SESSION XV TUTORIALS AND GENERAL APPLICATIONS

Howard L. Kaplan, *Presider*

Effective random seeding of random number generators

HOWARD L. KAPLAN

Addiction Research Foundation, Toronto, Ontario M5S 2S1, Canada

To obtain different sequences of values from a random number generating algorithm, we must start with different seed values or discard a variable number of initial samples. For maximum independence among sequences, differences must include the lowest order seed bits, and some generators require one to wait many samples for differences to propagate to the significant bits. While it is straightforward to generate two statistically independent sequences, it is surprisingly difficult to generate three or more such sequences.

Let us consider this question: How can one generate a random seed for a random number generator? The problem of seeding random number generators arises from the fact that so-called random number generators do not, in fact, generate random numbers at all. What they generate are very deterministic sequences of numbers, which will always be repeated if the same seed numbers are used to initialize them. The question actually has two components: Where can one obtain a set of distinct seed numbers for starting random generators at different points in their sequences? And, how can one be sure that similar but nonidentical seeds generate quite distinct sequences of random numbers? The first question is primarily one of operating system details: how the time and date are recorded, whether programs can determine the state of ongoing I/O operations, and whether computations can occur during input operations. The second question is much more algebraic and statistical in nature, related to word lengths and to arithmetic rather than to computer resource management.

TWO CLASSIC GENERATORS

Two of the best known algorithms for generating pseudorandom numbers, or sequences of numbers that have properties that we would want from random numbers, are the multiplicative and additive generators. This will be a somewhat informal and intuitive explanation of how they work, leaving the more formal and careful presentation to sources such as Knuth (1969). It should be stressed, however, that parameters such as word length, choice of multiplier or list length, and

radix (binary or decimal) of arithmetic must be chosen and tested very carefully. Both algebraic theory and empirical tests are available to help with these choices, and, to quote Knuth (1969, p. 5), "Random numbers should not be generated with a method chosen at random."

A multiplicative generator is often used on 32-bit computers, such as the IBM 370 series. Here is one in FORTRAN:

```
FUNCTION RINIT(NEWSED)
  INTEGER *4 SEED, NEWSED, MULT/1220703125/
  SEED = NEWSED
  DIVIDE = 2.**31
  RINIT = 0.
  RETURN
ENTRY RAND(X)
  SEED = SEED*MULT
  RAND = ABS(SEED/DIVIDE)
  RETURN
END
```

Before this generator is called for the first time, an

odd-number NEWSED must be passed as a function argument to the entry point RINIT. The value of MULT is large enough, close to the maximum value that can be stored in one computer word, that changes in the lowest order bits of the old value of SEED may cause changes in the highest order, most significant bits of the new value of SEED. In the process of multiplying the old SEED by MULT, many bits also propagate out to the left of the space available for recording the product, a condition that is formally called overflow, but which, in our case, is an intentional discarding of the most informative part of the product. The remaining part of the product, the lowest order 32 bits, carries the least information about the magnitude of the old SEED, even though it was completely determined by the bits of the old SEED. Because the part of the product that we keep is, for all practical purposes, unrelated to the magnitude of the old SEED, we can use the sequence of products so generated as effectively uncorrelated, uniformly distributed magnitudes: All we need to do is to scale them down to the range 0 to 1, by dividing by 2^{31} and taking the absolute value.

For computers without hardware multiply instructions or with multiply instructions that do not maintain low-order bits after overflow, a rather different form of generator is often used. This generator is based on a circular list of seed numbers, circular in the sense that the first item logically follows the last item. To generate a random integer, the previous random integer is added to the next item in the list, and the sum both replaces that next item and is used as the current random integer and is then scaled down to a real number between 0 and 1. Here is a FORTRAN implementation of that idea:

```

FUNCTION RINIT(LIST)
  INTEGER *4 LIST(22), OLDNUM
  IPOINT = 1
  DIVIDE = 2.**31
  RINIT = 0.
  RETURN
  ENTRY RAND(X)
  IPOINT = IPOINT + 1
  IF (IPOINT.EQ.23) IPOINT = 1
  OLDNUM = OLDNUM + LIST(IPOINT)
  LIST(IPOINT) = OLDNUM
  RAND = ABS(OLDNUM/DIVIDE)

