GENERATING AND TESTING PSEUDORANDOM NUMBERS

BY CHARLES A. WHITNEY
Analyze haphazard occurrences with linear congruential generators

F A DRUNKARD starts from a lamppost and randomly staggers away. How far will he have progressed in one thousand steps? Expressed in varying forms, the "drunkard's walk" has become a staple of mathematical physics. How far will an impurity atom migrate in a crystal lattice? How many steps will be required for a photon to emerge from a foggy atmosphere? They are all the same question, and they can be treated with "Monte Carlo" calculations. These calculations are finding increasing applications in business, as well. For example, they provide an analysis of how best to serve customers arriving haphazardly at a restaurant.

You can find numerical examples of random series all around you. The final integers in a list of telephone numbers gives a good random series in the range 0 to 9. The face values of cards drawn from a well-shuffled deck and the final plate numbers on passing cars are usually quite reliable. (But the first plate numbers are not."

The property that defines a randomly generated series is that each number is independent of all earlier numbers. In other words, the process that generates the random series has no memory. Therefore, even if you know all the previous numbers, you cannot predict with certainty the next one. (And if you have been losing at a truly random game, you have no reason to think you will start winning.) That's why flipping a coin is such a good method of generating a random series of zeros and ones. Each flip is entirely independent of the others.

Before going further, I should try to clarify a point of language that can lead to confusion. Strictly speaking, no such thing as a random number exists-only a random process. The number 12345 is neither less nor more random than the number 32719. In a series of 100,000 randomly generated numbers, both of these would have the same chance of occurring. The idea that 12345 is less random comes from comparing it with a simple pattern of ascending digits. Thus, instead of saying you are trying to produce random numbers, you should say you are trying to construct a method that will produce a series of numbers that imitates the results of a random process. But for the purpose of this article, I will use the word "random" more loosely to refer to a random sequence or a random number, without worrying about the niceties. If a sequence looks like it was generated by a random process, I will call it random and will put off the question of how to judge appearances until I discuss methods of testing random-number generators.

To carry out a Monte Carlo calculation, you need to work with random numbers inside a computer. But you are faced with the fact that a purely digital computer is a deterministic machine-except on its "off" days-and such a device cannot truly generate a random process. You have to be satisfied with deterministic algorithms that imitate random processes. You can thus generate "pseudorandom" numbers with some of the earmarks of randomness. Naturally, some random-number generators work better than others, and you must be wary.

LINEAR CONGRUENTIAL GENERATORS

Mathematicians have suggested many methods for generating pseudorandom numbers with a digital computer. Happily, the most common and powerful one involves simple arithmetic. This method is the linear congruential generator, or LCG for short. An LCG produces a series of numbers, \( l_i \), where the subscript \( i \) indicates the location of the number. \( l_1 \) indicates the first number, \( l_3 \) is the third, and so on. Since each successive term, \( l_{i+1} \), is computed from its predecessor, you can see right away that this series is not truly random because it has memory. That is why the LCG is called a "pseudorandom"-number generator.

To understand the LCG, look at the following linear expression:

\[
l_{i+1} = a l_i + c \quad (a > 0, c \geq 0)
\]

This expression multiplies each number by the factor \( a \) and then adds \( c \) to find the next member. It produces an ascending series of numbers whose differences are given by the expression

\[
l_{i+1} - l_i = (a-1) l_i + c
\]

We call the process "mapping" because it carries the integer \( l_i \) to \( l_{i+1} \), from one point to the next in a one-dimensional space, as shown in figure 1.

Suppose \( a = 2 \) and \( c = 1 \); then you have 1, 3, 7, 15. As it stands, this series doesn't look random. To create a series that looks random, you need a method of scrambling the output, perhaps by cutting off the upward run. One way to achieve this appearance is to imagine you've subdivided the number line representing the \( l \)-space into segments of length \( m \). If you make the multiplier, \( a \), in the LCG suff

Charles A. Whitney is a physicist at the Harvard-Smithsonian Center for Astrophysics (60 Garden St., Cambridge, MA 02138) and is a professor of astronomy at Harvard.
Random Numbers

(continued from page 129)

To start, you can look at the following series:

\[ I_0 = 0, I_1 = 1, I_2 = 2, I_3 = 3, I_4 = 4, I_5 = 5, I_6 = 6, I_7 = 7, I_8 = 0, \]

and taking the difference between terms is \( I_{n+1} = I_n \mod m \). The result is a series in the range \( 0 \leq x < m \).

