Pitfalls in Computing with Pseudorandom Determinants*

Bernd Gärtner
Institut für Theoretische Informatik, ETH Zürich
ETH Zentrum, CH-8092 Zürich, Switzerland
gaertner@inf.ethz.ch

ABSTRACT

It has been known for 30 years that pseudorandom number generators in the class of *linear congruential generators* (LCG) exhibit strong and predictable regularities. A widely used generator in this class is `drand48`. While the regularity is not problematic in most applications, I show that it can produce very misleading results in testing geometric algorithms that involve determinant computations. By presenting scenarios where LCG behave 'nonrandom' (sometimes in a spectacular way), I want to raise awareness for possible problems with LCG and pseudorandom numbers in general.

1. INTRODUCTION

In recent years, more and more papers published in computational geometry conferences and journals report about implementations. The results usually include tests with 'random input'. Statements like "Table 4 gives the runtime for n points randomly chosen from the unit square, for different values of n" are typical.

The main point of this paper is that such a statement can be quite meaningless, if it does not mention the method according to which the random choices were made. And even if it does, this particular method might be inappropriate, thus invalidating the results.

This is not a new insight; on the contrary, similar statements have been made by many people, including Knuth who remarked that "random numbers should not be generated by a method chosen at random"[10]. In fact, it is computer sciene folklore that pseudorandom number generation bears some danger, and many even know about one particularly bad random number generator used in the sixties. This generator (actually an LCG; we come back to it below) has the property that the threedimensional points it generates are concentrated in very few planes.

Still, a widespread attitude is that as long as some 'good' generator (according to standard benchmarks) is used, it does not matter which one it is in particular. Then statements like the one quoted above are considered valid—by the author as well as by the reader—under the silent assumption that the random number generator used in the tests is reasonably up-to-date (which hopefully holds for most generators used nowadays).

This attitude, however, misses the main point. Namely, even the 'best' and most recent generator can be inappropriate in a certain application. It seems that the 'likelihood' of such an event is small, but we will see that the whole class of LCG is problematic in applications dealing with determinants. Because LCG are widely used, and determinant computations are ubiquitous in computational geometry, the problems are just around the corner.

It should be clear that results for pseudorandom or truly random input are only vaguely (if at all) related to the performance of an algorithm in practice, because the input distributions one typically observes are quite nonrandom. Despite this fact, tests with pseudorandom input are frequently being made, with the clear intention of performance evaluation. One reason for that is a lack of real data, often observed in connection with more theoretically oriented research; another attractive feature is that pseudorandom data are seemingly beyond the control of the test person, documenting that the algorithm can handle inputs not explicitly 'prepared' by the author.

This paper does not make any statement about testing with 'real-world data', obtained without involving a pseudorandom source. It does make a statement regarding test scenarios that simply *pretend* to simulate real-world behavior by using pseudorandom input. I want to argue that the latter scenarios can potentially be dangerous, more often than one might think.

In the next section, I will introduce LCG and state the regularity theorem. Section 3 presents three scenarios where this theorem and consequences of it lead to nonrandom behavior. Finally, Section 4 contains a proof of the regularity

*This work was supported by a grant from the Swiss Federal Office for Education and Science (Project ESPRIT IV LTR No. 28155 GALIA). Parts of the material have already appeared in the *Mitteilungen der Deutschen Mathematiker-Vereinigung* (DMV), Volume 2, 1999, pp. 55-60 (in German).

theorem, which provides insights into the structure of LCG in general.

The regularity theorem as I present it, along with its implications for determinant computation, is new. However, it is a simple consequence of known and classical results regarding the structure of LCG [11, 10].

2. LCG AND THE REGULARITY THEOREM

Let a, c, m be integers with $0 \le a, c < m$. The LCG with parameters a, c, m outputs a sequence $X_1, X_2, \ldots$ of pseudorandom integers in the range $[0, m)$ according to the formula

$$X_{t+1} = (aX_t + c) \mod m, \quad t \ge 0. \tag{1}$$

Here, X_0 is the *seed*, m the *modulus*, a the *multiplier*, and c the *increment*.

The generator `drand48`, for example, has the parameters

$$a = 25214903917, \quad c = 11, \quad m = 2^{48}. \tag{2}$$

To get floating-point numbers in the unit interval, the generated numbers X_i are scaled by 2^{-48}.

