 1$), requires only one extra addition:

$$\begin{aligned}\left(\left(2^w + a_{-}\right)x\right) \bmod 2^{2w} &= \left(\left(2^w + a_{-}\right)x_{-} + \left(2^w + a_{-}\right)\left(x^{-} \cdot 2^w\right)\right) \bmod 2^{2w} = \\ &\left(2^w x_{-} + a_{-}x_{-} + 2^w \cdot a_{-}x^{-}\right) \bmod 2^{2w} = \left(a_{-}x_{-} + 2^w \cdot \left(x_{-} + a_{-}x^{-}\right)\right) \bmod 2^{2w}.\end{aligned}$$

Modern compilers know the reduction above and will reduce the strength of operations involved as necessary.

Even without the help of the compiler, we can push this idea further to multipliers of the form $2^{w+k} + a$, where k is a small positive integer constant:

$$\begin{aligned}\left(\left(2^{w+k} + a\right)x\right) \bmod 2^{2w} &= \left(\left(2^{w+k} + a\right)x_{-} + \left(2^{w+k} + a\right)\left(x^{-} \cdot 2^w\right)\right) \bmod 2^{2w} = \\ &\left(2^{w+k}x_{-} + ax_{-} + 2^w \cdot ax^{-}\right) \bmod 2^{2w} = \left(ax_{-} + 2^w \cdot \left(2^k x_{-} + ax^{-}\right)\right) \bmod 2^{2w}.\end{aligned}$$

In comparison to the $(w+1)$-bit case, we just need an additional shift to compute $2^k x_{-}$. In the interest of efficiency, it thus seems interesting to study in more detail the quality of small multipliers.

In Figure 1 we show code generated by the `clang` compiler that uses 64-bit instructions to multiply a 128-bit value (in registers `rsi` and `rdi`) by (whimsically chosen) constants of various sizes. The first example shows that if the constant is of size 64, indeed only two 64-bit by 64-bit multiply instructions (one producing a 128-bit result and the other just a 64-bit result) and one 64-bit add instruction are needed. The second example shows that if the constant is of size 65, indeed only one extra 64-bit add instruction is needed. For constants of size 66 and above, more sophisticated strategies emerge that use `leaq` (the quadword, that is, 64-bit form of `lea`) and shift instructions and even subtraction. In Figure 2 we show three examples of code generated by `clang` for the ARM processor: since its RISC architecture [1] can only load constant values 16 bits at a time, the length of the sequence of instructions grows as the multiplier size grows. On the other hand, note that the ARM architecture has a multiply-add instruction `madd`.

4. Bounds

Our first result says that if the multiplier is smaller than the root of order d of the modulus, there is an upper bound to the value that the figure of merit f_d can attain:

Theorem 4.1. *Consider a full-period generator with modulus m and multiplier a. Then, for every $d \geq 2$, if $a < \sqrt[d]{m}$ we have $\nu_d = \sqrt{a^2+1}$, and it follows that*

$$f_d = \frac{\sqrt{a^2+1}}{\gamma_d^{1/2}\sqrt[d]{m}}$$

Proof. The length ν_d of the shortest vector of the dual lattice Λ_d^* can be easily written as

$$\nu_d = \min_{(x_0,\ldots,x_{d-1})\neq(0,\ldots,0)} \Big\{ \sqrt{x_0^2+x_1^2+\cdots+x_{d-1}^2} \;\Big|\; x_0 + a x_1 + a^2 x_2 + \cdots + a^{t-1}x_{t-1} \equiv 0 \mod m \Big\}, \tag{4.1}$$

where $(x_0,\ldots,x_{d-1}) \in \mathbf{Z}^d$, due to the simple structure of the basis of Λ_d^* [9, §3.3.4]. Clearly, in general $\nu_d \leq \sqrt{a^2+1}$, as $(-a,1,0,0,\ldots,0) \in \Lambda_d^*$. However, when $a < \sqrt[d]{m}$ we have $\nu_d = \sqrt{a^2+1}$, as no vector shorter than $\sqrt{a^2+1}$ can fulfill the modular condition.

To prove this statement, note that a vector $(x_0,\ldots,x_{d-1}) \in \Lambda_d^*$ shorter than $\sqrt{a^2+1}$ must have all coordinates smaller than a in absolute value (if one coordinate has absolute value a, all other coordinates must be zero, and the vector cannot belong to Λ_d^*). Then, for every $0 \leq j < d$

$$\left|\sum_{i=0}^{j} x_i a^i\right| \leq \sum_{i=0}^{j} |x_i| a^j < a^{j+1} < m,$$

so the modular condition in (4.1) must be fulfilled by equality with zero. However, let t be the index of the last nonzero component of $(x_0,\ldots,x_{d-1})$ (i.e., $x_i = 0$ for $i > t$): then, $\left|\sum_{i=0}^{t-1} x_i a^i\right| < a^t$, whereas $|x_t a^t| \geq a^t$, so their sum cannot be zero. □

Note that if $m = a^d$, then the vector that is a in position $d-1$ and zero elsewhere is in Λ_d^*, but by the proof above shorter vectors cannot be, so

$$f_d = \frac{a}{\gamma_d^{1/2}\sqrt[d]{m}} = \frac{1}{\gamma_d^{1/2}}.$$

Using the approximation $\sqrt{a^2+1} \approx a$, this means that if $a \leq \sqrt[d]{m}$ then for $2 \leq d \leq 8$, f_d cannot be greater than approximately

$$(4/3)^{-1/4} \approx 0.9306, \quad 2^{-1/6} \approx 0.8909, \quad 2^{-1/4} \approx 0.8409, \quad 2^{-6/10} \approx 0.8122,$$
$$(64/3)^{-1/12} \approx 0.7749, \quad 4^{-3/14} \approx 0.7430, \quad 2^{-1/2} \approx 0.7071$$

Bits	Multiplier	Code

64	`0xCAFEF00DDEADF00D`	`movabsq $0xCAFEF00DDEADF00D, %rax`
		`imulq   %rax, %rsi`
		`mulq    %rdi`
		`addq    %rsi, %rdx`

65	`0x1CAFEF00DDEADF00D`	`movabsq $0xCAFEF00DDEADF00D, %rcx`
		`imulq   %rcx, %rsi`
		`mulq    %rcx`
		`addq    %rdi, %rdx`
		`addq    %rsi, %rdx`

66	`0x2CAFEF00DDEADF00D`	`movabsq $xCAFEF00DDEADF00D, %rcx`
		`imulq   %rcx, %rsi`
		`mulq    %rcx`
		`leaq    (%rdx,%rdi,2), %rdx`
		`addq    %rsi, %rdx`

67	`0x4CAFEF00DDEADF00D`	`movabsq $0xCAFEF00DDEADF00D, %rcx`
		`imulq   %rcx, %rsi`
		`mulq    %rcx`
		`leaq    (%rdx,%rdi,4), %rdx`
		`addq    %rsi, %rdx`

67	`0x5CAFEF00DDEADF00D`	`movabsq $0xCAFEF00DDEADF00D, %rcx`
		`mulq    %rcx`
		`imulq   %rcx, %rsi`
		`leaq    (%rdi,%rdi,4), %rcx`
		`addq    %rcx, %rdx`
		`addq    %rsi, %rdx`

67	`0x7CAFEF00DDEADF00D`	`movabsq $0xCAFEF00DDEADF00D, %r8`
		`mulq    %r8`
		`leaq    (,%rdi,8), %rcx`
		`subq    %rdi, %rcx`
		`addq    %rcx, %rdx`
		`imulq   %r8, %rsi`
		`addq    %rsi, %rdx`

96	`0xFADC0C0ACAFEF00DDEADF00D`	`movl    $0xFADC0C0A, %ecx`
		`movabsq $0xCAFEF00DDEADF00D, %r8`
		`mulq    %r8`
		`imulq   %rdi, %rcx`
		`addq    %rcx, %rdx`
		`imulq   %r8, %rsi`
		`addq    %rsi, %rdx`

128	`0xAB0DE0FBADC0FFEECAFEF00DDEADF00D`	`movabsq $0xAB0DE0FBADC0FFEE, %rcx`
		`movabsq $0xCAFEF00DDEADF00D, %r8`
		`mulq    %r8`
		`imulq   %rdi, %rcx`
		`addq    %rcx, %rdx`
		`imulq   %r8, %rsi`
		`addq    %rsi, %rdx`

FIGURE 1. `clang`-generated Intel code for the multiplication part of a 128-bit LCG using multipliers of increasing size. The code generated for more than 96 bits (not shown here) is identical to the 128-bit case.

