Behavior Research Methods, Instruments, & Computers
1991, 23 (1) 45-60

— COMPUTER TECHNOLOGY —

Algorithms for randomness in the behavioral sciences: A tutorial

MARC BRYSBAERT
University of Leuven, Leuven, Belgium

Simulations and experiments frequently demand the generation of random numbers that have specific distributions. This article describes which distributions should be used for the most common problems and gives algorithms to generate the numbers. It is also shown that a commonly used permutation algorithm (Nilsson, 1978) is deficient.

Numerous studies in the behavioral sciences make use of "randomness": Subjects must be randomized over conditions, stimuli are to be presented in an unpredictable sequence, simulations involve an unsystematic component, or events must take place at random time intervals. Unfortunately, "randomness" is not an unambiguous concept. There are several "types" of randomness, each of which is appropriate only under well-specified conditions. This article consists of an attempt to give an idea of the most common types of randomness and the situations in which they are to be applied. It is intended as a practical guide for researchers, with mathematical proofs and justifications omitted as much as possible. This may make the text unsatisfactory for mathematically oriented scientists, but it should make it readable for everyone who wants a short review and a brief answer to problems that can occur in the actual use and generation of random numbers. Emphasis has been placed on problems encountered in experimental psychology. Researchers interested in simulations will find a groundwork in the text, but they may additionally wish to consult more specialized texts such as Kennedy and Gentle (1980), Knuth (1981), or Ripley (1987).

GENERATING RANDOM NUMBERS

Although in a strict sense a sequence of random numbers can only be obtained from a truly random phenomenon, practical limitations have led to the almost exclusive use of pseudorandom number generators in science. These are mathematical functions that are essentially deterministic, but ones that mimic the properties of a sequence of independent uniformly distributed random variables (i.e., variables of which each value has the same probability of occurring). These sequences can further be translated into samples from other distributions (e.g., from the standard normal distribution; see below). First, we will consider the generation of uniformly distributed random numbers.

The author wishes to thank G. d'Ydewalle, as well as the editor and three reviewers, for helpful comments on earlier drafts of the manuscript. Correspondence should be addressed to Marc Brysbaert, University of Leuven, B-3000 Leuven, Belgium.

Uniform Distribution

By far the most successful pseudorandom number generators known today are special cases of the (linear) congruential method first proposed by Lehmer (Knuth, 1981; Ripley, 1987). A sequence of random numbers is generated with the use of the following equation:

$$X_{n+1} = (aX_n + c) \bmod m \qquad (1)$$

in which

a, c = positive constants, $a > 0$, $c \geq 0$,
m = the modulus, $m > a$, $m > X_0$, $m > c$,
X_0 = the starting value or seed,
X_n = the nth value of the sequence, and
mod = the modulus operator (returns the remainder of the division of two integer operands, e.g., 10 mod 3 = 1, because $3*3+\underline{1} = 10$; likewise, 9 mod 3 = 0, and 11 mod 3 = 2).

In most cases, the numbers generated by Equation 1 are further divided by the modulus m, in order to obtain a (uniform) distribution of real numbers between 0 and 1. However, not all values of a, c, and m yield a good random generator. There is a large literature about which values to use (see Nance & Overstreet, 1972; Sahai, 1979; Sowey, 1972, 1978, 1986, for bibliographies). Table 1 gives some of the best values that have been suggested hitherto, and references to where more information may be found.

As shown in Table 1, the modulus of the generators is quite large. This is because all congruential random number generators ultimately get into a loop, producing a sequence of numbers that is repeated endlessly. The length of the repeating sequence is called the period of the generator;

Table 1
Some Good Random Number Generators Given in the Literature

X_0	m	a	c	Source	Period
odd	2^{59}	13^{13}	0	Ripley (1983)	2^{57}
odd	2^{48}	44485709377909	0	Ripley (1983)	2^{46}
	2^{35}	5^{15}	odd	Knuth (1981)	2^{35}
	2^{32}	69069	odd	Ripley (1983)	2^{32}
	2^{32}	100485	1	Atkinson (1980)	2^{32}
	2^{32}	1589013525	odd	Ripley (1983)	2^{32}
odd	2^{32}	663608941	0	Dudewicz and Ralley (1981)	2^{30}
$<>0$	$2^{31}-1$	630360016	0	Dudewicz and Ralley (1981)	$2^{31}-2$
$<>0$	$2^{31}-1$	764261123	0	Dudewicz and Ralley (1981)	$2^{31}-2$
$<>0$	$2^{31}-1$	950706376	0	Fishman and Moore (1986)	$2^{31}-2$

it is always less than or equal to the modulus m. A generator with modulus 8 thus yields a period of 8 different numbers at most. For instance, a generator with $a = c = 5$ and $m = 8$ always gives the sequence ..., 1, 2, 7, 0, 5, 6, 3, 4, 1, 2, 7, ..., irrespective of the starting value (the only difference that the starting value makes consists of where in the period the process is started). The importance of the values of a and c can be illustrated if we take, for instance, $a = c = 2$, $m = 9$. This leads not only to a period shorter than the maximum period (i.e., 9), but also to a period that depends on the starting value, as can be seen below

$$X_0 = 0 \rightarrow 0, 2, 6, 5, 3, 8, 0, \ldots$$
$$X_0 = 1 \rightarrow 1, 4, 1, 4, \ldots$$
$$X_0 = 2 \rightarrow 2, 6, 5, 3, 8, 0, 2, \ldots$$
$$X_0 = 3 \rightarrow 3, 8, 0, 2, 6, 5, 3, \ldots$$
$$X_0 = 4 \rightarrow 1, 4, 1, 4, \ldots$$
$$X_0 = 5 \rightarrow 5, 3, 8, 0, 2, 6, 5, \ldots$$
$$X_0 = 6 \rightarrow 6, 5, 3, 8, 0, 2, 6, \ldots$$
$$X_0 = 7 \rightarrow 7, 7, 7, \ldots$$
$$X_0 = 8 \rightarrow 8, 0, 2, 6, 5, 3, 8, \ldots$$

All the generators in Table 1 produce satisfactory sequences of random numbers if the requirements with respect to the starting value are met (in some cases this value must be odd or different from zero). One problem, however, seriously limits their use in everyday scientific life. Because round-off errors must not occur, the algorithms are difficult to implement on the 16-bit microprocessors frequently used in psychological laboratories. This problem for a long time seemed unsolvable, because of the need for a sufficiently large modulus, but in 1982 Wichmann and Hill presented a rather simple solution. They showed that adding several congruential generators and taking the fractional part led to a new congruential generator with a much larger modulus and much better statistical properties. More specifically, they used the following three multiplicative (i.e., $c = 0$) generators:

$$\begin{array}{lll} \text{generator 1:} & a = 171 & m = 30269, \\ \text{generator 2:} & a = 172 & m = 30307, \\ \text{generator 3:} & a = 170 & m = 30323. \end{array} \quad (2)$$

The composite generator is equivalent to a simple multiplicative congruential generator with a = 1655 54252 64690 and m = 2781 71856 04309 (Zeisel, 1986), and it has an estimated period length of $6.95 * 10^{12}$ (Wichmann & Hill, 1984). In addition, Wichmann and Hill (1982) provided an implementation of the algorithm that requires arithmetic only up to 30323 and therefore is easy to implement on a 16-bit microprocessor (the maximum value of a 16-bit integer is 32767). Wichmann and Hill's (1982) implementation is the following:

Algorithm 1
Wichmann and Hill's (1982) Random Number Generator

```
1. Define 3 starting values (one for each subgenerator)
     set seed1 = 0 < seed1 < 30269,
     set seed2 = 0 < seed2 < 30307,
     set seed3 = 0 < seed3 < 30323,
2. Calculate random number and redefine 3 starting values
     set seed1 =
       171 * (seed1 mod 177) - 2 * (seed1 div 177)[1]
     set seed2 =
       172 * (seed2 mod 176) - 35 * (seed2 div 176)
     set seed3 =
       170 * (seed3 mod 178) - 63 * (seed3 div 178)
     if seed1 < 0 then set seed1 = seed1 + 30269
     if seed2 < 0 then set seed2 = seed2 + 30307
     if seed3 < 0 then set seed3 = seed3 + 30323
     set random number =
       fractional part of (seed1/30269 + seed2/30307
                                       + seed3/30323)
3. Optional, to check for rounding-off errors (McLeod, 1985)
     if random number ≤ 0
       then set random number = 1E-30
     if random number ≥ 1
       then set random number = 0.9999999999
4. Return random number
```

The generator needs three seeds to start (see Part 1 of the algorithm). All seeds must be larger than 0 and smaller than their modulus. They only need to be defined before the first random number is calculated, because they are updated every time a new number is produced. Taking the same seeds leads to the same sequence of random numbers, which may be appropriate in simulations to test the effect of a small variation in one of the parameters, but which usually is not necessary. "Random" seeds can be obtained by using either the time-of-day clock information or the random number generator of the machine (i.e., the routines RANDOMIZE and RND in Microsoft BASIC). However, the best way to guarantee that two subsequent sequences are independent is to take the last values of the first series as the seeds of the second series. "Random" seeds involve the (small) risk that the generator starts somewhere in the sequence of the previous series and thus produces two related strings of random numbers. In the worst case, all numbers in the second series match the sequence of numbers in the first series.