Because the sequence (X_i) is periodic with period length at most m, the modulus should be large, and the period length close to the maximum. `drand48` has full period. As a small running example, let us consider the full-period generator `knuth8` [10], specified by

$$a = 137, \quad c = 187, \quad m = 2^8. \tag{3}$$

THEOREM 2.1. *(Regularity Theorem.) For fixed m, a, c, X_0, let*

$$X_{t+1} = (aX_t + c) \bmod m,$$

for $t \ge 0$, and fix a positive integer d as well as two sequences of nonnegative integers $i_1, \ldots, i_d$ and $j_1, \ldots, j_d$. Consider the $d \times d$-matrices

$$A := \begin{pmatrix} X_{i_1+j_1} & X_{i_1+j_2} & \cdots & X_{i_1+j_d} \\ X_{i_2+j_1} & X_{i_2+j_2} & \cdots & X_{i_2+j_d} \\ \vdots & \vdots & & \vdots \\ X_{i_d+j_1} & X_{i_d+j_2} & \cdots & X_{i_d+j_d} \end{pmatrix}$$

and

$$A' := \begin{pmatrix} X_{i_1+j_1} & X_{i_1+j_2} & \cdots & X_{i_1+j_{d-1}} & 1 \\ X_{i_2+j_1} & X_{i_2+j_2} & \cdots & X_{i_2+j_{d-1}} & 1 \\ \vdots & \vdots & & \vdots & \vdots \\ X_{i_d+j_1} & X_{i_d+j_2} & \cdots & X_{i_d+j_{d-1}} & 1 \end{pmatrix}.$$

Then both $\det(A)$ and $\det(A')$ are integer multiples of m^{d-2}.

I prove the theorem in Section 4. For $d = 3$, one gets the first nontrivial statement, showing that for example,

$$\det \begin{pmatrix} X_1 & X_2 & 1 \\ X_3 & X_4 & 1 \\ X_5 & X_6 & 1 \end{pmatrix}, \quad \det \begin{pmatrix} X_1 & X_4 & 1 \\ X_2 & X_5 & 1 \\ X_3 & X_6 & 1 \end{pmatrix}$$

and

$$\det \begin{pmatrix} X_1 & X_2 & X_3 \\ X_2 & X_3 & X_4 \\ X_3 & X_4 & X_5 \end{pmatrix}$$

are divisible by the modulus m. To give concrete numbers in case of `knuth8`, one computes (with $X_0 = 0$)

$$\begin{aligned} \det \begin{pmatrix} X_1 & X_2 & 1 \\ X_3 & X_4 & 1 \\ X_5 & X_6 & 1 \end{pmatrix} &= \det \begin{pmatrix} 187 & 206 & 1 \\ 249 & 252 & 1 \\ 151 & 138 & 1 \end{pmatrix} \\ &= -2560 = -10 \cdot 256. \end{aligned}$$

The theorem is quite general in the sense that many natural ways of filling the matrix A or A' from a stream of pseudorandom numbers lead to regular behavior. For example, one can proceed row- or columnwise, skip or repeat full rows or columns, or use only every k-th number, for some k. Still, knowing the pattern, it is not difficult to break it. For example, *omitting* every k-th number from the stream for $k \ge 3$ leads to a matrix with substantially less regularity (also depending on whether k and d are relative prime).

In the following, the term 'pseudorandom' always refers to LCG-generated numbers.

3. THREE NONRANDOM SCENARIOS

The important consequence of Theorem 2.1 is that determinants with pseudorandom entries have unnaturally small bit complexity when the modulus is a power of two. Among the three scenarios below, the first two elaborate on the practical implications of this phenomenon. The third scenario deals with a different effect that happens when lower-order bits of an LCG sequence are used to define pseudorandom point coordinates.

To argue that certain effects are nonrandom, one must show that they do not occur with truly random data. In all scenarios discussed below, this is either obvious, or can easily be done. Still, to illustrate the effects, I have performed tests that compare pseudorandom with truly random input. To obtain the latter, I have used David Walker's `HotBits` which are generated by radioactive decay and can be downloaded in chunks from Walker's web site (`http://www.fourmilab.ch/hotbits/`). Since it is impossible to decide whether a concrete stream of bits is 'random', or whether the random source is trustworthy at all, those tests should not be taken too seriously. They simply provide good practical estimates for the impact of the effects I am going to describe.

Exact computations: too fast.

Exact evaluations of pseudroandom determinants can lead to the largest miscalculations of an algorithm's performance; therefore, I want to begin with this scenario. Although it is mostly the sign rather than the exact value one is interested in, there are situations where one actually needs to compute the full determinant. For example, to obtain benchmarks for the efficiency of a floating-point filter, one typically compares the runtime of the filtered approach with the runtime of the exact evaluation. Also, in degenerate situations where the filtered approach fails for many sign tests, the efficiency

	drand48			HotBits	
d	t	b	b_{norm}	t	b
5	0.12	138	89	0.12	234
10	0.14	134	86	0.19	474
20	0.37	139	92	1.78	956
30	0.97	143	96	9.87	1442
40	2.14	152	104	34.4	1929
50	4.06	160	113	92.0	2411

Table 1: Exact matrix inversion over pseudorandom and truly random input

of the exact evaluation becomes an issue. In geometric optimization (most notably in low-dimensional linear programming), systems of linear equations have to be solved; in this case, it is much less clear how floating-point filters can be applied effectively, and exact computation is often the only viable alternative. Solving those linear systems, however, often boils down to the computation of (sub-)determinants.