Bits	Multiplier	Code

64	`0xCAFEF00DDEADF00D`	`mov x8, #0xF00D`
		`movk x8, #0xDEAD, lsl #16`
		`movk x8, #0xF00D, lsl #32`
		`movk x8, #0xCAFE, lsl #48`
		`umulh x9, x0, x8`
		`madd x1, x1, x8, x9`
		`mul x0, x0, x8`

65	`0x1CAFEF00DDEADF00D`	`mov x8, #0xF00D`
		`movk x8, #0xDEAD, lsl #16`
		`movk x8, #0xF00D, lsl #32`
		`movk x8, #0xCAFE, lsl #48`
		`umulh x9, x0, x8`
		`add x9, x9, x0`
		`madd x1, x1, x8, x9`
		`mul x0, x0, x8`

67	`0x7CAFEF00DDEADF00D`	`mov x8, #0xF00D`
		`movk x8, #0xDEAD, lsl #16`
		`movk x8, #0xF00D, lsl #32`
		`movk x8, #0xCAFE, lsl #48`
		`lsl x9, x0, #3`
		`umulh x10, x0, x8`
		`sub x9, x9, x0`
		`add x9, x10, x9`
		`madd x1, x1, x8, x9`
		`mul x0, x0, x8`

96	`0xFADC0C0ACAFEF00DDEADF00D`	`mov x8, #0xF00D`
		`movk x8, #0xDEAD, lsl #16`
		`movk x8, #0xF00D, lsl #32`
		`movk x8, #0xCAFE, lsl #48`
		`mov w9, #0x0C0A`
		`movk w9, #0xFADC, lsl #16`
		`umulh x10, x0, x8`
		`madd x9, x0, x9, x10`
		`madd x1, x1, x8, x9`
		`mul x0, x0, x8`

128	`0xAB0DE0FBADC0FFEECAFEF00DDEADF00D`	`mov x9, #0xF00D`
		`mov x8, #0xFFEE`
		`movk x9, #0xDEAD, lsl #16`
		`movk x8, #0xADC0, lsl #16`
		`movk x9, #0xF00D, lsl #32`
		`movk x8, #0xE0FB, lsl #32`
		`movk x9, #0xCAFE, lsl #48`
		`movk x8, #0xAB0D, lsl #48`
		`umulh x10, x0, x9`
		`madd x8, x0, x8, x10`
		`madd x1, x1, x9, x8`
		`mul x0, x0, x9`

FIGURE 2. `clang`-generated ARM code for the multiplication part of a 128-bit LCG using multipliers of increasing size. Note how the number of `mov` and `movk` instructions depends on the size of the multiplier.

for $d = 2, \ldots, 8$. For $d > 2$ this is not a problem, as such very small multipliers are not commonly used. However, choosing a multiplier that is smaller than or equal to $\sqrt{m}$ has the effect of making it impossible to obtain a figure of merit close to 1 in dimension 2. Note that, for any d, as a drops well below $\sqrt[d]{m}$ the figure of merit f_d degenerates quickly; for example, if $a < \sqrt{m}/2$ then f_2 cannot be greater than $(4/3)^{-1/4}/2 \approx 0.4653$.

Nonetheless, as soon as we allow a to be even a tiny bit larger than $\sqrt{m}$, ν_2 (and thus f_2) is no longer constrained: indeed, if $m = 2^{2w}$, a $(w+1)$-bit multiplier is sufficient to get a figure of merit in dimension 2 very close to 1 (see Table 1).

MCGs with power-of-two moduli cannot achieve full period: the maximum period is $m/4$. It turns out that the lattice structure, however, is very similar to the full-period case, once we replace m with $m/4$ in the definition of the dual lattice. Correspondingly, we have to replace $\sqrt[d]{m}$ with $\sqrt[d]{m/4}$ (see [9, §3.3.4, Exercise 20]):

Theorem 4.2. *Consider an MCG with power-of-two modulus m, multiplier a, and period $m/4$. Then for every $d \geq 2$ and every $a < \sqrt[d]{m/4}$ we have $\nu_d = \sqrt{a^2+1}$, and it follows that*

$$f_d = \frac{\sqrt{a^2+1}}{\gamma_d^{1/2}\sqrt[d]{m/4}}.$$

Note that Theorem 4.2 imposes limits on the figures of merit for $(w-1)$-bit multipliers for $2w$-bit MCGs, but does not impose any limits on w-bit multipliers for $2w$-bit MCGs. In Table 2, observe that the 31-bit multipliers necessarily have figures of merit f_2 smaller than $(4/3)^{-1/4} \approx 0.9306$ (though one value for f_2, namely 0.930577, is quite close), but for multipliers of size 32 and greater we have been able to choose examples for which f_2 is well above 0.99.

5. Beyond spectral scores

In view of Theorem 4.1, it would seem that using a $(w+1)$-bit multiplier gives us the full power of a $2w$-bit multiplier: or, at least, this is what the spectral scores suggest empirically. We now show that, however, on closer inspection, the spectral scores are not telling the whole story.

Hörmann and Derflinger [7] studied multipliers close to the square root to the modulus for LCGs with 32 bits of state, and devised a statistical test that makes generators using such multipliers fail: the intuition behind the test is that with such multipliers there is a relatively short lattice vector $\boldsymbol{s} = (1/m, a/m) \in \Lambda_2$ that is almost parallel to the y axis. The existence of this vector creates bias in pairs of consecutive outputs, a bias that can be detected by generating a distribution using the rejection method: if at some point the density of the distribution increases sharply, the rejection method will underrepresent certain parts of the distribution and overrepresent others.

We applied an instance of the Hörmann–Derflinger test to congruential generators (both LCG and MCG) with 64 bits of state using a Cauchy distribution on the interval $[-2\,..\,2)$. We divide the interval into 10^8 slots that contain the same probability mass, repeatedly generate by rejection 10^9 samples from the distribution, and compute a p-value using a χ^2 test on the slots. We consider the number of repetitions after which the p-value is very close to zero[2] a measure of the resilience

[2]More precisely, when the p-value returned by the Boost library implementation of the χ^2 test becomes zero, which in this case happens when the p-value goes below $\approx 10^{-16}$.

of the multiplier to the Hörmann–Derflinger test, and thus a positive feature (that is, a larger number is better).

The results are reported in Tables 1 and 2. As we move from small to large multipliers, the number of iterations necessary to detect bias grows, but within multipliers with the same number of bits there is a very large variability.[3]

The marked differences have a simple explanation: incrementing the number of bits does not translate immediately into a significantly longer vector $\boldsymbol{s}$. To isolate generators in which $\boldsymbol{s}$ is less pathological, we have to consider larger multipliers, as $\|\boldsymbol{s}\| = \sqrt{a^2+1}/m$. In particular, we define the simple figure of merit λ for a full-period LCG as

$$\lambda = \frac{\|\boldsymbol{s}\|}{1/\sqrt{m}} = \frac{\sqrt{a^2+1}/m}{1/\sqrt{m}} = \frac{\sqrt{a^2+1}}{\sqrt{m}} \approx a/\sqrt{m}$$

In other words, we measure the length of $\boldsymbol{s}$ with respect to the threshold $1/\sqrt{m}$ of Theorem 4.1. In general, for a set of multipliers bounded by B, $\lambda \leq B/\sqrt{m}$.

Note that because of Theorem 4.1, if $a < \sqrt{m}$

$$f_2/\lambda = \frac{\sqrt{a^2+1}}{\gamma_2^{1/2}\sqrt{m}} \Big/ \frac{\sqrt{a^2+1}}{\sqrt{m}} = \gamma_2^{-1/2} \approx 0.9306,$$

that is, for multipliers smaller than $\sqrt{m}$ the two figures of merit f_2 and λ are linearly correlated. Just one additional bit, however, makes the two figures independent (see the entries for 33-bit multipliers in Table 1, as well as the entries for 32-bit multipliers in Table 2).

For MCGs with power-of-two modulus m, $\boldsymbol{s} = (4/m, 4a/m)$, and, in view of Theorem 4.2, we define

$$\lambda = \frac{\|\boldsymbol{s}\|}{1/\sqrt{m/4}} = \frac{\sqrt{a^2+1}/(m/4)}{1/\sqrt{m/4}} = \frac{\sqrt{a^2+1}}{\sqrt{m/4}} \approx 2a/\sqrt{m}$$

In Tables 1 and 2 we report a few small-sized multipliers together with the figures of merit f_2 and λ, as well as the number of iterations required by our use of the Hörmann–Derflinger test: larger values of λ (i.e., larger multipliers) correspond to more resilience to the test.

