A NEW CLASS OF RANDOM NUMBER GENERATORS

BY

GEORGE MARSAGLIA AND ARIF ZAMAN

FSU STATISTICS REPORT #M802
A New Class of Random Number Generators

by

George Marsaglia and Arif Zaman

FSU Technical Report No. M-802

March, 1989

Supercomputer Computations Research Institute
and
Department of Statistics
The Florida State University
Tallahassee, Florida 32306-3033

Research supported by NSF Grant DMS-8807976

AMS (MOS) subject classifications: 65C10, 10A30.

Key Words: random number generators, Monte Carlo, lagged-Fibonacci, add-with-carry, subtract-with-borrow.
A New Class of Random Number Generators

George Marsaglia*
Arif Zaman*
Supercomputer Computations Research Institute
and
Department of Statistics
The Florida State University
Tallahassee, Florida 32306

Abstract

We introduce a new class of generators of two types: add-with-carry
and subtract-with-borrow. Related to lagged-Fibonacci generators, the
new class has interesting underlying theory, astonishingly long periods and
provable uniformity for full sequences. Among several that we mention, we
recommend particularly promising ones that will generate a sequence of
$2^{1750}$ bits, or a sequence of $2^{1177}$ 32-bit integers, or a sequence of $2^{226}$ reals
with 24-bit fractions—all using simple computer arithmetic (subtraction)
and a few memory locations.

1 Introduction

This section briefly describes the most commonly used random number genera-
tors to provide the context for a new class of generators. The new generators can
be grouped into four classes: two add-with-carry and two subtract-with-borrow
generators. In section 2 we give a simple example of an add-with-carry generator,
and section 3 describes the two subtract-with-borrow generators. Section 4
provides an analysis of the periods of these generators, along with a description
of the second type of add-with-carry generator, in section 4.4. In sections 5–7 we
subject the proposed generators to theoretical and experimental tests to suggest
the quality of random numbers produced. Section 8 summarizes the conclusions
and provides recommendations for implementation.

*Research supported by NSF Grant DMS-8807976

Key Words: random number generators, Monte Carlo, lagged-Fibonacci,
add-with-carry, subtract-with-borrow.

AMS (MOS) subject classifications: 65C10, 10A30.
Virtually all random number generators are based on theory which may be
described as follows: We have a finite set $X$ and a function $f : X \rightarrow X$ that
takes elements of $X$ into other elements of $X$. Given an initial (seed) value,
$x \in X$, the generated sequence is

$$x, f(x), f^2(x), f^3(x), \ldots,$$

where $f^2(x)$ means $f(f(x))$, $f^3(x)$ means $f(f(f(x))) = f(f(x))$ and so on. The
three most common classes of random number generators are 1) Congruential,
2) Shift-register and 3) Lagged-Fibonacci.

For congruential generators, the finite set $X$ is the set of reduced residues
of some modulus $m$ and $f(x) = ax + b \mod m$. Thus, with an initial element
$x_0 \in X$, the generated sequence is

$$x_0, x_1, x_2, \ldots \text{ with } x_{n+1} = ax_n + b \mod m.$$

A wide variety of choices for $a, b$ and $m$ have been described in the literature;
see, particularly, Knuth[1] or Marsaglia [3] for methods for finding periods and
establishing structure of congruential sequences.

For shift-register generators, the finite set $X$ is the set of $1 \times k$ binary
vectors $x = (b_1, b_2, \ldots, b_k)$ and the function $f$ is a linear transformation,
$f(x) = xT$, with $T$ a $k \times k$ binary matrix and all arithmetic mod 2. With an initial
binary vector $x$ the sequence is

$$x, xT, xT^2, xT^3, \ldots$$

with the matrix $T$ chosen so that the period is long and multiplication by
$T$ is reasonably fast in computer implementation. See Marsaglia and Tsay
[4] for methods for finding periods and establishing structure of shift-register
sequences. Such generators are sometimes called Tausworthe generators.

For lagged-Fibonacci generators, the finite set $X$ is the set of $1 \times r$
vectors $x = (x_1, x_2, \ldots, x_r)$ with elements $x_i$ in some finite set $S$ on which there
is a binary operation $\circ$. The function $f$ is defined by

$$f(x_1, x_2, \ldots, x_r) = (x_2, x_3, x_4, \ldots, x_r, x_1 \circ x_{r+1-s}).$$

Informally, we describe a lagged-Fibonacci sequence as a set of $r$ seed values
followed by the rule for generating succeeding values:

$$x_1, x_2, \ldots, x_r, x_{r+1}, \ldots \text{ with } x_n = x_{n-r} \circ x_{n-s},$$

but to formally define and establish the period and structure of such sequences
we view them as iterates $x, f(x), f^2(x), \ldots$ on the set $X$ of $1 \times r$ vectors with
elements in the set $S$ on which the binary operation $\circ$ is defined.

Various choices for $S$ and $\circ$ lead to interesting sequences—for example, when
$S$ is the set of reduced residues of some modulus $m$ and $\circ$ is addition or sub-
traction mod $m$; $S$ is the set of reduced residues relatively prime to $m$ and $\circ$ is
multiplication; \( S \) is the set of \( 1 \times k \) binary vectors and \( \circ \) is addition of binary vectors (exclusive-or); \( S \) is the set of floating-point computer numbers \( 0 \leq x < 1 \) having 24-bit fractions and \( x \circ y = \{ \mathrm{if} \ x > y \ \mathrm{then} \ x - y \ \mathrm{else} \ x - y + 1 \} \). Such generators are described in [5,6,7], and methods for establishing periods are in [4].

While examples of generators of each of the three standard methods described above are widely used and—for most purposes—work quite well, it is worth considering new methods. All standard generators (with the exception of lagged-Fibonacci using multiplication) fail one or more stringent tests of randomness such as those described in [3], and many of them have periods too short for the huge samples that current computer speeds make possible. For these reasons, we offer the class of add-with-carry and subtract-with-borrow generators we now proceed to define.

## 2 The New Class: add-with-carry generators

We introduce add-with-carry generators with a simple example. Consider the classical Fibonacci sequence

\[ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots, \]

with each element the sum of the previous two. If we take this sequence mod 10, we have an example of a lagged-Fibonacci sequence with lags \( r = 2 \) and \( s = 1 \) and binary operation \( v \circ w = v + w \) mod 10:

\[ 0, 1, 2, 3, 5, 8, 3, 1, 4, 5, 9, 4, 3, 7, \ldots. \]

The informal description of the sequence is \( x_n = x_{n-2} + x_{n-1} \) mod 10, but to formally describe it and define and establish its period we need the finite set \( X \) of \( 1 \times 2 \) vectors \( x = (x_1, x_2) \) with elements reduced residues of 10 and the iterating function \( f \) defined by \( f(x_1, x_2) = (x_2, x_1 + x_2 \text{ mod } m) \). Since \( f \) has an inverse, for any initial vector \( x \in X \) the sequence

\[ x, f(x), f^2(x), f^3(x), \ldots \]

is strictly periodic. Depending on the initial vector \( x \), there is a longest cycle of period 60 and shorter cycles of periods 1, 3, 4, 12 and 20. Each period is the lcm of the periods for moduli 2 and 5.

