Form C20-8011

IBM Data Processing Techniques

Random Number Generation and Testing

Design

Random sampling—simulation studies—Monte Carlo methods—have been in use for many years. Papers describing various aspects of these topics have appeared in technical journals and textbooks available to a relatively small percentage of computer users. Meanwhile, applications requiring random numbers are becoming more important and more common in business and industry as well as purely scientific areas. Therefore, this manual has been prepared which gives the mathematical development of the power residue method, outlines computer techniques for implementing it, and also offers brief comments on other methods. An appendix provides programming illustrations for binary and decimal computers.

Preface

There are two facts relating to random numbers which undoubtedly are not sufficiently recognized:

1. Many commercial, as well as technical, computer applications require random numbers.
2. Random numbers are harder to come by than one might suspect.

The power residue method for obtaining random numbers is in many ways superior to other methods and also is entirely satisfactory if used properly. However, the method requires choosing certain parameters, and an understanding of the mathematical basis for the method is necessary in order to choose these optimally.

This manual describes the mathematical development of the power residue method, outlines computer techniques for implementing it, and also offers brief comments on other methods. An appendix provides programming illustrations for binary and decimal computers.

Contents

Introduction

There are quite a few problems for which a probabilistic model can be formed, so that a solution may be found by a statistical sampling technique—a "Monte Carlo" method. Hence a need for random numbers. Examples include not only algebraic problems such as those involving permutations and combinations of variables, and certainly many standard statistics problems, but also the evaluation of definite integrals, the solution of ordinary and partial differential equations, solution of integral equations, solution of linear algebraic equations, and matrix inversion. Information on these and other examples may be found in the references listed at the end of this paper. The method is most useful when only little accuracy is desired, since increased accuracy generally requires greatly increased sample size.

To illustrate, a definite integral representing an unknown volume V may be found approximately by enclosing V in a cube of volume C, splattering N points at random within the cube, and counting the number n of these that also lie in V; then $V = (n/N)C$, approximately. It may be observed in this case that the points need not be truly random, but may be only uniformly dense, since their order is immaterial. However, a random number generator has the merit of being applicable no matter how many points are used, and additional points may be added by the same subroutine if greater accuracy is desired.

A much more important group of applications can be solved *only* by employing random numbers. These problems include mathematical "games of strategy," which involve incomplete information, and problems which involve variables that can only be described statistically—for example, simulation studies in business and industry, or science. To illustrate, an IBM 704 program designed to simulate the operation of a job shop may use about 5,000 random numbers in a 15-minute run, these numbers being used to develop statistical variables such as transit time from machine to machine. Again, computers are used to simulate automobile traffic flow for purpose of highway and signal system design. Statistics describe the overall flow, and random numbers place individual vehicles so as to study the flow in detail. One of the best known applications of the Monte Carlo method is to a simulation study in nuclear physics.

Strictly speaking, a random number exists only as the result of a random process. It is often suggested, therefore, that a mechanical or electronic device—a perfect roulette wheel—be built to supply truly random numbers on demand. This is not done because (1) nature tends to be systematic, so that construction and maintenance of such a device—which must output millions of times in an unsystematic manner, and at millisecond or even microsecond speeds—is not at all cheap or easy; (2) fast arithmetic procedures do exist whose results, though of course not random, nevertheless do furnish a satisfactory substitute; (3) it is sometimes desirable to repeat a calculation exactly—for example, in checking machine operation. This is possible with arithmetic schemes, but of course not possible if a random device is involved.

Another alternative to an arithmetic scheme would be to record random numbers on magnetic tape, say from the Rand table [1]*, and read in as required. This is not done because it is too slow, because using an internal subroutine is easier, and because large problems require more numbers than have been published. (You cannot double the size of a table by reading it twice, just as you cannot read only the first page over and over.) Also, one might object to reading the same "random" data for every problem; arithmetic schemes avoid this objection by allowing the user to specify starting values and/or other parameters.

* See References at end of text.

The procedures to be described produce numbers u_n that are (approximately) uniformly distributed over the unit interval. These numbers will be called random, even though they are not, rather than use a more awkward term like pseudo-random, quasi-random, etc. It is important to distinguish between random *numbers* (points on the unit interval) and random *digits*—or bits, in a binary machine. The procedures described obtain numbers since these are what is most generally desired. If random digits are needed, they should be taken from the high-order end of the numbers; the comments following the description of the recommended methods will elaborate on this problem. It should also be noted that many of the statistical tests commonly used to examine the randomness of a sequence of numbers are really tests on the digits appearing in the numbers; in view of the comments above, tests will be emphasized that treat the numbers as points.

Often one needs numbers that have a specified statistical distribution other than a uniform distribution on the unit interval. It so happens that this more general problem is easily solved once a supply of uniformly distributed random numbers is obtained. For, suppose F(x) is the cumulative distribution function for the desired probability density function f(x), and let u be uniformly distributed on (0,1)*. Then the variable $F^{-1}(u)$ has f(x) for its probability density function, since

$$P\left(F^{-1}(u) \leq x\right) = P\left(u \leq F(x)\right) = F(x).$$

Example 1. Alternative decisions A,B,C are to be chosen with respective probabilities 1/6, 2/6, 3/6. Use a uniformly distributed (on (0,1)) random variable u to determine the decision.