6. Potency

Potency is a property of multipliers of LCGs: it is defined as the minimum s such that $(a-1)^s$ is a multiple of the modulus m. Such an s always exists for full-period multipliers, because one of the conditions for full period is that $a-1$ be divisible by every prime that divides m (when m is a power of two, this simply means that a must be odd).

Multipliers of low potency generate sequences that do not look very random: in the case m is a power of two, this is very immediate, as a multiplier a with low potency is such that $a-1$ is divisible by a large power of two, say, 2^k. In this case, the k lowest bits of ax are the same as the k lowest bits of x, which means that changes to the k lowest bits of the state depend only on the fact that we add c.

[3] We also tested a generator with 128 bits of state and a 64-bit multiplier, but at that size the bias is undetectable even with a hundred times as many (10^{10}) slots.

Bits	a	f_2	λ	H–D
32	`0xfffeb28d`	0.930586	1.00	6
	`0xcffef595`	0.756102	0.81	4
33	`0x1dd23bba5`	0.998598	1.86	19
	`0x112a563ed`	0.998387	1.07	7
34	`0x3de4f039d`	0.998150	3.87	72
	`0x2cfe81d9d`	0.992874	2.81	46
35	`0x78ad72365`	0.995400	7.54	313
	`0x49ffd0d25`	0.991167	4.62	109

Table 1. A comparison of small LCG multipliers for $m = 2^{64}$. In the 32-bit case, f_2 and λ are linearly correlated, and f_2 is necessarily smaller than approximately 0.9306. For sizes above 32 we show multipliers with almost perfect f_2 but different λ. The last column shows the corresponding number of iterations of the Hörmann–Derflinger test.

Bits	a	f_2	λ	H–D
31	`0x7ffc9ef5`	0.930509	0.50	2
	`0x672a3fb5`	0.750046	0.40	1
32	`0xef912f85`	0.994558	0.94	4
	`0x89f353b5`	0.997577	0.54	2
33	`0x1f0b2b035`	0.996853	1.94	22
	`0x16aa7d615`	0.994427	1.42	11
34	`0x3c4b7aba5`	0.992314	3.77	81
	`0x2778c3815`	0.998339	2.47	37
35	`0x7d3f85c05`	0.998470	7.83	354
	`0x40dde345d`	0.996172	4.05	87

Table 2. A comparison of small MCG multipliers for $m = 2^{64}$. In the 31-bit case, f_2 and λ are linearly correlated, and f_2 is necessarily smaller than approximately 0.9306. For each size above 31 we show multipliers with almost perfect f_2 but different λ. The last column shows the corresponding number of iterations of the Hörmann–Derflinger test.

For this reason, one ordinarily chooses multipliers of maximum possible potency,[4] and since for full period if m is a multiple of four, then $a-1$ must be a multiple of four, we have to choose a so that $(a-1)/4$ is odd, that is, $a \bmod 8 = 5$.

Potency has an interesting interaction with the constant c, described for the first time by Durst [3] in response to proposals from Percus and Kalos [13] and Halton [6] to use different constants to generate different streams for multiple processors. If we take a multiplier a and a constant c, then for every $r \in \mathbf{Z}/m\mathbf{Z}$ the generator with multiplier a and constant $(a-1)r + c$ has the same sequence of the first one, up to addition with r. Indeed, if we consider sequences starting from x_0 and $y_0 = x_0 - r$, we have[5]

$$y_n = ay_{n-1} + (a-1)r + c = a(x_{n-1} - r) + (a-1)r + c = x_n - r.$$

That is, for a fixed multiplier a, the constants c are divided into classes by the equivalence relation of generating the same sequence up to an additive constant.

How many classes do exist? The answer depends on the potency of a, as it comes down to solving the modular equation

$$c' - c = (a-1)r$$

If a has low potency, this equation will be rarely solvable because there will be many equivalence classes: but for the specific case where m is a power of two and $a \bmod 8 = 5$, it turns out that there are just *two* classes: the class of constants that are congruent to 1 modulo 4, and the class of constants that are congruent to 3 modulo 4. All constants in the first class yield the sequence $x_n = ax_{n-1} + 1$, up to an additive constant, and all constants in the second class yield the sequence $x_n = ax_{n-1} - 1$, up to an additive constant. It follows that if one tries to use three (or more) different streams, even if one chooses different constants for the streams, at least two of the streams will be correlated.

If we are willing to weaken slightly our notion of equivalence, in this case we can extend Durst's considerations: if we consider sequences starting from x_0 and $y_0 = -x_0 + r$, then

$$y_n = ay_{n-1} - ((a-1)r + c) = a(-x_{n-1} + r) - (a-1)r - c = -x_n + r.$$

Thus, if we consider the equivalence relation of generating sequences that are the same up to an additive constant *and* possibly a sign change, then *all* sequences generated by a multiplier a of maximum potency for a power-of-two modulus m are the same, because to prove equivalence we now need to solve just *one* of the two modular equations

$$c' - c = (a-1)r \qquad \text{and} \qquad c' + c = (a-1)r,$$

and while the first equation is solvable when the residues of c and c' modulo 4 are the same, the second equation is solvable when the residues are different.

---

[4]Note that "maximum possible potency" is a quite rough statement, because potency is a very rough measure when applied to multipliers that are powers of primes: for example, when $m = 2^{2w}$ a generator with $a-1$ divisible by 2^w (but not by 2^{w+1}) and a generator with $a-1$ divisible by 2^{2w-1} have both potency 2, but in view of the discussion above their randomness is very different. More precisely, here we choose to consider only multipliers which leave unchanged that smallest possible number of lower bits.

[5]All remaining computations in this section are performed in $\mathbf{Z}/m\mathbf{Z}$.

7. Using spectral data from MCGs

The case of MCGs with power-of-two modulus is different from that of LCGs because the maximum possible period is of length $m/4$ [9, §3.2.1.2, Theorem C]. Thus, there are two distinct orbits (remember that the state must be odd). The nature of these orbits is, however, very different depending on whether the multiplier is congruent to 5 modulo 8 or to 3 modulo 8: let us say such multipliers are of *type* 5 and *type* 3, respectively.

For multipliers of type 5, each orbit is defined by the residue modulo 4 of the state (i.e., 1 or 3), whose value depends on the second-lowest bit.[6] Thus, the remaining upper bits (above the second) go through all possible $m/4$ values. More importantly, the lattice of points described by the upper bits is simply a translated version of the lattice Λ_d associated with the whole state, so the figures or merit we compute on Λ_d^* describe properties of the generator obtained by discarding the two lowest bits from the state. Indeed, for every MCG of type 5 there is an LCG with modulus $m/4$ that generates "the same sequence" if the two low-order bits of every value produced by the MCG are ignored [9, §3.2.1.2, Exercise 9].

For multipliers of type 3, instead, each orbit is defined by the residue modulo 8 of the state: one orbit alternates between residues 1 and 3, and one orbit alternates between 5 and 7.[7] In this case, there is no way to use the information we have about the lattice generated by the whole state to obtain information about the lattice generated by the part of state that is changing; indeed, there is again a correspondence with an LCG, but the correspondence involves an alternating sign (again, see [9, §3.2.1.2, Exercise 9]). For this reason, we (like L'Ecuyer [10]) will consider only MCG multipliers of type 5.

Note that a and $-a \bmod m = m - a$ have different residue modulo 8, but the same figures of merit [9, §3.2.1.2, Exercise 9]. Moreover, in the MCG case the lattice structure is invariant with respect to inversion modulo m, so for each multiplier its inverse modulo m has again the same figures of merit. In the end, for each multiplier a of maximum period $m/4$ there are three other related multipliers $a^{-1} \bmod m$, $(-a) \bmod m$ and $(-a^{-1}) \bmod m$ with the same figures of merit; of the four, two are of type 3, and two of type 5.

8. Tables

In this section we provide tables of good multipliers for 32, 64 and 128 bits of state, updating some of the lists in the classic paper by L'Ecuyer [10, Tables 4 and 5].

For LCGs, only multipliers a such that $a \bmod 8$ is either 1 or 5 achieve full period [9, §3.2.1.2, Theorem A], but we (like L'Ecuyer) consider only the case of maximum potency, that is, the case when $a \bmod 8$ is 5. For MCGs, as we already discussed in Section 7, we consider only multipliers of type 5. In the end, therefore, we consider in both cases (though for different reasons) only multipliers whose residue modulo 8 is 5.

