The Monty Python Method for Generating Random Variables

GEORGE MARSAGLIA
The Florida State University
and
WAI WAN TSANG
The University of Hong Kong

We suggest an interesting and fast method for generating normal, exponential, t, von Mises, and certain other important random variables used in Monte Carlo studies. The right half of a symmetric density is cut into pieces, then, using simple area-preserving transformations, reassembled into a rectangle from which the x-coordinate—or a linear function of the x-coordinate—of a random point provides the required variate. To illustrate the speed and simplicity of the Monty Python method, we provide a small C program, self-contained, for rapid generation of normal (Gaussian) variables. It is self-contained in the sense that required uniform variates are generated in-line, as pairs of 16-bit integers by means of the remarkable new multiply-with-carry method.

Categories and Subject Descriptors; G.3 [**Mathematics of Computing**]: Probability and Statistics; I.6.1 [**Simulation and Modeling**]: Simulation Theory

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Monty Python method, normal variates, t variates, von Mises variates

1. INTRODUCTION

Consider the rectangle in Figure 1, base $b = \sqrt{2\pi}$, height $1/b$, so the area is 1. You may generate a normal random variable by choosing a random point (x, y) in that rectangle, and then

(1) If the point is in region **F**, $(x < a)$, return $\pm x$. (Note: this step avoids generation of y.)
(2) If the point is in region **G**, $(y < f(x))$, return $\pm x$.
(3) If the point is in region **H**, $(y > g(x))$, return $\pm s \times (b - x)$.

Marsaglia's research was supported by the National Science Foundation.
Author's address: Department of Statistics, The Florida State University, Tallahassee, FL 32306; email: geo@stat.fsu.edu; W. W. Tsang, The University of Hong Kong.

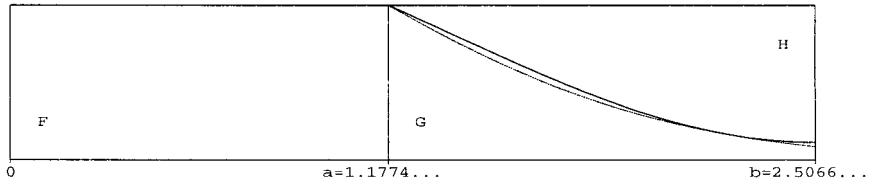

Fig. 1. The Monty Python method applied to the normal distribution.

(4) Else generate $x = -\ln U_1/b$, $y = -\ln U_2$ until $y + y > x \times x$, then return $\pm(b + x)$.

Here f is the absolute normal density; $f(x) = 2e^{-(1/2)x^2}/\sqrt{2\pi}$ on $a < x < b$; and g is the density from 0 to a, but rotated (and stretched) to fit into the upper right corner—the essence of the Monty Python method. Thus, $g(x) = 1/b - s[f(s(b - x)) - 1/b]$, with $s = a/(b - a)$, and a is the x-coordinate of the point where f crosses the rectangle: $a = \sqrt{\ln 4}$.

The Monty Python Method has the following features:

—Since $x < a$ ensures the random point (x, y) will be in region **F**, whatever y, we generate the required variate about half the time by means of one uniform variate, and a test on magnitude.

—The rest of the time, we do require a second uniform, $0 < y < 1/b$, and tests $y < f(x)$ or $y > g(x)$. But the need to evaluate these can be reduced to perhaps 2% by fitting a quadratic $q(x) = c_1 + x(c_2 + xc_3)$ below f so that, for $a < x < b$, $q(x) < f(x) < g(x) < q(x) + \epsilon$.

—Since the rectangle has base b and height $1/b$, its area is 1. Thus the area between f and g is the area of the right tail, $\int_b^\infty f(x)dx$, and step (4) of the algorithm provides a normal variate x from the tail $x > b$.

—The random $\pm$ can be provided in a single machine cycle from the sign bit of a random 32-bit integer, with the rightmost 31 bits used to provide a uniform variate. The integer random number generator should be in-line, without recourse to calling a subroutine or procedure. Then the time for generating a normal variable will be little more than the time for calling a random integer subroutine, since the bulk of the cost of such fast routines is the overhead for saving and restoring registers.