McLeod (1985) pointed to the possibility that round-off errors in some systems may yield random numbers equal to 0 or 1. In the remainder of this article, random numbers are always assumed to be larger than 0 and smaller than 1, and therefore it may be advisable to include Part 3 of the algorithm. Otherwise, problems may arise. For instance, in the randomization procedure (see below), there will be a range error if the random number equals 1, and in many of the algorithms for standard normal distributions, to take the logarithm will be impossible if the random number equals 0.

Wichmann and Hill's (1982) random number generator has been tested several times (Wichmann & Hill, 1982; MacLaren, 1989; see also below), and it produces a very satisfactory output. Therefore, its use is strongly recommended. An additional advantage is that it can easily be reproduced in different laboratories, because the algorithm yields the same sequences of numbers on different systems (at least if the starting values are known).

The only disadvantage of the algorithm is that it is rather slow. On our system (an IBM AT 286 clone running at 8.9 MHz according to the Landmark CPU speed test), with Turbo Pascal 4.0 software (Borland), it takes 1.38 msec to generate one random number. This is 9.2 times slower than the built-in random number generator (0.15 msec per number). However, although things may have improved for recent versions of languages, one should be skeptical about the performance of the built-in generators (see, e.g., Afflerbach, 1985, on Commodore and Apple; Aldridge, 1987, on the Apple II; Edgell, 1979, on the DECsystem-10; Lordahl, 1988, on IBM; Modianos, Scott, & Cornwell, 1987, on several PCs; Strube, 1983, on the Commodore VIC-20). Therefore, if "true" randomness is essential, researchers should at least do some empirical tests on the appropriateness of their system (see below) if Wichmann and Hill's (1982) generator is not to be used. Researchers should also check to see that the built-in algorithm is reseeded every time a new sequence of numbers is desired. Microsoft BASIC and Turbo Pascal, for instance, always return the same sequence if they are not reseeded with the RANDOMIZE statement.

Of course, the choice of random number generators to a large extent depends on what is investigated. For some applications, the most important requirement is that all values have the same probability of occurring, a requirement that most built-in generators meet. For instance, if one wants to estimate the probability that $x^2 < y$ (x and y being uniformly distributed random variables between 0 and 1), a built-in generator will create approximately the same results as will Wichmann and Hill's (1982) generator. The only difference will be that the latter takes more time (for those who are interested, the exact value of $P(x^2 < y) = 2/3$; 100,000 trials with Algorithm 1 gave an estimate of 0.6647, and 100,000 trials with the IBM built-in generator yielded 0.6641). If, however, the independence of the numbers in a sequence is predominant, most built-in generators will fail (see, e.g., Lordahl, 1988).

Since many processes do not have a uniform (rectangular) distribution, the random numbers generated by Algorithm 1 are only occasionally useful without transformation. Additional algorithms are needed to transform the numbers into samples from other distributions. This article only deals with two of these distributions, the standard normal and the standard exponential. The normal distribution is considered because of its importance in many simulations, the exponential distribution because we need it for randomness in time. Algorithms for other distributions (Student's t, F, chi-square) can be found in Kennedy and Gentle (1980) or Ripley (1987), or in preprogrammed statistical simulation packages such as DATASIM (Bradley, 1988; Bradley, Senko, & Stewart, 1990). It should also be noted that it is possible to convert uniformly distributed random numbers into random numbers from any distribution by using the simple fact that the cumulative density function (cdf) of any distribution is uniform between 0 and 1. All that is necessary is to generate a uniformly distributed random variable (which denotes a point on the cdf) and take the inverse of the cdf function whose distribution you desire to sample from. Examples are given below for the standard normal and the standard exponential distribution, but the rule can be extended to any distribution.

Normal Distributions

There are numerous ways to convert a uniform distribution of numbers between 0 and 1 to a standard normal distribution (mean equal to 0 and variance equal to 1). Five of them will be discussed here. They are chosen because they are reasonably fast and accurate, and they require but a small amount of memory.

As indicated above, a first way to convert a random number generated by Algorithm 1 into a standard normal deviate is to consider each number as a value of the cdf of the standard normal. We all know that a z value of -1.96 corresponds to a cdf value of 0.025 and a z value of 1.96 to a cdf value of 0.975, because we have all used it to calculate (two-tailed) statistical significance. Thus, what we need is an algorithm that converts cdf numbers such as 0.025 or 0.975 into their corresponding z values of -1.96 and $+1.96$. Brophy (1985) compares several of these algorithms, one of which (Hill & Davis's) will be used in Algorithm 2. This algorithm has been chosen because it is quite accurate (maximum absolute error of 0.00035) and relatively fast (see below). Other algorithms may be preferred if either speed or accuracy is to predominate (see Brophy, 1985, for these algorithms). In the following algorithms, Z denotes a random variate from the standard normal distribution.

Algorithm 2
Standard Normal via Inverse Function

```
generate $U_1$ = random number
if $U_1 > 0.5$ then set $U_2 = 1 - U_1$ else set $U_2 = U_1$
if $U_2 < 1E - 20$ then set $Z = 10$
else se-
t       $A = sqrt(-2*ln(U_2))$
        $Z = A-((7.45551*A+450.636)*A+1271.059)/$
            $(((A+110.4212)*A+750.365)*A+500.756)$
if $U_1 > 0.5$ then set $Z = -Z$
return Z
```

A second method for the normal distribution owes to Box and Muller (1958). The underlying rationale is rather simple (see, e.g., Ripley, 1987, p. 54), but, for the sake of brevity, it will not be explained here. The algorithm is the following:

Algorithm 3
Standard Normal According to Box-Muller

```
generate $U_1$ = random number, set $A = 2\pi U_1$
generate $U_2$ = random number
set $B = -ln(U_2)$, $C = sqrt(2B)$
return $Z_1 = C*cos(A)$, $Z_2 = C*sin(A)$
```

Algorithm 3 produces two independent standard normal deviates, Z_1 and Z_2, at least if the random number generator is good. If the generator is not good, the (Z_1,Z_2) pairs are likely to be situated on a limited number of circles or radii (see the cautionary tale in Ripley, 1987, pp. 55-59). We have plotted several hundreds of thousands of these (Z_1,Z_2) pairs based on Algorithm 1 to check whether they are indeed dispersed throughout the whole plane. Algorithm 1 passed the test very well.

A third method to generate standard normal deviates is Marsaglia's (1962) polar method, a modification of the Box-Muller algorithm. It avoids evaluation of sines and cosines.

Algorithm 4
Standard Normal via the Polar Method

```
repeat
  generate $U_1$ = random number, set $V_1 = 2*U_1 - 1$
  generate $U_2$ = random number, set $V_2 = 2*U_2 - 1$
until $W = V_1^2 + V_2^2 < 1$
set $A = sqrt(-2*ln(W)/W)$
return $Z_1 = AV_1$, $Z_2 = AV_2$
```

Marsaglia and Bray (1964) published a modification of the polar method that is slightly more complicated but faster. Speed is acquired by introducing simple auxiliary functions that can be assessed most of the time, and by restricting the time-consuming polar algorithm to fill in the gaps between the theoretical distribution and the approximation.