The following case study models the situation as it occured in the implementation of an exact linear programming solver [5]. The suspiciously high efficiency of the solver on pseudorandom input—even for larger dimensions— made me 'discover' the underlying regularities of the input, as they are given in Theorem 2.1.

By Cramer's rule, the entries of the inverse of a $(d \times d)$-matrix A can be computed as

$$A^{-1}_{ij} = (-1)^{i+j} \frac{\det(A^{ji})}{\det(A)}, \tag{4}$$

where A^{ji} is the submatrix obtained by deleting row i and column j. In the linear programming solver, A is a matrix with floating-point entries, and A^{-1} is used as a device to solve linear systems $Ax = y$. To represent A^{-1} without roundoff errors, its entries are stored as rational numbers according to (4), where numerator and denominator are of the form $s \cdot 2^e$, with s and e multiple precision integers.

A technique called *Q-pivoting* [4, 5] is used to update A^{-1} in case a column of A is exchanged (this corresponds to a pivot step in the solver and a *rank-1 update* of A^{-1}). The update step performs $O(d^2)$ arithmetic operations, many operands being subdeterminants $\det(A^{ji})$.

Q-pivoting can be applied to build A^{-1} from scratch, using d rank-1 updates. This process is used in Table 1 to evaluate the efficiency of multiple precision integer arithmetic in this context. Note that other exact arithmetic methods for solving linear systems (e.g. the division-free Gauss elimination described in [6]) are similar and boil down to computations with subdeterminants as well. All these methods are therefore subject to the phenomenon described here.

For various values of d, the matrix A was filled with floating-point numbers of 48 bits precision, in one case generated from `drand48`, in the other case from a stream of `HotBits`. In both cases, t denotes the absolute time in seconds needed to invert A.[1]

[1]All tests have been done on a SUN Ultra 2 machine, without compiler optimization, and using the multiple precison `integer` type of LEDA (`http://www.mpi-sb.mpg.de/LEDA/`) to represent the mantissa s. However, this is not important here, because we only consider the runtimes relative to each other.

b is the number of significant bits of $\det(A)$ in the form $s \cdot 2^e$, as it was computed by the Q-pivoting. b_{norm} is the number of significant bits after normalizing, which moves powers of 2 contained in s to the exponent. Because in case of `HotBits`, $b - b_{norm} \leq 1$ holds in all tests that were done, the normalized bit complexity is not given.

The table shows that Theorem 2.1 strikes in an impressive manner here. A was filled row by row, thus has the form

$$A = \frac{1}{2^{48}} \begin{pmatrix} X_1 & X_2 & \cdots & X_d \\ X_{d+1} & X_{d+2} & \cdots & X_{2d} \\ \vdots & \vdots & & \vdots \\ X_{(d-1)d+1} & X_{(d-1)d+2} & \cdots & X_{d^2} \end{pmatrix}$$

where the X_i are generated according to (2). For d small, the integer value $2^{48} \det(A)$ requires no more than roughly $48d$ bits of precision. On the other hand, according to Theorem 2.1, the least significant $48(d-2)$ bits must be zero, so roughly 96 significant bits remain. This of course also holds for $\det(A)$ itself, which then has a mantissa s of no more than roughly 96 bits in the normalized form $s \cdot 2^e$. This tallies quite well with the figures in Table 1.

Moreover, the numbers are essentially that small throughout the whole Q-pivoting; only the final step does not perform normalization, which explains the gap between b and b_{norm}.

In comparison, $\det(A)$ really requires about $48d$ bit of precision in case of `HotBits`. In dimension 50, this is a factor of 15 larger than the precision one gets from `drand48`. Because the complexities of multiplication and integer division (the latter occurs during the Q-pivoting) are superlinear in the bit complexities of their operands, the runtime is higher even by a factor of 23.

In other words, if `drand48` (or any other LCG whose modulus is a power of two) is chosen to generate the entries of A, the efficieny of the exact inversion of A is overestimated by a substantial factor, already for small d.

It might seem that the problem is specific to the choice of representing numbers in the form $s \cdot 2^e$ (although this is a canonical way to generalize floating-point numbers). But the same effect can be observed when other space-efficient representations are chosen, for example Shewchuck's *nonoverlapping expansions* [12]. In this format, a multiple precision floating-point number f is stored as a sum of ordinary floating-point numbers, and if f has small mantissa, only few summands are needed.