Although this example has a simple, rather obvious solution, it will show that the above theory applies. Let the decisions (A,B,C) correspond to integers (0,1,2) as determined by the intervals $(0 \leq x < 1, 1 \leq x < 2, 2 \leq x < 3)$; then distribution functions for this problem can be defined as follows:

$$\begin{array}{lll} f(x) = 0 \ , & F(x) = 0 \ , & x < 0 \\ f(x) = 1/6, & F(x) = x/6 \ , & 0 \leq x < 1 \\ f(x) = 2/6, & F(x) = (2x - 1)/6, & 1 \leq x < 2 \\ f(x) = 3/6, & F(x) = (3x - 3)/6, & 2 \leq x < 3 \\ f(x) = 0 \ , & F(x) = 1 \ , & 3 \leq x \end{array}$$

Solving for F^{-1} gives

$$\begin{array}{rll} F^{-1}(u) & = 6\,u \ , & 0 \leq u < 1/6 \\ & = (6\,u + 1)/2, & 1/6 \leq u < 3/6 \\ & = (6\,u + 3)/3, & 3/6 \leq u < 1 \end{array}$$

Thus, if u is uniformly distributed over the unit interval, then

1/6 of the time we have $0 \leq F^{-1}(u) < 1$

2/6 of the time we have $1 \leq F^{-1}(u) < 2$

3/6 of the time we have $2 \leq F^{-1}(u) < 3$

Example 2. Use a variable u, uniformly distributed on (0,1), to obtain a variable N, normally distributed with mean m and standard deviation s.

As in example 1, we could construct a table for a probability distribution f(x), integrate to obtain F(x), and solve for $F^{-1}(u)$. However, in special cases one often finds special procedures; hence a special method will be used to illustrate this example.

The uniformly distributed variable u has mean 1/2 and variance $\int_0^1 (x - 1/2)^2 dx = 1/12$, so the statistical "central limit theorem" states that if n values of u are added and the sum is divided by n to obtain $\bar{u}$, then the variable $x = (\bar{u} - 1/2)\sqrt{12n}$ is approximately normally distributed with mean 0 and standard deviation 1 and approaches this normal distribution as n becomes infinite. Therefore, one may set $N = m + s(\bar{u} - 1/2)\sqrt{12n}$ and then choose a suitable value of n. If $n = 12$, which is probably large enough for most purposes, then $N = m + s(\Sigma_1^{12} u_i - 6) = (m - 6s) + s\Sigma_1^{12} u_i$.

* Definitions: The cumulative distribution function F(x) for a random variable r gives the probability that $r \leq x$; the probability density function $f(x) = dF(x)/dx$; $F^{-1}(u)$ is the ordinary inverse function to F(u). The variable u is uniformly distributed on (0,1) if it has the cumulative distribution function $F(x) = x, 0 \leq x < 1$.

Number Theory Background for the Power Residue Method

Any sequence of numbers produced by a subroutine with finite input will eventually repeat, since the computer has only a finite number of "stable states." Thus, the first problem in determining a procedure to produce random numbers is to assure a long period for the repeating sequence. Number theory discloses ways of obtaining long periods with simple arithmetic procedures; of course, the results must still be studied statistically before they can be accepted as random. The two principal number theoretic ideas employed are congruences and power residues; the pertinent information is summarized here.

Congruence, Modulus, Residue (algebraic symbols represent integers):

The congruence $x \equiv y \pmod{m}$, read "x is congruent to y modulo m" or "x is congruent to y mod m", means that $(x-y)$ is divisible by m. Either x or y is a residue, and m is the modulus. Commonly, the residue is the remainder after division by m.

Examples: $5591 \equiv 7 \equiv -1 \pmod{8}$; $2^{p-1} \equiv 1 \pmod{p}$ if p is an odd prime.

The special case $x \equiv y \pmod{1}$ is interpreted to mean that x and y may not be integers but that their difference is an integer.

If $a \equiv b \pmod{m}$ and $x \equiv y \pmod{m}$, then $a \pm x \equiv b \pm y \pmod{m}$ and $ax \equiv by \pmod{m}$; however, division involves special considerations. If $(d,m) = 1$, that is, d and m are relatively prime, so that the greatest common divisor (gcd) of d and m is 1, and if $dx \equiv dy \pmod{m}$, then $x \equiv y \pmod{m}$; but if the gcd of d and m is $(d,m) = g$, and $dx \equiv dy \pmod{m}$, then it can only be concluded that $x \equiv y \pmod{m/g}$. Note also that if $x \equiv y \pmod{m}$ and d divides m, then $x \equiv y \pmod{d}$.

Congruences are often used to permit working with smaller numbers. For example, time is measured in hours modulo 12 and in days modulo 7 or 365. Again, one can determine that $N = 1 + 147\,(2^{147})$ is exactly divisible by 31 without computing the 47 digit value of N. Taking all congruences mod 31,

$$2^5 \equiv 1 \pmod{31},$$

so $(2^5)^{29} = 2^{145} \equiv 1 \pmod{31}$

and $2^{147} = 2^{145}(4) \equiv 4 \pmod{31}$.