You can accomplish such a subdivision of the \( I \)-space by introducing the modulus function

\[ \text{mod}(I, m) = I - \lfloor I/m \rfloor m \]

where \( \lfloor I/m \rfloor \) is the integer part of the quotient, derived by truncation.

When, for example, \( \text{mod}(11, 3) = 2 \), the result always lies between 0 and \( m-1 \).

In some forms of BASIC, this is written \( 11 \mod 3 \).

Now rewrite the mapping as

\[ I_{n+1} = \text{mod} (aI_n + c, m) \]

This equation is a general form for the LCG, and it produces a series of integers in the range \( 0 < I < m-1 \).

Three parameters describe this mapping: the multiplier (a), the difference (c), and the modulus (m).

Often it is convenient to "normalize" the values, dividing each by the modulus. The result is a series in the range \( 0 < x < (m-1)/m \).

To see some of the properties of series that are generated this way, look at the following series:

\[ I_{n+1} = \text{mod} (5I_n + 3, 8) \]

By starting with an arbitrary value, the "seed," and taking \( I_1 = 1 \), you find

\[ 1, 0, 5, 4, 7, 6, 1, 0, 5, 4, \ldots \]

This series starts off with a haphazard appearance, but since it repeats itself every 8 terms, it is said to have a "period" of 8. It is not hard to see why this is the period. First, the modulus of the series is 8, so the series cannot have more than 8 integers. (You remove larger integers by subtracting the modulus.) Second, the series is deterministic. Each appearance of a particular integer must be followed by a uniquely determined integer. That is, each appearance of "1" must be followed by "5." As a result, the series must repeat itself with a period no longer than its modulus.

This reasoning suggests adopting a large modulus if you want a long period. But it isn’t the only possibility. Some generators will skip many of the possible numbers and give an in-
complete set of “random” numbers. A series that generates all of the \( m \) distinct integers \( (0 < n < m-1) \) during each period is called a “full period.” Whether a series will have a full period or not depends in large measure on the values you choose for the parameters \((a, c, m)\). Table I illustrates some of the series with \( a = 5 \) and \( m = 8 \) for various values of the additive constant, \( c \). Take the first series as an example. If you use a seed, 6, you obtain the pseudorandom series 6, 7, 4, 5, 2, … . If you use the seed 5, you get 5, 2, 3, 0, 1, 6, ….

Because several of the series in Table I do not have a full period, they generate subsets of integers with many useful properties. In the first place, the sum of the periods of the subsets equals the modulus of the series. This property helps you decide whether you have found all the subsets. Second, if the series does not have a full period, different seeds start each subset. For example, the series with \( c = 4 \) has six subseries of periods 1 or 2, and the seed you use will determine which subseries you generate.

When you set up a random-number generator, look for a long and full period because it will produce the richest set of numbers. Several established rules for selecting the parameters will achieve a full period. These rules are discussed in Donald Knuth’s The Art of Computer Programming, referenced at the end of this article. One rule is that the modulus, \( m \), and the constant, \( c \), must have no factors in common. Another rule is that \( a \) must be greater than the square root of \( m \) (\( a > \sqrt{m} \)) to avoid the serial correlation that upward runs produce. (This rule ensures that the mapping quickly takes the number out of the current segment, the same condition mentioned earlier.) Finally, you will get the longest possible periods if the modulus is a prime number equal to or less than the largest integer your computer can handle. (For a 16-bit processor, this condition implies that \( m \leq 32,768 \).)

Two additional examples of LCGs with short, full periods follow:

\[
l_{i+1} = \text{mod}(71, + 5.9)
\]

\[
l_{i+1} = \text{mod}(71, + 7.9)
\]

In practice, you develop a pseudorandom-number generator in a cut-and-try process, testing and modifying various possibilities.