```

RETURN

END

In this generator, the calling program does not simply provide one 32-bit seed number but, instead, provides a whole list of seed numbers, at least one of which is odd. Rather than the extremely complex transformation of bits provided by the multiplicative generator, this generator applies a largely magnitude-preserving addition operation to the current number and to a number from the relatively distant past, thereby achieving a good degree of uncorrelation between successive values. The list length of 22 is not arbitrary but is based on the existence of certain properties of 22nd-degree polynomials, ensuring that the sequence achieves maximum length before repeating. Unlike the multiplicative generator, the word length has very little effect on the statistical properties of this generator, except to change the period before the list of random numbers repeats. Insignificant low-order bits are not needed to carry information that gets transformed into the significant bits of the next random value returned. Instead, that information is carried by the other words of the list. Therefore, the word length need only be wide enough to represent the precision required from the random samples. If the generator is to be used only to choose among 256 alternatives, then it is reasonable to implement it in 8-bit single-precision arithmetic on a microcomputer, avoiding the unnecessary complications of multiple-precision arithmetic. Knuth (1969) says very little about the properties of additive generators; more information, including a table of optimal list lengths, is available in Jansson (1966).

The raw output of a random number generator is generally scaled to be a set of real numbers between 0 and 1, not including the upper endpoint. If a random integer is required instead, then standard practice is to multiply the output by the width of the range of integers, truncate the result, and add it to the lower end of the range. For example, to get a random integer between 6 and 10, we use this code:

$$J = 6 + \text{IFIX}(5. * \text{RAND}(X))$$

The product has a range from 0 to 5, not including the upper endpoint, which truncates to one of the five integers 0 to 4. Adding a constant 6 translates the range. While this logic is easy to implement on a computer with floating-point facilities in hardware or software, computers that require the use of the additive generator may not have such floating-point capability. In that case, another algorithm can be used to calculate the random integer from 0 to any desired maximum within the computer's word size. If M is the maximum, then let Q be the smallest number of the form $2^n - 1$ that equals or exceeds M . That is, let Q be a binary mask that retains only as many digits as are needed to express

M. Such a mask is easy to find: A word consisting entirely of 1 bits is shifted right while M is shifted left until it overflows. If a random binary integer, masked by Q, exceeds M, then it is discarded and another random selection is made. When a masked selection is found that does not exceed M, it is taken as the random integer to be chosen. In the best case, in which M itself is of the form $2^n - 1$, no numbers are rejected. In the worst case, in which M is of the form 2^n , this algorithm requires half of the raw random integers to be discarded. Even with a 50% rejection rate, the speed of this technique makes it a practical alternative for computers in which multiplication and division are slow operations.

SEEDING THE GENERATORS

Despite their differences, the additive and multiplicative generators (and most other reasonable generators on computers with 2s complement arithmetic) share one subtle requirement in their seed numbers: Although we are generally most interested in differences in the high-order bits of the numbers generated, any differences among different seed numbers should be concentrated into the lowest order bits of the seed. Any bit changes in the seed are propagated to the left only, and therefore high-order seed changes can never result in low-order bit changes in the subsequently chosen random numbers.

Since the multiplicative generator needs only 32 bits of seed information, I will begin by considering sources for that seed and later will consider ways to seed the other generator. The context of randomizing trials for a real-time experiment will be considered, although similar methods can be used for simulations, preexperimental design plans, and similar non-real-time problems. Reasonable sources are the system's automatic date and time recorder, the information used to identify different subjects and conditions on the hard-copy or magnetic media record of the experiment, and the inconsistent reaction time of the computer operator in responding to an opportunity to start the experiment.

Many current computer systems are provided with real-time clocks, which maintain time-of-day information in addition to allowing interevent timing. Systems that are never turned off may also maintain current date information; others may collect that information each day when they are turned on. Over a 10-year period, the current date provides somewhat more than 11 bits of real information (3,652 days), and its representation as a six-digit number can be converted to a BCD representation of at most 24 bits (4 bits/digit). Similarly, the time of day to the nearest 1 sec provides 86,400 possibilities, just over 16 bits of real information or 24 bits of apparent information as 6 BCD digits. If the computer's internal representation is condensed, so that the date and time are available as 12- and 17-bit

numbers, then simply concatenating those 2-bit strings in either order and following the concatenation with a 1 bit to insure an odd number provides a practically unlimited set of distinct seed numbers. There is no need for the date and time to be correct, only distinct from session to session: On the CompuColor II home computer, the "star date" keyed into the Star Trek game serves solely as a seed number for randomization.