Also, $147 \equiv -8 \pmod{31}$

so $N \equiv 1 + (-8)(4) = -31 \equiv 0 \pmod{31}$.

The "casting out 9's" method of checking arithmetic is based on the fact that $10^n \equiv 1 \pmod{9}$ for all n. Some oft repeated algebra problems involve remainders and are much more easily solved using congruence notation. We will use congruences both theoretically and in actually computing random numbers.

Residue Class: A class of all integers which are mutually congruent for a given modulus. There are m distinct residue classes mod m, corresponding to the terms of a complete residue system mod m; collectively they comprise the class of all integers.

Complete Residue System: For a given modulus m, a set of m numbers congruent in some order to the residues $0, 1, \cdots, m-1$.

Reduced Residue System: A subset of a complete residue system, containing all terms which are relatively prime to m.

Euler's phi-function $\phi(m)$ denotes the number of positive integers less than m and prime to m, so a reduced residue system contains $\phi(m)$ terms. If the prime factorization of m is $m = \Pi p_i^{e_i}$, then $\phi(m) = \Pi (p_i - 1) p_i^{e_i - 1}$.

Power Residues: The residues of the successive powers of a number: $x^n \pmod m$ for $n = 1,2,3, \cdots$.

Order of x mod m: When x and m are relatively prime, the least positive exponent h with $x^h \equiv 1 \pmod m$. Euler showed that $x^{\phi(m)} \equiv 1 \pmod m$, from which it follows that h is a divisor of $\phi(m)$.

Primitive Root of m: A number x whose order $h = \phi(m)$.

The powers of a primitive root x generate a reduced system of residues mod m (the largest possible set), and all primitive roots occur in this set as those powers whose exponents are relatively prime to $\phi(m)$. Therefore, if m has a primitive root, it has $\phi(\phi(m))$ primitive roots. However, only numbers of the form p^n or $2p^n$, for p an odd prime, have primitive roots, with the singular exception $m = 4$. Unfortunately, there is no easy method of finding primitive roots for a large modulus m. Any number x can be tested individually, however, for a prime modulus p, as follows: let p_i be the distinct prime factors of $p - 1$, and compute each of the quantities $r_i \equiv x^{(p-1)/p_i} \pmod p$; if no $r_i \equiv 1 \pmod p$ then x is a primitive root of p, otherwise not. If the modulus $m = p^n$ and x is a primitive root of p, then x is a primitive root of m unless $x^{p-1} \equiv 1 \pmod{p^2}$ in which case $x + p$ is a primitive root of m.

The power residue method for generating random numbers, which is most widely used, computes power residues with appropriately selected x and m. Lehmer [2] originally suggested choosing m to be a large prime p and x a primitive root of p; however, it is easier, faster, and for most purposes just as good, to let m represent the word size of the computer and choose x so as to assure a long period for the repeating sequence $x^n \pmod m$. It turns out to be easy to assure a long period, and although the method must be checked statistically, intuition suggests it may prove to be quite good since successive multiplications mod m give points that hop about in a somewhat erratic manner.

Examples of power residue calculations

1. $m = 31$, $x = 5$: the sequence is 5,25,1 and then repeat, showing that the order of 5 (mod 31) is only 3.
2. $m = 31$, $x = 3$: the sequence is 3, 9, 27, 19, 26, 16, 17, 20, 29, 25, 13, 8, 24, 10, 30, 28, 22, 4, 12, 5, 15, 14, 11, 2, 6, 18, 23, 7, 21, 1 and then repeat, showing that 3 is a primitive root of the prime modulus 31: all 30 positive integers less than 31 appear.
3. $m = 24$, $x = 2$: the sequence is 2, 4, 8, 16, 8, 16, 8, $\cdots$; it does not return to the starting point because $(x, m) \neq 1$.

Notes on power residues:

1. If $(x, m) = 1$ then $x^n \pmod m$ repeats by returning to the starting point since $x^r \equiv x^s \pmod m$ implies $x^{r-s} \equiv 1 \pmod m$; the division is possible because $(x,m) = 1$.
2. If the terms $x^n \pmod m$ are multiplied by a constant a that is relatively prime to m, then the resulting sequence has the same length and still repeats by returning to the beginning. In fact, if x is a primitive root of m then the sequence $ax^n \pmod m$ is the same sequence as $x^n \pmod m$ but with a different starting point (a, instead of x). Therefore, our procedures for computing random numbers will obtain $u_n \equiv ax^n \pmod m$, for suitable a,x,m; and the choice of the starting value a — at least — will be left to the user. It is important, however, that a be chosen relatively prime to m, for otherwise the common factor will be repeated in all terms, and, worse, the period will be shortened because the modulus has been effectively reduced by the same factor.

   Examples: $u_n \equiv 7 \cdot 3^n \pmod{100}$: 7, 21, 63, 89, 67, 1, 3, 9, 27, 81, 43, 29, 87, 61, 83, 49, 47, 41, 23, 69, 7 — (repeats).