Algorithm 5
Standard Normal According to Marsaglia-Bray

```
generate U = random number
if $U < 0.8638$ then
  generate $U_1,U_2,U_3$ = random numbers
  set $Z = 2(U_1 + U_2 + U_3) - 3$
else if $U < 0.9745$ then
  generate $U_1,U_2$ = random numbers
  set $Z = 1.5(U_1 + U_2 - 1)$
else if $U < 0.9973002039$
  repeat
    generate $U_1$ = random number
    set $V = 6*U_1 - 3$
    generate $U_2$ = random number
  until $0.358*U_2 \leq g(V)$ *
  set Z = V
else
  repeat
    repeat
      generate $U_1,U_2$ = random numbers
      set $V_1 = 2*U_1 - 1$, $V_2 = 2*U_2 - 1$
    until $W = V_1^2 + V_2^2 < 1$
    set $A = sqrt((9-2*ln(W))/W)$
    set $B = AV_1$, set $C = AV_2$
  until $|B| > 3$ or $|C| > 3$
  if $|B| > 3$ then Z = B else Z = C
endif
return Z
```

$$*g(v) = \begin{cases} ae^{-v^2/2} - 2b(3-v^2) - c(1.5-|v|), & |v| < 1 \\ ae^{-v^2/2} - b(3-|v|)^2 - c(1.5-|v|), & 1 \leq |v| < 1.5 \\ ae^{-v^2/2} - b(3-|v|)^2, & 1.5 \leq |v| < 3 \end{cases}$$

$a = 17.49731196$, $b = 2.36785163$, $c = 2.15787544$

A final algorithm we will include is the ratio of uniforms (Best, 1979; Knuth, 1981, pp. 125-127; Ripley, 1987, p. 82).

Algorithm 6
Standard Normal via Ratio of Uniforms

```
1. generate $U_1,U_2$ = random number
   set $V = 0.8578*(2*U_2 - 1)$
   set $Z = V/U_1$
   set $A = 0.25*Z^2$
2. if $A < 1 - U_1$ then go to 3
   if $A > 0.259/U_1+0.35$ then go to 1
   if $A > -ln(U_1)$ then go to 1
3. return Z
```

An algorithm is good if it is reasonably fast and accurate in the tails, and if it returns deviates with a cdf value close to the expected value. Table 2 gives these results for the five algorithms presented above. The first three columns give the tail probabilities at the low end of the distribution, the next three at the high end. The seventh column tabulates the maximal absolute difference between the obtained and the expected cdf, and the eighth column returns the average time needed for the generation of one deviate.[2] All algorithms used random numbers generated with the use of Algorithm 1. Because the generation of these random numbers is relatively slow, the average number of uniform deviates needed for the calculation of a normal deviate will be a considerable factor in the speed of an algorithm; this value is therefore given in the next to

Table 2
Performance of Algorithms 2–6 for the Generation of Normal Deviates; Estimates Based on 100,000 Trials

	Tails								Mean Number of Random Numbers Required	Mean Number of Time-Consuming Functions
	Lower			Upper			Maximum	Mean Time/		
Algorithm	.00050	.00500	.02500	.02500	.00500	.00050	$\lvert cdf_o - cdf_t \rvert$*	Deviate†	per Deviate	per Deviate
Inverse	.00051	.00490	.02532	.02437	.00498	.00055	.00256	9.37	1.00	2.00
Box-Muller	.00042	.00500	.02563	.02519	.00521	.00069	.00325	6.22‡	2.00	3.00
Polar	.00039	.00504	.02533	.02481	.00485	.00047	.00351	5.65‡	2.54	2.00
Marsaglia-Bray	.00049	.00509	.02583	.02561	.00511	.00045	.00229	6.51	3.93	0.06
Ratio of uniforms	.00053	.00516	.02535	.02551	.00540	.00050	.00372	7.25	2.73	0.52

*Maximum absolute difference between obtained and expected cdf. Normal cumulative density function calculated using equation 26.2.17 of Zelen and Severo (1964), which has an error $< 7.5*10^{-8}$. †Time in milliseconds. Estimated with an IBM AT 286 clone running at 8.9 MHz according to the Landmark CPU Speed Test. Turbo Pascal 4.0 (Borland) software. ‡Algorithm returns two standard normal deviates.

the last column of Table 2. Another important aspect of the speed is the average number of time-consuming operations (logarithms, exponentials, square roots, sines and cosines) that need to be evaluated. This figure is presented in the last column of Table 2. All estimates are based on 100,000 trials.

The accuracy in the tails and the maximum absolute deviation of observed and expected cdf values are good and comparable for all five algorithms. Only the time needed to evaluate a standard normal deviate differs and ranges from 5.65 msec for the polar method to 9.37 msec for the inverse cdf method (for the system and the language used). Because the Marsaglia-Bray method requires the most random numbers (i.e., 3.9), the results are more in favor of it if random number generation is fast. One way of speeding it up might be to evaluate the random number in the first step with the built-in generator. The major requirement of this number is that it be uniformly distributed, as is the case for most built-in generators (e.g., in our system, estimates based on 1,000,000 trials yielded an error smaller than $1.4 * 10^{-4}$ between the observed probabilities and the probabilities required for the Marsaglia-Bray algorithm). The time needed for the generation of one standard normal deviate then drops from 6.51 to 5.34 msec.

Exponential Distributions

Just like normal deviates, exponential deviates can be generated by using the inverse cdf function. The cdf of an exponential distribution is $F(x) = 1 - e^{-\lambda x}$, and the inverse is $F^{-1}(U) = -\ln(1-U)/\lambda$. If $\lambda = 1$, we have the standard exponential. U is a uniformly distributed random number between 0 and 1, so that, for programming purposes, it makes no difference whether we take $1-U$ or U. This gives us the following algorithm (E denotes a random variate from the standard exponential distribution):

Algorithm 7
Standard Exponential via Inverse Function

```
generate U = uniformly distributed random number
set E = −ln(U)
return E
```

A second way to generate exponential deviates is to split the range of E up into intervals. More specifically, the exponential distribution is considered as the compound of a geometric and a new exponential distribution with pdf $e^{-x}/(1-e^{-1})$. The following algorithm owes to von Neumann (1951). Its advantage is that it avoids the explicit use of the logarithm function (which is rather time-consuming).

Algorithm 8
Standard Exponential According to von Neumann

```
   let I = 0
1. generate $U_1$ = random number
   set A = $U_1$
2. generate $U_2$ = random number
   if $U_1 \leq U_2$ then return E = I + A
3. generate $U_3$ = random number
   if $U_3 \leq U_2$ then go to 2
4. set I = I + 1
   goto 1.
```

The last algorithm that we will discuss for generating exponential deviates makes use of the ratio-of-uniforms method. The fourth, the fifth, and the sixth steps are optional pretests to avoid calculation of the logarithm in step seven. More information on the algorithm is to be found in Ripley (1987, pp. 69-71; note, however, the mistake in the algorithm outline on p. 71).