The order of concatenation should concentrate changes within any one experiment into the low-order bits. On systems in which it is inconvenient to convert BCD representations to other forms, it may be necessary to discard some of the apparent 48 bits of seed information contained in the character representations. For experiments that start at somewhat uncertain times of day because of scheduling difficulties with human subjects, it is best to discard the high-order date information (year and month), which changes little over the course of most experiments, and to retain all of the time-of-day bits. For long-term experiments conducted in the chambers in which animal subjects live, automatically scheduled to begin at the same time every day, clearly the opposite advice would hold: Use the varying date information as the low-order part of the seed and discard the high-order time-of-day bits. Although the seeds generated on two successive sessions may be similar, the extreme scrambling caused by the multiplicative algorithm means that sequences formed from different but similar seeds are uncorrelated after only one multiplication, that is, as of the first random number after seeding. Figure 1 shows 120 consecutive values output by a multiplicative generator, using both an original odd seed value and the next higher odd seed value. It is clear from the graph that the two output

Figure 1. Consecutive values output by a 30-bit multiplicative random number generator using two different seeds, the original odd seed (filled circles), and the next higher odd number (open circles). The scattergrams at the right have the original seed's outputs on the abscissa.

sequences are immediately uncorrelated: Scattergrams of three 40-pair samples provide a graphic view of the lack of correlation.

For systems without automatic time maintenance, an equally good alternative is to use the information collected to identify subjects, sessions, experimental conditions, and so on. On the FRIVLOS operating system at the University of Toronto, subjects sign themselves into experimental sessions with their names, subject numbers, and session numbers. This information, in addition to its use in ensuring that conditions are run in the proper order, forms the seed list for the kind of additive generator discussed above. The seed list is also stored on tape along with other experimental results, making it possible to reconstruct the randomization of stimulus conditions in greater detail than is normally available from the trial-by-trial record. For systems using a multiplicative generator, it is possible to let one or two characters of name information be used in the seed, along with a few bits each of subject number, session number, and any between-session condition code the computer derives from the subject and session numbers, as in a Latin square design.

The random seeding technique used in C. D. Creelman's PSYCLE system at the University of Toronto is, on the surface, the simplest of them all. Whenever the system is loaded, the same random seed list is used; what varies is the number of random values that are discarded before the sequence is used. In general, when the computer is waiting for Teletype input, it is simply idling, but when it is waiting for a response to 'READY? TYPE "GO"', it is calling the random number generator as many times as possible between character inputs. Since the operator response time is very large compared with the time to call the generator, on the order of at least 1,000 calls/character, it is clear that we are dealing with a distribution of at least hundreds of reasonably likely sequence starting points. The reason that I call this technique simple on the surface only is that it is difficult or impossible to implement in some operating systems. If the operating system considers the reading of characters as an activity that alternates with arithmetic computation, rather than as one that can occur in parallel, then there is no way to simultaneously ask for random numbers and monitor the keyboard for input. It is, for example, impossible in standard FORTRAN, in which it can be accomplished only by special subroutine calls analogous to the PEEK function in BASIC, and then only if the operating system allows unrequested input to be typed. With such machine-level functions in FORTRAN or BASIC, the technique can be implemented and is reasonable, providing that the time to call the generator is a very small fraction of the operator's response time.

One randomization technique that cannot be recommended is the simple reliance on the vendor's RANDOMIZE function for starting a random number generator at a supposedly random point. On the Hewlett-Packard 9845 system, for example, such a function call

will start the random number generator at 1 of only 116 points. While this may be adequate for some games, it is not adequate for an experiment in which 200 different randomly ordered word lists are needed. If there is no other way to seed the random number generator, then the only safe course is to combine the vendor's technique with the third technique discussed above, discarding an inconsistent number of samples before beginning serious randomization.

AN ANALYSIS OF MINIMAL RESEEDING

Although most of these methods will provide somewhat irregularly spaced seed numbers, it is unwise to rely on any particular distribution of experiment starting times or on the spellings of subjects' names to ensure the proper randomization of experiments. Whatever reseeding techniques we use should be powerful enough that we can transform raw session numbers, 1, 2, 3, and so on, into acceptable seed values. To see what is required of such a transformation, we must study the effects of the minimal possible changes in seed values on the subsequent numbers generated. For an additive generator, a minimal reseeding is adding 1 to the first seed list item; for a multiplicative generator, it is adding 2 to the odd seed to produce the next odd number.

Multiplicative generators are seed-and-go types, because they immediately distribute changes in any seed bit over the higher order bits of the updated seed. In the additive generators, however, any changes in low-order bits propagate only slowly to higher order bits and, hence, only slowly to the significant bits of the real numbers returned by the generator. Another way of viewing this effect is to imagine applying the same 32-bit seed to a multiplicative and an additive generator. In the multiplicative case, one multiplication immediately distributes the randomization over the entire history; in the additive case, one addition propagates the randomization only over an additional 1/22 of the history. When the changes in the seed occur only in the lowest order bits, as happens when the current date or the session number forms the low-order part of one additive generator seed list word, the generator must be called many times in order to propagate those changes to the significant, high-order bits. It is therefore important to know how many numbers must be discarded after seeding an additive generator.