—The cost of a generating procedure is sometimes given in terms of the average number of uniform variates required. For the Monty Python Method, that average ranges from about 1.5 to 1.8, depending on the ratio a/b and the small frequency (ϵ) of the tail procedure. The formula is $2 - a/b + \epsilon T$, with T the number required for the tail.

2. THE MONTY PYTHON METHOD

As we shall see, the above method for normal variates may be applied to exponential, t, von Mises, and other important variates. Many methods have been developed in the past—see, for example, the comprehensive book by Devroye [1986]—and methods are usually given names to identify them.

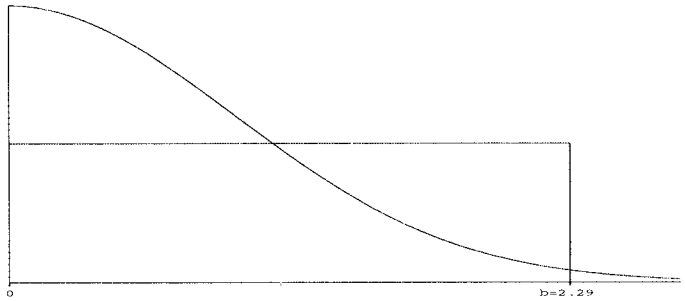

Fig. 2. The basic rectangle for applying the Monty Python method.

(I, Marsaglia, called this the Monty Python method when it was developed, some 20 years ago, because opening graphics on the British television show Monty Python's Flying Circus resembled the essential element of the method. The zany Monty Python crew pictured a stylized head with a hinged top that folded open, with all kinds of silliness pouring out. The Monty Python Method has an analogous hinged top that is folded over to suggest, among other things, an interesting way to generate a random variable. You may judge for yourself its silliness.)

The fastest method for sampling from decreasing densities is probably the ziggurat method of Marsaglia and Tsang [1984]. That method is related to the Monty Python method, but, for extreme speed, requires slicing the density horizontally into many pieces of equal area, with resulting complex programs and large arrays. The method described below provides programs that are nearly as fast as those of the ziggurat method, but simpler, with no arrays.

2.1 Getting Started

Given a decreasing density f on $0 < x$, the first step is to choose a base b, then draw a rectangle of area 1, so the height is $1/b$. The idea is to guess at a b for which the "cap," the region above the rectangle, seems to be an inverted version of the northeast corner of the rectangle.

For example, with $f(x) = 2e^{-(1/2)x^2}/\sqrt{2\pi}$ and $b = 2.29$ you have a picture such as Figure 2.

The x-coordinate of a point chosen uniformly under f will have density f, but of course it is difficult to choose a point uniformly under f in a direct way. But it is easy to choose (x, y) uniformly from a rectangle, and the idea of the Monty Python method is to choose the rectangle so that the area under f can be folded neatly into it, and in such a way that the resulting variate can be quickly provided by means of a few simple tests on magnitude.

We consider two ways for cutting and folding f into a rectangle: first, by cutting off the cap and putting it directly into the upper right corner, leading to the simplest but not the most efficient method, or second, by stretching the cap so as to better fit it into the upper right corner of the rectangle. In both cases, we are cutting f into pieces and reassembling them

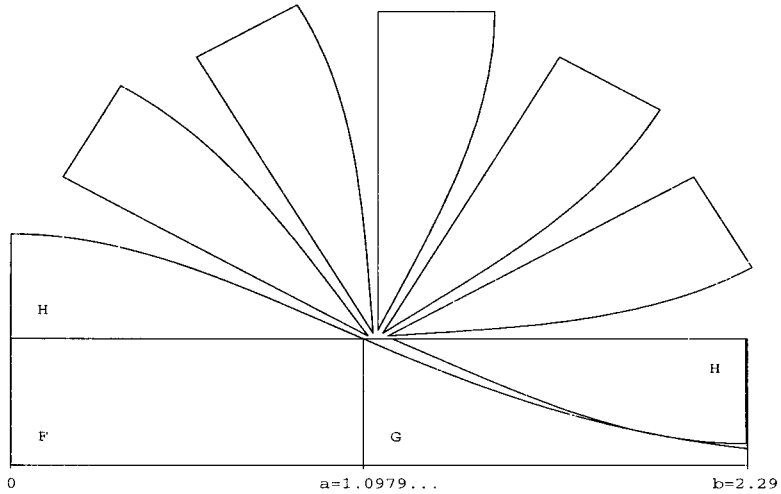

Fig. 3. Rotate and slide the cap to the northeast corner.