Now consider the add-with-carry version of this generator. We assign two initial values, say 0, 1 and an initial "carry bit", say 0. Then each new digit is the sum of the previous two digits plus the carry bit. The result is taken mod 10 and the next carry bit set to 1 or 0 according to whether or not the sum exceeds 10. Using a superscript to indicate the carry bit, the sequence of digits becomes

\[ 0, 1^0, 1^0, 2^0, 3^0, 5^0, 8^0, 3^1, 2^1, 6^0, 8^0, 4^1, 3^1, 8^0, 1^1, 0^1, 2^0, \ldots \]

3
Formally, as before, we have a sequence of iterates \( x, f(x), f^2(x), \ldots \). But now our \( x' \)'s come from the set \( X \) of \( 1 \times 3 \) vectors \( x = (x_1, x_2, c) \) with \( x_1, x_2 \) reduced residues of 10 and \( c \) the "carry bit", 0 or 1. Then the iterating function \( f \) is

\[
f(x_1, x_2, c) = \begin{cases} 
(x_2, x_1 + x_2 + c, 0) & \text{if } x_1 + x_2 + c < 10 \\
(x_2, x_1 + x_2 + c - 10, 1) & \text{if } x_1 + x_2 + c \geq 10
\end{cases}
\]

For initial vectors \( x = (x_1, x_2, 0) \) with \( x_1 < x_2 \) or \( x = (x_1, x_2, 1) \) with \( x_1 > x_2 \), the sequence of iterates \( x, f(x), f^2(x), \ldots \) is strictly periodic with period 108. If the initial vector \( x \) is not of those two types, and not \((0, 0, 0)\) or \((9, 9, 1)\), then the sequence beginning with \( f(x) \) is strictly periodic with period 108, but the "seed" vector \( x \) may not reappear in the sequence. We develop rules for finding the period and assigning seed vectors for add-with-carry generators below.

As with lagged-Fibonacci sequences, a whole class of such generators can be created by altering the lags from the values \( r = 2 \) and \( s = 1 \) used in the previous example. The general add-with-carry generator has a base \( b \), lags \( r \) and \( s \) with \( r > s \), a seed vector \( x = (x_1, x_2, \ldots, x_r, c) \) with elements "digits" of the base \( b \). Then the generated sequence is \( x, f(x), f^2(x), f^3(x), \ldots \) with

\[
f(x_1, \ldots, x_r, c) = \begin{cases} 
(x_2, \ldots, x_r, x_{r+1-s} + x_1 + c, 0) & \text{if } x_{r+1-s} + x_1 + c < b \\
(x_2, \ldots, x_r, x_{r+1-s} + x_1 + c - b, 1) & \text{if } x_{r+1-s} + x_1 + c \geq b
\end{cases}
\]

3 The New Class: subtract-with-borrow generators

We illustrate a subtract-with-borrow generator with a numerical example using the same parameters as the add-with-carry generator above: base 10, lags \( r = 2 \), \( s = 1 \), but now each new digit is a difference: the lag-1 digit is subtracted from the lag-2 digit and the carry bit is subtracted as well. If the result is positive, it becomes the new element with carry bit set to 0; if negative, 10 is added and the new carry bit is set to 1. Thus, with seeds 0, 1 and initial carry 0, the sequence becomes

\[
0, 1^0, 9^1, 1^1, 7^0, 4^1, 2^0, 2^0, 0^0, 2^0, 8^1, 3^1, 4^0, 1^1, 2^0, 9^1, 2^1, 6^0, \ldots
\]

Formally, for this generator, \( X \) is the set if \( 1 \times 3 \) vectors \( x = (x_1, x_2, c) \), with \( x_1, x_2 \) reduced residues for base 10 and \( c \in \{0, 1\} \). But now the iterating function \( f \) is

\[
f(x_1, x_2, c) = \begin{cases} 
(x_2, x_1 - x_2 - c, 0) & \text{if } x_1 - x_2 - c \geq 0 \\
(x_2, x_1 - x_2 - c + 10, 1) & \text{if } x_1 - x_2 - c < 0
\end{cases}
\]

This makes precise the meaning of the generator we informally describe by \( x_n = x_{n-2} - x_{n-1} - c \).
Since the order of subtraction matters, a different generator would be formed by \( x_n = x_{n-1} - x_{n-2} - c \). More generally, the informal rules for subtract-with-borrow generators with lags \( r \) and \( s \) are \( x_n = x_{n-r} - x_{n-s} - c \) and \( x_n = x_{n-s} - x_{n-r} - c \). (Here and throughout, we assume for the two lags \( r \) and \( s \) that \( r > s \).) To formally define the generators and establish their periods we have the finite set \( X \) of \( 1 \times (r + 1) \) vectors \( x = (x_1, x_2, \ldots, x_r, c) \) with the \( x \)'s reduced residues of some base \( b \) and the iterating function \( f \) defined for the \( x_n = x_{n-s} - x_{n-r} - c \) case by

\[
f(x_1, \ldots, x_r, c) = \begin{cases} 
(x_2, \ldots, x_r, x_{r+1-s} - x_{1} - c, 0) & \text{if } x_{r+1-s} - x_{1} - c \geq 0 \\
(x_2, \ldots, x_r, x_{r+1-s} - x_{1} - c + b, 1) & \text{if } x_{r+1-s} - x_{1} - c < 0
\end{cases}
\]

and with \( f \) for the \( x_n = x_{n-r} - x_{n-s} - c \) case by

\[
f(x_1, \ldots, x_r, c) = \begin{cases} 
(x_2, \ldots, x_r, x_{1} - x_{r+1-s} - c, 0) & \text{if } x_1 - x_{r+1-s} - c \geq 0 \\
(x_2, \ldots, x_r, x_{1} - x_{r+1-s} - c + b, 1) & \text{if } x_1 - x_{r+1-s} - c < 0
\end{cases}
\]

With appropriately chosen base \( b \), lags \( r \) and \( s \) and seed vector \( x \) the generated sequence \( x, f(x), f^2(x), \ldots \) will be strictly periodic with period \( b^r - b^s - 2 \) for the \( x_n = x_{n-s} - x_{n-r} - c \) generator, and period \( b^r - b^s - 2 \) for \( x_n = x_{n-r} - x_{n-s} - c \).

4 Periods of the New Generators

The fundamental result for establishing periods comes from recognizing that these generators behave very much like the operation of long addition with carry (that some of us in the pre-calculator age learned in school). Once this addition is explicitly written, it is easy to recognize that the sequence of digits formed by the add-with-carry or subtract-with-borrow operation is, in reverse order, the same as the sequence of digits of a proper fraction \( k/(b^r \pm b^s \pm 1) \). Here \( r, s \) and \( b \) are the lags and the base, respectively, and choices of \( \pm \) depend on the particular generator.