For each multiplier, we considered figures of merit up to dimension 8, that is, we computed f_2, f_3, f_4, f_5, f_6, f_7, and f_8. For reasons of space, we present only f_2 through f_6 in the tables. We also present two different scores that summarize these

[6] This is a consequence of the fact that multipliers of type 5 do not change the two lowest bits.

[7] Multipliers of type 3 always leave the lowest bit and the third-lowest bit of the state unchanged.

figures of merit: the customary *minimum* spectral score (over all seven dimensions 2 through 8) and a novel *harmonic* spectral score (also over all seven dimensions 2 through 8). The tables present not only the multipliers with the best minimum spectral scores that we found and the multipliers with the best harmonic spectral scores that we found, but also multipliers that exhibit a good balance between the two scores, as described below.

Traditionally, when examining the figures of merit of the spectral test up to dimension d, the *minimum spectral score (up to dimension d)* is given by the *minimum* figure of merit over dimensions 2 through d. L'Ecuyer's paper [10] uses the notation $M_d(m, a)$ for this aggregate score for a generator with modulus m and multiplier a. We prefer to distinguish the minimum spectral scores of LCGs and MCGs, because the figures of merit f_d are computed differently for the two kinds of generator when the modulus is a power of two: we use the notation

$$\mathscr{M}_d^+(m, a) = \min_{2 \le i \le d} f_i(m, a)$$

to denote the minimum spectral score up to dimension d for an LCG, and we use the notation $\mathscr{M}_d^*(m, a)$ to denote the analogous score for an MCG.

The use of the minimum spectral score seems to have originated in the work of Fishman and Moore [5], where, however, no motivation for this choice is provided. The definition has been referred to and copied several times, but even Knuth argues that the importance of figures of merit decreases with dimension, and that "the values of ν_t for $t \ge 10$ seem to be of no practical significance whatsoever" [9, §3.3.4]. Therefore, while L'Ecuyer's paper reports three different minimum figures of merit $M_8(m, a)$, $M_{16}(m, a)$, and $M_{32}(m, a)$ for each multiplier, here we will report only $\mathscr{M}_8^+(m, a)$ or $\mathscr{M}_8^*(m, a)$.

The disadvantage of the minimum spectral score is that it tends to flatten the spectral landscape—it is easy, even using small multipliers, to get figures of merit up to dimension 8 greater than 0.77. But smaller dimensions should be given more importance, as a lower figure of merit in a lower dimension is more likely to have an impact on applications, and a multiplier with a very high minimum spectral score over $2 \le d \le 8$ may have an unremarkable value for, say, f_2.

We therefore suggest considering also a second aggregate figure of merit:

Definition 8.1. Let $f_i(m, a)$, $2 \le i \le d$, be the figures of merit of an LCG multiplier a with modulus m. Then, the *harmonic spectral score (up to dimension d)* of a with modulus M is given by

$$\mathscr{H}_d^+(m, a) = \frac{1}{H_{d-1}} \sum_{2 \le i \le d} \frac{f_i(m, a)}{i - 1},$$

where $H_n = \sum_{k=1}^{n} \frac{1}{k}$ is the n-th harmonic number.[8] Analogously, the notation $\mathscr{H}_d^*(m, a)$ denotes the harmonic spectral score (up to dimension d) for an MCG multiplier a with modulus m.

The effect of the harmonic spectral score is to weight each dimension progressively less, using weights $1, 1/2, 1/3, \ldots, 1/(d-1)$, and the sum is normalized so that the score is always between 0 and 1.

[8] We have used script letters $\mathscr{M}$ and $\mathscr{H}$ to denote spectral scores so that the harmonic spectral score function $\mathscr{H}_8$ will not be confused with the harmonic number H_8.

An example of the difference in sensitivity between the minimum spectral score and the harmonic spectral score is that the minimum spectral score is in practice not limited by Theorem 4.1; for example, the largest minimum spectral score of a 32-bit multiplier for a 64-bit LCG is 0.774103, and the largest minimum spectral score for a 33-bit multiplier is almost the same: 0.776120 (an increase of about 0.002). But the largest spectral harmonic score goes from 0.867371 for 32-bit multipliers to 0.890221 for 33-bit multipliers (an increase of almost 0.03), reflecting the fact that f_2 can get arbitrarily close to 1 (indeed, there are 33-bit multipliers for which $f_2 = 0.998598$).

Another empirical observation in favor of the harmonic spectral score is that as soon as we look into multipliers with a high harmonic score, we see that their minimum score can be chosen to be just a few percentage points below the best possible, but at the same time the low-dimensional figures of merit, which are more relevant, have an increase an order of magnitude larger. These empirical observations are based on multipliers of at most 35 bits, which we have enumerated and scored exhaustively, but the same phenomenon appears to happen in larger cases, which we have sampled randomly.

Following a suggestion by Entacher, Schell, and Uhl [4], we compute figures of merit using the implementation of the ubiquitous Lenstra–Lenstra–Lovász basis-reduction algorithm [12] provided by Shoup's NTL library [14]. For $m = 2^{64}$ and $m =^{128}$ we recorded in an output file all tested multipliers whose minimum spectral score is at least 0.70 (we used a lower threshold for $m = 2^{32}$). Overall we sampled approximately 6.5×10^{11} multipliers, enough to ensure that for each pair of modulus and multiplier size reported, we recorded at least one million multipliers. (In several cases we recorded as many as 1.5 million or even two million multipliers.) As a sanity check, we also used the same software to test multipliers of size 63 for LCGs with $m = 2^{128}$; as expected, in view of Theorem 4.1 and its consequences, a random sample of well over 10^{10} 63-bit candidates revealed *none* whose minimum spectral score is at least 0.70.

In theory, the basis returned by the algorithm is only approximate, but using a precision parameter $\delta = 1 - 10^{-9}$ we found only very rarely a basis that was not made of shortest vectors: we checked all multipliers we selected using the LatticeTester tool,[9] which performs an exhaustive search after basis-reduction preprocessing, and almost all approximated data we computed turned out to be exact; just a few cases (usually in high dimension) were slightly off, which simply means that we spuriously stored a few candidates with minimum below 0.70.

Besides half-width and full-width multipliers, we provide multipliers with up to three bits more than half-width for $m = 2^{32}$ and $m = 2^{64}$, and up to seven bits more than half-width for $m = 2^{128}$, as well as multipliers of three-fourths width (24 bits for $m = 2^{32}$, 48 bits for $m = 2^{64}$, 96 bits for $m = 2^{128}$), because these are experimentally often as fast as smaller multipliers. Additionally, we provide 80-bit multipliers for $m = 2^{128}$ because such multipliers can be loaded by the ARM processor with just five instructions, and on an Intel processor one can use a multiply instruction with an immediate 16-bit value.

For small multipliers, we try to find candidates with a good λ: in particular, we require that the second-most-significant bit be set. For larger multipliers, we consider only spectral scores, as the effect of a good λ becomes undetectable. Since

[9] `https://github.com/umontreal-simul/latticetester`

when we consider $(w+c)$-bit multipliers we select candidates larger than 2^{w+c-1}, in our tables $2^{c-1} \leq \lambda \leq 2^c$ for LCGs and $2^c \leq \lambda \leq 2^{c+1}$ for MCGs.

More precisely, for each type (LCG or MCG), every $m \in \{2^{32}, 2^{64}, 2^{128}\}$ and for every multiplier size (in bits) tested, we report (in Tables 3 through 10) four multipliers:

- the best multiplier by harmonic spectral score;
- the best multiplier by harmonic spectral score within the top millile of minimum spectral scores.
- the best multiplier by minimum spectral score;
- the best multiplier by minimum spectral score within the top millile of harmonic spectral scores.

In case the first-millile criterion provides a duplicate multiplier for a given score, we try the same strategy with the first *decimillile*, and mark the multiplier with an asterisk, or with the first *centimillile*, marking with two asterisks, and so on.

The rationale for these reporting criteria is that the best score gives an idea of how far we went in our sampling procedure, but in principle the best score within the first millile of the alternative score gives a more balanced multiplier: indeed, within every list of four, the *second* multiplier (best multiplier by harmonic spectral score within the top millile of minimum spectral scores) is our favorite candidate.

All multipliers we provide are *Pareto optimal* for our dataset: that is, for each type, modulus, and size there is no other multiplier we examined that is at least as good on both scores, and strictly improves one. In particular, for each type, modulus, and size, the multipliers with distinct scores are pairwise incomparable (i.e., for each pair, the harmonic spectral score increases and the minimum spectral score decreases, or vice versa).