into a rectangle. The transformations are area-preserving, and except for the narrow tail region, simple enough that the required variate is a simple function of the x-coordinate of the random point (x, y).

For the simplest method, all we need do is rotate and slide the "cap" of the density to the upper right corner of the rectangle, as in Figure 3.

Then for $x < a$ return x; for (x, y) in **G**, return x; for (x, y) in **H**, return $b - x$. Of course we cannot completely fold the area under f into the rectangle—the tail is still hanging out. But the area of the tail is exactly what we have remaining to fill in the rectangle. So if our random point in the rectangle falls in that narrow space between f and g, we just return a variate from the normal tail by means the method of Marsaglia or the general tail method of Marsaglia and Tsang [1984].

The choice of b is not critical, but requires care. If b is too big, then the rotated and shifted cap will intersect the lower curve. If b is small, there is plenty of room for the cap, but then the tail method is required too often.

2.2 Rotating and Stretching

The choice $b = 2.29$ is about the maximum possible—unless one not only rotates, but stretches, the cap so that it fits exactly from a to b. The cap density is $f(x) - 1/b$, $0 < x < a$. To stretch it, we let $x = st$, with s the stretch factor, $s = a/(b - a)$. Then the density of t is $s[f(st) - 1/b]$, and the rotated, stretched cap leads to the test function $g(x) = 1/b - s[f(s(b - x)) - 1/b]$. Stretching and rotating the cap about the point $(a, 1/b)$ provides a figure such as Figure 4, (with each successive cap stretched by a factor of $s^{1/6}$):

Now we have regions **F**, **G**, and (stretched) **H** from which the required variate is easily provided, as either x or $s(b - x)$. The tail region is encountered with probability about 0.012, compared to 0.022 for the unstretched cap with $b = 2.29$. The choice $b = \sqrt{2\pi}$ for the stretched cap is

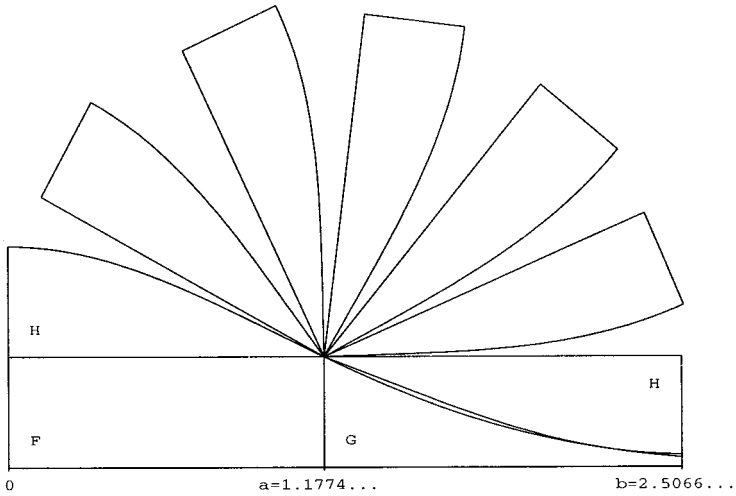

Fig. 4. Rotating and stretching the cap.

not critical. Any value in 2.506 to 2.5074 will do, but $b = \sqrt{2\pi}$ and the resulting $a = \sqrt{\ln 4}$ are easily identified choices to unlimited precision.

3. MONTY PYTHON FOR EXPONENTIAL-LIKE DENSITIES

The direct Monty Python method—draw a rectangle of area 1 and rotate and stretch the cap so that it neatly fits into the upper right corner of the rectangle—works well for normal, t, and von Mises densities. But it does not work well for exponential, or exponential-like, densities. If you try to rotate and stretch the cap, a too-large lenticular shape remains in the middle.