Even if the exact inverse of an *integer* matrix is computed, using plain multiple precision integers as the exact number type, it is not guaranteed that everything behaves as expected. For example, the integer type might still be able to handle powers of 2 more efficiently than other numbers. This is even very likely, if the numbers are stored in binary representation, in which case many zeros in the representa-

tion make the arithmetic operations faster.

The only reasonable advice I can give in this scenario is to stay away from LCG completely.

Floating-point computations: too exact.

Consider three points p, q, r in the plane. It is well known [2] that the orientation of the triple (p, q, r) is given by the sign of the 3×3-determinant

$$\begin{aligned} D &= \det \begin{pmatrix} p_x & p_y & 1 \\ q_x & q_y & 1 \\ r_x & r_y & 1 \end{pmatrix} \\ &= (q_x - p_x)(r_y - p_y) - (q_y - p_y)(r_x - p_x), \end{aligned} \tag{5}$$

where the entries are the coordinates of the points. Assume those coordinates are of the floating-point type `double`, allowing 53 bits of precision [8], but they are initialized with pseudorandom values from `drand48`, row- or columnwise. Then the sign of the determinant is *always* evaluated correctly over the type `double`. If pseudorandom $d+1$-tuples of points in d-space are chosen that way, it still holds for $d = 3$ (and a few more values of d) that the sign of D is evaluated correctly unless $D = 0$.

For the argument, let us concentrate on the 2-dimensional case now. D will be of the form $D = s \cdot 2^{-96}$, where s is an integer multiple of 2^{48} by Theorem 2.1. But then we can normalize D to $s' \cdot 2^{-48}$. In particular, D has absolute value at least 2^{-48} if it is nonzero. This is much larger than the absolute error that can occur during the computation of D using (5)—recall that we have 53 bits of precision for that. Thus, the sign of D is computed correctly, if $D \neq 0$. In case $D = 0$, both expressions

$$(q_x - p_x)(r_y - p_y), \quad (q_y - p_y)(r_x - p_x)$$

evaluate to the same `double` value, hence their difference will evaluate to 0. To see this note that the four differences will be evaluated exactly, because they only need 49 bits of precision. Then, both products will be the same closest `double` approximation to the real product $\alpha = (q_x - p_x)(r_y - p_y) = (q_y - p_y)(r_x - p_x)$.

Theorem 2.1 shows that in *any* dimension d, the pseudorandom orientation determinant

$$D = \frac{1}{2^{48}} \begin{pmatrix} X_1 & X_2 & \cdots & X_d & 1 \\ X_{d+1} & X_{d+2} & \cdots & X_{2d} & 1 \\ \vdots & \vdots & & \vdots & \vdots \\ X_{d(d-1)+1} & X_{d(d-1)+2} & \cdots & X_{d^2} & 1 \end{pmatrix}$$

has absolute value $|D| \geq 2^{-48}$ if $D \neq 0$, giving a spectacularly good separation bound. For small d, this bound is still larger than the absolute error made in computing D. Hence, $D \neq 0$ implies that its sign is computed correctly.

Even for $d = 2$, the practical relevance of this observation is limited, because a truly random 3-tuple of points leads to a value of $|D| < 2^{-48}$ only with very small probability—in fact so small that this event is unlikely to occur even in years of testing.

As shown by Henze [7], the density function of $|D|$ for three points randomly chosen from the unit square can be computed explicitly (although the formula is pretty complicated), and one obtains

$$\text{prob}(|D| < 2^{-48}) \approx 4.7 \cdot 10^{-14}.$$

Devillers and Preparata [3] have derived a simpler formula for the density of $|D|$ if $p = (p_x, p_y) = (0, 0)$, and only the points q, r are chosen at random. Both distributions behave similarly, but only the former correctly models the situation we consider here.

If we do not compute with double precision, but use single precision numbers of type `float` (24 bits of precision), the effect described above becomes noticeable. From Henze's formula, we get

$$\text{prob}(|D| < 2^{-24}) \approx 7.2 \cdot 10^{-7}.$$

This means for example that if we generate $10,000$ truly random points, we expect about 12,000 point triples with $0 < |D| < 2^{-24}$, while no such triple will occur when the input coordinates are chosen from an LCG with modulus at most 2^{24}. In this situation, we expect more roundoff errors in computing D for the truly random triples. However, since generators with such small moduli are not in practical use, the danger of actually running into the described anomalies is still small.

As we show in the next scenario, the situation becomes critical if the 24-bit coordinates are chosen in a more realistic way, namely as the lower-order bits of the numbers generated by some LCG with higher modulus.