Bits	$\mathscr{H}_8^+(m,a)$	$\mathscr{M}_8^+(m,a)$	a	f_2	f_3	f_4	f_5	f_6	λ
16	<u>0.8219</u>	0.6374	`0xfb85`	0.9143	0.8680	0.6484	0.8172	0.7354	0.98
	<u>0.7696</u>	0.7002	`0xd09d`	0.7583	0.7760	0.8228	0.8101	0.7293	0.81
	0.7696	<u>0.7002</u>	`0xd09d`	0.7583	0.7760	0.8228	0.8101	0.7293	0.81
	0.8219	<u>0.6374</u>	`0xfb85`	0.9143	0.8680	0.6484	0.8172	0.7354	0.98
17	<u>0.8432</u>	0.5813	`0x1d6cd`	0.9778	0.7842	0.8273	0.7970	0.7439	1.84
	<u>0.8065</u>	0.6731	`0x19c05`	0.9083	0.8057	0.7119	0.7306	0.7369	1.61
	0.8065	<u>0.6731</u>	`0x19c05`	0.9083	0.8057	0.7119	0.7306	0.7369	1.61
	0.8432	<u>0.5813</u>	`0x1d6cd`	0.9778	0.7842	0.8273	0.7970	0.7439	1.84
18	<u>0.8391</u>	0.6961	`0x3956d`	0.9477	0.8807	0.7409	0.6961	0.7439	3.58
	<u>0.7478</u>	0.7226	`0x342dd`	0.7692	0.7360	0.7265	0.7226	0.7478	3.26
	0.7478	<u>0.7226</u>	`0x342dd`	0.7692	0.7360	0.7265	0.7226	0.7478	3.26
	0.8391	<u>0.6961</u>	`0x3956d`	0.9477	0.8807	0.7409	0.6961	0.7439	3.58
19	<u>0.8492</u>	0.5873	`0x6ebd5`	0.9601	0.8854	0.8637	0.5873	0.7062	6.92
	<u>0.7870</u>	0.6960	`0x7c8a5`	0.8405	0.7499	0.8023	0.7497	0.7654	7.78
	0.7622	<u>0.7052</u>	`0x6d7f5`	0.7700	0.8063	0.7347	0.7052	0.7755	6.84
	0.8295	<u>0.6462</u>	`0x7e57d`	0.9458	0.9319	0.6462	0.7213	0.6675	7.90
24	<u>0.8682</u>	0.6219	`0xe027a5`	0.9806	0.9001	0.8145	0.7783	0.7429	224.15
	<u>0.8371</u>	0.7224	`0xc083c5`	0.9397	0.8395	0.7224	0.7268	0.7976	192.51
	0.8174	<u>0.7448</u>	`0xca7b35`	0.8900	0.7489	0.7847	0.8048	0.7934	202.48
	0.8489	<u>0.7061</u>	`0xe8fd45`	0.9311	0.8898	0.8340	0.7061	0.7464	232.99
32	<u>0.8875</u>	0.7289	`0xadb4a92d`	0.9759	0.9362	0.7558	0.8776	0.7513	4.4×10^4
	<u>0.8486</u>	0.7552	`0xa13fc965`	0.8969	0.8469	0.8044	0.8452	0.7939	4.1×10^4
	0.7925	<u>0.7649</u>	`0x8664f205`	0.7996	0.7987	0.7792	0.7945	0.7649	3.4×10^4
	0.8720	<u>0.7395</u>	`0xcf019d85`	0.9512	0.8868	0.9037	0.7427	0.7474	5.3×10^4

TABLE 3. Good multipliers for LCGs with $m = 2^{32}$.

Bits	$\mathscr{H}_8^*(m,a)$	$\mathscr{M}_8^*(m,a)$	a	f_2	f_3	f_4	f_5	f_6	λ
15	<u>0.8101</u>	0.6443	`0x7dc5`	0.9144	0.7346	0.7798	0.7887	0.6443	0.98
	<u>0.7629</u>	0.6487	`0x756d`	0.8537	0.7045	0.7223	0.7400	0.7110	0.92
	0.7502	<u>0.6814</u>	`0x72ed`	0.8356	0.6814	0.6978	0.7213	0.6909	0.90
	0.8101	<u>0.6443</u>	`0x7dc5`	0.9144	0.7346	0.7798	0.7887	0.6443	0.98
16	<u>0.8211</u>	0.5625	`0xf7b5`	0.9642	0.7428	0.7486	0.7958	0.7526	1.94
	<u>0.8141</u>	0.6665	`0xc075`	0.8897	0.8276	0.8297	0.6665	0.7077	1.50
	0.8141	<u>0.6665</u>	`0xc075`	0.8897	0.8276	0.8297	0.6665	0.7077	1.50
	0.8211	<u>0.5625</u>	`0xf7b5`	0.9642	0.7428	0.7486	0.7958	0.7526	1.94
17	<u>0.8336</u>	0.5294	`0x1d205`	0.9666	0.8503	0.8361	0.8077	0.5294	3.64
	<u>0.7926</u>	0.6772	`0x1c77d`	0.8593	0.7686	0.8230	0.7018	0.7321	3.56
	0.7926	<u>0.6772</u>	`0x1c77d`	0.8593	0.7686	0.8230	0.7018	0.7321	3.56
	0.8336	<u>0.5294</u>	`0x1d205`	0.9666	0.8503	0.8361	0.8077	0.5294	3.64
18	<u>0.8381</u>	0.6914	`0x305d5`	0.9534	0.8630	0.7066	0.6914	0.7401	6.05
	<u>0.7988</u>	0.6963	`0x3c965`	0.8486	0.8932	0.7034	0.6963	0.7216	7.57
	0.7788	<u>0.7134</u>	`0x31e2d`	0.8116	0.7419	0.8130	0.7960	0.7134	6.24
	0.8381	<u>0.6914</u>	`0x305d5`	0.9534	0.8630	0.7066	0.6914	0.7401	6.05
19	<u>0.8467</u>	0.5506	`0x7ecc5`	0.9852	0.9230	0.7840	0.5506	0.7409	15.85
	<u>0.7982</u>	0.6808	`0x728cd`	0.8905	0.7981	0.6808	0.7472	0.7110	14.32
	0.7106	<u>0.6838</u>	`0x6be35`	0.6878	0.6838	0.7657	0.7570	0.7369	13.49
	0.8428	<u>0.5625</u>	`0x76e3d`	0.9574	0.8667	0.8285	0.7575	0.6780	14.86
24	<u>0.8615</u>	0.6620	`0xc00e35`	0.9896	0.8766	0.7794	0.7114	0.8090	384.11
	<u>0.8495</u>	0.7433	`0xc7fb6d`	0.9428	0.8251	0.7561	0.7964	0.8169	399.96
	0.8495	<u>0.7433</u>	`0xc7fb6d`	0.9428	0.8251	0.7561	0.7964	0.8169	399.96
	0.8615	<u>0.6620</u>	*`0xc00e35`	0.9896	0.8766	0.7794	0.7114	0.8090	384.11
32	<u>0.8799</u>	0.7395	`0xae3cc725`	0.9789	0.9054	0.8330	0.7532	0.7741	8.9×10^4
	<u>0.8311</u>	0.7523	`0x9fe72885`	0.8576	0.8584	0.8799	0.7589	0.7565	8.2×10^4
	0.8239	<u>0.7616</u>	`0xae36bfb5`	0.8405	0.8791	0.7703	0.7887	0.8276	8.9×10^4
	0.8799	<u>0.7395</u>	`0x82c1fcad`	0.9789	0.9054	0.8330	0.7532	0.7741	6.7×10^4

TABLE 4. Good multipliers for MCGs with $m = 2^{32}$.