Algorithm 9
Standard Exponential via Ratio of Uniforms

```
1. generate $U_1, U_2$ = random numbers
   set V = 2/e * $U_2$
   set E = V/$U_1$
   if $E/2 \leq (1+\ln(a) - a*U_1)$ then go to 2
   if $E/2 > b_1/U_1 - (1+\ln(b_1))$ then go to 1
   if $E/2 > b_2/U_1 - (1+\ln(b_2))$ then go to 1
   if $E/2 > -\ln(U_1)$ then go to 1
2. return E
   a = 1.6487 $b_1$ = 0.105 $b_2$ = 0.773
```

Accuracy in the tails, maximum absolute deviation between observed and expected cdf values, and speed of the

Table 3
Performance of Algorithms 7–9 for the Generation of Exponential Deviates; Estimates Based on 100,000 Trials

	Tails									
	Lower			Upper						
Algorithm	.00050	.00500	.02500	.02500	.00500	.00050	Maximum $\lvert cdf_o - cdf_t \rvert$*	Mean Time/ Deviate†	Mean Number of Random Numbers Required per Deviate	Mean Number of Time-Consuming Functions per Deviate
Inverse	.00046	.00484	.02450	.02418	.00483	.00055	.00353	5.80	1.00	1.00
Neumann	.00057	.00470	.02483	.02518	.00461	.00052	.00263	6.59	4.29	0.00
Ratio of uniforms	.00049	.00521	.02501	.02404	.00456	.00046	.00272	8.71	2.95	0.23

*Maximum absolute difference between obtained and expected cdf. †Time in milliseconds. Estimated with an IBM AT 286 clone running at 8.9 MHz according to the Landmark CPU Speed Test. Turbo Pascal 4.0 (Borland) software.

three algorithms are listed in Table 3. Again, accuracies are similar, but this time the inverse cdf function (Algorithm 7) is fastest, at least if uniformly distributed random numbers are generated with the use of Algorithm 1. The average sum of random numbers needed to generate an exponential deviate shows that von Neumann's (1951) method will be superior if random number generation is faster (e.g., with the built-in generator, von Neumann's algorithm only takes 1.32 msec, against 4.35 for the inverse, and 5.06 for the ratio of uniforms).

TESTING RANDOM NUMBER GENERATORS

Empirical tests of random number generators can be divided into two broad categories. The first category consists of tests to examine whether the distribution of generated numbers corresponds to the theoretical distribution. This can be done with either a chi-square goodness-of-fit test or the Kolmogorov-Smirnov (K-S) test. The latter test is more powerful when continuous functions are involved, but care should be taken with respect to which source is consulted. Many textbooks in the behavioral sciences do not provide a correct description of the K-S test (Kraner, Mohanty, & Lyons, 1980). The obtained value of the fit statistic (chi-square as well as K-S) should be close to the expected value and not to zero, since small values indicate that the sequence fits the distribution too well. For instance, if we want to check whether Wichmann and Hill's (1982) random number generator produces an equal number of digits between 0 and 9 when multiplied by 10 and truncated, we should find a chi-square value around 9—that is, the number of degrees of freedom of the frequency test (100 chi-square tests based on 1,000 numbers each yielded an average value of 8.87, which is indeed close to the expected value). Similarly, a frequency test of the numbers 0–99 after multiplication by 100 and truncation should have an expected value of 99 (100 chi-square tests based on 1,000 numbers generated by Algorithm 1 yielded an average value of 97.90). The test can be made more precise, because not only should the average value of the fit statistic be close to the expected value, but also the distribution of obtained values should coincide with the theoretical distribution (see, e.g., Dudewicz & Ralley, 1981). Thus, the distribution of 100 chi-square values based on the frequency test of the digits 0–9 (see above) should correspond to a chi-square distribution with 9 degrees of freedom. This again can be examined with the use of a K-S test or a chi-square goodness-of-fit test.

The second category consists of empirical tests to investigate whether the numbers in a sequence are well spread. It is not enough for Algorithm 1 to generate a uniform distribution of numbers. The numbers of the sequence must also be independent, and this is where most generators fail (see above). There is virtually an infinite number of tests that can be conceived to measure this property quantitatively. Chi-square and correlation tests can be used to test whether subsequent numbers are unrelated, although attention should be paid to the fact that, for some tests, the data are not independent and therefore a modified version is required (see Knuth, 1981, and Ripley, 1987, for more information). Independence can also be examined by making predictions from the (assumed) independence and checking whether the data do indeed conform to these predictions. For instance, if digits between 0 and 9 are generated, we can tabulate the frequency of the interval lengths between two identical digits and compare the obtained frequencies with the expected ones. This test is known as the gap test. We can also look at the monotone increasing and decreasing subsequences and see whether their frequencies conform to the expected probabilities (a test known as the runs test). Or we can consider the lengths of sequences needed to "collect" all digits (coupon collectors' test), look at the number of matching digits in subsequences of four (the poker test), or calculate the frequency of the middle digit's being the maximum in a chain of three (the maximum test), and so on.

More information about such empirical tests can be found in Kennedy and Gentle (1980), Knuth (1981), and Ripley (1987), mentioned in the introduction, or in Gruenberger and Jaffray (1965; but see below). The tests are not discussed at full length here, because all generators of Table 1 and Algorithm 1 are known to pass them successfully. For the same reason, theoretical tests that can be applied on random number generators (Atkinson, 1980;

Knuth, 1981; MacLaren, 1989; Ripley, 1987) are omitted in this article.

THE USE OF RANDOM NUMBERS

To know how to generate random numbers is important, but it is only a first step. Once we have the source of randomness, we need to apply it correctly. In the remainder of the article, three selected topics will be discussed: randomization of stimuli and subjects, random sampling, and randomness in time. These topics have been chosen because of their importance in experimental psychology, and because they tend to be neglected in more general texts dealing with simulations.

Randomization of Stimuli and Subjects

Strings of independent random numbers are not very interesting for randomizing subjects or stimuli, because they may lead to serious imbalances. Suppose an experimenter needs a string of 100 binary digits (zero and one) in order to determine whether a subject is included in Condition A or Condition B. Generating such a string with a random number generator involves a risk of at least 0.27 that more than 55 of the 100 subjects are included in one condition, and fewer than 45 in the other. To avoid these imbalances in randomization, sampling without replacement is a better technique. This is achieved by first listing all alternatives and then making a random permutation of the list.

Before presenting a good permutation algorithm, however, we would like to give a cautionary tale. For years we used an algorithm (Nilsson, 1978) in our laboratory that at first sight seemed very sound and that we actually were going to defend in this article. Yet the algorithm failed on the first and most basic test that was applied to it in our analysis for the present article. Nilsson's (1978) algorithm is the following. First, all alternatives are listed in an array with *N* elements (e.g., for the example above, an array of 50 0s and 50 1s would be created). Then, a random permutation of the array is made by applying the following algorithm:

```
set I = 0
repeat
  set I = I + 1
  set U = integer random number ranging from 1 to N
  set A = X[I] (i.e., the ith element from the array)
  set X[I] = X[U]
  set X[U] = A
until I = N
```

Each element *I* of the array can be exchanged with all possible alternatives, which leads to a total sum of imaginable rearrangements equal to N^N (e.g., an array of 10 digits can be rearranged in 10^{10} different ways). At the time, this sounded very convincing to us, and we applied the algorithm without further testing. However, when for testing purposes we listed the 5! = 120 possible orderings of 5 digits and made 12,000 simulations to check whether the 5^5 = 3,125 possible rearrangements were equally divided over the orderings, we always obtained chi-square values around 700, even though values around 119 were expected (see above). This indicated that something was wrong. If we looked further at the first digit of the permuted array, we saw that the probability was 0.194 that this digit was the first of the original array, 0.245 that it was the second, 0.214 that it was the third, 0.183 that it was the fourth, and 0.164 that it was the fifth (the expected value each time was 0.200). That is, the probability (based on the 12,000 simulations) of the first digit's being 1 was good, but then the probability decreased monotonically from the first digit's being 2 to the first digit's being 5. The inequality increased as the number of elements in the string was augmented. For instance, the probability that the second element of a 100-item array was the first in the permuted array amounted to 0.014 (100,000 simulations; expected value, 0.010), whereas the probability of the 100th element's being first was only 0.007 (again 0.010 expected). Remember that

Table 4
Distribution of Data from a 10-Item Array after Permutation with Nilsson's Algorithm (100,000 simulations); Entries Are Conditional Probabilities of Item's Being in Final (Column) Position, Given Initial (Row) Position

Position in Original Array	Position of Item after Permutation									
	1	2	3	4	5	6	7	8	9	10
1	.100	.099	.099	.101	.100	.098	.101	.101	.102	.100
2	.130	.095	.094	.095	.097	.096	.098	.098	.099	.099
3	.120	.124	.090	.091	.090	.095	.094	.098	.099	.099
4	.111	.116	.121	.087	.090	.091	.093	.094	.096	.099
5	.103	.109	.112	.119	.085	.087	.092	.095	.097	.101
6	.099	.102	.107	.111	.118	.087	.089	.091	.096	.101
7	.092	.096	.102	.106	.112	.122	.086	.090	.096	.098
8	.087	.091	.096	.102	.107	.113	.119	.090	.094	.100
9	.081	.086	.091	.096	.103	.107	.116	.124	.096	.100
10	.077	.082	.087	.093	.097	.105	.111	.120	.127	.101

an array of 100 elements could be rearranged in 100^{100} different ways. Table 4 gives the distribution of a 10-item array after permutation (the data are based on 100,000 simulations). The differences between observed and expected probability are largest in the lower left corner.