4.1 Results from number theory

We need some background material from number theory to establish periods of the new class of generators. This elementary material has been known for hundreds of years, but it is seldom mentioned in modern books. We summarize it here. It concerns the decimal expansions of fractions—expansions to a base \( b \), rather than the customary base 10—but we illustrate with the more familiar base 10.

Let the modulus \( m \) be chosen and consider the group \( G \) of \( \phi(m) \) reduced residues of \( m \) relatively prime to \( m \). For \( k \) in \( G \) we want the base-\( b \) expansion of \( k/m \). That expansion is strictly periodic with period the order of \( b \) in the group \( G \). (This requires the assumption that \( b \in G \).) The cyclic subgroup generated
by \( b \) partitions \( G \) into cosets. Two elements \( g \) and \( h \) of \( G \) are equivalent (belong to the same coset) if \( g = hb^j \) for some \( j \). If two elements \( g, h \) belong to the same coset then \( g/m \) and \( h/m \) have the same base-\( b \) expansion with period the order of \( b \), except that the "digits" in their periods are cyclic permutations of one another.

Example: modulus \( m = 39 \), base \( b = 10 \). The powers of \( 10 \mod m \) generate the cyclic subgroup \( \{1, 10, 22, 25, 16, 4\} \), so the order of \( 10 \) for modulus 39 is 6. Successive elements of that subgroup have a common set of digits in the period-6 decimal expansion of \( k/m \), each a cyclic permutation of the previous one:

\[
\frac{1}{39} = .025641025 \ldots, \quad \frac{10}{39} = .256410256 \ldots, \quad \frac{22}{39} = .564102564 \ldots,
\]
\[
\frac{25}{39} = .641025641 \ldots, \quad \frac{16}{39} = .410256410 \ldots, \quad \frac{4}{39} = .102564102 \ldots.
\]

Now choose an element not in the first coset, say 2. Its coset is \( \{2, 20, 5, 11, 32, 8\} \), and ratios \( k/m \) with \( k \) from that coset all have the same digits in their period-6 decimal expansions, shifted by one because of successive multiplications by the base 10:

\[
\frac{2}{39} = .051282051 \ldots, \quad \frac{20}{39} = .512820512 \ldots, \quad \frac{5}{39} = .128205128 \ldots,
\]
\[
\frac{11}{39} = .282051282 \ldots, \quad \frac{16}{39} = .820512820 \ldots, \quad \frac{8}{39} = .205128205 \ldots.
\]

For our purposes, we want to choose bases \( b \) related to computer implementation, particularly \( b = 2, 2^{16}, 2^{24} \) and \( 2^{32} \). We want the modulus \( m \) to be a prime and \( b \) a primitive root. In some cases we will have to settle for \( b \)'s not quite of full period, because of we cannot find primes \( m \) for which \( b \) is a primitive root.

### 4.2 Periods of the new generators: add-with-carry

We now illustrate methods for establishing the periods of the add-with-carry or subtract-with-borrow generators. In the simplest case, consider the classical Fibonacci sequence with starting values 1,2 and new terms the sum of the previous two, but with carry mod 10. The sequence is 1, 2, 3, 5, 8, 3, 2, 6, 8, 4, 3, 8, ..... Let \( I \) be the integer whose digits are the first twelve of that sequence, in reverse order:

\[
I = 834862385321.
\]

Now shift \( I \) left one (form 10\( I \)) and add it to \( I \), getting

\[
\begin{align*}
I & = 834862385321 \\
10I & = 8348623853210 \\
11I & = 9183486238531
\end{align*}
\]
Because of the rule for forming the sequence, there will be a substring
of digits common to each of those three levels; let \( S \) be the integer formed by
those digits—in this case, \( S = 8348623853 \). Then
\[
10^2 S + 21 + 10^3 S + 210 = 91 \times 10^{11} + 10^1 S + 1,
\]
leading to
\[
(10^2 + 10 - 1) S = 91 \times 10^{10} - 23.
\]
Thus
\[
S = \frac{10^{10} \times 91}{109} - 23.
\]
Now \( S \) is an integer, so the fractional part of \( 10^{10}(91/109) \) must cancel that
of 23/109, and it follows that \( S \) is the integer part of \( 10^{10}(91/109) \), that is, \( S \)'s
digits are the first ten digits of the decimal expansion of 91/109.

We may apply such an argument to the reversed digits in an arbitrarily long
finite string formed by \( x_n = x_{n-r} + x_{n-s} + c \) mod \( b \). The period of the sequence
will be the period of the base-\( b \) expansion of a proper fraction of the form \( k/m \)
with \( m = b^r + b^s - 1 \). When \( k \) is relatively prime to \( m \) the period will be the
order of \( b \) for modulus \( m \). Making \( m \) a prime ensures this, of course.

4.3 Periods of the generators: subtract-with-borrow

For sequences generated by \( x_n = x_{n-r} - x_{n-s} - c \) mod \( b \), we may establish the
period as follows: choose \( r \) starting values and generate a sequence of arbitrary
length. As before, let \( I \) be the integer formed by those digits in reverse order. For
example, with \( b = 10 \), \( r = 5 \), \( s = 3 \) and starting values 5, 9, 7, 7, 7 the reversed
string of 15 digits yields \( I = 304285901877795 \). Shift \( I \) left by \( s \) positions
(multiply by \( b^s \)) and add to get

\[
\begin{align*}
I &= 304285901877795 \\
10000I &= 304285901877795000 \\
1001I &= 304590187779672795
\end{align*}
\]

The string \( S = 59018779 \) appears in each line. Cancel the leading 304 and the
trailing 5 to get a simplified equation for \( S \):
\[
30428b^9 + S + 28b^{13} + b^3 S + 500 = b^8 S + 672795.
\]
Thus
\[
58428b^9 - 64779 = (b^5 - b^3 - 1) S,
\]
and, with \( m = b^5 - b^3 - 1 = 98999 \),
\[
S = \frac{58428}{98999} - \frac{64779}{98999}.
\]
Since \( S \) is an integer, it must be the integer part of the first term, that is, the digits of \( S \) are the leading 9 digits of the decimal expansion of \( 58428/98999 \):
\[
S = 590187779.
\]

We may apply such an argument to the reversed string of an arbitrary length sequence with \( r \) initial values and, for \( n > r \), \( x_n = x_{n-r} - x_{n-s} - c \mod b \). Its period will be the order of \( b \) for modulus \( m = \beta^r - b^r - 1 \), except for seed vectors \((0, 0, \ldots, 0, 1)\) or \((b-1, b-1, \ldots, b-1, 1)\), for which \( f(x) = x \) and the period is 1.