Lower order bits: duplicated points.

To choose pseudorandom values $Y_1, Y_2, \ldots$ in some range $\{0, \ldots, k-1\}$, a natural way to proceed is to generate values $X_1, X_2, \ldots$ from some LCG with modulus m, such that $m > k$, and set

$$Y_i := X_i \mod k.$$

For example, to get a sequence of pseudorandom `float` values, we might generate a sequence of `double` values using `drand48`, and take the 24 least significant bits of each value. In this case, m is a multiple of k, and the following theorem shows that this is a bad choice for generating pseudorandom points in d-dimensional space.

THEOREM 3.1. *With k, m, X_i and Y_i as above, m a multiple of k, and $j_1, \ldots, j_d$ a sequence of nonnegative integers, define*

$$p_i := (Y_{i+j_1}, Y_{i+j_2}, \ldots, Y_{i+j_d}).$$

Then the set $P = \{p_i \mid i > 0\}$ has size at most k.

This is remarkable, because the number of different points $(X_{i+j_1}, X_{i+j_2}, \ldots, X_{i+j_d})$ equals the period length of the generator, thus is close or even equal to m. We prove the theorem in the next section, where it is an easy consequence of the proof of Theorem 2.1. Let us derive the undesirable implications of it now. Assume for the moment that for every i, p_i is a random point in P. An argumentation similar to the one used to establish the birthday-paradox [1] then shows that after choosing only about $\sqrt{k}$ points, we expect

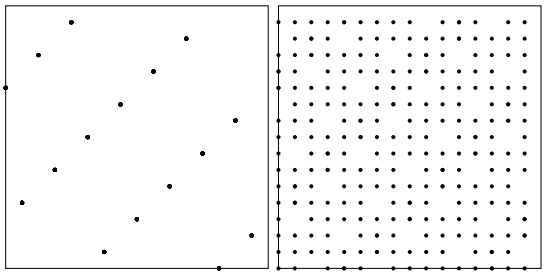

Figure 1: Two-dimensional points obtained from the 4 lower-order bits (left) resp. the 4 higher-order bits (right) of `knuth8`

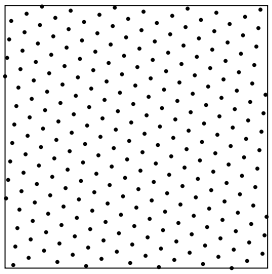

Figure 2: Two-dimensional points obtained from `knuth8`

to have selected some point twice! Coming back to the case where $k = 2^{24}$, this means that only about 2^{12} points can be chosen before we obtain a duplicated point. As the size of the sample gets larger, more and more duplicated points will occur.

In contrast, if we choose truly random d-dimensional points with coordinates at most k, we need to sample roughly

$$\sqrt{k^d} = k^{d/2}$$

points before we expect any duplicated point. Also, taking the k *most* significant bits of X_i to define Y_i usually has less dangerous effects, because the size of the set P as defined above will still be close to m.

Theorem 3.1 is visualized in Figure 1 for the generator `knuth8` defined in (3). To the left, the set of points

$$P = \{(X_i \bmod 16, X_{i+1} \bmod 16) \mid 1 \le i \le 256\}$$

obtained from the 4 lower-order bits is drawn in the square $[0, 16]^2$. To the right, the point set

$$P' = \{(X_i \operatorname{div} 16, X_{i+1} \operatorname{div} 16) \mid 1 \le i \le 256\}$$

obtained from the 4 higher-order bits is depicted. While P' still looks reasonably 'random', P consists of only 16 points of high multiplicity each. Figure 2 shows the distribution of the original points

$$\{(X_i, X_{i+1}) \mid 1 \le i \le 256\},$$

drawn in the square $[0, 256]^2$.

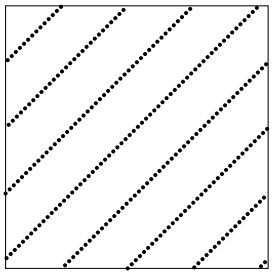

Figure 3: Distribution of pairs $(X_i/256, X_{i+1}/256)$ for `adhoc8`

The practical consequences are quite apparent: if we choose — as in the previous scenario — a set of many (w.r.t. $\sqrt{k}$) pseudorandom points from lower-order bits, we will get many duplicates. A huge number of orientation determinants D of point triples will then actually be zero. This means, a point configuration obtained in this way is extremely degenerate, just not what you would expect from truly random points.