In describing the LCG, I assumed that the coefficients were all integers. You can increase the series’ apparent period by taking real (decimal) values. For example, if you substitute \( 5.1 \) and \( 3.111 \) for the coefficients in the series above, the terms won’t repeat precisely after a period of eight terms. But the terms in successive cycles will

**Graphics with Gray Matter.**


The GP-29 delivers the graphics and text versatility of powerful, expensive terminals at significant cost savings. And you can buy the GP-29 as a ready to use terminal or as a retrofit board for the Zenith Z29.

Expand your applications with dual plane graphics. Create images with “shades of gray.” Overlay two separate images. Animate the images. Store multiple images in local memory.

The GP-29 gives you low cost graphics you never imagined possible. 1024 x 500 hi-res and 512 x 250 lo-res selectable resolution. 128k of display memory. And our dedicated graphics processor provides selective vector erase, area erase, area fill, area move, arc drawing, pan and zoom.

You also get Tektronix 4014 compatibility as well as DEC VT100 and VT220 compatibility.

GP-19 board for the Zenith Z19 terminal is also available for $695.
show only a slight shift, and the overall pattern of the cycle will be the same—except for occasional jumps when the fractional parts accumulate sufficiently. Thus, the increase of period is only illusory, and since I don't want to fool you with this illusion, I will restrict myself to integer coefficients.

**Statistical Tests for Randomness**

Often, you can look at a series and see that it was probably not generated by a random process. For example, who would claim that a real coin would lead to a series of heads (H) and tails (T) such as HTHHHTTTTTHHHHTTTTT? It could happen (with a probability of about \((1/2)^{29} \approx 0.000000001\)), but you wouldn't expect it in your lifetime.

But some series are not so obvious, and you need a more reliable test than the eyeball and a hunch. For this, you must compare some statistical properties of the series with some theoretical predictions you make after assuming that the series was generated by a truly random process. When I refer to a statistical property, I'm talking about one that is independent of the seed you used to start the series, including those tests that are oblivious to the order of terms in the series. (The mean value is one such property.) These statistical tests reveal how likely it is that a random process generated a certain series.

No statistical test is a sure bet, and few tests are reliable in themselves. Some pseudorandom series will pass one test with flying colors, only to fail miserably in another. Therefore, you have to apply several different tests. I will apply some tests to the LCG and the generator that the IBM PC's Advanced BASIC supplies. Then I'll discuss how to develop a more powerful random-number generator that anyone can use.

Let's start with the simplest test: determining the period of the series.

**Period**

You can determine a period by noting a first number and then stepping through it while computing one number after another until the first number recurs, that is, until \(x_i = x_{i+1}\). Then \(n-1\) is the period of the series. (In order to ensure that \(x_i\) is, indeed, the start of a repeat cycle, the first few values, \(x_1 - x_{1+n}\), can be saved for comparison with \(x_n - x_{n+i}\)).

Figure 3 shows the unfortunate effect of a short period on the random sequence.
special distribution laws (the normal distribution, for example), but I will consider only the ones that are intended to produce uniformly distributed numbers. If I normalize an LCG, my program should produce numbers in the range 0 to 1.0 with equal probability. However, the numbers won't arrive in a perfectly uniform way. They will exhibit a tendency to clump, just as the flips of a real coin will show runs of more than one head or tail instead of HTHHTH.

Figure 3: A random walk generated with a pseudorandom-number generator of the type in table 1, with a period of 127. Each step upward or downward was determined by simulated flip of a coin. This diagram illustrates the repetitive pattern of some random-number generators.

Figure 4: Similar to figures 1 and 2, this walk was generated with the RND function of the Microsoft Advanced BASIC supplied with the IBM PC operating under MS-DOS 2.0. It shows an approximate periodicity of about 18,000 steps, although the rigorous period is about 64,000. Using such a function for Monte Carlo simulations requiring more than 8000 steps could produce misleading results.
RANDOM NUMBERS

walk generated by one of the LCGs in table 1. You can easily spot the periodicity, and you wouldn't want it as the imitation of a very long random walk. (This walk was generated from a normalized LCG by stepping upward if \( x > 0.5 \) and downward if \( x < 0.5 \).)

When I applied this test to the pseudorandom-number generator supplied with the Advanced BASIC Interpreter of the IBM Personal Computer (PC), it showed a period of 65,535. This result was as good as I could have hoped, but a detailed plot of a walk shows that it also has a much shorter wave-like cycle superposed. Figure 4 shows such a plot and reveals a subcycle that is about 18,000 steps long.