Bits	$\mathscr{H}_8^+(m,a)$	$\mathscr{M}_8^+(m,a)$	a	f_2	f_3	f_4	f_5	f_6	λ
32	0.8674	0.7258	`0xff2826ad`	0.9275	0.8787	0.8567	0.8662	0.7258	1.00
	0.8482	0.7552	`0xf691b575`	0.8963	0.9209	0.7658	0.7887	0.7748	0.96
	0.8255	0.7741	`0xf2fc5985`	0.8833	0.7741	0.7963	0.8098	0.8030	0.95
	0.8602	0.7512	`0xff1cd035`	0.9274	0.9176	0.8053	0.7512	0.7524	1.00
33	0.8881	0.7341	`0x1dce91c05`	0.9875	0.8824	0.8275	0.8824	0.7370	1.86
	0.8480	0.7537	`0x19a28f105`	0.9418	0.8234	0.7936	0.7630	0.7724	1.60
	0.8324	0.7761	`0x1e5a5a195`	0.8648	0.8362	0.8458	0.7775	0.7928	1.90
	0.8856	0.7438	`0x1e179ae9d`	0.9900	0.8792	0.8394	0.8069	0.7566	1.88
34	0.8825	0.7124	`0x3dd03af2d`	0.9911	0.9289	0.8234	0.7187	0.7399	3.86
	0.8405	0.7573	`0x3af78c385`	0.9299	0.8115	0.7843	0.7700	0.7573	3.69
	0.8249	0.7849	`0x3631069bd`	0.8678	0.7891	0.8220	0.7930	0.8029	3.39
	0.8697	0.7351	`0x30761063d`	0.9526	0.9001	0.7725	0.8274	0.7562	3.03
35	0.8902	0.7352	`0x6bf6b1a55`	0.9918	0.9580	0.7494	0.7916	0.7352	6.75
	0.8835	0.7556	`0x758d4ae8d`	0.9500	0.9287	0.8192	0.8511	0.7754	7.35
	0.8736	0.7762	`0x69803d095`	0.9615	0.8395	0.8422	0.8217	0.7817	6.59
	0.8835	0.7556	*`0x758d4ae8d`	0.9500	0.9287	0.8192	0.8511	0.7754	7.35
48	0.8983	0.7083	`0x87338161ef95`	0.9916	0.9714	0.8686	0.7083	0.7676	3.5×10^4
	0.8896	0.7653	`0xb67a49a5466d`	0.9871	0.9351	0.7696	0.7965	0.7966	4.7×10^4
	0.8172	0.7892	`0x8616afca102d`	0.8204	0.8543	0.7933	0.8016	0.8084	3.4×10^4
	0.8790	0.7820	`0xbc1afb38ad6d`	0.9848	0.8601	0.7966	0.7842	0.7881	4.8×10^4
64	0.8992	0.7602	`0xd1342543de82ef95`	0.9586	0.9375	0.8708	0.8223	0.8204	3.5×10^9
	0.8924	0.7689	*`0xaf251af3b0f025b5`	0.9871	0.8989	0.8258	0.8160	0.7702	2.9×10^9
	0.8519	0.7950	`0xb564ef22ec7aece5`	0.9099	0.8358	0.8094	0.7950	0.8099	3.0×10^9
	0.8799	0.7842	`0xf7c2ebc08f67f2b5`	0.9637	0.8803	0.8270	0.7850	0.8135	4.2×10^9

TABLE 5. Good multipliers for LCGs with $m = 2^{64}$.

Bits	$\mathscr{H}_8^*(m,a)$	$\mathscr{M}_8^*(m,a)$	a	f_2	f_3	f_4	f_5	f_6	λ
32	0.8804	0.7129	`0xe9c5aaa5`	0.9806	0.8735	0.8475	0.7494	0.8419	1.83
	0.8489	0.7606	`0xf8e86b9d`	0.8673	0.8713	0.9052	0.7832	0.8166	1.94
	0.7921	0.7727	`0xd3733915`	0.8057	0.7727	0.8024	0.7898	0.7792	1.65
	0.8741	0.7334	`0xecbce6ad`	0.9705	0.9204	0.7488	0.8117	0.7334	1.85
33	0.8913	0.7087	`0x1efc38315`	0.9839	0.9546	0.8578	0.7574	0.7451	3.87
	0.8505	0.7637	`0x1feec73b5`	0.9038	0.8348	0.8036	0.8622	0.8201	3.99
	0.8197	0.7704	`0x1d5e995ed`	0.8557	0.7954	0.8117	0.7760	0.8317	3.67
	0.8722	0.7425	`0x1ec77d545`	0.9757	0.8636	0.8254	0.7425	0.7837	3.85
34	0.8903	0.7008	`0x32a4e0b8d`	0.9722	0.9127	0.8762	0.7607	0.8601	6.33
	0.8645	0.7557	`0x3dd6e1fa5`	0.9442	0.8131	0.8465	0.8512	0.8038	7.73
	0.8479	0.7710	`0x36b370ff5`	0.8997	0.8669	0.8259	0.7777	0.7710	6.84
	0.8731	0.7426	`0x37900045d`	0.9594	0.9256	0.7884	0.7570	0.7602	6.95
35	0.8870	0.7567	`0x76826be35`	0.9813	0.8629	0.8433	0.8289	0.8068	14.81
	0.8473	0.7704	*`0x77a0b8d0d`	0.9371	0.8048	0.7809	0.7824	0.7904	14.95
	0.8473	0.7704	`0x77a0b8d0d`	0.9371	0.8048	0.7809	0.7824	0.7904	14.95
	0.8870	0.7567	`0x76826be35`	0.9813	0.8629	0.8433	0.8289	0.8068	14.81
48	0.9012	0.7105	`0xe1aadae62835`	0.9968	0.9483	0.8156	0.8721	0.7478	1.2×10^5
	0.8988	0.7652	`0xf6473f07ba5d`	0.9773	0.9636	0.8230	0.8097	0.7829	1.3×10^5
	0.8730	0.7947	`0xc3be54e6b3dd`	0.9293	0.8773	0.8434	0.7947	0.8384	1.0×10^5
	0.8894	0.7722	*`0xbdcdbb079f8d`	0.9855	0.8937	0.7973	0.8466	0.7867	9.7×10^4
64	0.9016	0.7107	`0xcc62fceb9202faad`	0.9976	0.9569	0.8906	0.7893	0.7107	6.9×10^9
	0.8910	0.7606	`0xcb9c59b3f9f87d4d`	0.9825	0.9135	0.8524	0.7929	0.7630	6.8×10^9
	0.8365	0.7881	`0xfa346cbfd5890825`	0.8738	0.8413	0.7985	0.8215	0.7924	8.4×10^9
	0.8748	0.7761	`0x83b5b142866da9d5`	0.9633	0.8791	0.8012	0.7903	0.7761	4.4×10^9

TABLE 6. Good multipliers for MCGs with $m = 2^{64}$.

Bits	$\mathscr{H}_8^+(m,a)$	$\mathscr{M}_8^+(m,a)$	a	f_2	f_3	f_4	f_5	f_6	λ
64	0.8819	0.7432	`0xfdeb119694293925`	0.9230	0.9305	0.8853	0.7828	0.8733	0.99
	0.8709	0.7614	`0xff37f1f758180525`	0.9278	0.9140	0.8282	0.8245	0.7753	1.00
	0.8453	0.7894	`0xeb2bbdba79726cb5`	0.8549	0.9157	0.8095	0.7934	0.8197	0.92
	0.8638	0.7802	`0xf656d1ff96c39a1d`	0.8955	0.9063	0.8723	0.7897	0.8006	0.96
65	0.8993	0.7020	`0x1df77a66a374e300d`	0.9937	0.9656	0.8086	0.7964	0.7839	1.87
	0.8987	0.7615	`0x1d605bbb58c8abbfd`	0.9919	0.9079	0.8310	0.8380	0.7804	1.84
	0.7967	0.7850	`0x1d7d8dd3a6a72b43d`	0.7895	0.7995	0.8036	0.7850	0.8323	1.84
	0.8814	0.7758	`0x1f20529e418340d05`	0.9613	0.9008	0.8115	0.7967	0.8130	1.95
66	0.8980	0.7113	`0x3c4d319b99f8cc105`	0.9799	0.9276	0.8939	0.8291	0.7807	3.77
	0.8960	0.7933	`0x3e0d997a645f176dd`	0.9844	0.8455	0.8914	0.8430	0.8033	3.88
	0.8960	0.7933	`0x3e0d997a645f176dd`	0.9844	0.8455	0.8914	0.8430	0.8033	3.88
	0.8980	0.7113	***`0x3c4d319b99f8cc105`	0.9799	0.9276	0.8939	0.8291	0.7807	3.77
67	0.8982	0.7592	`0x77808d182e9136c35`	0.9721	0.9324	0.8682	0.8175	0.7995	7.47
	0.8743	0.7722	*`0x677b06feba6314a95`	0.9316	0.8980	0.7913	0.8817	0.8100	6.47
	0.8490	0.7861	`0x728917326ee7fe425`	0.8620	0.8866	0.8723	0.7943	0.8112	7.16
	0.8721	0.7759	`0x7136fb7b10e963785`	0.9289	0.8911	0.8114	0.8122	0.8558	7.08
68	0.8921	0.7076	`0xca7592823d4c35535`	0.9988	0.8698	0.8744	0.8410	0.7801	12.65
	0.8874	0.7611	`0xc478db86929909e45`	0.9880	0.8897	0.7622	0.8763	0.7901	12.28
	0.8367	0.7823	`0xd290438662882ec8d`	0.9037	0.7823	0.8121	0.8076	0.7872	13.16
	0.8766	0.7664	`0xc5bf2cf11c47e0f1d`	0.9842	0.8610	0.7863	0.7953	0.7664	12.36
69	0.9012	0.7164	`0x1c50d68cca5d0c811d`	0.9938	0.9361	0.8520	0.8365	0.7535	28.32
	0.8923	0.7765	`0x19fc3e7a00d672e85d`	0.9849	0.8770	0.8420	0.8211	0.8141	25.99
	0.8485	0.7900	`0x1a03cbacd725e7559d`	0.9161	0.8049	0.8304	0.7924	0.8099	26.01
	0.8923	0.7765	`0x19fc3e7a00d672e85d`	0.9849	0.8770	0.8420	0.8211	0.8141	25.99