In short, for \( x_{n-r} - x_{n-s} - c \) with \( r > s \), write an initial string backwards, shift it left \( s \) positions and add. The resulting three lines will have a common string \( S \) that leads to a linear equation whose solution is \( S = b^d \frac{k}{m} - \frac{j}{m} \) with \( m = b^r - b^s - 1 \) and \( 1 \leq k, j \leq m \), so that the base-\( b \) digits of \( S \) are the first \( d \) digits of the base-\( b \) expansion of the fraction \( k/m \).

For the case \( x_n = x_{n-r} - x_{n-s} - c \) with \( r > s \) a similar derivation holds, except that \( I \) is added to \( b^r I \) and the common string \( S \) has a solution of the form \( S = b^d \frac{j}{m} - \frac{j}{m} \) with \( m = b^r - b^s + 1 \).

4.4 Periods: the complementary add-with-carry generator

Finally, we consider sequences formed by the more complicated rule

if \( t = x_{n-r} + x_{n-s} + c < b \) then \( x_n = b - 1 - t \) and \( c = 0 \)

else \( x_n = 2b - 1 - t \) and \( c = 1 \).

This rule leads to expansions of \( k/m \) with \( m = b^r + b^s + 1 \), so that certain bases and choices \( r, s \) may lead to large primes \( m \) so desirable that the more complicated generating procedure may be justified.

We illustrate with this example: \( b = 10 \), \( r = 3 \), \( s = 1 \) and initial values 5, 3, 9. Then the sequence, through 17 terms, is

\[
5, 3, 9, 5, 0, 0, 4, 5, 4, 1, 3, 2, 6, 0, 7, 2.
\]

As before, form the integer \( I \) with those digits in reverse order. Then add it to \( b^3 I \), getting

\[
\begin{align*}
I &= 26706231454005935 \\
1000I &= 26706231454005935000 \\
1001I &= 26732937685459940935
\end{align*}
\]

Identify the string \( S = 267062314540059 \) in \( I \) and \( b^3 I \). The bottom line does not contain \( S \), but it contains the 9's complement of \( S \): \( 732937685459940 = b^{16} - 1 - S \). This provides a linear equation in \( S \),
\[
(b^3 + b + 1)S = 26b^{16} - 350 + b^{10} - 10.
\]
leading to

\[(b^3 + b + 1)S = 26b^{10} - 350 + b^{10} - 10,\]

or, with \(b = 10\),

\[S = 10^{11} \frac{270}{1011} - \frac{360}{1011}.\]

Once again, since \(S\) is an integer, its digits must be the leading digits of the expansion of \(k/m\), this time with \(k = 270\) and \(m = b^r + b^s + 1 = 1011\).

A similar argument applies to any finite sequence generated by the complicated recursion—the period is the order of \(b\) for modulus \(m = b^r + b^s + 1\).

### 4.5 Summary for periods and legitimate seed vectors

When implementing these generators, we will always choose \(r, s\) so that \(m\) is a prime and, whenever possible, \(b\) is a primitive root of \(m\). Occasionally we will have to settle for primes \(m\) for which the order of \(b\) is nearly, but not quite, \(m - 1\).

Any starting (seed) vector of \(r\) digits and a carry will produce a sequence that is ultimately periodic, but it may not be strictly periodic—after a few iterations (at most \(r\), the periodic cycle begins. We say that a seed vector is “legitimate” if it produces a strictly periodic sequence: the first vector to be repeated in the sequence \(x, f(x), f^2(x), \ldots\) is the seed vector \(x\) itself. Finding conditions which make a seed vector legitimate is pretty much an academic exercise, done out of curiosity. We emphasize that \textit{whatever the seed vector, except for the two trivial seeds, the add-with-carry and subtract-with-borrow sequences become periodic after a few iterations of the generating function, and the periods are the order of the base \(b\) for the appropriate modulus \(m = b^r \pm b^s \pm 1\).}

The following table summarizes rules for legitimate seed vectors and periods for subtract-with-borrow and add-with-carry generators, with base \(b\), lags \(r\) and \(s\), \(r > s\). If \(r\) and \(s\) are chosen so that \(m\) is prime and \(b\) is a primitive root of \(m\), the (long) period of the sequence will be \(m - 1\). (There are two short periods, each of length 1, for the trivial seed vectors \((0, \ldots, 0, 0)\) and \((b - 1, \ldots, b - 1, 1)\). Legitimate seed vectors have the form \((x_1, x_2, \ldots, x_r, c)\) with rules for their formation given for each method. When we write a succession of symbols such as \(x_8x_7\ldots x_1\) we mean the integer for which that is the base-\(b\) representation.

- **Method 1:** \(x_n = x_{n-s} - x_{n-r} - c \mod b; \quad m = b^r - b^s + 1.\)
  
  \[c = 0 \text{ and } x_{r+1} \ldots x_{r+s} > x_{r-s} \ldots x_1\]
  
  \[c = 1 \text{ and } x_{r+1} \ldots x_{r+s} < x_{r-s} \ldots x_1\]

- **Method 2:** \(x_n = x_{n-r} - x_{n-s} - c \mod b; \quad m = b^r - b^s - 1.\)
  
  \[c = 0 \text{ and } x_{r+1} \ldots x_{r+s} + x_{r-s} \ldots x_1 < b^r - b^s - 1\]
  
  \[c = 1 \text{ and } x_{r+1} \ldots x_{r+s} + x_{r-s} \ldots x_1 > b^r - b^s - 1\]
• Method 3: \( x_n = x_{n-r} + x_{n-s} + c \mod b; \quad m = b^r + b^s - 1. \)

\[ \begin{align*}
  c = 0 \text{ and } x_r \ldots x_{s+1} & \leq x_{r-s} \ldots x_1 \\
  c = 1 \text{ and } x_r \ldots x_{s+1} & \geq x_{r-s} \ldots x_1
\end{align*} \]

• Method 4: Form \( x_n \) and carry as in method 3, but replace \( x_n \) by its complement, \( b - 1 - x_n \); \quad \[ m = b^r + b^s + 1. \]

\[ \begin{align*}
  c = 0 \text{ and } x_r \ldots x_{s+1} + x_{r-s} \ldots x_1 & \leq b^{r-s} - 1 \\
  c = 1 \text{ and } x_r \ldots x_{s+1} + x_{r-s} \ldots x_1 & \geq b^{r-s} - 1
\end{align*} \]

5 Structure of the Sequences

In this section we clarify what we meant by “provable uniformity for full sequences”, mentioned in the abstract, and give details on exactly which initial vectors lead to strictly periodic sequences, and which \( r \)-tuples can or cannot appear in full-period sequences.