**Distribution**
A random sequence ought to contain representative numbers from all parts of the permitted range. Some programs generate numbers that follow

\[
\binom{51}{1} < 0.5 .
\]

This result was as good as one could have hoped, but a detailed plot of a walk shows that it also has a much shorter wave-like cycle superposed. Figure 4 shows such a plot and reveals a subcycle that is about 18,000 steps long.

**Distribution**
A random sequence ought to contain representative numbers from all parts of the permitted range. Some programs generate numbers that follow

\[
\binom{51}{1} < 0.5 .
\]

Expression: \( l_{i+1} = \text{mod}(l_i + c, m) \)

Table 1: Because several of these series do not have a full period, they generate subsets of integers with several useful properties, for example, the sum of the periods of the subsets equals the modulus of the series.

<table>
<thead>
<tr>
<th>Expression</th>
<th>( l_i )</th>
<th>Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>5( l_i + 1 )</td>
<td>1, 6, 7, 4, 5, 2, 3, 0</td>
<td>8</td>
</tr>
<tr>
<td>5( l_i + 2 )</td>
<td>1, 7, 5, 3, 1, 7, 5, 3</td>
<td>4</td>
</tr>
<tr>
<td>5( l_i + 3 )</td>
<td>4, 6, 0, 2, 4, 6, 0, 2</td>
<td>4</td>
</tr>
<tr>
<td>5( l_i + 4 )</td>
<td>1, 0, 3, 2, 5, 4, 7, 6</td>
<td>8</td>
</tr>
<tr>
<td>5( l_i + 5 )</td>
<td>1, 1, 1, 1, 1, 1, 1</td>
<td>1</td>
</tr>
<tr>
<td>5( l_i + 6 )</td>
<td>2, 6, 2, 6, 2, 6, 2</td>
<td>2</td>
</tr>
<tr>
<td>5( l_i + 7 )</td>
<td>3, 3, 3, 3, 3, 3, 3</td>
<td>1</td>
</tr>
<tr>
<td>5( l_i + 8 )</td>
<td>4, 0, 4, 0, 4, 0, 4</td>
<td>2</td>
</tr>
<tr>
<td>5( l_i + 9 )</td>
<td>5, 5, 5, 5, 5, 5, 5</td>
<td>1</td>
</tr>
<tr>
<td>5( l_i + 10 )</td>
<td>7, 7, 7, 7, 7, 7, 7</td>
<td>1</td>
</tr>
<tr>
<td>5( l_i + 11 )</td>
<td>3, 3, 3, 3, 3, 3, 3</td>
<td>1</td>
</tr>
<tr>
<td>5( l_i + 12 )</td>
<td>1, 2, 7, 0, 5, 6, 3, 4</td>
<td>8</td>
</tr>
<tr>
<td>5( l_i + 13 )</td>
<td>1, 3, 5, 7, 1, 3, 5, 7</td>
<td>4</td>
</tr>
<tr>
<td>5( l_i + 14 )</td>
<td>2, 0, 6, 4, 2, 0, 6, 4</td>
<td>4</td>
</tr>
<tr>
<td>5( l_i + 15 )</td>
<td>1, 4, 3, 6, 5, 0, 7, 2</td>
<td>8</td>
</tr>
</tbody>
</table>

Figure 1: Schematic diagram of mapping in a linear congruential generator (LCG). Each successive term in the series is larger than the preceding. This series does not imitate a random series, but it is the first step in that direction. See figure 2.

Figure 2: Schematic of an LCG, showing how the division of the number line into equal intervals, \( m \), can produce pseudorandom numbers. The location of each number inside the corresponding interval is haphazard. It results from using the modulus function and leads to a pseudorandom series.
Table 2 shows the result for $N = 1000$ numbers in 100 bins computed with the BASICA RND. The observed values cluster about the expected mean. $<NB> = N/O = 10$. When you run the test several times, the excesses and deficiencies appear in different bins. As a result, no evidence appears that any particular bins consistently receive more than $1/O$ of the counts.

