SIAM J. SCI. STAT. COMPUT.
Vol. 5, No. 2, June 1984

A FAST, EASILY IMPLEMENTED METHOD FOR SAMPLING FROM DECREASING OR SYMMETRIC UNIMODAL DENSITY FUNCTIONS*

GEORGE MARSAGLIA† AND WAI WAN TSANG†

Abstract. The fastest computer methods for sampling from a given density are those based on a mixture of a fast and slow part. This paper describes a new method of this type, suitable for any decreasing or symmetric unimodal density. It differs from others in that it is faster and more easily implemented, thereby providing a standard procedure for developing both the fast and the slow part for many given densities. It is called the ziggurat method, after the shape of a single, convenient density that provides for both the fast and the slow parts of the generating process. Examples are given for REXP and RNOR, subroutines that generate exponential and normal variates that, as assembler routines, are nearly twice as fast as the previous best assembler routines, and that, as Fortran subroutines, approach the limiting possible speed: the time for Fortran subroutine linkage conventions plus the time to generate one uniform variate.

Key words. random numbers, normal random variables, exponential random variables, ziggurat method, Monte Carlo, simulation

AMS(MOS) subject classifications, mr categories. 65C10, 65C05, 68A55

1. Introduction.

It has long been known that, in principle, computer sampling from any given density can be made very rapid; if enough memory space is available and enough attention is paid to details that provide, most of the time, rapid return of the required variate by means of a few operations on a uniform variate [5].

Applications of this principle have led to very fast subroutines for normal and exponential variates, but these implementations have been ad hoc, tailored to the features of each density and each particular machine. The fastest seem to be those based on the rectangle-wedge-tail method of references [6], [8] and described in Knuth [3, p. 211].

This article seeks to improve on this situation by suggesting a general method that is easily applied to any decreasing or symmetric unimodal density $f(x)$. The method leads to exact, even faster, subroutines, using a table of $n+1$ entries, with the user free to choose n on the basis of time and complexity before setting up the table.

The time and space complexity of the algorithm may be inferred from the following outline, in which UNI means a uniform random variable and a table $x_0, x_1, \cdots, x_n$ and constants a, b, c are given:

```
ALGORITHM Z.
1. Choose $j$ uniformly from $\{1, 2, \cdots, n\}$.
2. Form $X = x_j$*UNI; if $X < x_{j-1}$, return $X$.
   (Some 99% of the time when $n = 256$, 96% when $n = 64$.)
3. Else form $x = (X - x_{j-1})/(x_j - x_{j-1})$, $y = $ UNI.
4. If $y > c - af(b - bx)$, return $b - bx$.
5. If $f(x_j) + y/(nx_j) < f(X)$ return $X$.
6. Else return an $X$ from the tail density, $c'f(x)$, $x > x_n$.
```

The algorithm returns the required variate some 96–99% of the time from step 2, which requires two table look-ups, one multiplication and one uniform variate (since a single uniform variate can provide j in step 1 and the UNI in step 2). If the fast

* Received by the editors November 17, 1981. This research was supported by the National Science Foundation under grant MCS80-05789.

† Computer Science Department, Washington State University, Pullman, Washington 99164.

part of the algorithm, steps 1 and 2, is coded as an assembler subroutine and the remaining, slower, part as a Fortran subroutine, the composite will be very fast—nearly twice as fast as the previous fastest.

The entire algorithm can also be easily implemented as a standard Fortran subroutine, to make it machine independent, but there is a time penalty, primarily because of Fortran linkage conventions, which are more costly than the generating procedure itself. But even as a standard Fortran subroutine the procedure is very fast and general, suited for decreasing or symmetric unimodal densities. The resulting average execution times approach the limit that a standard Fortran random variable subroutine can attain: the time for the linkage conventions plus the time to generate one UNI.

Development of the algorithm is in § 2, followed in § 3 by a straightforward procedure for setting up the table $x_0, x_1, \cdots, x_n$ and the three constants a, b, c needed in the algorithm. The set-up procedure accepts the table size n and the given decreasing density $f(x)$, $x \geqq 0$, or a symmetric unimodal density, in which case a uniform variate on $(-1, 1)$ is used.