   $u_n \equiv 5 \cdot 3^n \pmod{100}$: 5, 15, 45, 35, 5 — already repeating; shorter since $(5,100) \neq 1$.

   (Corresponds to 3^n mod 20)

3. If x is a primitive root of $m = p^e$, $(p > 2)$, then the second half of the sequence $ax^n \pmod m$ is just the negative of the first half. This is because $\phi(m)$ is even so $x^\phi \equiv 1$ implies $(x^{\phi/2} - 1)(x^{\phi/2} + 1) \equiv 0$; now $x^{\phi/2} \not\equiv 1$ since x is a primitive root of m, and p cannot divide both factors since they differ by only 2, so $x^{\phi/2} \equiv -1$, which implies the preceding statement.

   Example: In the sequence $3^n \pmod{31}$ of example 2 above, the 15th term is $30 \equiv -1$, the 16th is $28 \equiv -3$, etc.

4. It is possible to back up and compute the terms

$u_n \equiv ax^n \pmod{m}$ in reverse order, if desired, as long as $(x,m) = 1$. For the equation $xy \equiv 1 \pmod{m}$ can be solved for y by trial or by standard techniques of number theory, and then $u_{n+1} \equiv xu_n \pmod{m}$ is equivalent to $u_n \equiv yu_{n+1} \pmod{m}$.

5. If $m = ab$ with $(a,b) = 1$, and the order of x (mod a) is h_a and the order of x (mod b) is h_b, then the order of x (mod m) is $h = \text{lcm}\,[h_a,h_b]$, the least common multiple of h_a and h_b. This theorem is used in the following note and will be needed again in Paragraph II, following.

   Proof: $x^n \equiv 1 \pmod{m}$ implies both $x^n \equiv 1 \pmod{a}$ and $x^n \equiv 1 \pmod{b}$, and the statement "h_a is the order of x (mod a)" implies that the series $x^n \pmod{a}$ repeats only after blocks of length some multiple of h_a; similarly h_b; therefore $x^n \equiv 1 \pmod{m}$ implies n is a multiple of both h_a and h_b. Next, $x^n \equiv 1 \pmod{a}$ and $x^n \equiv 1 \pmod{b}$, $m \equiv ab$ and $(a,b) = 1$, together imply $x^n \equiv 1 \pmod{m}$, so the order of x (mod m) is the least common multiple (lcm) of h_a and h_b.

6. Lehmer's method for the Eniac, "multiply an 8 digit number by 23, remove the top two digits from the ten digit product and subtract them from the bottom," is a power residue method with modulus $m = 10^8 + 1$. This follows since $u_{n+1} = 23u_n - k\,(10^8 + 1)$. Now $m = 10^8 + 1 = 17 \cdot 5882353$ is not prime, but the second factor is, and 23 is a primitive root of this factor. Since $\phi\,(17) = 16$ divides $\phi\,(5882353) = 5882352$, the order of 23 mod m is 5882352, so this is the number of terms in the repeating sequence. Of course, if $m \neq 10^8 + 1$ then the number 23 may not be a good multiplier.

With this background, some specific procedures for generating random numbers can now be developed. The power residue method has been chosen because considerable experience with it, as well as repeated investigations of other methods, has shown it to be a best choice for quality of results in addition to being extremely simple and fast. Note that the calculation of power residues must be done by fixed point integer arithmetic, and division by m, retaining only the remainder, is implied by the (mod m) reduction. An additional division by m is required to convert the number to an appropriate point on the unit interval.

I. **For a binary machine.** The modulus $m = 2^b$, representing the word size of the machine, is chosen for two reasons: first, reduction mod m involves merely keeping the low b bits, and second, conversion to the unit interval involves only assigning the binary point to the left of the number; therefore both divisions are circumvented. The main problem is to determine x to assure a long period.

The order h of any number x (mod 2^b) is a divisor of $\phi\,(2^b) = 2^{b-1}$, and so is of the form 2^e for $e \leq b - 1$. In fact, $e < b - 1$ for $b > 2$ since then $m = 2^b$ has no primitive roots. Of course, x will have to be odd, and all odd numbers are either of the form $8t \pm 1$ or $8t \pm 3$. We observe that $(8t \pm 1)^{2^{b-3}} = 1 \pm 2^b t + \ldots \equiv 1 \pmod{2^b}$ so that the order of $x = 8t \pm 1 \pmod{2^b}$ is a divisor of 2^{b-3}. However, $(8t \pm 3)^{2^{b-3}} \equiv 3^{2^{b-3}} = (4 - 1)^{2^{b-3}} \equiv 1 - 2^{b-1} \not\equiv 1 \pmod{2^b}$, although $(8t \pm 3)^{2^{b-2}} \equiv (4 - 1)^{2^{b-2}} \equiv 1 \pmod{2^b}$, so the order of $x = 8t \pm 3 \pmod{2^b}$ is $h = 2^{b-2}$ and this order is maximal. This leads to the following procedure: choose any odd integer a for the starting value, choose an integer x of the form $x = 8t \pm 3$ for a multiplier, and compute $ax^n \pmod{2^b}$.