TABLE 7. Good multipliers for LCGs with $m = 2^{128}$ (smaller sizes).

Bits	$\mathscr{H}_8^+(m,a)$	$\mathscr{M}_8^+(m,a)$	a	f_2	f_3	f_4	f_5	f_6	λ
70	$\underline{0.8978}$	0.7590	`0x3fed4ae17a065d9795`	0.9769	0.8996	0.8374	0.8684	0.8323	63.93
	$\underline{0.8665}$	0.7687	*`0x3c4a964d52d0d8c5fd`	0.9672	0.8046	0.8254	0.8221	0.7687	60.29
	0.8498	$\underline{0.7815}$	`0x3ad59455e2c68e647d`	0.9281	0.8258	0.7983	0.7828	0.7856	58.83
	0.8785	$\underline{0.7652}$	`0x3f20bb6a991aebfa0d`	0.9739	0.8906	0.7675	0.7652	0.8096	63.13
71	$\underline{0.8951}$	0.7387	`0x7ff1ef4da80c020645`	0.9976	0.9632	0.7812	0.7792	0.7724	127.95
	$\underline{0.8867}$	0.7656	`0x78527790da229db3c5`	0.9747	0.8911	0.8217	0.8120	0.8042	120.32
	0.8415	$\underline{0.7821}$	`0x654ecfa104a2b561b5`	0.9090	0.8069	0.7931	0.7900	0.8142	101.31
	0.8867	$\underline{0.7656}$	`0x78527790da229db3c5`	0.9747	0.8911	0.8217	0.8120	0.8042	120.32
72	$\underline{0.8942}$	0.7139	`0xc043dd12a44daa4c7d`	0.9915	0.9236	0.8833	0.7728	0.7455	192.27
	$\underline{0.8830}$	0.7592	`0xce12be0ad9384349d5`	0.9548	0.9322	0.8139	0.7592	0.8244	206.07
	0.7984	$\underline{0.7807}$	`0xc8a0bcbd37f06521c5`	0.8056	0.7996	0.8018	0.7807	0.7883	200.63
	0.8753	$\underline{0.7646}$	`0xc35bd29e4b5db7ff6d`	0.9572	0.8983	0.8184	0.7844	0.7797	195.36
80	$\underline{0.9105}$	0.7385	`0xc41f4d6a9941f620e1ad`	0.9932	0.9115	0.8978	0.8528	0.8169	5.0×10^4
	$\underline{0.8839}$	0.7563	`0xd66d53ac13109e1ccd25`	0.9910	0.8651	0.8630	0.7563	0.7738	5.5×10^4
	0.8629	$\underline{0.7838}$	`0xfe2d84b0671aa6f869bd`	0.9491	0.8101	0.8142	0.8229	0.8054	6.5×10^4
	0.8761	$\underline{0.7745}$	`0xddb80b9ccb1066bee495`	0.9579	0.9098	0.7779	0.8016	0.7808	5.7×10^4
96	$\underline{0.8966}$	0.7187	`0x974b29a2bd0ead10a63b07a5`	0.9858	0.8645	0.8890	0.8560	0.7892	2.5×10^9
	$\underline{0.8903}$	0.7771	`0xdfe1956283473c8e63b49445`	0.9599	0.9367	0.8193	0.8042	0.8205	3.8×10^9
	0.8530	$\underline{0.7853}$	`0xef6fddb4a090a95008d79ecd`	0.9118	0.8344	0.8376	0.8023	0.7938	4.0×10^9
	0.8903	$\underline{0.7771}$	`0xdfe1956283473c8e63b49445`	0.9599	0.9367	0.8193	0.8042	0.8205	3.8×10^9
128	$\underline{0.8989}$	0.7423	`0xde92a69f6e2f9f25fd0d90f576075fbd`	0.9886	0.9410	0.8257	0.8232	0.7716	1.6×10^{19}
	$\underline{0.8935}$	0.7572	`0xff44119c81341bdbcbfa54949296b3b5`	0.9800	0.9194	0.7852	0.8603	0.8274	1.8×10^{19}
	0.8299	$\underline{0.7905}$	`0x87ea3de194dd2e97074f3d0c2ea63d35`	0.8196	0.8784	0.8537	0.7938	0.8252	9.8×10^{18}
	0.8763	$\underline{0.7759}$	`0xf48c0745581cf801619cd45257f0ab65`	0.9729	0.8173	0.8680	0.7759	0.8143	1.8×10^{19}

TABLE 8. Good multipliers for LCGs with $m = 2^{128}$ (larger sizes).

Bits	$\mathscr{H}_8^*(m,a)$	$\mathscr{M}_8^*(m,a)$	a	f_2	f_3	f_4	f_5	f_6	λ
63	0.8802	0.7260	`0x7fee2f589372a885`	0.9301	0.9769	0.8563	0.7498	0.8114	1.00
	0.8692	0.7605	`0x7f060b620c8cf005`	0.9235	0.9244	0.8568	0.7661	0.7679	0.99
	0.8152	0.7822	`0x71af0ab5118a6e6d`	0.8265	0.8008	0.8209	0.8493	0.7902	0.89
	0.8618	0.7757	`0x7ebcd75629af014d`	0.9214	0.8435	0.8282	0.8472	0.8051	0.99
64	0.8960	0.7088	`0xee3892fb8a5e8e75`	0.9938	0.9224	0.7980	0.8551	0.8247	1.86
	0.8826	0.7690	`0xea1fbb3aa44f0b9d`	0.9755	0.9129	0.7725	0.7763	0.8293	1.83
	0.8385	0.7846	`0xe9db2851bd2dd4ad`	0.8361	0.9034	0.8551	0.7918	0.7846	1.83
	0.8793	0.7793	`0xde01abbf8f022f55`	0.9685	0.8904	0.7793	0.7882	0.8106	1.73
65	0.8955	0.7141	`0x1ec15c1dd17c6f745`	0.9936	0.9085	0.8546	0.8337	0.7959	3.84
	0.8884	0.7565	`0x1ec18fa24ae54a1dd`	0.9702	0.9456	0.8288	0.7856	0.7676	3.84
	0.8583	0.7818	`0x1e3a6c660c9c8d2ed`	0.9418	0.8058	0.8406	0.7826	0.7818	3.78
	0.8707	0.7719	`0x18da4f2ec25b600c5`	0.9076	0.9210	0.8695	0.8083	0.7719	3.11
66	0.8965	0.7111	`0x31b5ded2927f31a55`	0.9665	0.9253	0.8980	0.8199	0.7910	6.21
	0.8791	0.7557	`0x3e895c103064eff15`	0.9389	0.9512	0.7949	0.8198	0.7557	7.82
	0.8660	0.7870	`0x3edab7c1a1f9078fd`	0.9494	0.7967	0.8592	0.8041	0.8277	7.86
	0.8731	0.7755	`0x36bfab71e57b81a9d`	0.9485	0.8622	0.7966	0.8650	0.7995	6.84
67	0.8966	0.7197	`0x6a876400b76f60395`	0.9822	0.9648	0.7843	0.8630	0.7210	13.32
	0.8802	0.7616	`0x6efbe29439fbde605`	0.9792	0.8999	0.7984	0.7710	0.7616	13.87
	0.8240	0.7802	`0x6b677bf1402c4b5f5`	0.8346	0.8722	0.7947	0.8065	0.7842	13.43
	0.8720	0.7725	`0x71217c8b506a03245`	0.9590	0.8654	0.8263	0.7899	0.7750	14.14
68	0.8980	0.7262	`0xfcb2c4e9f685e90fd`	0.9653	0.9618	0.8654	0.8477	0.7307	31.59
	0.8821	0.7592	`0xdb9db0f421d1a042d`	0.9764	0.9310	0.7774	0.7633	0.7592	27.45
	0.8162	0.7844	`0xf8edf6d981eda7d25`	0.8232	0.8402	0.8091	0.7918	0.7844	31.12
	0.8736	0.7671	`0xd702f582b6b36c565`	0.9536	0.8671	0.8395	0.7844	0.8096	26.88

TABLE 9. Good multipliers for MCGs with $m = 2^{128}$ (smaller sizes).