A quantitative measure of performance is the conventional chi-square test, which evaluates a measure of the spread (see "The Chi-Square Test" on page 446). This test estimates how likely it is that the actual value will be different from the expected value in a randomly generated series. If you look at table 2, you find no bins with less than 5, two bins with $NB = 5$, seven with $NB = 6$, and so on. The chi-square test examines all the bin populations and tells how often you can expect this particular distribution of populations from a randomly generated series, where you expect $NB = 10$ on the average.

Applying the chi-square test to the bin populations of table 2 and then for much longer runs using the BASICA generator, you will find that if the random-number generator is pushed to 30,000 terms, it still performs well. The story changes as soon as you get close to the full period of about 65,000 terms. There, all bins are more or less equally filled and the histogram of bin populations, $NB(i)$, becomes tightly peaked about the mean value, $<NB>$, because all possible values have been achieved. The generator has displayed its entire full period. Near this extreme limit, the generator fails the chi-square test because the chances are small that actual values will be any different from the expected value.

What happens when a random-number generator comes to the end of its period is similar to what happens in a game of blackjack when the cards are not collected into the deck after each hand. When 51 cards have been laid out, there is no doubt what the next card will be. You pro-

You can test the distribution of numbers by setting up $O$ bins and putting each member of the series into one of the bins. For example, if the numbers are restricted to the interval $0 < x < 1$, each can be put into bin $j$, where $j$ is computed from $$j = \text{int}(O \cdot x), \quad 0 < j < O$$

On each occurrence of $j$, the bin count, $NB(j)$, is incremented, so that $NB(j) = NB(j) + 1$.
It features, simple and uniform commands, and utters and numbers your files. It ing files—a critical difference from other utilities that search and destroy, without informing you type in their menu number!

Unlike some utility programs that are a headache computer housekeeping, each with its own commands other commands are customizing software into one 16K integrated package, so you do things to memorize. Your programs won't run. (And POWER! still has sensitive data from prying eyes with ..)

We first developed POWER! two years ago, and a stack of testimonials from FORD to XEROX testify to its excellence. For IBM-PC™ users, special features like managing sub-directories, [CHANGE], and a separate creation of up to 8 simultaneous, on-screen [WINDOWS] have been added.

MONEY-BACK GUARANTEE AND A 10 DAY TRIAL

POWER! hasn't the Seal of Approval from the Professional Software Programmers Association, and you, too, must be happy with POWER!—or your money back! For only $149 you can now really be in control of your computer. Call Computing! at 415-567-1634, or your local dealer. For IBM-PC or any CP/M machine. Please specify disk format.

TO ORDER CALL 800 TOLLFREE

IBM and IBM-PC are registered trademarks of International Business Machines Corporation.

You can probably use a generator out to one-half of its period. For small simulations this limit should be good enough.

AMPLITUDE SPECTRUM FROM FOURIER TRANSFORM

I've already remarked that the random-walk pattern in figure 4, generated from BASICA RND, shows clear signs of waves. The Fourier amplitude spectrum lets you quantitatively measure the waves' size (see "Fourier Spectrum" on page 464). You can derive this spectrum from the fundamental definition of the Fourier coefficients that you'll find in introductory books on applied mathematics. Or you can derive it from the fast Fourier transform subroutines in some software packages. As an example, figure 5 shows the frequency spectrum of the random walk in figure 3, which was generated with an LCG with a period of 127. This spectrum shows the relative amplitudes of waves of various frequencies. You plot the frequencies in terms of the walk's full length, namely 1000 steps, so that the primary period of 127 shows as a peak in the spectrum at about 8 cycles (1000/127) on this spectrum. Because it quickly becomes repetitive, you can't use such an LCG for simulations involving more than a few dozen steps, and the Fourier spectrum puts you on guard. Figure 6 shows the amplitude spectrum for the BASICA RND pseudorandom-number generator that comes with
the IBM PC's MS-DOS 2.0 operating system. In this diagram, the frequency is the number of peaks in every 64,000-step run. For example, significant waves show 4, 12, 21, 28, and 37 peaks per 64,000 steps. The most prominent is the wave with frequency 4, which accounts for the main random-walk plot pattern. The fact that you can divide two of the higher frequencies, 12 and 28, by 4 accounts for the repeated pattern of details on the waves. This test is a clear call for caution in using BASICA RND, and it implies that you need an improved generator.