Now the desired output is a sequence of (random) numbers u_n on the unit interval; however, the arithmetic of the subroutine producing these numbers is necessarily integer arithmetic. Therefore, although the u_n are interpreted as *integers* while they are being calculated, merely reassigning the fixed binary point or appropriately converting to floating decimal form gives values $0 < u_n < 1$ for final results.

Procedure: Choose any odd integer u_o for the starting value and choose an integer x of the form $x = 8t \pm 3$ for a multiplier. Compute $u_{n+1} \equiv xu_n \pmod{2^b}$ using fixed point integer arithmetic, but interpret the result as a b-bit binary fraction.

Notes.

1. This procedure will produce 2^{b-2} terms before repeating.
2. Some values of x are better than others; see the section on statistical tests. (A good choice is $x \doteqdot \sqrt{m}$.) However, any starting value u_o is as good as any other as long as $(u_o, m) = 1$.
3. Computing time can be shortened by choosing x to have few 1's, and using shift and add commands instead of multiply.
4. $u_n \equiv x^n u_o \pmod{2^b}$ implies $u_n \equiv x^n u_o \pmod{2^e}$ for $e \leq b$, so we see that the low-order bits of u_n are far from random. In fact, if b_1 is the low-order bit of u_n, b_2 the next to last, b_3 the 3rd from last, etc., then

   $b_1 = 1$ for all u_n

   b_2 and b_3 either do not change or else alternate as n changes

b_4 has period 2^2

.

.

b_r has period 2^{r-2}

.

.

the high-order bit has period 2^{b-2}

This last note explains the comment in the introduction that if random bits are needed, they should be taken from the high-order end of the u_n.

II. **For a decimal machine.** We choose the modulus $m = 10^d$, representing the word size of the machine, again to simplify both arithmetic mod m and conversion to the unit interval. Now x must be relatively prime to 10, and the order h of x (mod 10^d) is at most lcm $[2^{d-2}, 4 \cdot 5^{d-1}] = 5 \cdot 10^{d-2}$ for $d > 3$. There are numbers x with this maximum order (mod 10^d); they must be of the form $x \equiv \pm 3$ (mod 8) so as to have maximum order mod 2^d, and must simultaneously have order 5^{d-1} or $2 \cdot 5^{d-1}$ or $4 \cdot 5^{d-1}$ (mod 5^d). We find that all numbers x with maximum order mod 10^d reside in 32 different residue classes mod 200, represented by $x \equiv \pm$ (3, 11, 13, 19, 21, 27, 29, 37, 53, 59, 61, 67, 69, 77, 83, 91) (mod 200).*

Example: x = 10011 is a suitable multiplier, since $x \equiv +11$ (mod 200).

Procedure: Choose any integer u_0 not divisible by 2 or 5 as a starting value and choose an integer x from one of the above 32 residue classes (mod 200) as a multiplier. Compute $u_{n+1} \equiv xu_n$ (mod 10^d) using fixed point integer arithmetic, but interpret the result as a d-digit decimal fraction.

Notes.

1. This procedure will produce $5 \cdot 10^{d-2}$ terms before repeating.
2. See notes 2, 3, 4 following the procedure for a binary machine, above.

Note on Shuffling.

The problem of "shuffling" the numbers from 1 to k—that is, generating a random permutation of k integers—is rather similar to the problem of generating random numbers, and can be satisfactorily solved by the above techniques. Choose a prime p slightly greater than k, and find a primitive root x of p†; then the sequence ax^n (mod p) shuffles the numbers from 1 to $p - 1$, which include those from 1 to k. Alternatively, for a binary machine one can choose a power of two slightly greater than k, $2^d > k$, choose a multiplier $x \equiv 5$ (mod 8), and compute ax^n (mod 2^{b+2}) for an odd starting value a. The sequence produces 2^b terms before repeating, each of $b + 2$ bits. The low-order two bits will be the same in all numbers produced, so dropping them leaves the 2^b integers in shuffled order.

* Although the calculations are omitted here, it can be seen that all of these numbers are $\equiv \pm 3$ (mod 8), and their order mod 5^d can be checked by applying the binomial theorem. Similarly, it can be verified that any number not included above has a smaller order mod 10^d.

† A table of smallest primitive roots, from which others can be generated as indicated following the above definition of primitive root, is published in NUMBER THEORY by I. M. Vinogradov (Dover Publications, Inc., 1954) for primes up to 4000.

Statistical Considerations

The sole objective of the entire preceding section was to obtain a simple arithmetic process with a long period. It is true that the power residue method was singled out from many others because it is also good statistically, but a discussion of this aspect of the problem has been postponed until now.

The statistical properties desired for the numbers u_n are exactly those that would result if the u_n were obtained by an idealized chance device which selected points from the unit interval independently and with all points $0 \leq x < 1$ equally likely. Clearly, the numbers produced by a computer subroutine are not random in this sense, since they are completely determined by the starting data and since they have limited precision; therefore, various specific properties implied by the above concept of randomness are looked for and output of the subroutine is tested, both theoretically and empirically, for these properties.