Section 5 discusses implementations and gives Fortran programs for normal and exponential variates, but the method provides equally fast subroutines for decreasing or symmetric densities, such as Student's t for given degrees of freedom. Section 5 also gives a general method for handling the tail.

Section 6 discusses timing of subroutines and gives times for assembler and standard Fortran versions of the new method, followed by a summary in § 7.

2. The ziggurat method. This section will describe a method that leads to exceptionally fast algorithms for generating variates with decreasing or unimodal, symmetric density functions. Suppose we are given a decreasing $f(x)$ and want to generate a random variable X with density f. Choose a rectangle of unit area, one that crosses $f(x)$ twice, as pictured in Fig. 1.

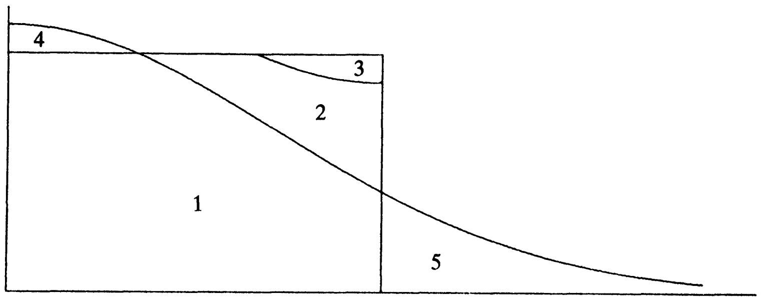

FIG. 1. *Efficient use of a uniform point in a rectangle to get a point under $y = f(x)$.*

This leads to five regions. Region 4 is called the *cap*. The cap is rotated and translated to get the corner region 3, having the same area. Then region 2 has the same area as region 5. (For densities unbounded at the origin, the exact-approximation method [9] may be used to provide a bounded cap.)

Now to generate a variate X with density $f(x)$: Choose (X, Y) uniformly from the rectangle. If (X, Y) is in region 1, return X; if in region 3, return $a-X$; else return a new X from the tail density $c'f(x), x > a$.

This idea is worth implementing if region 1 dominates the rectangle, and we can make it do so by means of what we call a ziggurat density, illustrated for the exponential density e^{-x}, $x > 0$, in Fig. 2. The ziggurat function $z(x)$ is the step function drawn with

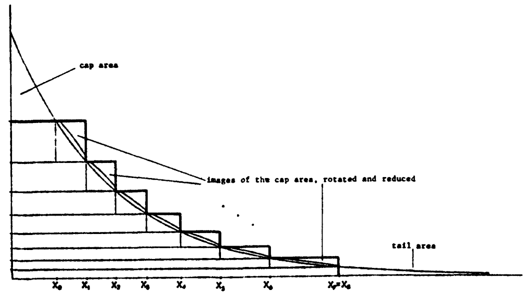

FIG. 2. *Crossing the exponential density twice: a ziggurat density with* 8 *layers of area* 1/8.

heavy lines. It is a density function, made up of n layers, each with area $1/n$. (Drawn with $n=8$—too small to be practical but chosen to provide detail.)

A point (X, Y) is easily generated uniformly under the ziggurat: Choose j uniformly from $\{1, 2, \cdots, n\}$, then (X, Y) uniformly from the jth layer. Most of the time, Y in the appropriate range will not be required, since $X < x_{j-1}$ ensures that the resulting (X, Y) will be under the curve $y = f(x)$.

Each layer of the ziggurat can be represented as in Fig. 3, and this leads to the outline of an algorithm with very fast average execution time:

1. Choose j uniformly from $\{1, 2, \cdots, n\}$.
2. Form $X = x_j^* UNI$; *if* $X < x_{j-1}$, return X.
3. Else choose Y uniformly in the range $f(x_j) < y < f(x_j) + p/x_j$. The resulting (X, Y) is uniform in $B \cup C \cup D$. If in B, return X; if in D, return a multiple of $(x_j - X)$; if in C, return an X from the tail.