To fix ideas, consider the prime \( 10^5 + 10^2 + 1 = 99901 \), for which 10 is a primitive root. The subtract with borrow sequence \( x_n = x_{n-2} - x_{n-5} - c \mod 10 \) is strictly periodic with period 99900 if, and only if, the seed vector is one of these two types:

\[ (x_1, x_2, x_3, x_4, x_5, 0) \] with integers \( x_5x_4x_3 > x_3x_2x_1 \)

\[ (x_1, x_2, x_3, x_4, x_5, 1) \] with integers \( x_5x_4x_3 < x_3x_2x_1 \)

There are \( 10^6 = 100,000 \) possible 5-tuples \( x_1x_2x_3x_4x_5 \), but only 99,900 of them appear in the full-period sequence. Which ones are missing? Evidently the 5-tuples which cannot be used to form a legitimate seed vector, that is, 5-tuples with \( x_5x_4x_3 = x_3x_2x_1 \). These have the form \( xyzyx \), 100 in number. Other than those 100 exceptions, every 5-tuple appears exactly once in the full period of 99,900 5-tuples. Thus the sequence has nearly full-period uniformity, in that virtually all of the possible 5-tuples appear exactly once, and only a few—those of the form \( xyzyx \)—do not appear.

More generally, for the subtract-with-borrow generators \( x_n = x_{n-r} - x_{n-s} - c \mod b \) with maximal period \( b^r - b^s \): every possible \( r \)-tuple \( x_1x_2 \cdots x_r \) appears exactly once in the full-period sequence, except for the \( b^s \) \( r \)-tuples whose reversed digits form integers such that \( x_r \cdots x_{s+1} = x_{r-s} \cdots x_1 \).

Similar conclusions apply for the subtract-with-borrow \( x_n = x_{n-r} - x_{n-s} - c \mod b \) or add-with-carry \( x_n = x_{n-r} + x_{n-s} + c \mod b \) generators of maximal periods \( b^r - b^s - 2 \) and \( b^r + b^s - 2 \): Virtually all of the possible \( r \)-tuples appear exactly once in the full-period sequences. Only a small proportion, some \( b^s/b^r \) of them, fail to appear (in the \( x_{n-r} - x_{n-s} - c \) generator) or appear twice (in the \( x_{n-r} + x_{n-s} + c \) generator).

We conclude this section with examples of rules for legitimate starting vectors \((x_1, x_2, x_3, x_4, x_5, c)\) for the particular case \( b = 10 \) and \( r = 5, s = 2 \). Similar
rules apply for the more general case; they were given above. In order that the sequence be strictly periodic, the seed vector must have one of two possible forms:

- For the generator $x_n = x_{n-r} - x_{n-s} - c \mod 10$,
  
  \begin{align*}
  & (x_1, x_2, x_3, x_4, x_5, 0) \text{ with integers } x_5 x_4 x_3 > x_3 x_2 x_1 \\
  & (x_1, x_2, x_3, x_4, x_5, 1) \text{ with integers } x_5 x_4 x_3 < x_3 x_2 x_1 \\
  \end{align*}

- For the generator $x_n = x_{n-r} - x_{n-s} - c \mod 10$,
  
  \begin{align*}
  & (x_1, x_2, x_3, x_4, x_5, 0) \text{ with integers } x_5 x_4 x_3 + x_3 x_2 x_1 < 999 \\
  & (x_1, x_2, x_3, x_4, x_5, 1) \text{ with integers } x_5 x_4 x_3 + x_3 x_2 x_1 > 999 \\
  \end{align*}

- for the generator $x_n = x_{n-r} + x_{n-s} + c \mod 10$,
  
  \begin{align*}
  & (x_1, x_2, x_3, x_4, x_5, 0) \text{ with integers } x_5 x_4 x_3 \leq x_3 x_2 x_1 \\
  & (x_1, x_2, x_3, x_4, x_5, 1) \text{ with integers } x_5 x_4 x_3 \geq x_3 x_2 x_1 \\
  \end{align*}

- for the generator $x_n = 9 - x_{n-r} - x_{n-s} - c \mod 10$,
  
  \begin{align*}
  & (x_1, x_2, x_3, x_4, x_5, 0) \text{ with integers } x_5 x_4 x_3 + x_3 x_2 x_1 \leq 999 \\
  & (x_1, x_2, x_3, x_4, x_5, 1) \text{ with integers } x_5 x_4 x_3 + x_3 x_2 x_1 \geq 999 \\
  \end{align*}

6 Choosing the Lags $r$ and $s$

In searching for generators that have a long period (for all but trivial starting vectors), we want one of $b^r + b^s + 1$, $b^r + b^s - 1$, $b^r - b^s + 1$ or $b^r - b^s - 1$ to be prime and $b$ a primitive root for that prime. Good choices for $b$ are $2^{24}$, which permits direct generation of floating point numbers, and $b = 2^{32}$, which permits generation of 32-bit integers, one of the most common computer sizes. We want the larger lag, $r$, to be of reasonable size—say 15 or more. For such $r$ we must seek primes on the order of $2^{800}$ that have $b$ as a primitive root. If we want to generate 32-bit integers, we need $b = 2^{32}$, so that numbers on the order of $2^{480}$ must be examined for primality. This is feasible—Monte Carlo methods can virtually assure primality for primes of several thousand bits. But verifying that $b$ is a primitive root, or finding its order, for such a large prime $m$ is much more difficult—it amounts to finding the prime factors of $m - 1$. For that reason, when the base $b$ is large, such as $2^{24}$ or $2^{32}$, we concentrate on primes of the form $m = b^r - b^s + 1$ or $m = b^r + b^s + 1$, for then we have a reasonable chance of factoring $m - 1$.

The subtract-with-borrow generators are particularly attractive because they may be used to generate floating-point numbers directly, without the usual method of generating an integer then dividing by the modulus. This feature was exploited in the "universal" generator described in [7]. The choice $b = 2^{24}$
allows direct generation of computer reals with 24 bit fractions—the most frequent size for single precision. Larger values, such as \( b = 2^{48} \) or \( b = 2^{64} \) for higher precision reals, make very difficult the task of finding primes of the form \( b^r - b^s + 1 \) having \( b \) as a primitive root. We are searching for suitable \( r \) and \( s \) for large \( b \)’s, but have found a number of good pairs \( r, s \) for \( b = 2, 24, 31 \) and 32, which we report in what follows.

For lags \( r \) and \( s \) and subtract-with-borrow generators \( x_n = x_{n-s} - x_{n-r} - c \mod b \) with \( b = 2^{24} \), we want \( m \) to be prime and \( b \) a primitive root. Here are some choices that make \( m \) prime: \( m = b^{24} - b^{10} + 1 \), \( m = b^{25} - b^{11} + 1 \), \( m = b^{39} - b^{25} + 1 \). The bad news is that \( b \) is a primitive root of none of these. But there is good news: we are able to factor \( m - 1 \) and show that the order of \( b \) is not significantly smaller than \( m - 1 \). Here is how to do it: For each of these primes \( m \), the prime factors of \( m - 1 \) are

\[
2, 3, 5, 7, 13, 17, 29, 43, 97, 113, 127, 241, 257, 337, 673, 1429, 13361, 5153, 5419, 14449, 15790321, 25629623713, 8895982481, 54410972897, 153853959564161.
\]

(Since, in each case, \( m - 1 = b^k(b^7 - 1)(b^7 + 1) \), we “only” need the prime factors of \( 2^{108} - 1 \) and \( 2^{108} + 1 \), listed, with 2, above.) With these prime factors of \( m - 1 \) we may find the order of \( b = 2^{24} \) for modulus \( m \). In none of these cases does \( b \) have order \( m - 1 \), but the order is large enough to provide sequences with extremely long periods. These three generators are among those recommended in the table in the summary, section 8. They are:

- \( r = 24, s = 10, b = 2^{24}, m = b^{24} - b^{10} + 1 \). The order of \( b \) for modulus \( m \) is \( (m - 1)/48 \). For any \( k \) in \( 1 \leq k < m \) the base-\( b \) expansion of \( k/m \) has period \( (m - 1)/48 \), with “digits” in the period a cyclic permutation of one of 48 possible sets of \( (m - 1)/48 \) digits—in this case, about \( 2^{573}/3 \).