For example, on the left of this figure is about the best that can be done for the exponential density e^{-x} with rotate and stretch ($b = 2.3585$), while on the right is improvement arising from the exact-approximation transformation of the cap (Figure 5).

This is the idea. We want to change the shape of the cap so that, when rotated and stretched, it will better fit in the upper right corner and allow a larger value for b, providing a much smaller tail probability. Some simple transformation of the cap density is called for. We want to transform the cap of the density, $f(x) - f(a)$, $0 < x < a$, to a density on $0 < z < a$ by means of $x = C(z)$, with C a cubic, $C(z) = c_1 z + c_2 z^2 + c_3 z^3$. The density of z must be $d(z) = C'(z)[f(C(z)) - f(a)]$, $0 < z < a$ in order that $x = C(z)$ have density $f(x) - f(a)$, $0 < x < a$. This is the exact-approximation method of Marsaglia [1984].

We want $C(a) = a$, (so that x and z have the same range) and $sd(0) = f(a) - f(b)$ (to make the rotated, stretched d fit exactly at $x = b$).

Thus $c_1 + c_2 a + c_3 a^2 = 1$ and $c_1 s[f(0) - f(a)] = f(a) - f(b)$. This leaves one free parameter, say c_3, in determining C, with the hope that the resulting $d(x)$, when rotated and stretched about the point $(a, 1/b)$, will fit

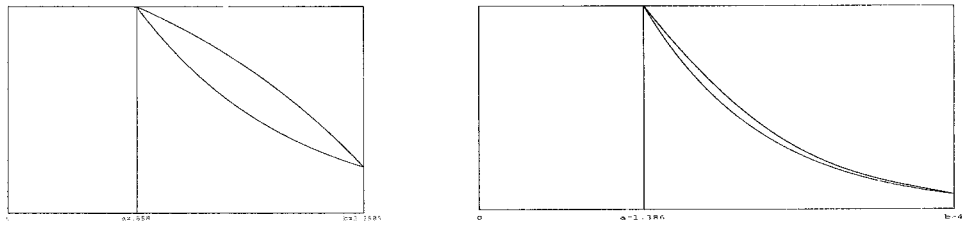

Fig. 5. Improvements for exponential-like densities.

nicely in the available space. The picture on the right shows the transformed cap resulting from $C(z) = 0.58242z + 0.16259z^2 + 0.1z^3$. When rotated about the point $(a, 1/b)$, and stretched by the factor $s = a/(b - a)$, the result fits nicely in the upper right corner, as indicated in Figure 5 above. There, the upper curve is $g(x) = 1/b - sd(s(b - x))$, with $d(z) = C'(z)[f(C(z)) - 1/b]$. For points (x, y) in the upper right region, the value $C(s(b - x))$ is returned—in practice, as a Horner-form cubic in x.

Next, Figure 6 shows the exponential density and the transformed cap density $d(z)$ arising from $x = C(z)$. The transformed cap may then be rotated and stretched to fit as shown in the right side of Figure 5.

4. MONTY PYTHON AND THE VON MISES DISTRIBUTION

The von Mises density is $f(x) = ce^{k \cos(x)}$, symmetric on $-\pi < x < \pi$. The constant is $c = [\pi I_0(k)]^{-1}$. (Twice the ordinary one, in order that the right half have area one.) We put a random $\pm$ on a variate from that right half. We want to choose b and the resulting a so that the rotated and stretched cap density just fits in the upper right corner of the rectangle with base $[0, b]$ and altitude $h = 1/b = f(a)$.

As with other applications of the Monty Python method, choosing b determines h, a, s, and g by $h = 1/b$, $f(a) = h$, $s = a/(b - a)$, and $g(x) = h - s[f(s(b - x)) - h]$. The problem is to choose b so that the resulting g puts the rotated and stretched cap into the northeast corner with no overlapping of f and g and the area between them very small. The "best" b is the largest b that makes this true.