Nilsson's (1978) algorithm learns two things. First, that something looks complicated and/or seems appropriate does not make it random; and second, not all procedures published in established journals and/or books have been well tested (though we admit that mistakes are sometimes very difficult to trace both by authors and by reviewers).

If we then look for a good permutation algorithm, we have to return to the basic process we want to simulate. What is needed is random sampling without replacement. This can be compared with a bowl that contains N elements, from which one element after another is picked out and processed. Each item has a probability of $1/N$ to be picked out first. If it is not picked out the first time, it has a chance of $1/(N-1)$ to be picked out second, a chance of $1/(N-2)$ to be picked out third, and so on, until all items are removed from the bowl. The following algorithm does just this. It draws an element I with chance $1/(N-I+1)$ from the array and places it at the end. Note that the number of possible rearrangements is smaller than that for Nilsson's algorithm ($N!$ instead of N^N) yet, it produces much better results.

Algorithm 10
Permutation

```
set I = N + 1
repeat
  set I = I - 1
  generate U = integer random number from 1 to I
  set A = X[I]
  set X[I] = X[U]
  set X[U] = A
until I = 2
```

Twenty simulations in which 12,000 permutations of a five-item array were generated yielded a mean chi-square value of 116.16 for the differences between the observed and expected frequencies over the 120 possible orderings (see above). This is close to the expected value of 119. Algorithm 10 was first proposed by Moses and Oakford (1963) and Green (1963).

There are two more things to be said about randomization. First, Algorithm 10 produces truly random sequences of items. There is no need to "correct" it by adding constraints, as is sometimes seen in the literature. For instance, it is not necessary to alter the sequences with more than three stimuli belonging to the same condition, in order to make the sequence more random. Actually, these "corrections" are usually mistakes due to human failure to produce randomness without special training. Their net result is more often an increase of information rather than a decrease (e.g., excluding all sequences with more than three subsequent stimuli of the same condition informs the subject about the fact that if three stimuli of the same type have been presented, the fourth will surely be one of a different type). Those "corrections" should be avoided, unless experimental tests of models require such "nonrandom" constraints. A second common mistake with respect to permutation is the idea that it suffices to make just one random permutation of a stimulus series and to present that permuted series to all subjects. The major aim of randomization is to preclude sequence effects, and because this is largely done by "averaging out" influences, every systematization may involve a bias. With the ubiquitous use of microcomputers, it is not difficult to generate a new permutation for each subject and/or experimental session.

Random Sampling

Two procedures for drawing a random sample from a population can be distinguished, depending on the need to preserve the order of the subjects/stimuli. If the order is of no importance, Algorithm 10 can be used. For instance, if 10 stimuli must be drawn from a population of 100, the algorithm is applied from $I = 100$ till $I = 91$, and the last 10 items of the array are used as the sample. If, on the other hand, the order of the stimuli is critical, either a sorting algorithm (see, e.g., Dreger, 1989; Dwyer & Critchfield, 1978; Ellis, 1985; Knuth, 1973; Press, Flannery, Teukolsky, & Vetterling, 1986) must be added to Algorithm 10, or another algorithm must be used. Bissell (1986) proposed the following procedure:

Algorithm 11
Random Sampling with Order Preservation

```
1. set population size = N, sample size = n
   set A = N - n, N' = N, A' = A
2. generate U = random number
   set B = 1
3. set B = B * A'/N'
   if B ≤ U then select item N-N'+1
                 set N' = N' - 1
                 if N' > 0 then go to 2 else stop
   else set N' = N' - 1
        set A' = A' - 1
        if N' > 0 then go to 3 else stop
```

To test Algorithm 11, all possible samples of 5 elements drawn from a population of 10 elements were listed. This yielded a total of $10!/5!5! = 252$ samples. Twenty replications of 25,200 sample drawings were completed, which gave an average chi-square value for the difference between the observed and the expected frequencies equal to 248.93, very close to the expected value of 251.

Randomness in Time: The Exponential and Geometric Distribution

The generation of exponential deviates has been included in this paper because the exponential distribution is the only one that yields true randomness in time. Suppose, for instance, that an experimenter wants to control eye fixations in a visual word recognition task. The experimenter does so by flashing a digit instead of a word at the fixation location from time to time. Subjects have to identify the tachistoscopically presented digit, and if too many errors are made, the session is called invalid. Luce (1986, pp. 13-15) argues that in such a case it would be a bad strategy to use a random variable with a uniform distribution—say, a time-interval varying from 0 to 9 stimuli, with each value having the same probability—because such a procedure changes the amount of information between different values of the variable. Immediately after presentation of a digit, chances are 1/10 that a new digit will be presented. However, if no digit has been presented on the 1st trial, chances become 1/9 that it will be shown on the 2nd trial. Similarly, if no digit was shown in the first two trials, chances are 1/8 that it will appear on the 3rd trial, and so on. Finally, after 9 trials without a digit, the probability of a digit on the 10th trial reaches 1, which is a complete lack of randomness. Thus, what is needed is a procedure that will keep the probability of presenting a digit constant at each trial. If chances are 1/10 that a digit is presented immediately after another digit, the probability that a digit is presented on Trial 2 if no digit has been presented on Trial 1 must also be 1/10. Or to put it differently, the probability that a digit is presented on Trial 2 must equal 9/10 * 1/10 = 0.09 (i.e., the probability that no digit has been presented on Trial 1 times the probability that a digit is presented on Trial 2). Similarly, the probability of a digit on Trial 3 is 0.9 * 0.9 * 0.1 = .081, and so on. More formally, the probability that a digit is presented on Trial i equals $(1-p)^{i-1} * p$, which is the geometric distribution, the discrete equivalent of the exponential function. An algorithm for sampling from a geometric distribution is the following:

Algorithm 12
Geometric Distribution

```
generate E = random standard exponential deviate
set A = −ln(1−p)
return G = integer part of E/A      G = 0,1,2,...
```

The exponential and/or geometric distribution should be utilized whenever randomness in time is required (e.g., also for random foreperiods in reaction time studies; see Luce, 1986, pp. 54-55). Exponential distributions can be used for simulations as well. For instance, Strube (1983; see also Gruenberger & Jaffray, 1965) proposed the following test to check the usefulness of a random number generator. Integers between 0 and 9 are generated and the average interval between repetitions of the digits in the series is examined. The distribution of these intervals is geometric with probability density function = $0.9^{i-1} * 0.1$, expected value = 9 = $(1-p)/p$, and variance = 90 = $(1-p)/p^2$ (Luce, 1986, p. 41). This test is known in the literature as the gap test (see above). However, whereas most authors (see, e.g., Knuth, 1981) verify the usefulness of a generator by comparing the observed and the expected frequencies of the different gaps with the use of a chi-square test, Strube (1983) proposed to compare the average gap with the expected value and to compute a t test. Furthermore, he calculated the variance of the gaps and compared it with the expected value of 90 via a chi-square test. However, both the t test of means and the chi-square test of variances assume normality of data (Hays, 1988, pp. 292-293 and 327-331). To check whether the geometric distribution of the raw data distorted the test statistics, 10,000 means and variances of 100 geometric deviates were calculated and compared with the expected $t(99)$ and $\chi^2(99)$ distributions. More specifically, the probabilities at the .005 and .025 tails were evaluated. For the t test, this gave lower tail values of .016 and .047, respectively, and upper tail values of .001 and .011. That is, the t test was too conservative at the upper part and too liberal at the lower part. The chi-square test, which was even worse, gave rather enhanced values of .097 and .173 at the low end, and .085 and .137 at the high end. Therefore, Strube's (1983) gap test (see also Gruenberger & Jaffray, 1965) should not be used to test the usefulness of a random number generator, unless better tests than t and chi-square are available (an alternative might be to run a number of simulations and estimate the critical values).