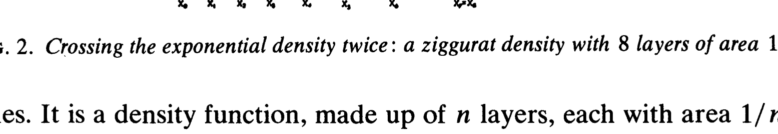

FIG. 3. *The jth layer of the ziggurat, not drawn to scale. Typically, region A occupies* 99% *of the layer.*

The D region of each layer of the ziggurat has the shape of the cap density, rotated, translated and reduced so that the sum of all the D areas is the area of the cap. Thus if a point (X, Y) is uniform in the D region of the jth layer, then a multiple of $x_j - X$ has the cap density.

The total area of C regions is the tail area, so that the algorithm returns a variate X from the cap and tail densities with the proper frequency.

3. The set-up.

The ziggurat density is a step function with n layers, each having area $p = 1/n$. The abscissas of the steps are $x_0, x_1, \cdots, x_n$. They may be computed by letting x_n be the rightmost solution to $xf(x) = p$ (there will be two solutions); then $x_{n-1} = x_n$ and the remaining x's computed from $f(x_i) = f(x_{i+1}) + p/x_{i+1}$ for $i = n-2, \cdots, 1$.

The generating process chooses one of the n layers of the ziggurat at random, say the jth, then forms $X = x_j^*\text{UNI}$ and returns X if $X < x_{j-1}$. This happens most of the time (99% when $n = 256$) and accounts for the speed of the method. Occasionally, further tests must be made after forming $Y = f(x_j) + \text{UNI}^* p/x_j$ so that (X, Y) is uniform in the rectangle at the end of the jth layer—region $B \cup C \cup D$ in Fig. 3.

Details of the additional tests are made easier if the point (X, Y), uniform in $B \cup C \cup D$, is made a point (x, y) uniform in the unit square by means of the transformation

$$x = (X - x_{j-1})/(x_j - x_{j-1}), \qquad y = (Y - f(x_j))x_j/p.$$

When transformed to the unit square, the test regions look like those in Fig. 4.

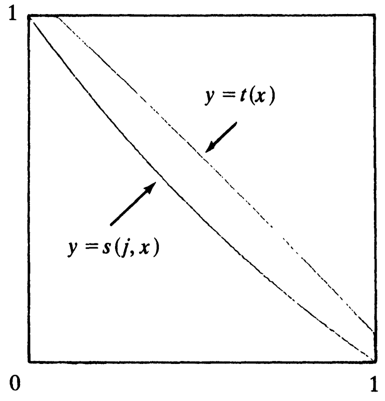

FIG. 4. *The rectangle at the end of the jth layer, transformed to the unit square.*

Then the test $(X, Y) \in B$ is equivalent to $y < s(j, x)$ and $(X, Y) \in D$ is equivalent to $y > t(x)$.

The region $y < s(j, x)$ changes with j:

$$s(j, x) = nx_j[f(x_{j-1} + x(x_j - x_{j-1})) - f(x_j)], \qquad 0 \leqq x \leqq 1.$$

The region $y > t(x)$ does not change with j:

$$t(x) = 1 - \frac{x_0[f(b - bx) - f(x_0)]}{b[f(0) - f(x_0)]}, \qquad 1 - x_0/b \leqq x \leqq 1,$$

with b to be determined so that the area above $y = t(x)$ is correct.