1. The values of u should be uniformly distributed over the unit interval regardless of the number of values computed.

   The power residue method has stood this test very well, and seems insensitive to choice of both starting value and multiplier. Many independent tests have been made, dividing (0,1) into subintervals, tallying the number of u_n in each, and applying a chi-square test to confirm the reasonableness of the results.

2. Successive values of u should be independent.

   Interestingly, this apparently impossible objective is in large part realized by the power residue method. For suppose, for a binary machine, that $x > 2^k$, and divide the unit interval into 2^k equal parts: then the knowledge that u_n is in the i'th subinterval gives no information at all as to which subinterval will contain u_{n+1}, since multiplying by x expands each subinterval to at least a full unit interval. If $xy \equiv 1 \pmod{m}$ so that y is the multiplier that obtains u_{n-1} from u_n, and if both $x > 2^k$ and $y > 2^k$, then no information as to location of *either* u_{n+1} or u_{n-1} results from the knowledge that u_n is in a particular subinterval of length 2^{-k}. Similar remarks apply to decimal machines.

   This observation suggests choosing the multiplier x as large as possible while maintaining $y \geq x$, in order to achieve maximum independence for the u_n. A good choice is $x \doteqdot \sqrt{m}$ ($m = 2^b$ or 10^d) because then the left half of each u_n forms a sequence of points that are actually independent in the above sense. Larger values of x exist, still with $y \geq x$, but there does not seem to be an easy way to find them—while still satisfying the preceding number theoretic conditions.

3. Another measure of independence is furnished by an auto-correlation coefficient for the u_n, defined by $c_h = \frac{1}{N}\sum_{n=1}^{N} u_n u_{n+h}$. Statistical theory shows that truly random samples give values of c_h that are approximately normally distributed with mean $= 1/4$ and standard deviation $= .22/\sqrt{n}$, for $h > 0$, and with mean $= 1/3$ and standard deviation $= .30/\sqrt{n}$ for c_0. Some tests of this nature have been made for the power residue method, all giving satisfactory results. M. L. Juncosa [5] has made a similar test, again satisfactory, by dividing (0,1) into k parts and tallying in a k x k matrix: tally 1 in row i, column j, when u_n in the i'th subinterval is followed by u_{n+1} in the j'th. The expected result is equal numbers in all positions of the matrix, and a chi-square test is used to confirm reasonableness of actual results.

4. A study of runs up and down, which describes the oscillatory nature of the magnitudes u_n, appears to be a good test: it shows, for example, that the Fibonacci Series method (see Notes on Other Methods, p. 9) of generating random numbers is

not a good one, and that some multipliers are better than others with the power residue method. This test evidently is not described in detail elsewhere, so will be treated rather completely here.

For N points $u_1, \ldots, u_N$ we write an $N - 1$ bit binary sequence S whose n'th term is 0 if $u_n < u_{n+1}$ and is 1 if $u_n > u_{n+1}$. A subsequence of k zeroes, bracketed by ones at each end, forms a run of zeroes of length k; similarly for runs of ones; and the test involves counting the number of occurrences of runs of different lengths and comparing to expected results. The expected results, based on a truly random sample, are: $(2N - 1)/3$ runs total, $(5N + 1)/12$ runs of length 1, $(11N - 14)/60$ runs of length 2, and in general, $2\{(k^2 + 3k + 1)N - (k^3 + 3k^2 - k - 4)\}/(k + 3)!$ runs of length k for $k < N - 1$; and finally $2/N!$ runs of length $N - 1$. A common characteristic of non-random sets u_n is an excess of long runs. As in note (2) above, one should not use too small a value of x for a multiplier, and y, the reciprocal of x (mod m) also should not be too small, since using y instead of x would give the same sequence but in reverse order—which would show the same pattern of runs. Even beyond this point, some values of x seem to be better than others, but there is no further theory to indicate a good choice.

A convenient way to obtain the formulas for expected number of runs is to first observe that if the u_n (assumed random) are arranged in increasing order, $u_{n_1} < u_{n_2} < \cdots < u_{n_N}$, then all N! permutations of the subscripts are equally likely, since $P(u_r < u_s) = P(u_r > u_s) = 1/2$. Therefore, an equivalent problem is to consider the N! permutations of the first N integers, form the $N - 1$ bit binary sequence S as before for each permutation, and count the number of occurrences of a run of length k in the whole set of S's. The process is lengthy but straightforward. For example, consider $k = 1$: in this instance, four consecutive values of u_n are needed for an "inside" run of length 1, as in $u_n < u_{n+1} > u_{n+2} < u_{n+3}$ which gives $\cdot\cdot 010 \cdot\cdot$ for S, but only three values of u_n are needed for an "end" run, as in $u_1 < u_2 > u_3$ which gives $01 \cdots$ for S and the leading 0 makes the run. The total number of "end" runs of length 1 is $2 \cdot 2 \cdot 2 \cdot \binom{N}{3} (N - 3)!$, where the first 2 indicates two ends to choose, the second 2 corresponds to choice of up or down, the third 2 indicates two ways to make such a pattern with three numbers (abc or cba), the $\binom{N}{3}$ is the number of ways to choose three numbers from N to form the run, and $(N - 3)!$ arranges the rest. Similarly, there are $2 \cdot 5 \cdot \binom{N}{4} (N - 4)! (N - 3)$ "inside" runs, and so $(5N + 1)\, N!/12$ total.