Exactly in this context, Kettner and Welzl [9] have performed tests to evaluate the precision of floating-point arithmetic, as follows. A sequence of n random points is generated, and the *consistency* of every triple is checked. A triple is called consistent if the different ways to assign the points to p, q and r in (5) all lead to the same sign of D. A triple can only be inconsistent if the true value of D is small. In our scenario, such a test would report far too many inconsistent triples, because of the many triples with $D = 0$. Interestingly, when (5) is used, this will not happen, because the formula correctly computes $D = 0$ when two of the points are equal. In higher dimensions, or for more complicated predicates sensitive to degeneracies, this is not necessarily the case; it is not even the case for $d = 2$, if instead of (5), D is evaluated according to the less robust formula

$$D = p_x q_y + r_x p_y + q_x r_y - r_x q_y - p_x r_y - q_x p_y.$$

The advice in this scenario is not to use the lower-order bits of an LCG sequence, because the actual 'entropy' of the sequence is contained in the higher-order bits.

4. THE STRUCTURE OF LCG

If an LCG with modulus m has full period, the numbers $X_i/m, i = 1, \ldots, m$ define an equidistant partition of the unit interval, as fine as possible according to the given parameters. With respect to this measure of quality, the generator `adhoc8`, defined by the parameters

$$a = 193, \quad c = 73, \quad m = 2^8$$

is as good as `knuth8`, because it also has full period. A difference in quality becomes apparent when we consider *pairs* of numbers $(X_i/m, X_{i+1}/m), i = 1, \ldots, m$. Again, it is desirable that the distribution of those pairs approximates the uniform distribution over the unit square in some sense. From Figure 2 it is intuitively clear that `knuth8` reaches this to some extent, while Figure 3 shows that `adhoc8` has clear deficiencies.

The so-called *spectral test* formalizes this intuition (in one

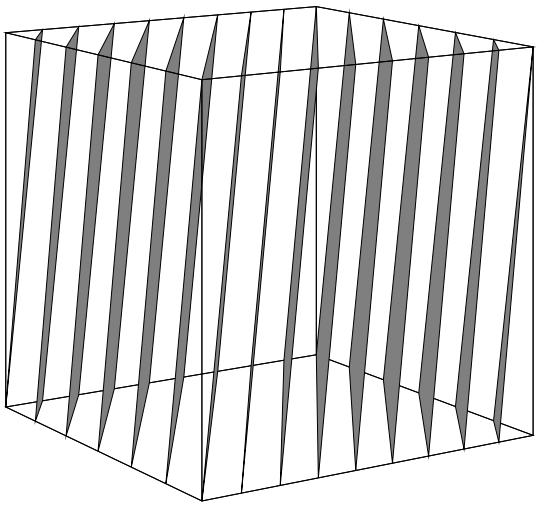

Figure 4: The 15 planes of `randu`

possible way) and defines a procedure for measuring the quality of a generator. It considers strips (regions between parallel lines) in the unit square. The widest strip not containing a point $(X_i/m, X_{i+1}/m)$ defines the quality of the generator: the inverse of its width is the (two-dimensional) measure of quality.

According to this measure, the quality of `adhoc8` is $\sqrt{32}$, while `knuth8` attains $\sqrt{274}$. One can show that the best possible quality attainable by any LCG is about $\sqrt{m}$, meaning that `knuth8` is very good.

The spectral test can also be carried out in higher dimensions; for example, in three dimensions, one considers points $(X_i/m, X_{i+1}/m, X_{i+2}/m)$ in the unit cube, and the quality is defined by the widest empty strip between parallel planes. The spectral test has become famous for showing that the generator `randu`—widely used in the sixties—is actually a very bad generator. `randu` has the parameters

$$a = 65539, \quad c = 0, \quad m = 2^{31},$$

and is therefore a *multiplicative* generator. The value $c = 0$ is quite common, although multiplicative generators cannot have full period. `randu` has a period length of 2^{29} which is best possible under the given parameters. The multiplier $a = 2^{16}+3$, however, has turned out to be very poor (according to Knuth, it is almost the worst conceivable choice [10]). The result is that all 2^{29} points $(X_i/m, X_{i+1}/m, X_{i+2}/m)$ live in only 15 parallel planes which are pretty far apart, see Figure 4.

The threedimensional quality of this generator is $\sqrt{118}$, while the desirable value is roughly $\sqrt[3]{m} \approx 1000$. This means, whenever a set of random points in 3-space is to be generated, `randu` behaves spectacularly 'nonrandom'.

As it turns out, the consideration of d-tuples underlying the spectral test in dimension d is the key to the structure of LCG in general, and also leads to proofs of Theorems 2.1 and 3.1.

As the Figures 2, 3 and 4 already suggest, the set of d-dimensional points $(X_i, X_{i+1}, \ldots, X_{i+d-1})$ has the structure of an *integer lattice*. To prove this, we consider a slightly more general scenario which directly allows us to prove Theorem 2.1.