CONCLUSION

Algorithms have been described to generate random numbers with a uniform, a normal, and an exponential (geometric) distribution. The utility of these numbers was illustrated with procedures for randomization, random sampling, and randomness in time. Other uses are simulations, numerical approximations of compound mathematical equations, and the creation of nonrandom sequences in which various forms of autocorrelation are present (for these procedures, see Malmi, 1986, and Box & Jenkins, 1976, pp. 46-84).

AVAILABILITY

In addition to the algorithm descriptions in the text, Appendix B provides Turbo Pascal listings of all procedures discussed. However, it is our experience that a gap exists between the availability of algorithms and their actual implementation. Small mistakes are easily made, so that one is obliged to rerun some elementary tests in order to check the correctness of the implementation. Therefore, Appendix A displays the first few numbers generated by the algorithms when all seeds are equal to 1. In this way, everyone can check the correctness of their implementation. Turbo Pascal and BASIC implementations can also be obtained by sending a formatted disk in a returnable box to the author. For administrative costs, $10 must be included.

REFERENCES

Afflerbach, L. (1985). The pseudo-random number generators in Commodore and Apple microcomputers. *Statistische Hefte*, **26**, 321-333.
Aldridge, J. W. (1987). Cautions regarding random number generation on the Apple II. *Behavior Research Methods, Instruments, & Computers*, **19**, 397-399.

Atkinson, A. C. (1980). Tests of pseudo-random numbers. *Applied Statistics*, **29**, 164-171.

Best, D. J. (1979). Some easily programmed pseudo-random normal generators. *Australian Computer Journal*, **11**, 60-62.

Bissell, A. F. (1986). Ordered random selection without replacement. *Applied Statistics*, **35**, 73-75.

Box, G. E. P., & Jenkins, G. M. (1976). *Time series analysis: Forecasting and control*. San Francisco: Holden-Day.

Box, G. E. P., & Muller, M. E. (1958). A note on the generation of random normal deviates. *Annals of Mathematical Statistics*, **29**, 610-611.

Bradley, D. R. (1988). *DATASIM*. Lewiston, ME: Desktop Press.

Bradley, D. R., Senko, M. W., & Stewart, F. A. (1990). Statistical simulation on microcomputers. *Behavior Research Methods, Instruments, & Computers*, **22**, 236-246.

Brophy, A. L. (1985). Approximation of the inverse normal distribution function. *Behavior Research Methods, Instruments, & Computers*, **17**, 415-417.

Dreger, R. M. (1989). A BASIC program for the Shell-Metzner sort algorithm. *Educational & Psychological Measurement*, **49**, 619-622.

Dudewicz, E. J., & Ralley, T. G. (1981). *The handbook of random number generation and testing with TESTRAND computer code*. Columbus, OH: American Sciences Press.

Dwyer, T., & Critchfield, M. (1978). *BASIC and the personal computer*. Reading, MA: Addison-Wesley.

Edgell, S. E. (1979). A statistical check of the DECsystem-10 FORTRAN pseudorandom number generator. *Behavior Research Methods & Instrumentation*, **11**, 529-530.

Ellis, J. K. (1985). Distribution counting as a method for sorting test scores. *Behavior Research Methods, Instruments, & Computers*, **17**, 419-420.

Fishman, G. S., & Moore, L. R., III (1986). An exhaustive analysis of multiplicative congruential random number generators with modulus $2^{31}-1$. *SIAM Journal on Scientific & Statistical Computing*, **7**, 24-45.

Green, B. F. (1963). *Digital computers in research: An introduction for behavioral and social scientists*. New York: McGraw-Hill.

Gruenberger, F., & Jaffray, G. (1965). *Problems for computer solution*. New York: Wiley.

Hays, W. L. (1988). *Statistics*. New York: Holt, Rinehart and Winston.

Kennedy, W. J., & Gentle, J. E. (1980). *Statistical computing*. New York: Marcel Dekker.

Knuth, D. E. (1973). *The art of computer programming: Vol. 3. Sorting and searching*. Reading, MA: Addison-Wesley.

Knuth, D. E. (1981). *The art of computer programming: Vol. 2. Seminumerical algorithms*. Reading, MA: Addison-Wesley.

Kraner, H. C., Mohanty, S. G., & Lyons, J. C. (1980). Critical values of the Kolmogorov-Smirnov one-sample test. *Psychological Bulletin*, **88**, 498-501.

Lordahl, D. S. (1988). Repairing the Microsoft BASIC RND function. *Behavior Research Methods, Instruments, & Computers*, **20**, 221-223.

Luce, R. D. (1986). *Response times*. New York: Oxford University Press.

MacLaren, N. M. (1989). The generation of multiple independent sequences of pseudorandom numbers. *Applied Statistics*, **38**, 351-359.

McLeod, A. I. (1985). A remark on Algorithm AS183: An efficient and portable pseudo-random number generator. *Applied Statistics*, **34**, 198-200.

Malmi, R. A. (1986). Intuitive covariation estimation. *Memory & Cognition*, **14**, 501-508.

Marsaglia, G. (1962). Random variables and computers. In J. Kozesnik (Ed.), *Information theory, statistical decision functions, random processes: Transactions of the Third Prague Conference* (pp. 499-510). Prague: Czechoslovak Academy of Sciences.

Marsaglia, G., & Bray, T. A. (1964). A convenient method for generating normal variables. *SIAM Review*, **6**, 260-264.

Modianos, D. T., Scott, R. C., & Cornwell, L. W. (1987). Testing intrinsic random-number generators. *Byte*, **12**, 175-178.

Moses, L. E., & Oakford, R. V. (1963). *Tables of random permutations*. Stanford, CA: Stanford University Press.

Nance, R. E., & Overstreet, C. L. (1972). A bibliography on random number generators. *Computer Review*, **13**, 495-508.

Nilsson, T. H. (1978). Randomization without replacement using replacement without losing your place. *Behavior Research Methods & Instrumentation*, **10**, 419.

Press, W. H., Flannery, B. P., Teukolsky, S. A., & Vetterling, W. T. (1986). *Numerical recipes: The art of scientific computing*. Cambridge, U.K.: Cambridge University Press.

Ripley, B. D. (1983). Computer generation of random variables: A tutorial. *International Statistical Review*, **51**, 301-319.

Ripley, B. D. (1987). *Stochastic simulation*. New York: Wiley.

Sahai, H. (1979). A supplement to Sowey's bibliography on random number generation and related topics. *Journal of Statistical Computation & Simulation*, **10**, 31-52.

Sowey, E. R. (1972). A chronological and classified bibliography on random number generation and testing. *International Statistical Review*, **40**, 355-371.

Sowey, E. R. (1978). A second classified bibliography on random number generation and testing. *International Statistical Review*, **46**, 355-371.

Sowey, E. R. (1986). A third classified bibliography on random number generation and testing. *Journal of the Royal Statistical Society*, **149A**, 83-107.

Strube, M. J. (1983). Tests of randomness for pseudorandom number generators. *Behavior Research Methods & Instrumentation*, **15**, 536-537.

von Neumann, J. (1951). Various techniques in connection with random digits. *NBS Applied Mathematics Series*, **12**, 36-38.

Wichmann, B. A., & Hill, J. D. (1982). Algorithm AS183: An efficient and portable pseudo random number generator. *Applied Statistics*, **31**, 188-190.

Wichmann, B. A., & Hill, J. D. (1984). An efficient and portable pseudo random number generator: Correction. *Applied Statistics*, **33**, 123.

Zeisel, H. (1986). A remark on Algorithm AS183: An efficient and portable pseudo-random number generator. *Applied Statistics*, **35**, 89.

Zelen, M., & Severo, N. C. (1964). Probability functions. In M. Abramowitz & I. A. Stegun (Eds.), *Handbook of mathematical functions* (pp. 925-995). New York: Dover.

NOTES

1. The div operation stands for integer division (i.e., 14 div 5 = 2). The div and mod operation are available in most software packages.