5. Another type of run test—"above and below the mean"—has often been used. From N values of u_n an N-bit binary sequence S is formed with the n'th term 0 if $u_n < 1/2$ or 1 if $u_n > 1/2$. Again runs in S are counted; the expected number of runs of length k is $(N - k + 3)2^{-k-1}$, and expected total number of runs is $(N + 1)/2$. A chi-square test is used to confirm reasonableness of actual results. The power residue method has always passed this test satisfactorily.

6. Applications with k variables will likely assign every k'th random number to the same variable, so it is important to know that such a subset from a random number generator will still be random. With the power residue method, taking every k'th number using x as a multiplier is the same as taking every number while using x^k as a multiplier. The cycle length may still be maximum or may be reduced, depending on x,m, and k. Of course, it will not be less than (1/k) times the maximum. The statistical properties of the subset will also depend x,m, and k; in particular, it should be checked that x^k (mod m) is not too small or too large—cf. notes 2 and 4 above. However, the power residue method usually gives good results when used in this manner; good results can be assured by studying x,m, and k, and at least it apparently gives results as good or better than any other method.

7. Many tests have been applied to the digits of "random" numbers. The frequency test is, usually, a chi-square test applied to observed frequency of occurrence of each digit or of various digit combinations. Sometimes the distribution of many such counts is studied. The poker test is a special frequency test for certain 5 digit (or n digit) combinations. The gap test is a chi-square test applied to observed frequency of distances separating two like digits.

   The power residue method has always passed these tests satisfactorily. However, the tests on digits seem to be less critical, for purpose of comparing methods, than some of the tests on successive numbers—e.g., runs up and down—and anyway most applications call for numbers rather than digits. Therefore, the earlier paragraphs of this section are emphasized.

Notes on Other Methods

The center square method, one of the earliest, squares a 2n digit number and takes the middle 2n digits from the 4n digit product for the next number. There are several variations, but the method has generally been discarded because it does not give very long periods, and in fact, the period is only found by trial for each starting number. The sequence generally does not return to the starting data to repeat. Also, it is relatively slow.

The Fibonacci Series method assumes two starting numbers u_o, u_1, and computes $u_{n+1} \equiv u_n + u_{n-1}$ (mod m); it is convenient to choose $m = 2^b$ or 10^d. The method is fast, easy, and the period is satisfactorily long ($3 \cdot 2^{b-1}$ for $m = 2^b$, $3 \cdot 10^{d-1}$ or $15 \cdot 10^{d-1}$ for $m = 10^d$, depending on starting values—which should not both have a common factor with m), but the results are not so good statistically: runs up and down, in particular, show lack of randomness. The method is essentially a power residue method with a multiplier $x = (1 + \sqrt{5})/2$, which is too small.

A great improvement results by taking only every k'th number, where k is chosen to make $[(1 + \sqrt{5})/2]^k$ satisfactorily large. With this modification, the Fibonacci Series method is as good as any other statistically, but for most computers is not as fast or easy to use as the power residue method.

Numerous other methods have been devised, some of which will be found in references listed at the end of this write-up. Most are slower and more cumbersome, but none has been so extensively studied statistically and so thoroughly tested as the power residue method.

Appendix: Programming the Power Residue Method

This section is included for those persons unfamiliar with the mathematical concepts employed in the previous discussion. It outlines the series of steps required in programming the procedure for either a binary or a decimal computer.

Procedure for a binary computer

In the following discussion the computer is assumed to have a word size of b-bits. Furthermore, the arithmetic assumes the binary point to be at the extreme right of the word. Thus, all of the numbers involved are integers—including the random numbers generated. Once the result is available, however, the programmer should consider the binary point to be at the extreme left in order to obtain random numbers distributed over the unit interval. The following procedure will produce 2^{b-2} terms before repeating.

1. Choose for a starting value any odd integer u_o.
2. Choose as a constant multiplier an integer x of the form $x = 8t \pm 3$ where t is any integer. (A value of x close to $2^{b/2}$ is a good choice.)
3. Compute xu_o. This produces a product 2b-bits long; the high-order b-bits are discarded and the b low-order bits are the value u_1.
4. Each successive random number u_{n+1} is obtained from the low-order bits of the product xu_n.

Example: For simplicity, assume that $b = 4$. The procedure will produce only 4 terms before repeating but will illustrate the principle.

1. Choose $u_o = 1001$.
2. The choice $t = 1$ gives either 1011 or 0101 for x. Since $2^{b/2} = 4$, the value 0101 is selected for a multiplier.
3. $xu_o = (0101)(1001) = 00101101$. Therefore, $u_1 = 1101$.
4. $xu_1 = (0101)(1101) = 01000001$. Therefore, $u_2 = 0001$.
5. $xu_2 = (0101)(0001) = 00000101$. Therefore, $u_3 = 0101$.
6. $xu_3 = (0101)(0101) = 00011001$. Therefore, $u_4 = 1001 = u_o$ and the sequence repeats.