**SHUFFLING A GENERATOR**

How can you extend the period? As mentioned earlier, an LCG's period has two limitations: only integers less than the modulus, $m$, are generated; and the series is deterministic, meaning a particular number always has a particular sequel. For a computer capable of handling integers smaller than a fixed limit, $l_{max}$, you can do nothing about the first restriction. You can, however, alleviate the second restriction and alter the simple determinism of the series using a technique called "shuffling."

Consider an LCG with a period of 8. Each member of the series is an integer from 0 to 7, and if the selection is purely sequential, the period inevitably will be 8. But suppose you set up a secondary list of five numbers and use a second LCG to select the next member of the series from one of them. Then, after each five selections, you replace the five numbers in the secondary list with a randomly selected set. By using two LCGs, you can shuffle the series and extend the effective period. You do not increase the possible integers but prevent an

(continued)
Circle 381 on inquiry card.

If your computers are not on speaking terms, introduce them to MOVE-IT.

Now you can move files, programs and data between incompatible computers. When adding a new computer, data and programs from one system can be transferred to another with MOVE-IT, even if the two have different disk formats and use different operating systems.

A mini-network can be set up by equipping each computer with MOVE-IT, increasing utilization by sharing information freely between machines.

Error Free. A 16 bit error checking technique assures accurate and error-free data transmission even at high speeds and over "dirty" telephone lines or long cable runs. MOVE-IT is the ideal program for telephone line use with modems as well as direct wire applications.

WOOLF SOFTWARE SYSTEMS, INC.
6754 ETON AVE., CANOGA PARK, CA 91303  •  (818) 703-8112
© COPYRIGHT 1984 WOOLF SOFTWARE SYSTEMS, INC.

SEE US AT COMDEX

Series 36: 35 cps - all-purpose interface
630 API: 40 cps - all-purpose interface

Treat your personal computer to famous Diablo letter-quality printing.

Don't settle for less: your choice of hundreds of printwheel styles, fully-formed characters, interfaces with IBM, Apple, Radio Shack, Commodore IEEE 488, Centronics "Parallel" and RS232C "Serial". In MTI's opinion, the best-letter-quality printers on the market. Whether you lease or buy, you'll find MTI is the one source for all the computer and data communications equipment, applications expertise and service you'll ever need. At hard-to-beat prices. Call us.

New York: 516/621-6200, 212/767-0677, 518/449-5959
New Jersey: 201/227-5552 Pennsylvania: 412/931-9351
Ohio: 216/464-6688, 513/891-7050
Outside N.Y.S.: 800/645-6530

"QED" Discounts
VISA & MasterCard

Computer & Data Communications Equipment
Distribution/Systems Integration/Maintenance

Random Numbers

An LCG's period has two limitations: only integers less than the modulus are generated; and the series is deterministic.

Early repetition of the pattern.

Another way to think of this manipulation is in terms of a fictitious game of solitaire. Suppose I have two decks of cards. One of them has been shuffled, but I am asked to produce the longest possible nonrepeating series by laying down the cards from the unshuffled deck. I must follow some well-defined rule of my own choosing, and when I have gone through the deck I can start again. Before continuing, however, I must put the cards back into their original order. Since I cannot shuffle the deck in the usual way, if I merely start over, I will have to come to the end of the deck and the series will repeat. What can I do with the other deck?

Here is one procedure, and you can invent others that will work just as well or better. They are analogs for shuffling in the LCG. (The two decks correspond to the two LCGs.) Lay the cards in the unshuffled deck into five piles. Then remove a few cards from the shuffled deck so that it will not have the same number as the unshuffled deck. Then draw a card from the shuffled deck and take its value modulo 5 to compute a number designating one of the five piles. Select the first card from that pile and put it into a discard pile. Continue drawing cards from the shuffled deck, computing the number of their piles and selecting a card from that pile until you exhaust the shuffled deck. Then, start again with the cards in the same order. When the five piles from the unshuffled deck are exhausted, pick up the discard pile, put the cards back into their original order, and lay out five piles again. Continue as before. You will probably find that the
order is quite different the second time and in succeeding sequences. The success of this method depends on having an appropriate number of piles and taking an appropriate number of cards out of the shuffled deck before starting. In a similar way, the success of the shuffled LCG depends on the two series having appropriate relationships between them.