Consider, as in the theorem, two sequences $i_1, \ldots, i_d$ and $j_1, \ldots, j_d$ which are fixed for the rest of the proof. Let us define

$$\mathbf{X}_i := (X_{i+j_1}, \ldots, X_{i+j_d}), \quad i \geq 0.$$

Furthermore, we observe that

$$X_{i+s} - X_{\ell+s} \equiv a^s (X_i - X_\ell) \pmod{m}, \quad i, \ell, s \geq 0,$$

which easily follows from (1) by induction on s. This implies

$$\mathbf{X}_i - \mathbf{X}_\ell \equiv (X_i - X_\ell) \begin{pmatrix} a^{j_1} \\ a^{j_2} \\ \vdots \\ a^{j_d} \end{pmatrix} \pmod{m},$$

for all i, ℓ, where congruence is defined componentwise. If we apply the definition of the congruence relation, we get

$$\mathbf{X}_i - \mathbf{X}_\ell = (X_i - X_\ell) \begin{pmatrix} a^{j_1} \\ a^{j_2} \\ \vdots \\ a^{j_d} \end{pmatrix} + m \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_d \end{pmatrix}, \qquad (6)$$

where the $u_1, \ldots, u_d$ are integers. W.l.o.g. assume that $j_1 = \min\{j_1, \ldots, j_d\}$. Then, $\mathbf{X}_i - \mathbf{X}_\ell$ is an integer linear combination of the $d+1$ vectors

$$\begin{pmatrix} 1 \\ a^{j_2-j_1} \\ \vdots \\ a^{j_d-j_1} \end{pmatrix}, \begin{pmatrix} m \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ m \\ \vdots \\ 0 \end{pmatrix}, \ldots, \begin{pmatrix} 0 \\ 0 \\ \vdots \\ m \end{pmatrix}. \qquad (7)$$

The vector $(m, 0, \ldots, 0)^T$ ist even redundant, because it is an integer combination of the others. Thus, for all i, ℓ, $\mathbf{X}_i - \mathbf{X}_\ell$ is a member of an *integer lattice*, generated by the integer linear combinations of a *lattice basis*

$$B = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ a^{j_2-j_1} & m & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a^{j_d-j_1} & 0 & \cdots & m \end{pmatrix}.$$

Now consider the matrices A and A' of Theorem 2.1. We get

$$\begin{aligned} \det(A) &= \det(\mathbf{X}_{i_1}, \mathbf{X}_{i_2}, \ldots, \mathbf{X}_{i_d}) \\ &= \det(\mathbf{X}_{i_1}, \mathbf{X}_{i_2} - \mathbf{X}_{i_1}, \ldots, \mathbf{X}_{i_d} - \mathbf{X}_{i_1}) \\ &= \frac{1}{m} \det(L), \end{aligned}$$

with

$$L := (m\mathbf{X}_{i_1}, \mathbf{X}_{i_2} - \mathbf{X}_{i_1}, \ldots, \mathbf{X}_{i_d} - \mathbf{X}_{i_1}).$$

All columns of L are elements of the lattice generated by B; for the last $d-1$ columns, we have already shown this, and

for the first column $m\mathbf{X}_{i_1}$ this follows from the fact that the set of vectors in (7) generates the lattice.

This again means that L is representable in the form

$$L = BQ$$

where Q is an integer matrix. In particluar, we then have

$$\begin{aligned} \det(A) &= \frac{1}{m}\det(L) = \frac{1}{m}\det(B)\det(Q) \\ &= m^{d-2}\det(Q). \end{aligned} \tag{8}$$

In case of A' we proceed similarly. With

$$\mathbf{Y}_i := (X_{i+j_1}, \ldots, X_{i+j_{d-1}}), \quad 0 \le i,$$

we have

$$\begin{aligned} & \det(A') \\ = \ & \det\left(\begin{matrix}\mathbf{Y}_{i_1}\\1\end{matrix}, \begin{matrix}\mathbf{Y}_{i_2}\\1\end{matrix}, \ldots, \begin{matrix}\mathbf{Y}_{i_d}\\1\end{matrix}\right) \\ = \ & \det\left(\begin{matrix}\mathbf{Y}_{i_1}\\1\end{matrix}, \begin{matrix}\mathbf{Y}_{i_2}-\mathbf{Y}_{i_1}\\0\end{matrix}, \ldots, \begin{matrix}\mathbf{Y}_{i_d}-\mathbf{Y}_{i_1}\\0\end{matrix}\right) \\ = \ & (-1)^{d-1}\det(\mathbf{Y}_{i_2}-\mathbf{Y}_{i_1}, \ldots, \mathbf{Y}_{i_d}-\mathbf{Y}_{i_1}), \end{aligned}$$

and a factor of m^{d-2} follows as before.