Bits	$\mathscr{H}_8^*(m,a)$	$\mathscr{M}_8^*(m,a)$	a	f_2	f_3	f_4	f_5	f_6	λ
69	$\underline{0.8972}$	0.7157	`0x1dc29acd0bbcd1618d`	0.9807	0.9725	0.8676	0.8019	0.7157	59.52
	$\underline{0.8858}$	0.7580	`0x1d86361eb9c5307f9d`	0.9612	0.8847	0.8598	0.8525	0.7894	59.05
	0.8469	$\underline{0.7819}$	`0x18e13ee9a22730b6b5`	0.9008	0.8391	0.8316	0.7911	0.7841	49.76
	0.8752	$\underline{0.7678}$	`0x1a72912e5a21cf1f4d`	0.9510	0.8808	0.8461	0.8072	0.7678	52.90
70	$\underline{0.8960}$	0.7086	`0x30606a40cc6cbe7895`	0.9815	0.9330	0.8724	0.7891	0.8068	96.75
	$\underline{0.8946}$	0.7701	`0x37a84527ebfc3b586d`	0.9709	0.8904	0.8904	0.8476	0.7765	111.31
	0.8026	$\underline{0.7890}$	`0x3782c32a82c5dbf4f5`	0.8133	0.7935	0.8078	0.7964	0.7890	111.02
	0.8763	$\underline{0.7787}$	`0x3b9044e6db80473695`	0.9062	0.8844	0.8724	0.9180	0.8025	119.13
71	$\underline{0.8965}$	0.7699	`0x7731be67a558124cdd`	0.9816	0.8935	0.8385	0.8521	0.8244	238.39
	$\underline{0.8353}$	0.7824	** `0x646bbc3142bc648dfd`	0.8360	0.8873	0.8262	0.8096	0.8276	200.84
	0.8120	$\underline{0.7849}$	`0x6af94974cd28cfa575`	0.7910	0.8457	0.8641	0.7974	0.7914	213.95
	0.8724	$\underline{0.7737}$	`0x70557806b726da3c95`	0.9666	0.8187	0.8585	0.8084	0.7905	224.67
72	$\underline{0.9039}$	0.7138	`0xd42fddd666ed5f2bbd`	0.9952	0.9662	0.8258	0.8230	0.7569	424.37
	$\underline{0.8825}$	0.7698	`0xd216c8b379531520ad`	0.9528	0.9232	0.8152	0.7764	0.8431	420.18
	0.8434	$\underline{0.7914}$	`0xc3d5e36abda23407a5`	0.8615	0.8884	0.7914	0.8102	0.8148	391.67
	0.8825	$\underline{0.7698}$	`0xd216c8b379531520ad`	0.9528	0.9232	0.8152	0.7764	0.8431	420.18
80	$\underline{0.8948}$	0.7154	`0xd33f378ea340c4eada65`	0.9923	0.9367	0.8864	0.7454	0.7154	1.1×10^5
	$\underline{0.8830}$	0.7630	`0xe8c67028b28c626d2185`	0.9254	0.9575	0.8430	0.8265	0.7630	1.2×10^5
	0.8541	$\underline{0.7832}$	`0xed126c68193f2a63846d`	0.8929	0.8771	0.8421	0.8000	0.7973	1.2×10^5
	0.8734	$\underline{0.7636}$	`0xdced41407dae02b88ded`	0.9177	0.9259	0.8666	0.8031	0.7636	1.1×10^5
96	$\underline{0.8946}$	0.7155	`0xc44887bda8f45c38ec440805`	0.9890	0.9612	0.8291	0.7875	0.7506	6.6×10^9
	$\underline{0.8846}$	0.7584	`0xc0d1f685e61b167aafc41545`	0.9610	0.8863	0.8422	0.8224	0.8187	6.5×10^9
	0.8529	$\underline{0.7916}$	`0x8da5d6bc4427ca1cfd32a8b5`	0.9093	0.8420	0.8205	0.8013	0.8017	4.8×10^9
	0.8721	$\underline{0.7625}$	`0xdc7bbe1f2cb3b43ab5a97905`	0.9470	0.9444	0.7742	0.7714	0.7625	7.4×10^9
128	$\underline{0.8995}$	0.7129	`0xace2628409311ff16a545ebdff0d414d`	0.9824	0.9484	0.8563	0.8698	0.7583	2.5×10^{19}
	$\underline{0.8945}$	0.7610	`0xf9f3608dc854565e41babd0cd07f7725`	0.9727	0.9170	0.8639	0.8040	0.7722	3.6×10^{19}
	0.8024	$\underline{0.7918}$	`0xfcf1dc21cdcc71ae30bcc1ec5be3c1a5`	0.7918	0.8157	0.8143	0.7953	0.8099	3.6×10^{19}
	0.8744	$\underline{0.7719}$	`0x8855c9aa096cdcc0eae76c902f3f2335`	0.9420	0.8509	0.8368	0.8722	0.7719	2.0×10^{19}

TABLE 10. Good multipliers for MCGs with $m = 2^{128}$ (larger sizes).

References

1. Arm Limited, 110 Fulbourn Road, Cambridge, England CB1 9NJ, *Arm Instruction Set, Version 1.0: Reference Guide*, October 2018.
2. Nicolas Bourbaki, *Algèbre. Chapitre 2: Algèbre linéaire*, Hermann, 1947.
3. Mark J. Durst, *Using linear congruential generators for parallel random number generation*, 1989 Winter Simulation Conference Proceedings, IEEE Press, 1989, pp. 462–466.
4. Karl Entacher, Thomas Schell, and Andreas Uhl, *Efficient lattice assessment for LCG and GLP parameter searches*, Mathematics of Computation **71** (2002), no. 239, 1231–1242.
5. George S. Fishman and Louis R. Moore III, *An exhaustive analysis of multiplicative congruential random number generators with modulus* $2^{31} - 1$, SIAM Journal on Scientific and Statistical Computing **7** (1986), no. 1, 24–45.
6. John H. Halton, *Pseudo-random trees: Multiple independent sequence generators for parallel and branching computations*, Journal of Computational Physics **84** (1989), no. 1, 1–56.
7. W. Hörmann and G. Derflinger, *A portable random number generator well suited for the rejection method*, ACM Trans. Math. Softw. **19** (1993), no. 4, 489–495.
8. Intel Corporation, *Intel 64 and IA-32 Architectures Software Developers Manual, Volume 2 (2A, 2B, 2C & 2D): Instruction Set Reference, A–Z*, October 2019.
9. Donald E. Knuth, *The art of computer programming, volume 2: Seminumerical algorithms*, third ed., Addison-Wesley, Reading, MA, USA, 1998.
10. Pierre L'Ecuyer, *Tables of linear congruential generators of different sizes and good lattice structure*, Math. Comput **68** (1999), no. 225, 249–260.
11. Derrick H. Lehmer, *Mathematical methods in large-scale computing units*, Annu. Comput. Lab. Harvard Univ. **26** (1951), 141–146.
12. Arjen K. Lenstra, Hendrik W. Lenstra, and László Lovász, *Factoring polynomials with rational coefficients*, Mathematische Annalen **261** (1982), no. 4, 515–534.
13. Ora E. Percus and Malvin H. Kalos, *Random number generators for MIMD parallel processors*, Journal of Parallel and Distributed Computing **6** (1989), no. 3, 477–497.
14. Victor Shoup, *NTL: A library for doing number theory*, 2019.

Oracle Labs
E-mail address: guy.steele@oracle.com

Università degli Studi di Milano, Italy
E-mail address: sebastiano.vigna@unimi.it