Let the area above $t(x)$ be r. Then r may be expressed in two different ways, providing a formula for the constant b in $t(x)$. Since each of the D regions is mapped into the region $y > t(x)$ of area r in the unit square, each D region must occupy the fraction r of the rectangle $B \cup C \cup D$. Thus the area of the jth D region is $r(1 - x_{j-1}/x_j)p$ and the sum of the D areas must be the cap area:

(1) $$r(1 - x_0/x_1)p + r(1 - x_1/x_2)p + \cdots + r(1 - x_{n-1}/x_n)p = \text{cap area}.$$

The cap area is $\int_0^{x_0} f(x)\,dx - x_0 f(x_0)$ and integration shows the area above $t(x)$ to be

(2) $$r = \int_{1-x_0/b}^{1} [1 - t(x)]\,dx = \frac{x_0(\text{cap area})}{b^2(f(0) - f(x_0))}.$$

Combining (1) and (2) provides the formula

$$b^2 = x_0\left(1 - p\sum_{i=1}^{n} \frac{x_{i-1}}{x_i}\right) \Big/ (f(0) - f(x_0))$$

as well as the interesting conclusion: *setting up the ziggurat does not require integration of f.*

We may summarize the ziggurat set-up procedure with this outline.

Given the decreasing density function $f(x)$, $x \geqq 0$, and $p = 1/n$, with n the number of layers:

1. Let x_n be the larger of the two solutions to $xf(x) = p$.
2. Put $x_{n-1} = x_n$ and solve for x_i from the relations

$$f(x_i) = f(x_{i+1}) + p/x_{i+1},\ i = n-2, \cdots, 2, 1.$$

(See the note below.)

3. Compute

$$b = \left[x_0\left(1 - p\sum_{i=1}^{n} x_{i-1}/x_i\right)\Big/(f(0) - f(x_0))\right]^{1/2},$$

$$a = x_0/[b(f(0) - f(x_0))],$$

$$c = 1 + af(x_0).$$

Note. For some densities, putting $x_{n-1} = x_n$ then solving for the remaining x's in step 2 may leave the cap so large that regions B and D of Fig. 3 overlap. This difficulty may be overcome by stacking more than the first two layers of the ziggurat, such as by setting $x_{n-2} = x_{n-1} = x_n$. For example, the normal and some of the t densities require $x_{n-3} = x_{n-2} = x_{n-1} = x_n$ before computing the remaining x's. Thus for some densities the set-up procedure is not completely automatic, and will require user intervention. An alternative method is to transform the cap by the exact-approximation method [9], so that regions B and D never overlap and the boundary of D is nearly linear. Such a device also allows one to use the ziggurat method for unbounded densities.

4. The algorithm. With n, p, a, b, c and $x_0, x_1, \cdots, x_n$ provided by the ziggurat set-up and recalling that $p = 1/n$, Algorithm Z will return a variate with density $f(x)$.

Since the boundary functions in Fig. 4 are nearly linear, we may use a linear squeeze test in order to avoid the need to evaluate f in steps 4 and 5, most of the time. Define

$$c_1 = \min_{\substack{0 \leqq x \leqq 1 \\ 1 \leqq j \leqq n}} [x + s(j, x)], \qquad c_2 = \max_{a - x_0/b \leqq x \leqq 1} [x + t(x)].$$

Then $c_2 - x > t(x)$ and the linear test $x + y > c_2$ will imply $y > c_2 - x > t(x)$. Similarly, $x + y < c_1$ will imply $y < s(j, x)$.

Thus a faster algorithm results from inserting two steps between steps 3 and 4 of Algorithm Z:

```
1.
2.
3.
3.1. If $x+y > c_2$ return $b - bx$.
3.2. If $x+y < c_1$ return $X$.
4.
5.
6.
```

A quadratic squeeze test may speed up the algorithm even more, but the improvement does not justify the complication, unless f is exceedingly difficult to evaluate. When $n \geqq 256$, even the linear squeeze is barely worth it.

5. Implementation. As examples of the ziggurat method, we list below two Fortran subroutines, REXP and RNOR, that return exponential and normal random variables. They are based on the method with $n = 64$. The generating procedure is short, some 15–17 lines of virtually linear code that follows the outline above.

Both of these subroutines must themselves call random number generators (RNG's). We do not address here the problem of providing suitable RNG's but have assumed those based on the conventions set in the McGill Random Number Package "Super-Duper" [7]:

IUNI: a random integer in $[0, 2^{31})$,

IVNI: a random integer in $[-2^{31}, 2^{31})$,

UNI: a random real in $[0, 1)$.