- \( r = 25, s = 11, b = 2^{24}, m = b^{25} - b^{11} + 1 \). The order of \( b \) for modulus \( m \) is \( (m - 1)/336 \). For any \( k \) in \( 1 \leq k < m \) the base-\( b \) expansion of \( k/m \) has period \( (m - 1)/336 \), with “digits” in the period a cyclic permutation of one of 336 possible sets of \( (m - 1)/336 \) digits—in this case, about \( 2^{596}/21 \).

- \( r = 39, s = 25, b = 2^{24}, m = b^{39} - b^{25} + 1 \). The order of \( b \) for modulus \( m \) is \( (m - 1)/672 \). For any \( k \) in \( 1 \leq k < m \) the base-\( b \) expansion of \( k/m \) has period \( (m - 1)/672 \), with “digits” in the period a cyclic permutation of one of 672 possible sets of \( (m - 1)/672 \) digits—in this case, about \( 2^{931}/21 \).

In the above cases, each “digit” is a 24-bit integer, or the 24-bit fraction of a floating point number, and only 24, or 25 or 39 seed values are required to initiate sequences of such lengths. A periodic sequence cannot have length greater than the number of possible seeds, and these generators provide virtually that maximum possible period.

The base \( b = 2 \) is of interest for generating a sequence of random bits. (It was a request for such a generator, for the Connection Machine—a computer
with 65536 independent 1-bit processors—that led to our development of the add-with-carry and subtract-with-borrow generators described here.) For \( b = 2 \) we want primes of the form \( m = 2^r \pm 2^s \pm 1 \) for which 2 is a primitive root. Finding prime \( m \)’s with \( r \) several hundred or more is feasible, using Monte Carlo primality tests, but factoring \( m - 1 \) may not be possible for \( m \)'s so large. So we take advantage of a suitable set of large primes of the form \( 2^p - 1 \)—that is, Mersenne primes. With \( 2^p - 1 \) a prime, we may search for primes of the form \( m = 2^r - 2^s + 1 \) with \( r = p + s \). Then \( m - 1 = 2^r (2^p - 1) \) and to verify that 2 is a primitive root of \( m \) we need only check that \( 2^k \neq 1 \mod m \) for \( k = 2^r \) and for \( k = (m - 1)/2 \).

We now list results of a search for good \( r \) and \( s \) values when \( b = 2 \). They lead to subtract-with-borrow bit generators \( x_n = x_{n-s} - x_{n-r} - c \mod 2 \) with periods \( 2^r - 2^s \).

Table 1: Pairs \( r \) and \( s \) for which \( m = 2^r - 2^s + 1 \) is prime and 2 is a primitive root of \( m \)

<table>
<thead>
<tr>
<th>( r )</th>
<th>( s )</th>
<th>( m )</th>
<th>( k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>23,20</td>
<td>57,38</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>25,12</td>
<td>59,28</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>25,18</td>
<td>61,54</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>29,26</td>
<td>65,60</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>31,24</td>
<td>68,66</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>32,30</td>
<td>69,8</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>33,20</td>
<td>73,68</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>37,18</td>
<td>79,72</td>
</tr>
<tr>
<td>10</td>
<td>8</td>
<td>38,36</td>
<td>95,92</td>
</tr>
<tr>
<td>13</td>
<td>8</td>
<td>41,10</td>
<td>101,70</td>
</tr>
<tr>
<td>14</td>
<td>12</td>
<td>43,41</td>
<td>103,84</td>
</tr>
<tr>
<td>17</td>
<td>14</td>
<td>49,30</td>
<td>105,74</td>
</tr>
<tr>
<td>19</td>
<td>12</td>
<td>53,34</td>
<td>109,78</td>
</tr>
<tr>
<td>20</td>
<td>18</td>
<td>53,50</td>
<td>117,56</td>
</tr>
<tr>
<td>23</td>
<td>4</td>
<td>55,52</td>
<td>121,114</td>
</tr>
</tbody>
</table>

7 A Practical Example: Throwing Dice

The new generators described here are easily implemented and have extremely long periods—virtually the maximum possible: the number of choices for seeds. But are the elements of these extremely long sequences suitably random? This can only be answered by subjecting them to extensive tests of uniformity and independence.
We have undertaken extensive testing, using DIEHARD, our battery of stringent tests partially described in [6]. Results indicate that the add-with-carry and subtract-with-borrow generators are better than, or at least as good as, any of the standard generators. We invite others to apply their own tests. To give an idea of some of the stringent tests that we apply, we give details for a particular generator—one that is used to generate the throws of a die.

Here, we assign the base \( b = 6 \) and use an add-with-carry generator, \( x_n = x_{n-21} + x_{n-2} + c \mod 6 \). Because \( 6 \) is a primitive root of the prime \( m = 6^{21} + 6^2 - 1 \), this generator will have period \( m - 1 = 6^{21} + 34 \) for any set of 21 seed values \( x_1, \ldots, x_{21} \) except all 0’s or all 5’s. We choose 21 seed values, say

\[
4, 3, 2, 2, 0, 1, 3, 5, 2, 4, 5, 1, 0, 0, 3, 4, 3, 0, 4, 2, 5 \text{ and initial carry bit } 0.
\]

Then successive generated values, \( x_{22}, x_{23}, \ldots \) (with superscripted carry bits) are: \( 0^0, 3^1, 3^0, 5^5, 3^0, \ldots \). If we add 1 to each of these residues of 6 we have a sequence of throws of a die: 1, 4, 4, 6, 4, 1, 2, 1, 5, 5, 1, 1, 4, 1, 2, 5, 4, 1, 2, 5, 4, 1, 2, 5, 4, 1, 2, 5, 4, 1, 2, 5, 4, 1, 2, 5, 4, 1, 2, 5. Is this sequence suitable for simulating the throws of a true die? To answer that question we report the results of a number of tests.