2. Note that for many algorithms the number of generated random numbers (and the time required) varies as a function of the algorithm, because random numbers are sampled (and discarded) until some criterion is met.

(Continued)

APPENDIX A
Numbers Generated by the Different Algorithms

```
seed1 = 1
seed2 = 1
seed3 = 1

first 10 numbers of Wichmann and Hill's random number generator (Algorithm 1):
  0.01693090620   0.89525391124   0.11149102121   0.93952679641   0.12822985510
  0.17800399298   0.29982708249   0.34971840637   0.05928746025   0.82197931465

first 10 numbers of standard normal, inverse cdf (Algorithm 2):
  2.12205889020  -1.25512190220   1.21877656770  -1.55109245260   1.13489054900
  0.92295174709   0.52458866900   0.38572616881   1.56106514540  -0.92288764201

first 10 numbers of standard normal, Box-Muller (Algorithm 3):
  0.46776157925   0.27003245504   1.28682417770  -0.44644375106   0.58321777179
  1.40685839470  -0.71746985100  -0.71278233544   1.07699514850  -0.28908727769

first 10 numbers of standard normal, polar method (Algorithm 4):
 -0.19407337327  -1.33042159440   2.19755506130  -0.59082236112   0.68175817609
  1.13620439410   0.87865940120  -0.50754615265  -0.17307865854   0.53106697446

first 10 numbers of standard normal, Marsaglia-Bray (Algorithm 5):
  0.89254345772  -1.34490103630   0.72689870961  -1.01316404230  -0.32030371023
  0.99555832695   0.82905654588   0.51709027840   0.12444994842  -0.22350462413

first 10 numbers of standard normal, ratio of uniforms (Algorithm 6):
 -0.85990598276  -0.66165288210  -0.03200237951  -1.68554875660   0.03422323645
  0.46775744684   0.58781477852   0.97552442825   0.31896217480  -0.46142694379

first 10 numbers of standard exponential, inverse cdf (Algorithm 7):
  4.07861455800   0.11064790124   2.19381121860   0.06237893855   2.05393088250
  1.72594929650   1.20454936220   1.05062700160   2.82535745820   0.19604004890

first 10 numbers of standard exponential, von Neumann (Algorithm 8):
  0.01693090620   0.11149102121   0.12822985510   0.29982708249   0.05928746025
  1.48791600110   0.51884426837   6.80740561510   1.09824758910   0.73856139688

first 10 numbers of standard exponential, ratio of uniforms (Algorithm 9):
  1.02135355940   0.85818939924   0.13362312147   0.55416121890   0.44833448843
  0.08964587217   0.75126375694   0.82019400019   0.66731253059   1.20040721920

randomization array of 10 stimuli (Algorithm 10):
  before randomization:  1  2  3  4  5  6  7  8  9 10
  after randomization:   3  5  4  2  6  8  7 10  9  1

random sample from population (Algorithm 11):
  population:  1  2  3  4  5  6  7  8  9 10
  sample:      5  6  8  9 10
```

(Continued)