Users who have different in-house RNG's, over possibly different ranges, should be able to replace calls to IUNI, IVNI, and UNI with suitable adaptation to their own system. Better yet is to insert Fortran code that generates the random IUNI and IVNI directly in the fast part of REXP and RNOR, for the subroutines are so fast that the time for the RNG in the fast part is a major determinant of the overall-speed.

EXPONENTIAL SUBROUTINE

```
      FUNCTION REXP(NULL)
C—RETURNS A RANDOM EXPONENTIAL VARIABLE, IGNORES ARGUMENT
      DATA A, B, C, RMAX/4.780222, .2339010, 4.807275, .4656613E-9/
      DATA C1, C2, P, XN/.9130147, 1.055764, .015625, 5.940712/
      REAL V(65)/.2275733, .2961199, .3568076, .4124534, .4645906,
     +.5141596, .5617859, .6079111, .6528617, .6968884, .7401897,
     +.7829269, .8252345, .8672267, .9090027, .9506499, .9922470,
     +1.033865, 1.075572, 1.117430, 1.159497, 1.201832, 1.244491,
     +1.287529, 1.331001, 1.374964, 1.419475, 1.464591, 1.510374,
     +1.556887, 1.604196, 1.652370, 1.701488, 1.751625, 1.802871,
     +1.855318, 1.909067, 1.964230, 2.020929, 2.079300, 2.139492,
     +2.201675, 2.266037, 2.332792, 2.402185, 2.474495, 2.550045,
     +2.629211, 2.712438, 2.800248, 2.893275, 2.992284, 3.098219,
     +3.212264, 3.335930, 3.471187, 3.620674, 3.788045, 3.978562,
     +4.200208, 4.465950, 4.799011, 5.247564, 5.940712, 5.940712/
C—FAST PART
      I = IUNI(0)
      J = MOD(I, 64) + 1
      REXP = I*RMAX*V(J + 1)
      IF (REXP.LE.V(J)) RETURN
C—————SLOW PART
        X = (REXP - V(J))/(V(J + 1) - V(J))
        Y = UNI(0)
        S = X + Y
        IF (S.GT.C2) GO TO 11
        IF (S.LE.C1) RETURN
        IF (Y.GT.C - A*EXP(B*X - B)) GO TO 11
        IF (EXP(-V(J + 1)) + Y*P/V(J + 1).LE.EXP(-REXP)) RETURN
C———————TAIL PART
          REXP = XN - ALOG(UNI(0))
          RETURN
11      REXP = B - B*X
        RETURN
      END
```

NORMAL SUBROUTINE

```
      FUNCTION RNOR(NULL)
C—RETURNS RANDOM NORMAL VARIABLE, IGNORES ARGUMENT.
      DATA AA,B,C,RMAX/12.37586, .4878992, 12.67706, .4656613E-9/
      DATA C1,C2,PC,XN/.9689279, 1.301198, .1958303E-1, 2.776994/
      REAL V(65)/.3409450, .4573146, .5397792, .6062427, .6631690,
     +.7136974, .7596124, .8020356, .8417227, .8792102, .9148948,
     +.9490791, .9820005, 1.013848, 1.044780, 1.074924, 1.104391,
     +1.133273, 1.161653, 1.189601, 1.217181, 1.244452, 1.271463,
     +1.298265, 1.324901, 1.351412, 1.377839, 1.404221, 1.430593,
     +1.456991, 1.483452, 1.510012, 1.536706, 1.563571, 1.590645,
     +1.617968, 1.645579, 1.673525, 1.701850, 1.730604, 1.759842,
     +1.789622, 1.820009, 1.851076, 1.882904, 1.915583, 1.949216,
     +1.983924, 2.019842, 2.057135, 2.095992, 2.136644, 2.179371,
     +2.224517, 2.272518, 2.323934, 2.379500, 2.440222, 2.507511,
     +2.583466, 2.671391, 2.776994, 2.776994, 2.776994, 2.776994/
C—FAST PART
      I=IVNI(0)
      J=MOD(IABS(I),64)+1
      RNOR=I*RMAX*V(J+1)
      IF (ABS(RNOR).LE.V(J)) RETURN
C————————SLOW PART; AA IS A*f(0)
      X=(ABS(RNOR)-V(J))/(V(J+1)-V(J))
      Y=UNI(0)
      S=X+Y
      IF (S.GT.C2) GO TO 11
      IF (S.LE.C1) RETURN
      IF (Y.GT.C-AA*EXP(-.5*(B-B*X)**2)) GO TO 11
      IF (EXP(-.5*V(J+1)**2)+Y*PC/V(J+1).LE.EXP(-.5*RNOR**2)) RETURN
C—————————TAIL PART: .3601016 IS 1./XN
  22        X=.3601016*ALOG(UNI(0))
            IF (-2.*ALOG(UNI(0)).LE.X**2) GO TO 22
  33        RNOR=SIGN(XN-X,RNOR)
            RETURN
  11    RNOR=SIGN(B-B*X,RNOR)
        RETURN
        END
```