The efficiency of the shuffling technique is quite spectacular. For instance, from a pair of LCGs with periods of only 8 and 9, you can generate a pseudorandom series with a period greater than 200. By tailoring the pair of LCGs to the word length of the computer, you can create shuffled LCGs with much longer periods.

Listing 1 is a BASIC shuffling program that generates a random walk consisting of NG groups of NS steps and prints the displacement after each group. By analyzing this listing, you can develop a random-number subroutine suitable for virtually any computer. Notice that subroutine 1010 initializes the program by filling the "piles" with numbers, as in laying out the five piles of cards described above. The program uses two distinct LCGs and their parameters are listed in statement 32. Before demonstrating the power of this program, I will describe how I selected these parameters.

TAILORING A SHUFFLED LCG

I looked for the largest modulus and the largest multiplier consistent with (continued)

Figure 6: Amplitude spectrum for the Advanced BASIC random-number generator, showing the prevalence of certain cycles. "Frequency" is the number of cycles in 64,000 steps. The peak at about 4 corresponds to the wavy pattern evident in figure 4.
In addition to receiving a monthly newsletter containing computer ads at reduced prices, subscribers also receive a subscription fee to the Computer Swap Shop. The shop offers a Risk-Free Guarantee: if you are not satisfied with the equipment, you can return it within 10 days and get a full refund. If you are a member of the Computer Swap Shop and hold a card, you can earn a commission on each sale. Otherwise, the seller receives a commission on each sale.

I first needed an expression for the largest integer my computer can handle and for a particular LCG, which must satisfy the condition \( a \cdot m < 181 \). Thus you can construct a shuffled LCG from a pair of LCGs with periods less than 181, with \( a \) a critical additive constant. The condition \( a \cdot m < 181 \) also constrains the multiplier vector to avoid serial correlations. These two constraints do not always hold for the critical polynomial.

Figure 7 shows a 50,000-step random walk generated on an IBM PC by a 16-bit computer. For \( a = \sqrt{m} \), which satisfies the conditions taking a small setting for the critical polynomial.

Random Numbers

Listing 1: A program in BASIC for generating pseudorandom sequence from two LCGs with shuffling.

```basic
10 OPEN "RANWALKS.OUT" FOR OUTPUT AS #2
20 OPEN "RANWALKS.RAN" FOR OUTPUT AS #5
30 dimension n(1000), u(1000)
40 for i = 1 to 1000
50 print #5, i, n(i)
60 n(i) = n(i-1) + u(i)
70 if n(i) > 1000 then n(i) = n(i) - 1000
80 print #5, i, n(i)
90 u(i) = u(i-1) + v(i)
100 if u(i) > 1000 then u(i) = u(i) - 1000
110 print #5, i, u(i)
```

This integer must satisfy \( \text{MOD} (\text{MAX} - \text{MIN}) < max - \text{MIN} \), which leads to the approximate relationship \( m = \sqrt{m} \). As for a 16-bit computer, it is approximately 32,768 random numbers that satisfy the condition \( a \cdot m < 181 \) for a particular LCG, which implies \( a = m \cdot \text{MIN} \). Thus you can construct a shuffled LCG from a pair of LCGs with periods less than 181, with \( a \) a critical additive constant. The condition \( a \cdot m < 181 \) also constrains the multiplier vector to avoid serial correlations. These two constraints do not always hold for the critical polynomial.

Figure 7 shows a 50,000-step random walk generated on an IBM PC by a 16-bit computer. For \( a = \sqrt{m} \), which satisfies the conditions taking a small setting for the critical polynomial.
RANDOM NUMBERS

Figure 7: A random walk with shuffling, tailored to a 16-bit computer. In the absence of shuffling, this generator would be limited to a period of several hundred. With shuffling, the period is longer than 50,000. This walk was generated with an IBM PC.

Figure 8: Amplitude spectrum of the random walk in figure 7, showing absence of significant periodicities and indicating that the shuffled pair of LCGs passes this test successfully.