APPENDIX B
Listing

```
(********    TURBO PASCAL ROUTINES FOR RANDOM NUMBER GENERATION    **********************)


VAR
    seed1,seed2,seed3 : integer;
   seed11,seed21,seed31 : longint;                      {optional, only if rndnumb1 used}


(********    Wichmann & Hill's random number generator.    ******************************

   Produces random numbers stricly larger than 0 and strictly smaller than 1.  Uniform
   distribution.  Needs three seeds to start.  Integer arithmetic up to 5212632 required.
   If danger of values 0.0 and 1.0 due to rounding (McLeod, 1985), add corrections.     *)

FUNCTION rndnumb1(xp1,xp2,xp3:longint): real;                    {xp = seed for generator}
{VAR rp : real;                                                  optional, rounding errors}
BEGIN
 seed11 := (171*xp1) mod 30269;
 seed21 := (172*xp2) mod 30307;
 seed31 := (170*xp3) mod 30323;
 rndnumb1 := frac(seed11/30269 + seed21/30307 + seed31/30323);  {take fractional part sum}
{ rp := frac(seed11/30269 + seed21/30307 + seed31/30323);
 IF rp <= 0.0 THEN rp := 0.0000000001;
 IF rp >= 1.0 THEN rp := 0.9999999999;
 rndnumb1 := rp;                                                 optional, rounding errors}
END;


(********    Wichmann & Hill's random number generator.    *****************************

  Better implementation of Wichmann & Hill's random number generator.  Produces the same
  numbers, but requires only arithmetic up to 30323 and usually is faster.  See also
  rounding error checking in rndnumb1.                                                 *)

FUNCTION rndnumb2(xp1,xp2,xp3:integer): real;                     {xp = seed for generator}
BEGIN
 seed1 := 171 * (xp1 mod 177) -  2*(xp1 div 177);
 seed2 := 172 * (xp2 mod 176) - 35*(xp2 div 176);
 seed3 := 170 * (xp3 mod 178) - 63*(xp3 div 178);
 IF seed1 < 0 THEN seed1 := seed1 + 30269;
 IF seed2 < 0 THEN seed2 := seed2 + 30307;
 IF seed3 < 0 THEN seed3 := seed3 + 30323;
 rndnumb2 := frac(seed1/30269 + seed2/30307 + seed3/30323);
END;


(********    Standard normal distribution inverse cdf    *********************************

  Function to produce random numbers with a standard normal distribution using the
  inverse cdf.  For information about the values used, see Brophy (1985).              *)

FUNCTION random_stand_norml: real;  {inverse F  Hill & Davis}
VAR tp1,tp2,tp3,tp4: real;
BEGIN
 tp1 := rndnumb2(seed1,seed2,seed3);
 IF tp1 > 0.5 THEN tp2 := 1-tp1 else tp2 := tp1;
 IF tp2 < 1E-20 THEN tp4 := 10
 ELSE
  BEGIN
   tp3 := sqrt(-2*ln(tp2));
   tp4 := tp3-((7.45551*tp3+450.636)*tp3+1271.059)/
          (((tp3+110.4212)*tp3+750.365)*tp3+500.756);
  END;
 IF tp1 > 0.5 THEN tp4 := -tp4;
 random_stand_norml := tp4;
END;
```

```
(********    Standard normal distribution Box-Muller     **********************************

  Function to produce random numbers with a standard normal distribution using
  the Box-Muller algorithm.  For information about the values used, see Ripley (1987).  *)

FUNCTION random_stand_norm2: real;  {Box-Muller}
VAR tp1,tp2,tp3,tp4,tp5 : real;
BEGIN
 tp1 := rndnumb2(seed1,seed2,seed3);
 tp2 := 2*pi*tp1;
 tp3 := rndnumb2(seed1,seed2,seed3);
 tp4 := sqrt(2*(-ln(tp3)));
 tp5 := tp4 * cos(tp2);
 random_stand_norm2 := tp5;
END;



(********    Standard normal distribution polar method     ********************************

  Function to produce random numbers with a standard normal distribution using
  the polar method.  For information about the values used, see Ripley (1987).         *)

FUNCTION random_stand_norm3: real;  {polar}
VAR tp1,tp2,tp3,tp4,tp5 : real;
BEGIN
 REPEAT
  tp2 := 2*rndnumb2(seed1,seed2,seed3)-1.0;
  tp3 := 2*rndnumb2(seed1,seed2,seed3)-1.0;
  tp4 := sqr(tp2) + sqr(tp3);
 UNTIL tp4 < 1.0;
 random_stand_norm3 := sqrt(-2.0*ln(tp4)/tp4) * tp2;
END;



(********    Standard normal distribution Marsaglia-Bray    *******************************

  Function to produce random numbers with a standard normal distribution using the
  Marsaglia-Bray algorithm.  For information about the values used, see Ripley (1987).  *)

FUNCTION random_stand_norm4: real;   {Marsaglia-Bray}
VAR tp1,tp2,tp3,tp4,tp5,tp6,tp7,tp8,tp9: real;
BEGIN
 tp1 := rndnumb2(seed1,seed2,seed3);
 IF tp1 < 0.8638 THEN
  begin
   tp2 := rndnumb2(seed1,seed2,seed3);
   tp3 := rndnumb2(seed1,seed2,seed3);
   tp4 := rndnumb2(seed1,seed2,seed3);
   tp5 := 2*(tp2+tp3+tp4) - 3;
  END
 ELSE IF tp1 < 0.9745 THEN
  BEGIN
   tp2 := rndnumb2(seed1,seed2,seed3);
   tp3 := rndnumb2(seed1,seed2,seed3);
   tp5 := 1.5*(tp2+tp3-1);
  END
 ELSE IF tp1 < 0.9973002039 THEN
  BEGIN
   REPEAT
    tp2 := rndnumb2(seed1,seed2,seed3);
    tp3 := 6*tp2 - 3;
    IF abs(tp3) < 1.0 THEN tp6 := 17.49731196*exp(-sqr(tp3)/2) - 4.73570326*(3.0-sqr(tp3))
                                 - 2.15787544*(1.5-abs(tp3))
    ELSE IF abs(tp3) < 1.5 THEN tp6 := 17.49731196*exp(-sqr(tp3)/2) -
                        2.36785163*sqr(3.0-abs(tp3)) - 2.15787544*(1.5-abs(tp3))
    ELSE IF abs(tp3) < 3.0 THEN tp6 := 17.49731196*exp(-sqr(tp3)/2) -
                        2.36785163*sqr(3.0-abs(tp3))
    ELSE writeln(tp3:10:4);
    tp4 := rndnumb2(seed1,seed2,seed3);
   UNTIL 0.358*tp4 <= tp6;
   tp5 := tp3;
  END
```

```
    ELSE
     BEGIN
      REPEAT
       REPEAT
        tp2 := rndnumb2(seed1,seed2,seed3);
        tp3 := rndnumb2(seed1,seed2,seed3);
        tp4 := sqr(2*tp2-1) + sqr(2*tp3-1);
       UNTIL tp4 < 1;
      tp6 := sqrt((9-2*ln(tp4))/tp4);
      tp7 := tp6*(2*tp2-1);
      tp8 := tp6*(2*tp3-1);
     UNTIL ((abs(tp7) > 3.0) OR (abs(tp8) > 3.0));
     IF abs(tp7) > 3.0 THEN tp5 := tp7
     ELSE tp5 := tp8;
    END;
    random_stand_norm4 := tp5;
   end;


(********    Standard normal distribution ratio-of-uniforms    ***************************

  Function to produce random numbers with a standard normal distribution using the
  ratio-of-uniforms method.  For information about the values used, see Ripley (1987).  *)

FUNCTION random_stand_norm5: real;  {ratio-of-unforms}
VAR tp1,tp2,tp3,tp4 : real;
LABEL stop;
BEGIN
 REPEAT
  tp1 := rndnumb2(seed1,seed2,seed3);
  tp2 := 0.8578*(2*rndnumb2(seed1,seed2,seed3) - 1);
  tp3 := tp2/tp1;
  tp4 := 0.25*sqr(tp3);
  IF tp4 < 1-tp1 THEN GOTO stop;                    {optional, to speed up the generation}
 UNTIL ((tp4 <= 0.259/tp1+0.35) and (tp4 <= -ln(tp1)));
 stop:
 random_stand_norm5 := tp3;
END;


(********    Exponential distribution inverse cdf    *************************************

  Function to generate random numbers with a standard exponential distribution.
  Inverse cdf;  For more information, see Ripley (1987)                                  *)

FUNCTION random_exp1: real; {inverse cdf}
BEGIN
 random_exp1 := -ln(rndnumb2(seed1,seed2,seed3));
END;


(********    Exponential distribution von Neumann    *************************************

  Function to generate random numbers with a standard exponential distribution.
  von Neumann;  For more information, see Ripley (1987)                                  *)

FUNCTION random_exp2: real; {von Neumann}
VAR ap1,ap2,ap3,ap4,ap5 :real;
LABEL stop,opnieuw,nog;
BEGIN
 ap1 := 0.0;
 opnieuw:
 ap2 := rndnumb2(seed1,seed2,seed3);
 ap5 := ap2;
 nog:
 ap3 := rndnumb2(seed1,seed2,seed3);
 IF ap2 < ap3 THEN goto stop;
 ap2 := rndnumb2(seed1,seed2,seed3);
 IF ap2 < ap3 THEN goto nog;
 ap1 := ap1 + 1.0;
 goto opnieuw;
 stop:
 random_exp2 := ap1 + ap5;
END;
```

```
(********    Exponential distribution ratio-of-uniforms    ******************************

  Function to generate random numbers with a standard exponential distribution.
  Ratio-of-uniforms;  For more information, see Ripley (1987)                        *)

FUNCTION random_exp3: real;   {ratio-of-uniforms}
VAR ap1,ap2,ap3,ap4,ap5,ap6,ap7,ap8,ap9,ap10 : real;
LABEL stop,opnieuw;
BEGIN
 ap4 := 2/2.7182818285;
 ap5 := 1.6487; ap8 :=  1.49998709858;
 ap6 := 0.105;  ap9 := -1.25379492880;
 ap7 := 0.773; ap10 :=  0.74252376960;
 opnieuw:
 ap1 := rndnumb2(seed1,seed2,seed3);
 ap2 := rndnumb2(seed1,seed2,seed3)*ap4;
 ap3 := ap2/ap1;
 IF ap3/2 <= ap8-ap5*ap1 THEN goto stop;
 IF ap3/2 > ap6/ap1-ap9  THEN goto opnieuw;
 IF ap3/2 > ap7/ap1-ap10 THEN goto opnieuw;
 IF ap3/2 > -ln(ap1) THEN goto opnieuw;
 stop:
 random_exp3 := ap3;
END;


(********    Geometric distribution    **************************************************

  Function to generate random numbers with a geometric distribution.                *)

FUNCTION random_geom(p:real): integer;
VAR tp1 : real;
BEGIN
 tp1 := ln(1-p);
 random_geom := trunc(ln(rndnumb2(seed1,seed2,seed3))/tp1);
END;


TYPE stim_array = array[1..100] of integer;


(********    Permutation routine    *****************************************************

  Procedure to make a random permutation of an array with Np stimuli.               *)

PROCEDURE permute(var data:stim_array;Np:integer);          {Np = number of Si in array}
VAR ip,rndp,datap : integer;
BEGIN
 FOR ip := Np downto 2 DO
  BEGIN
   rndp := trunc(rndnumb2(seed1,seed2,seed3)*int(ip)) + 1;   {important that rndnumb2 < 1}
   datap := data[ip];
   data[ip] := data[rndp];
   data[rndp] := datap;
  END;
END;


(********    Procedure to take a sample without replacement    **************************

  Procedure that takes a random sample of Np2 elements from a population of Np1 elements.
  Rankorder preservation.  Sample is placed at the beginning of the data array.        *)

PROCEDURE sample_without_replacement(var data:stim_array;Np1,Np2:integer);
VAR ip,Np1b,Np2b,counter1,counter2 : integer;
    pb,rndp : real;
    data1,data2 : stim_array;
LABEL stop;
BEGIN
 counter1 := 0;
 counter2 := 0;
 Np1b := Np1;
 Np2b := Np1-Np2;
 REPEAT
```

```
  pb := Np2b/Np1b;
  rndp := rndnumb2(seed1,seed2,seed3);
  WHILE pb > rndp DO
   BEGIN
    counter1 := counter1 + 1;
    data2[counter1] := data[Np1-Np1b+1];
    Np1b := Np1b-1;
    Np2b := Np2b-1;
    IF Np1b > 0 THEN pb := pb*Np2b/Np1b
    ELSE GOTO stop;
   end;
  counter2 := counter2 + 1;
  data1[counter2] := data[Np1-Np1b+1];
  Np1b := Np1b - 1;
 UNTIL Np1b = 0;
 stop:
 FOR ip := 1 TO Np2 DO
  data[ip] := data1[ip];
 FOR ip := Np2+1 TO Np1 DO
  data[ip] := data2[ip-Np2];
END;

(*****************************************************************************************)
```

(Manuscript received December 4, 1989;
revision accepted for publication November 2, 1990.)