We can take an analytical approach, based on the simplifying assumption of a seed list of all 0s except for the first item, which is a 1. Just as in the Fibonacci sequence, successive values of the current output term tend to approach geometric growth (Figure 2). Performing 22 growth steps on the current item is the same as performing 21 growth steps and then adding the current item, leading to this equation, where c is the current item and r is the ratio between successive items:

$$cr^{22} = cr^{21} + c.$$

Dividing through by c and solving by successive approximations yields $r = 1.10+$. With a 32-bit word length, to raise the original item to first overflow, or $2^{*}32$, therefore takes $\log(2^{*}32)/\log(1.10+)$, or 211 operations. This corresponds well to the empirical observation of first overflow after 222 operations. This difference, 11 operations, is half of the seed list length. Figure 2 shows the best-fitting line bisecting the first 22 no-growth samples. By the 256th operation, c is a 37-bit number and the lowest 32 bits are an effectively random sequence. It seems sufficient to discard the first 256 samples after providing this generator with a seed list that might differ from another seed list only in the lowest order bit of one item. A longer seed list requires more calls to distribute the effects of the first randomization, and a shorter word length requires fewer calls. Figure 3 shows the distributions of consecutive random numbers from a 22-item, 30-bit generator with its original seed list and a minimally modified list. The first 120 samples are not even shown, as the differences in Samples 121-160 are not revealed by the graph. Only after 160 samples are discarded do we begin to see any difference in the significant bits of the output, and only after 200 samples are discarded is the 40-pair correlation essentially 0.

Let us tentatively adopt lack of linear correlation as a measure of independence and look at the correlation between outputs from an additive generator and a

Figure 2. Consecutive values that would be output by an additive generator seeded with one 1 and 21 0s, were there no overflow. Once asymptotic growth is achieved, each value output is about 10% larger than the previous value. The ordinate, if rescaled, can also be read as the effect of a minimal reseeding on the output of any seed list.

Figure 3. Consecutive values output by an additive random number generator with word width of 30 bits and list length of 22 items. Points are from the original seed list (filled circles) and the addition of 1 to the first item of the seed list (open circles). Scattergrams at the right have the original seed's outputs on the abscissa.

Figure 4. (a) Distribution of the 10-item Pearson product-moment correlation between outputs of 600 original and minimally reseeded additive random number generators, as a function of the number of samples discarded before the correlation is evaluated. The three lines show the 10%, 50%, and 90% points on the observed distribution. (b) The same, but for a minimally reseeded multiplicative generator.

minimally reseeded version of the same generator. In Figure 4a, we see the distribution of the 10-item Pearson product-moment correlation coefficient between samples of original and reseeded additive generators. For each of 600 seed lists, samples were taken both before and after adding 1 to the first seed list item, and the correlation was calculated after first discarding 0 through 300 samples from each generator. The three lines on the figure show the 10%, 50%, and 90% points on the empirical distribution of correlations. For about the first 150 samples, the changes have almost no detectable effect on the most significant digits of the output, and it is not until about 200 samples have been discarded that we start to see a quasiasymptotic

distribution of correlation coefficients. However, for some numbers of samples discarded, the whole distribution tends to be shifted higher or lower than its usual range. At around 245 discards, for example, most of the correlations are negative and the median correlation is near $-.25$. This is not a sampling error effect; even if we were investigating 60,000 seed lists we would still see a median that is far from 0 after a certain number of discards.

The failure of the median correlation coefficient to approach 0 as the number of samples increases suggests that we are not studying 600 pairs of sequences such that all 1,200 are mutually independent. In particular, the difference between the two sequences in any pair is the same sequence (Mod 1), the lower 30 bits of the growth function that is graphed in Figure 2. At some points in that function, most of 10 consecutive output numbers are all near 0 or 1, so that there is a high correlation between the original and reseeded output. At other points, most of the current set of 10 are near .5, so that the original and reseeded outputs are on opposite sides of .5 and the correlation tends to be negative.

The problem is not peculiar to the additive generator; the multiplicative generator also fails to have the correlation approach 0 as the number of samples discarded after reseeding grows indefinitely. Figure 4b shows the same kind of distribution of 10-item correlation coefficients that is shown in Figure 4a, but for a multiplicative generator. Again, there are some numbers of discards that almost always yield a negative correlation. Just as in the case of the additive generator, there is a consistent difference sequence between the generators, the sequence that we would get if we were to use 2 (the difference between consecutive odd seeds) as a seed value. Unlike the additive generator, we can guarantee that the correlation function will immediately take on its quasiasymptotic behavior, but that behavior is no better than with the other generator.