We have chosen to provide the tables in the subroutines through DATA statements, rather than generate them on first entry to the routine. By doing this we avoid the need to use or even look at the arguments of the functions REXP(NULL) and RNOR(NULL). The integer arguments are ignored in the subroutines, but they must be included, since Fortran conventions require that a function subprogram have at least one argument, used or not.

The values in the tables may be generated separately through recursive use of two double precision values DX and DY. Here are Fortran segments that generate the tables, given the initial value DX, the larger of the two solutions to $xf(x)=1/n$, and DY = f(DX).

```
      DOUBLE PRECISION DX/5.940712090024669/,DY/.2630156076110249D-02/
      XN=DX
      DO 1 I=1,63
         IF (I.LE.2) V(63+I)=XN
         DY=DY+1.DO/(64.DO*DX)
         DX=-DLOG(DY)
1        V(64-I)=DX
```

```
      DOUBLE PRECISION DX/2.776994269662875/,DY/.1687976115474328D-1/
      XN=DX
      DO 2 I=1,61
          IF (I.LE.4) V(61+I)=XN
          DY=DY+1.DO/(64.DO*DX)
          DX=DSQRT(-2DO*DLOG(DY/.7978845608028653DO))
2         V(62-I)=DX
```

As the above examples show, there is a standard procedure for the set-up and generation parts of the ziggurat method, given a particular density—except for one thing: the tail part. This part of the overall procedure appears to require the user's intervention in what is otherwise an automatic process.

But even this part of the generating procedure can be made systematic, at least for most of the densities encountered in practice, in the following way: given the tail density $c'f(x)$, $x \geqq x_n$, choose a function $g(t)$ from the class $e^{-\beta t}$ or $(1+bt)^{-\beta}$ such that

$$f(x_n+t) \leqq f(x_n)g(t), \qquad t \geqq 0.$$

Then $G(x)=\int_x^\infty g(t)\,dt/D$ is easily inverted in terms of standard Fortran functions, where $D=\int_0^\infty g(t)\,dt$. To generate X from $c'f(x)$, $x \geqq x_n$, choose uniform U_1, U_2 until

$$U_2 f(x_n) g[G^{-1}(U_1)] \leqq f[x_n + G^{-1}(U_1)],$$

then return $X = x_n + G^{-1}(U_1)$.

The efficiency of this rejection method is $\int_{x_n}^\infty f(t)\,dt/[Df(x_n)]$, so the parameter(s) in $g(t)=e^{-\beta t}$ or $g(t)=(1+bt)^{-\beta}$ should be chosen to minimize D and still have g dominate f.