7.1 The craps test

First, the craps test. Can we use the sequence to play a satisfactory series of games of craps? In spite of its mundane connotations, this is quite a good test of a random number generator, for it tends to test both uniformity and independence of successive throws of the dice. The probability of winning for the thrower in craps is 244/495. Does our thrower win with frequency consistent with this value? Correlations between values many throws apart can affect the duration of the game; are the durations of our games consistent with the distribution that probability theory calls for? We also test the frequency of successive “passes”. In real-life craps, the dice thrower continues to hold the dice as long as he wins; successive wins are called “passes” and popular lore has reports of ten, fifteen or more successive passes. Finally, we also test the simplest and most obvious test of the throws: are the frequencies of 1’s, 2’s, \ldots, 6’s consistent with their expected values, each one-sixth of the total number of throws?

We “played” one million games of craps using the above seed values. The results, summarized in table 2, are indistinguishable from those that a truly random die would produce. All results are consistent with probabilities for the various aspects of the game: number of wins, duration of the game, frequency of consecutive “passes” and the values on the die. The die was “thrown” 6,747,174 times. There are plenty more “throws” available, however— the period is \( 6^{21} + 6^2 - 2 = 21,936,950,640,377,890 \) for the \( x_{n-21} + x_{n-2} + c \) generator, so it could easily run Las Vegas for 10,000 years with no one (but us) the wiser.
Table 2: Results of the Craps Test for One Million Games

<table>
<thead>
<tr>
<th>Duration of Game</th>
<th>Consecutive “passes”</th>
</tr>
</thead>
<tbody>
<tr>
<td>throws</td>
<td>exp</td>
</tr>
<tr>
<td>1</td>
<td>333333.3</td>
</tr>
<tr>
<td>2</td>
<td>188217.6</td>
</tr>
<tr>
<td>3</td>
<td>134773.7</td>
</tr>
<tr>
<td>4</td>
<td>96567.3</td>
</tr>
<tr>
<td>5</td>
<td>69257.1</td>
</tr>
<tr>
<td>6</td>
<td>49717.7</td>
</tr>
<tr>
<td>7</td>
<td>35725.1</td>
</tr>
<tr>
<td>8</td>
<td>25695.4</td>
</tr>
<tr>
<td>9</td>
<td>18499.3</td>
</tr>
<tr>
<td>10</td>
<td>13331.5</td>
</tr>
<tr>
<td>11</td>
<td>9616.6</td>
</tr>
<tr>
<td>12</td>
<td>6943.7</td>
</tr>
<tr>
<td>13</td>
<td>5018.6</td>
</tr>
<tr>
<td>14</td>
<td>3630.7</td>
</tr>
<tr>
<td>15</td>
<td>2629.2</td>
</tr>
<tr>
<td>16</td>
<td>1905.8</td>
</tr>
<tr>
<td>17</td>
<td>1382.7</td>
</tr>
<tr>
<td>18</td>
<td>1004.1</td>
</tr>
<tr>
<td>19</td>
<td>729.9</td>
</tr>
<tr>
<td>20</td>
<td>531.1</td>
</tr>
<tr>
<td>≥ 21</td>
<td>1435.6</td>
</tr>
</tbody>
</table>

Distribution of Values on the Die

<table>
<thead>
<tr>
<th>value</th>
<th>exp</th>
<th>obs</th>
<th>Number of Wins</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1125807</td>
<td>1125894</td>
<td>$p = 244/495 = 0.4929293$</td>
</tr>
<tr>
<td>2</td>
<td>1125807</td>
<td>1125613</td>
<td>exp</td>
</tr>
<tr>
<td>3</td>
<td>1125807</td>
<td>1126630</td>
<td>obs</td>
</tr>
<tr>
<td>4</td>
<td>1125807</td>
<td>1125269</td>
<td>492929.3</td>
</tr>
<tr>
<td>5</td>
<td>1125807</td>
<td>1124166</td>
<td>difference  = 244.7</td>
</tr>
<tr>
<td>6</td>
<td>1125807</td>
<td>1127270</td>
<td>sigma</td>
</tr>
</tbody>
</table>
7.2 The birthday-spacings test

Another stringent test of a generator is the Birthday-Spacings Test. This test is described in [6] and [7]. We produce a sequence of \( m \) integers from 1 to \( n \) and view them as \( m \) "birthdays" in a "year" of \( n \) days. We sort the birthdays and count the number of duplicate values in the list of spacings between birthdays. If \( n \) is large, the number of duplicate spacings should be approximately Poisson distributed with mean \( \lambda = m^2/(4n) \). Alternatively, the empirical distribution may be compared with that from other kinds of generators; discrepancies will show that at least one of those being compared fails the birthday-spacings test. Shift register and lagged-Fibonacci generators using +, − or exclusive-or fail this test. Congruential and lagged-Fibonacci generators using multiplication generally pass it.

The \( x_n = x_{n-21} + x_{n-2} + c \mod 6 \) generator passed the birthday-spacings test for a variety of choices of \( n \), the length of the year, and \( m \), the number of birthdays. We illustrate with details of two of the tests. In the first one, the add-with-carry mod 6 generator was used to produce ten "digits" of a base-6 integer from 0 to 60466175. The leftmost 24 bits of that integer produced a birthday in a year of \( 2^{24} \) days, then spacings among \( m = 512 \) birthdays were examined for duplicate values. This was repeated 1000 times, producing what should be a sample of 1000 from a Poisson distribution with \( \lambda = 2 \). The second test was similar, except that the nine digits of the base-6 generator were used to produce \( m = 512 \) birthdays from a year of \( 6^9 = 1077696 \) days, and the resulting 1000 duplicate-spacing counts should be a Poisson sample with \( \lambda = 3.3296 \).

<table>
<thead>
<tr>
<th>( m ) ( = ) 512</th>
<th>( n = 2^{24} ) ( \lambda = 2.00 )</th>
<th>( m = 512 ) ( n = 10077696 ) ( \lambda = 3.33 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean = 2.0640</td>
<td>Var = 1.9359</td>
<td>Mean = 3.2840</td>
</tr>
<tr>
<td>duplicate</td>
<td>number</td>
<td>number</td>
</tr>
<tr>
<td>spacings</td>
<td>observed</td>
<td>expected</td>
</tr>
<tr>
<td>0</td>
<td>117</td>
<td>135.335</td>
</tr>
<tr>
<td>1</td>
<td>274</td>
<td>270.671</td>
</tr>
<tr>
<td>2</td>
<td>273</td>
<td>270.671</td>
</tr>
<tr>
<td>3</td>
<td>191</td>
<td>180.447</td>
</tr>
<tr>
<td>4</td>
<td>96</td>
<td>90.224</td>
</tr>
<tr>
<td>5</td>
<td>35</td>
<td>36.089</td>
</tr>
<tr>
<td>( \geq 6 )</td>
<td>14</td>
<td>16.564</td>
</tr>
<tr>
<td>Chisquare with 6 d.o.f. = 3.96</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Probability = 0.31815</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

16
7.3 Monkey tests

The new kinds of generators have also passed a number of what we call Monkey Tests. These we called Overlapping m-tuple Tests in [6], but the more catchy term “monkey tests” seems to please students in our annual course on computational statistics, and it is readily understood from the popular image of a monkey randomly striking the keys of a typewriter until a particular string is produced. Numerous tests of this type were applied: suppose the monkey (our add-with-carry mod 6 generator) produces a sequence of base-5 digits such as 352201356531426451..., and consider, say, the sequence of four-letter “words” that result: 3522, 5220, 2201, 2013, .... Let \( w_{ijkl} \) be the number of times that the word \( ijk \) appears in a long sequence of \( n \) (overlapping) four-letter words. For a good generator, we would expect about \( n/6^4 \) appearances of each possible word, but the naive chi-square statistic \( \sum (w_{ijkl} - n/6^4)^2/6^4 \) is not a suitable test for randomness here, because of the dependence (covariance structure) among the individual cell counts.