In order that there be a maximal b, the set of b's for which $g(x) \geq f(x)$, $a \leq x \leq b$ must be shown to have at least one element. It seems preferable to argue that the set of a's for which $g(x) \geq f(x)$, with $b = 1/f(a)$, is not empty. We can arrange terms so that our condition becomes

$$d(x) = \frac{1}{b - a} - sf(s(b - x)) - f(x) \geq 0 \text{ for } a \leq x \leq b.$$

Now $d(a) = 0$ and $d'(x) = s^2 f'(b(s - x)) - f'(x)$. Furthermore, $d'(a) = (s^2 - 1)f'(a) > 0$. So our difference, $d(x)$, starts off at 0 and is rising as x "leaves" a. If a is small enough that $d'(x) \geq 0$ for $a \leq x \leq b$ then the

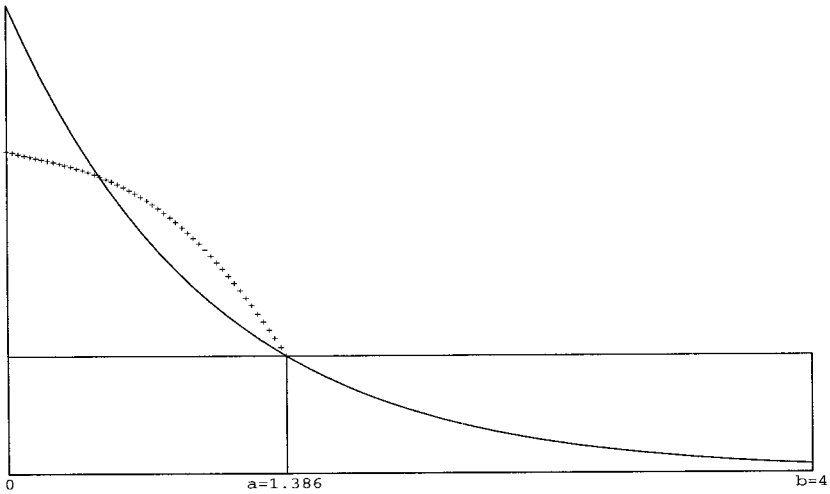

Fig. 6. The exponential density with the transformed cap.

condition $d(x) \geq 0$ for $a \leq x \leq b$ will be met. We may put the nonnegativity of $d'(x)$ in the form

$$\frac{s^2 f'(s(b-x))}{f'(x)} \leq 1,\ a \leq x \leq b.$$

Then, since $|f'(x)|$ is bounded from below on $a \leq x \leq b$, we can make a small enough that the condition holds.

Thus there are a's, and hence b's, for which $g(x) \geq f(x)$ for $a \leq x \leq b$, and we can be sure there is a best b. This should be clear from the graphical description of the Monty Python method. If a is very small, the cap is virtually a little triangle that gets rotated and stretched to the upper region of the rectangle. It starts equal to $f(x)$, but is farther and farther above it as x goes from a to b.

Finding maximal, or nearly maximal, b's is perhaps best done graphically, using, for example, Maple. One zeroes in on successive b's until the rotated and stretched cap fits with no overlap and the in-between area seems to be near its minimum. Good choices for b are shown in Figure 7, for $k = 0.5$ and $k = 4$.

Results from a representative set of k's can be used to develop a general formula for b as a function of k. The von Mises densities approach the normal as k grows, since $I(0, k) = (e^k/\sqrt{2\pi k})(1 + (1/k) + o(1/k^2))$. The asymptotic form for $I(0, k)$ puts a $-k$ in the exponent of e in the von Mises density, making it $k(\cos(x) - 1)$, with dominant term $-kx^2/2$, that of a normal density with variance $1/k$.