Theorem 3.1 can now easily be derived from the general structure. For k a divisor of m, (6) with $\ell = 0$ gives

$$(\mathbf{X}_i - \mathbf{X}_0) \mod k = (X_i - X_0)\begin{pmatrix} a^{j_1} \\ a^{j_2} \\ \vdots \\ a^{j_d} \end{pmatrix} \mod k,$$

which shows that the points $(\mathbf{X}_i - \mathbf{X}_0) \mod k$ can take only k different values over all i. The same is true for the points $p_i = \mathbf{X}_i \mod k$ forming the set P in the theorem.

5. CONCLUSION

I have described regularities in (orientation) determinants (matrices A and A' in Theorem 2.1), which are completely build from pseudorandom input, or have a single 'nonrandom' column of ones. The regularity theorem can easily be extended to other cases. In general, if a $d \times d$-matrix M contains ℓ rows or columns of A or A', its determinant still contains a factor of $m^{\ell-2}$, m the modulus of the LCG used to generate the entries. This follows by expanding the determinant of M according to Laplace's theorem.

In particular, the $(d+2) \times (d+2)$-determinant one evaluates for the d-dimensional *in-sphere test* is of this form for pseudorandom input, because it contains $d+1$ columns of a suitable orientation determinant A'. This means, already the twodimensional in-circle test is subject to the regularities described for the orientation test.

An important question is of course how to avoid regularities, in the scenarios above, and in other tests involving pseudorandom input. To get rid of the effects described here, one might simply not use LCG. Other generation methods are available, and if not too many random numbers are needed, I would even recommend `HotBits`. One should be aware that there is no way to avoid all possible pitfalls caused by using pseudorandom rather than truly random numbers. This implies that test results should be more carefully examined, in order to spot hidden regularities. Luckily, I did not have to learn this the hard way: *before* publishing the performance results of my linear programming solver [5], I accidentally discovered the phenomenon described in the first scenario above—during debugging of my Q-pivoting routine.

Another conclusion is that test reports should always mention the method according to which pseudorandom test data were generated. If the machine architecture and even the compiler (including the optimization level that was used) are mentioned, then why not the pseudorandom number generator? Reproducability of results is only guaranteed if all parameters of the test environment are known. As already observed in the introduction, the generator is often not considered as a crucial parameter; I hope I was able to argue that this is a mistake.

A completely different view of the regularity Theorem 2.1 is obtained when one asks whether it is possible to make constructive use of it. For example, a tempting idea to speed up algorithms using exact arithmetic is the following: round any input point to the nearest point $(X_i, \ldots, X_{i+d-1})$ in the integer lattice coming with some LCG (see Section 4). According to the first scenario, the arithmetic will be very fast on the rounded points. The catch of course is that rounding to a general integer lattice is an NP-hard problem, if d is a parameter, and still difficult if d is fixed: for this, we would have to perform exactly the full arithmetic we wanted to avoid by using the rounded points. Still, it is conceivable that the structure revealed by Theorem 2.1 might prove useful in certain applications.

6. ACKNOWLEDGMENT

I like to thank the SoCG reviewers for extensive and helpful comments.

7. REFERENCES

[1] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA., 1990.

[2] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, 1997.

[3] O. Devillers and F. P. Preparata. A probabilistic analysis of the power of arithmetic filters. Technical Report CS-96-27, Center for Geometric Computing, Dept. Computer Science, Brown Univ., 1996.

[4] J. Edmonds and J.-F. Maurras. Note sur les Q-matrices d'Edmonds. *Recherche Opérationnelle (RAIRO)*, 31(2):203–209, 1997.

[5] B. Gärtner. Exact arithmetic at low cost – a case study in linear programming. *Computational Geometry - Theory and Applications*, 13:121–139, 1999.

[6] M. Grötschel, L. Lovász, and A. Schrijver. *Geometric Algorithms and Combinatorial Optimization*, volume 2

of *Algorithms and Combinatorics*. Springer-Verlag, Berlin Heidelberg, 1988.

[7] N. Henze. Random triangles in convex regions. *J. Appl. Prob.*, 20:111–125, 1983.

[8] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, 1996.

[9] L. Kettner and E. Welzl. One sided error predicates in geometric computing. In K. Mehlhorn, editor, *Proc. 15th IFIP World Computer Congress*, Fundamentals–Foundations of Computer Science, pages 13–26, 1998.

[10] D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, 3rd edition, 1998.

[11] G. Marsaglia. Regularities in congruential random number generators. *Numerische Mathematik*, 16:8–10, 1970.

[12] J. R. Shewchuk. Robust adaptive floating-point geometric predicates. *Discr. Comput. Geom.*, 18(3):305–363, 1997.