But it turns out that the quadratic form in the weak inverse of the covariance matrix of the mean-adjusted cell counts, which is equivalent to the likelihood-ratio test that the cell counts are normally distributed with those specified means and covariances, has a simple representation. One merely computes \( Q_4 \), the naive chi-square statistic for 4-tuples, as well as \( Q_3 \), the naive \( \sum (\text{obs-exp})^2/\text{exp} \) for 3-tuples. Then \( Q_4 - Q_3 \) is the quadratic form in the weak inverse of the covariance matrix, and, under the null hypothesis, it should have a chi-square distribution with \( 6^4 - 6^3 \) degrees of freedom.

The \( x_n = x_{n-21} + x_{n-2} + c \mod 6 \) generator passed all monkey tests of this type—for 3, 4, 5, 6 and 7-letter words, as well as the OPSO test, a version of the monkey test when the number of possible words exceeds the number of memory locations that can be assigned to cell counts—see [6].

7.4 Other tests

This generator also passes the runs test, described in Knuth [1], as well as the other tests described there and derived from the earlier list of Maclaren and Marsaglia [2]. We have not found a test that the generator fails.

Of course, a base-6 generator is not well suited for computer implementation, but stringent tests for the new generators with bases \( 2 \) and \( 2' \) for \( j = 8, 16, 24, 31, 32 \) show similar promise. The simplicity of these add-with-carry or subtract-with-borrow generators, their astonishingly long periods and their good behavior on stringent tests of randomness, make them very attractive alternatives to the standard methods.
8 Summary and Recommendations

We have described new kinds of generators: add-with-carry and subtract-with-borrow. They produce exceptionally long sequences of "digits" \( x_1, x_2, \ldots \) of a base \( b \). Informally, the generators are of four types:

1. \[ x_n = x_{n-s} - x_{n-r} - c \mod b, \quad m = b^r - b^s + 1, \]
2. \[ x_n = x_{n-s} - x_{n-r} - c \mod b, \quad m = b^r - b^s - 1, \]
3. \[ x_n = x_{n-r} + x_{n-s} + c \mod b, \quad m = b^r + b^s - 1, \]
4. \[ x_n = b - 1 - x_{n-r} - x_{n-s} - c \mod b, \quad m = b^r + b^s + 1. \]

If \( r > s \) are chosen so that the designated value \( m \) is prime, then the sequence of \( x \)'s are the digits, in reverse order, of the base-\( b \) expansion of \( k/m \) for some \( 1 \leq k < m \). Formally, the sequence is \( x, f(x), f^2(x), \ldots \), where \( x \) is an initial vector of \( r \) seed digits and an initial "carry" bit \( c \). Rules for \( f \)'s that transform \( x \) into a new sequence of \( r \) digits and an associated carry bit are in sections 2.3 and 4.4, with examples. We conclude with a table giving good choices of \( r \) and \( s \) for various bases \( b \).

\begin{table}[!h]
\centering
\caption{Some recommended subtract-with-borrow generators \( x_n = x_{n-s} - x_{n-r} - c \mod b \)}
\label{tab:subtract_with_borrow}
\begin{tabular}{|ccccc|}
\hline
Number of seeds & base & lags & number of cycles & period of each cycle \\
and type & \( b \) & \( r \) & \( s \) & \\
\hline
847 bits & 2 & 847 & 240 & 1 & \( 2^{847} - 2^{840} \approx 10^{265} \) \\
1751 bits & 2 & 1751 & 472 & 1 & \( 2^{1751} - 2^{472} \approx 10^{527} \) \\
37 32-bit integers & \( 2^{32} \) & 37 & 24 & 64 & \( 2^{1178} - 2^{702} \approx 10^{354} \) \\
24 32-bit integers & \( 2^{32} \) & 24 & 19 & 1536 & \( \frac{1}{3} (2^{1070} - 2^{509}) \approx 10^{324} \) \\
21 32-bit integers & \( 2^{32} \) & 21 & 6 & 192 & \( \frac{1}{3} (2^{606} - 2^{186}) \approx 10^{200} \) \\
48 31-bit integers & \( 2^{31} \) & 48 & 8 & 210 & \( \frac{1}{108} (2^{1487} - 2^{247}) \approx 10^{445} \) \\
39 reals & \( 2^{24} \) & 39 & 25 & 672 & \( \frac{1}{21} (2^{931} - 2^{505}) \approx 10^{279} \) \\
28 reals & \( 2^{24} \) & 28 & 8 & 144 & \( \frac{1}{6} (2^{268} - 2^{188}) \approx 10^{200} \) \\
25 reals & \( 2^{24} \) & 25 & 11 & 336 & \( \frac{1}{21} (2^{590} - 2^{260}) \approx 10^{178} \) \\
24 reals & \( 2^{24} \) & 24 & 10 & 48 & \( \frac{1}{21} (2^{572} - 2^{235}) \approx 10^{171} \) \\
\hline
\end{tabular}

\footnote{Not counting the two trivial cycles of period 1.}
\footnote{Reals with 24-bit fractions. The floating-point version of the mod \( 2^{24} \) operation takes three inputs \( x, y \) and \( c \) then produces a new value \( z \) and new carry \( c' \):
\[ (z, c') = \begin{cases} (x - y - c, 0) & \text{if } x - y - c \geq 0, \\ (x - y - c + 1, 2^{24} - 1) & \text{else}. \end{cases} \]}
\end{table}
Only subtract-with-borrow generators are listed in the table. Add-with-carry generators of equally long periods surely exist, but the periods are so long that only for primes of the form $m = b^r - b^s + 1$ are we able to factor $m - 1$, a necessary step in establishing the period.

Various choices of $b$, $r$ and $s$ provide for long sequences of bits, bytes, 16-, 31- or 32-bit integers or floating point numbers with 24-, 48- or 64-bit fractions. The extremely long periods of the generators, their small memory requirements and the simple computer operations required, taken with excellent performance on stringent tests of randomness, make subtract-with-borrow and add-with-carry generators worth considering for general Monte Carlo use.

References