Procedure for a decimal computer

In the following discussion the computer is assumed to have a word size of d-digits. Furthermore, the arithmetic assumes the decimal point to be at the extreme right of the word. Thus, all of the numbers involved are integers—including the random numbers generated. Once the result is obtained, however, the programmer should consider the decimal point to be at the extreme left in order to obtain random numbers distributed over the unit interval. The following procedure will produce $5 \cdot 10^{d-2}$ terms before repeating (for d greater than 3).

1. Choose for a starting value any integer u_o not divisible by 2 or 5.
2. Choose as a constant multiplier an integer x of the form $x = 200t \pm r$

   where t is any integer and r is any of the values 3, 11, 13, 19, 21, 27, 29, 37, 53, 59, 61, 67, 69, 77, 83, 91. (A value close to $10^{d/2}$ is a good choice.)
3. Compute xu_o. This produces a product 2d-digits long; the high-order d-digits are discarded, and the d low-order digits are the value u_1.
4. Each successive random number u_{n+1} is obtained from the low-order digits of the product xu_n.

Example: For simplicity it will be assumed that $d = 4$. The procedure will, therefore, produce 500 terms before repeating.

1. Choose $u_0 = 2357$.

2. Since $10^{d/2} = 100$, a good choice of x would be either $x = (200)(0) + 91 = 91$ or $x = (200)(1) - 91 = 109$. The value 109 will be chosen for this illustration.

3. $xu_0 = (0109)(2357) = 00256913$. Therefore, $u_1 = 6913$.

4. $xu_1 = (0109)(6913) = 00753517$. Therefore, $u_2 = 3517$.

5. $xu_2 = (0109)(3517) = 00383353$. Therefore, $u_3 = 3353$.

6. $xu_3 = (0109)(3353) = 00365477$. Therefore, $u_4 = 5477$.

7. $xu_4 = (0109)(5477) = 00596993$. Therefore, $u_5 = 6993$, etc.

Notes on the procedures

1. The number of terms which can be obtained before the sequence repeats is extremely large. The 35-bit word length of the 704, 709, and 7090 makes it possible to generate a sequence of over 8.5 billion numbers. The ten-digit word length of the 650 and 7070 allows for a sequence of 500,000,000 terms.

2. In choosing the value for the constant multiplier, x, the timing characteristics of the computer being used should be considered. Multiplication time for the 709 and 7090 generally decreases as the number of l's in the multiplier decreases. For the 650 and 7070, the sum of the digits in the multiplier is a factor in timing. With some values of x a speed advantage can be obtained by using shift and add instructions instead of the multiply instruction.

3. The low-order digits of the numbers referred to here as random are far from random as evidenced by the two numerical examples. Actually, the periodicity of the digits in any particular digit position of the random number sequence increases as the order of the digit position increases. Therefore, if a random number smaller than a full word is required, the high-order digits of u_n should be used. The actual computation of the u_n, however, should be done using the full word length (or in the case of a variable word length machine such as the 705, with a sufficiently large word).

References

1. A MILLION RANDOM DIGITS: The Rand Corporation.

2. PROCEEDINGS OF A SECOND SYMPOSIUM ON LARGE SCALE DIGITAL CALCULATING MACHINERY: The Annals of the Computation Laboratory of Harvard University, Volume XXVI, 1951.

3. MONTE CARLO METHOD: U. S. Department of Commerce, National Bureau of Standards, Applied Mathematics Series. 12, 1951.

4. SYMPOSIUM ON MONTE CARLO METHODS: H. A. Meyer, Editor, John Wiley & Sons, Inc., 1956.

5. RANDOM NUMBER GENERATION ON THE BRL HIGH-SPEED COMPUTING MACHINES: M. L. Juncosa, BRL Report No. 855, 1953.

6. DECISION UNIT MODELS AND SIMULATION OF THE UNITED STATES ECONOMY; section on Random Number Generation: Martin Greenberger, Massachusetts Institute of Technology.

7. RANDOM DIGIT GENERATION; A. O. Arthur, Computing News, September, 1956.

8. MONTE CARLO METHODS: Chapter 12 of MODERN MATHEMATICS FOR THE ENGINEER, University of California, Engineering Extention Series, E. F. Beckenbach, Editor, McGraw-Hill Book Company, Inc., 1956.

9. THE MONTE CARLO METHOD: W. F. Bauer, INDUSTRIAL AND APPLIED MATHEMATICS, Volume 6, Number 4, December, 1958.

10. INTRODUCTION TO MATHEMATICAL STATISTICS: P. G. Hoel, John Wiley & Sons, Inc., 1956.

11. ELEMENTARY NUMBER THEORY: Uspensky and Heaslet, McGraw-Hill Book Company, Inc.

C20-8011

Printed in U.S.A. C20-8011

IBM®

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N.Y. 10601
[USA Only]

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
[International]