443
Eco-C Compiler  
Release 3.0

We think Rel. 3.0 of the Eco-C Compiler is the fastest full C available for the Z80 environment. Consider the evidence:

Benchmarks*  
(Seconds)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Eco-C</th>
<th>Aztec</th>
<th>Q/C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sieve</td>
<td>29</td>
<td>33</td>
<td>40</td>
</tr>
<tr>
<td>Fib</td>
<td>75</td>
<td>125</td>
<td>99</td>
</tr>
<tr>
<td>Deret</td>
<td>19</td>
<td>CNC</td>
<td>31</td>
</tr>
<tr>
<td>Matmult</td>
<td>42</td>
<td>115</td>
<td>N/A</td>
</tr>
</tbody>
</table>

*Times courtesy of Dr. David Clark  
CNC - Could Not Compile  
N/A - Does not support floating point

We’ve also expanded the library (120 functions), the user’s manual and compile-time switches (including multiple non-fatal error messages). The price is still $250.00 and includes Microsoft’s MACRO 80. As an option, we will supply Eco-C with the SLR Systems assembler - linker - librarian for $295.00 (up to six times faster than MACRO 80).

For additional information, call or write:

Eco-C Programmers  
6413 N. College Ave.  
Indianapolis, Indiana 46220  
(317) 255-6476

WALTZ LISP (TM)

The one and only adult Lisp system for CP/M users.

Waltz Lisp is a very powerful and complete implementation of the Lisp programming language. It includes features previously available only in large Lisp systems. In fact, Waltz is substantially compatible with Franz (the Lisp running under Unix), and is similar to MacLisp. Waltz is perfect for Artificial Intelligence Programming. It is also most suitable for general applications.

Much faster than other microcomputer Lises:  
• Long integers (up to 611 digits), Selectable radix  
• True dynamic character strings. Full string operations including fast matching/extraction  
• Flexibly implemented random file access.

• Binary files.  
• Standard CP/M devices.  
• Access to disk directories.  
• Functions of type lambda (expr), lambda (expr), map, macro.  
• Splicing and non-splicing character macros.  
• User control over all aspects of the interpreter.  
• Built-in pretty-printing and formatting facilities.  
• Complete set of error handling and debugging functions including user programmable processing of undefined function references.  
• Virtual function definitions.  
• Optional automatic loading of initialization files.  
• Powerful CP/M command line parsing.  
• Fast sorting/merging using user defined comparison predicates.  
• Full suite of mapping functions, iterators, etc.  
• Assembly language interface.  
• Over 250 functions in total.  
• The best documentation ever produced for a micro Lisp (300+ full size pages, hundreds of illustrative examples).

Waltz Lisp requires CP/M 2.2, 280 and 48K RAM (more recommended). All common 5” and 8” disk formats available.

Version 4.4  
$169


Call free 1-800-LIP-4000 Dept. #12  
in Oregon and outside USA call 1-503-684-3000

RANDOM NUMBERS

FOURIER SPECTRUM

Named for the French mathematician J. B. J. Fourier (1768-1830), the Fourier series is a representation of a series of data points, \( F(t) \), as a sum of harmonic terms in the form

\[ F(t) = \sum_n \left( a_n \sin(nwt) + b_n \cos(nwt) \right) \]

When \( F(t) \) is specified at a discrete set of points—such as the displacements in a random walk at uniformly spaced times—the coefficients \( a_n \) and \( b_n \) can be found by summing products of the data and the trigonometric functions. These coefficients comprise the amplitude spectrum, and the quantity \( \sqrt{a_n^2 + b_n^2} \) is a measure of the importance of harmonic frequency \( mw \) in the data. Large values of \( a_n \) or \( b_n \) indicate that the data has a significant component of variation with a period

\[ 2\pi/(nw) \]

using listing I, and figure 8 shows the corresponding amplitude spectrum. The spectrum shows a wide and fairly smooth distribution of frequencies, and the period of this shuffled LCG is evidently very long. A walk of this length represents a satisfactory type of pseudorandom-number generator for use with 16-bit computers. When 32-bit machines come along, changing the parameters to take advantage of the greater range of integers will be fairly straightforward.

CONCLUDING REMARKS


I’ve tried to show that the testing of random-number generators is important but not difficult. In fact, developing your own tests is an interesting game. There is no single right way, but the listing for the program provided here works quite well on my 16-bit machine.