Examples. The normal tail:

$$f(x)=c'e^{-x^2/2}, \quad x \geqq x_n, \qquad g(t)=e^{-x_n t}.$$

Tail, Student's t with d degrees of freedom:

$$f(x)=c(1+x^2/d)^{-(d+1)/2}, \qquad x \geqq x_n,$$
$$g(t)=(1+bt)^{-d-1}, \qquad b=x_n/(d+x_n^2).$$

Since most of the densities encountered in practice—normal, exponential, gamma, beta, F, t, as well as the stable densities—may be asymptotically, and closely, dominated by a g of the form $e^{-\beta t}$ or $(1+bt)^{-\beta}$, this general method may serve as a uniform procedure for convenient and reasonably fast sampling from tail densities.

6. Timing. Most articles that propose a new method for generating random variables claim that the method is fast, and often specific times are quoted. But many questions arise in assessing such claims: Is the method implemented as a standard subroutine with linkage-convention overhead, or as a faster assembler routine? Are the necessary uniform variates generated within the subroutine or called from separate subroutines with linkage penalties? Is the method for generating uniform variates one of the fast ones, such as multiplicative congruential, or one of the safer, but slower, ones that combine two different methods? If exponential or normal variates are used within the subroutine, how are they generated? What is the effect of the computer used—not only because of possibly faster arithmetic operations and high-speed memory, but the architecture—for example, modular arithmetic for powers of 2 is inherently faster in 2's complement machines.

Some methods are so slow that answers to these questions do not make much difference, but they can make a great difference for methods that approach the limiting speed for a random variable generator.

What is the best possible speed for a method? If it is to be implemented as a standard Fortran subroutine, with linkage-convention overhead, we suggest that the best possible speed on a particular computer can be estimated by timing the subroutine

```
FUNCTION FAST(T)
FAST = UNI(1)
RETURN
END
```

with UNI the name of the uniform generator subroutine.

Such a limiting speed may be estimated by timing the loop

```
    DO 2 I = 1,100000
2   X = FAST(1.)
```

then subtracting the time for the nugatory loop

```
    DO 3 I = 1,100000
3   X = 1.
```

(assuming that the compiler is not so smart that it will compress the latter loop).

Two questions still remain, however. Is the subroutine UNI itself a standard Fortran subroutine or a faster assembler routine, and what is the method for producing the uniform variates?

We contend that comparisons between two methods can be accurate only if both methods are implemented on the same computer, in the same way—as standard or as assembler subroutines—and using the same uniform generator.

Using this criterion, we will compare two methods for generating normal and exponential variates: the ziggurat method and the rectangle-wedge-tail method in the McGill package "Super-Duper" [7], chosen because it was the fastest we knew. None of the published methods such as in [1], [2], [4] provided implementations as simple or as fast as the ziggurat method. Readers may wish to compare these methods on their own systems.

Timing was done on an Amdahl V8, a computer with architecture patterned after that of the IBM 360/370. For the Amdahl V8, the limiting speed for a standard Fortran random variable generator was 6 microseconds (μs.), obtained by timing the above loops, with UNI the assembler routine in the McGill package [7] that generates uniform variates by combining a congruential and a shift-register generator.

The results are in Tables 1 and 2. No comparison could be made between standard Fortran versions, as the rectangle-wedge-tail method was available only as an assembler routine.

TABLE 1

Average speed, in microseconds, for the generation of normal and exponential variates using Fortran callable assembler routines, based on the ziggurat method and the rectangle-wedge-tail method in the Super-Duper package [7]. *For the Amdahl V8 computer.*

	Ziggurat		
	$n = 64$	$n = 256$	Super-Duper
RNOR	2.74	2.34	3.79
REXP	2.45	2.06	4.04

Table 1 compares the two methods with the fast parts implemented in assembler routines, the uniform variates generated within the subroutine. The slow parts are implemented as standard Fortran subroutines. The ziggurat method is faster and the choice of n has a greater effect than in Table 2.

Table 2 shows the speed, for $n = 64$ and 256, of the ziggurat method implemented as a standard Fortan subroutine. The times there, around 9 μs, should be compared to 6 μs, the limiting speed, on this machine, for generators as standard Fortran subroutines that call the subroutine UNI in the McGill Random Number Package [7].