In formal terms, what is happening is a consequence of the way in which the generators operate on their seeds. The multiplicative generator is clearly a linear operation on each seed to produce the next seed, with the word length as a modulus. That is, the subsequent seed generated by the sum of two seeds is the sum of their individual subsequent seeds. Although it is computationally efficient to operate on only two items of an additive generator at a time, the process is formally equivalent to rotating the entire seed list up one position and adding the former and new first items together to form a replacement first item. That, too, is a linear operation on the seed vector. In either generator, the same minimal reseeding will always produce the same difference in output sequences. The same reseeding phenomenon also occurs with a variant of the multiplicative generator known as the mixed congruential generator, in which a constant is added to the new seed after each multiplication. Although this is not a linear

transformation of the seed, it is similar enough to a linear transformation that any small seed change produces a consistent sequence of differences between the original and reseeded generators.

Perhaps the most striking demonstration of the reseeding problem arises when we ask what we mean by the claim that two or more sequences are statistically independent. A single sequence is an acceptable source of random numbers only if the n -tuples of consecutive values are uniformly distributed through n -space. (Knuth, 1969, discusses some more stringent definitions, which are not needed for this demonstration.) As an example, we expect the scattergram of pairs of consecutive outputs to be uniformly distributed throughout the unit square. It seems reasonable to state that two sequences $a_1, a_2, a_3, \dots$ and $b_1, b_2, b_3, \dots$ are statistically independent only if the interlaced sequence $a_1, b_1, a_2, b_2, a_3, b_3, \dots$ passes the same kinds of tests of randomness. We can easily extend such a definition to three or more such sequences.

In Figure 5 we have some three-dimensional distributions of triples of consecutive numbers taken from a single distribution, two interlaced distributions after a minimal reseeding, and three interlaced distributions after the same minimal reseeding has been applied again. That is, the three multiplicative seeds are $n, n+2$, and $n+4$. It is clear that the triples are uniformly distributed for one or two distributions but are concentrated onto three planes for three distributions. That is, the number from the third sequence is predictable (Mod 5) from the corresponding numbers from the first two sequences. If we instead use an irregular seed increment, with seeds of $n, n+2$, and $n+6$, as in Figure 6, we again find the results concentrated onto planes, but now there are more planes, spaced more closely together. In order to get the planes spaced closely enough together that the third sequence is, for all practical purposes, unpredictable from the first two sequences, the reseeding increments must have a greatest common divisor that is small compared with the larger increment. In other words,

Figure 5. Distribution of consecutive triples from one multiplicative random number generator and from the interlaced outputs of two and three generators. The three seeds are of the form $n, n+2$, and $n+4$.

Figure 6. Distribution of consecutive triples from one multiplicative random number generator and from the interlaced outputs of two and three generators. The three seeds are of the form n , $n + 2$, and $n + 6$.

increments such as 2 and 1,202, or 1,200 and 1,202 would provide a much better approximation to mutual independence among the three sequences. It is reasonable to conjecture that transforming a sequence of session numbers 1, 2, 3, . . . by any algebraically regular transformation, such as squares + 1 (2, 5, 10, 17, 26, . . .), would lead to similar algebraic interdependence among sequences. This would imply that it is necessary to provide randomly-spaced seed increments to produce

independent sequences. The question is still under investigation.

The phenomenon of the distribution of n -tuples falling onto widely spaced planes is not unique to reseeding of generators. It occurs in multiplicative generators with poorly chosen multipliers and is called low potency. What has been demonstrated here is that when the same algorithm is to be reused, we must also be concerned with low potency of the reseeding process, which can easily occur when evenly spaced integers are used to reseed the same generator. Another paper is now in preparation (Kaplan, Note 1) that will investigate these phenomena in more formal, algebraic terms and will provide results of some theoretical and empirical tests of possible solutions to this reseeding problem.

REFERENCE NOTE

1. Kaplan, H. L. *Generation of appropriate seed values for reuse of the same random number generator*. Manuscript in preparation, 1981.

REFERENCES

- JANSSON, B. *Random number generators*. Stockholm: Victor Pettersons Bookindustriab, 1966.
- KNUTH, D. E. Random numbers. In *The art of computer programming* (Vol. 2). *Seminumerical algorithms*. Reading, Mass: Addison-Wesley, 1969.