So we may expect a formula for b as a function of k in the form $b = (2.5074/\sqrt{k})(1 + a_1/k + a_2/k^2 + \cdots)$, and here is one, based on fitting a

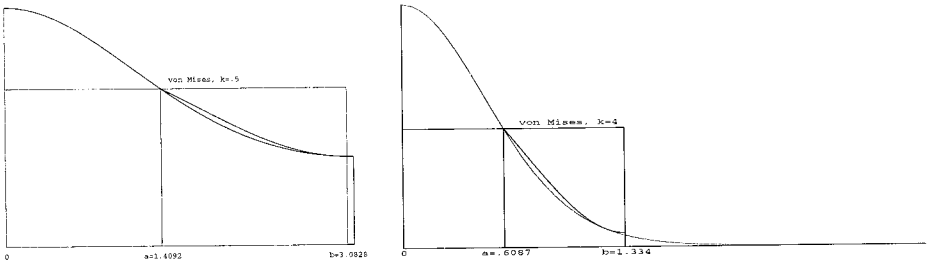

Fig. 7. The Monty Python method applied to two von Mises densities.

curve to b's for $k = 1, 2, 4, 8, 16, 32, 64$:

$$b = \frac{2.50715}{\sqrt{k}}\left(1 + \frac{0.19951}{k} - \frac{0.057465}{k^2} + \frac{1.80372}{k^3} - \frac{2.770225}{k^4} + \frac{0.9601616}{k^5}\right).$$

That "2.50715" can be recognized as a conservative estimate of the maximum possible for the normal, 2.5074 . . . .

The above formula for b is for $k \geq 1$. The von Mises densities approach the uniform distribution on $(0, \pi)$ as $k \to 0$. Plotting a few points for small k suggests a linear relation between $\ln(\pi - b)$ and $\ln(k)$, leading to conservative, yet accurate, b's by the formula

$$b = 3.14159 - 0.20086k^{2.0313} \text{ for } k < 1/8.$$

For the remaining range of k, fitting a curve to b values for $k = 1, 1/2, 1/4, 1/8$ leads to

$$b = 3.141388 + 0.00248k - 0.18047k^2 - 0.116028k^3, \text{ for } 1/8 \leq k \leq 1.$$

Note that the von Mises density has finite support, $-\pi < x < \pi$. So the "tail," extending from b to π, can be handled by a simple rejection technique when k is small, or the general tail method as described in Marsaglia and Tsang [1984] when k is large.

5. MONTY PYTHON AND THE t DISTRIBUTION

The t density is $f(x) = c(1 + x^2/n)^{-(n+1/2)}$, with $c = 2\Gamma(n + 1/2)/[\Gamma(n/2)\sqrt{n\pi}]$, doubled so the right half has area one.

To apply the Monty Python method, we must vary b until the resulting translated and stretched cap just fits in the northeast corner, and the tail area, between f and g, is small. Figure 8 shows good choices of b for $n = 8$ and $n = 4$.

For large n the average generating time is that for normals, very fast. But even for small n the resulting method is still very fast, with a slight

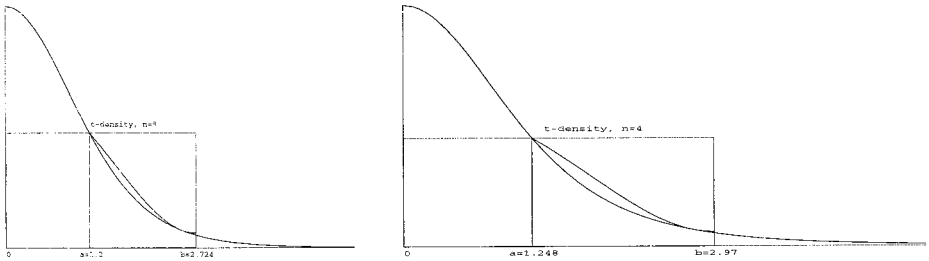

Fig. 8. The Monty Python method applied to two t densities.

loss, not so much because of increase in the tail area, but because the smaller ratio a/b means that a single uniform variate is returned perhaps 40% of the time, compared to the 48 to 50% for some densities.

The choices for b need not be precise. Thus it seems feasible to provide a formula that gives the (nearly) best b as a function of n. Here are suitable b's for $n = 1, 2, \ldots, 8$:

$$b_1 = 4.766,\ b_2 = 3.515,\ b_3 = 3.143,\ b_4 = 2.968,\ b_5 = 2.868,$$

$$b_6 = 2.783,\ b_7 = 2.756,\ b_8 = 2.724$$

For $n > 8$, use of $b_n = 2.5074 + 1.876n^{-1.042}$ will provide a conservative value quite close to the maximum possible. Its use, rather than the maximum possible, will have no measurable effect on the average running time of programs based on the Monty Python method.