TABLE 2
Average speed, in microseconds, for the generation of normal and exponential variates using the ziggurat method, implemented as a standard Fortran subroutine in the Amdahl V8.

	$n = 64$	$n = 256$
RNOR	9.53	9.11
REXP	9.49	9.20

Another means to put reported speeds of random variable subroutines into perspective is to compare them with the speeds of built-in Fortran functions for that machine. With the Fortran compiler for our Amdahl V8, the average times for SQRT, ALOG and EXP were 6.7, 7.1 and 8.6 microseconds, respectively, for arguments uniformly spread over (0, 1). Such built-in functions are, of course, written in assembly language, and do not have unnecessary linkage convention costs.

Still another measure of the speed for generating exponential variates is to compare REXP with -ALOG (UNI). For the Amdahl V8, the ziggurat speeds are 2.1 μs for REXP partially in assembler, 9.2 μs for REXP as a standard Fortran subroutine and 12 μs for -ALOG (UNI), with UNI itself a standard Fortran subroutine. If UNI is a fast assembler routine, then -ALOG (UNI) is as fast as the standard Fortran version of the ziggurat REXP, and much more convenient.

7. Summary.

For repeated, very rapid sampling from a given decreasing or symmetric unimodal density, the ziggurat method described here is simple and faster than any other we know. If the fast part—$X = x_j^*$UNI; if $X < x_{j-1}$ return X—is implemented as an assembler routine, with the slower part in Fortran, the composite is nearly twice as fast as the previous fastest methods. If the entire procedure is implemented as a standard Fortran subroutine, to make it machine independent, the result, while slower, still approaches the limiting attainable speed: the time for subroutine linkage conventions plus the time to generate one uniform variate.

The method requires setting up a table of $n + 1$ values, which is slow but straightforward. The set-up is easily implemented as part of a Fortran program that accepts the density function and table size as input.

The method is not suited for rapid sampling from a family of densities with shape parameters changing from call to call.

Because of their importance, we think that UNI, RNOR and REXP should be available to Monte Carlo reserchers as very fast, reliable subroutines. These are the bread-and-butter functions for such research, as important to many as the SIN, COS, SQRT, ALOG, EXP functions available—as assembler routines—with every Fortran compiler. Because of the standardized set-up and generating procedure for the ziggurat method, assembler routines for the fast part, and Fortran routines for the remaining, are easily implemented. We urge serious users to consider doing so.

REFERENCES

[1] J. H. Ahrens and K. D. Kohrt, *Computer methods for efficient sampling from largely arbitrary statistical distributions*, Computing, 26 (1981), pp. 19–31.

[2] A. J. Kinderman and J. G. Ramage, *Computer generation of normal random variables*, J. Amer. Statist. Assoc., 71 (1976), pp. 893–896.

[3] D. E. Knuth, *The Art of Computer Programming*, Vol. II: *Seminumerical Algorithms*, 2nd ed., Addison-Wesley, Reading, MA, 1981.

[4] R. A. Kronmal and A. V. Peterson, Jr. *Generating normal random variables using the alias-rejection-mixture method*, Proc. ASA 1979 Annual Meeting Computer Section, Washington, DC, 1980.

[5] G. Marsaglia, *Expressing a random variable in terms of uniform random variables*, Ann. Math. Statist., 32 (1961), pp. 894–899.

[6] G. Marsaglia, M. D. MacLaren and T. A. Bray, *A fast procedure for generating normal random variables*, Comm. ACM, 7 (1964), pp. 4–10.

[7] G. Marsaglia et al., *The McGill Random Number Package "Super-Duper,"* School of Computer Science, McGill University, Montreal, 1972 (available from the present authors, together with *How to Use the McGill Random Number Package Super-Duper*).

[8] G. Marsaglia, K. Ananthanarayanan and N. J. Paul, *Improvements on fast methods for generating normal random variables*, Inform. Process. Lett., 5 (1976), pp. 27–30.

[9] G. Marsaglia, *The exact-approximation method for generating random variables in a computer*, J. Amer. Statist. Assoc., to appear.