6. PROGRAMMING EXAMPLE

We conclude with an example of a C program for generating a normal variable by means of the Monty Python method. Other distributions will have similar programs; only the functions f, g, the constants b, a, and the tail method change.

We illustrate with our practice of including an in-line random integer generator, to avoid the overhead of a subroutine (procedure) call—in this case, concatenation of a pair of multiply-with-carry generators on 16 bits. Theory and description for the multiply-with-carry method is in the Postscript file mwc1.ps of the Marsaglia Random Number CDROM [Marsaglia 1996]. The define statements provide a random znew in the top 16 bits of a 32-bit word, while wnew provides the bottom 16 bits. So the C expression znew+wnew produces a random 32-bit integer that passes all tests in the DIEHARD battery of tests [Marsaglia 1996]. Note that seed values for z and w, containing the initial 16-bit integers and the "carries," are set by setseed( ) with two 32-bit integer arguments.

```
#define znew     ((z=36969*(z&65535)+(z>>16))<<16)
#define wnew     ((w=18000*(w&65535)+(w>>16))&65535)
static unsigned long z=362436069, w=521288629;
float rnor( ){
```

```
float x,y,v;
x=((signed long)(znew+wnew))*1.167239e-9;
if(fabs(x)<1.17741) return(x);
y=(znew+wnew)*2.328306e-10;
if(log(y)<.6931472-.5*(x*x)) return(x);
x=(x>0)? .8857913*(2.506628-x) : -.8857913*(2.506628+x);
if(log(1.8857913-y) < .5718733-.5*(x*x) ) return(x);
   do{
   v=((signed long)(znew+wnew))*4.656613e-10;
   x=-log(fabs(v))*.3989423;
   y=-log((znew+wnew)*2.328306e-10);
     }
   while(y+y<x*x);
   return( (v>0? 2.506628+x : -2.506628-x );
        }
void setseed(unsigned long i1,unsigned long i2){z=i1; w=i2;}
```

In many systems, inserting instructions for pretests may make the procedure run faster: after producing y, set $v = 2.8658 - |x|(2.0213 - 0.3605|x|)$. Then if $y < v$ return x; if $y > v + 0.0506$ return $\pm 0.8857913(2.506628 - |x|)$. With such pretests, transcendental functions f or g will be needed only about once in 37 calls. On a 120MHz PC, pretests led to 1.9 μsecs per call to rnor(), while the average was 2.5 μsecs without pretests (for gnu C on Linux). A (Microsoft) Fortran version averaged 0.9 μsecs with pretests, 3.6 without.

Remark. Similar ideas were used in the (more general, but also more complicated) patchwork rejection technique which resulted in competitive gamma (see Minh [1988]) and beta generators (see Zechner and Stadlober [1993]).

REFERENCES

DEVROYE, L. 1986. *Non-Uniform Random Variate Generation,* Springer-Verlag, New York.

MARSAGLIA, G. 1984. The exact-approximation method for generating random variables. *J. Am. Stat. Assoc. 79*, 218–221.

MARSAGLIA, G. 1996. *The Marsaglia Random Number CDROM, with the DIEHARD Battery of Tests of Randomness.* Department of Statistics, Florida State University. Available by ftp through the Department of Computer Science, Univ. of Hong Kong: http://www.cs.hku.hk/ftp.html

MARSAGLIA, G. AND TSANG, W. W. 1984. A fast, easily implemented method for sampling from decreasing or symmetric unimodal density functions. *SIAM J. Sci. Stat. Comput. 5*, 349–359.

MINH, D. L. 1988. *Generating gamma variates. ACM Trans. Math. Softw. 14*, 261–266.

ZECHNER, H. AND STADLOBER, E. 1993. *Generating beta variates via patchwork rejection. Computing 50*, 1–18.

Received May 1997; accepted October 1997