Efficient Generation of Exponential and Normal Deviates

by

Herman Rubin
Purdue University

Brad C. Johnson
Purdue University

Technical Report #04–02

Department of Statistics
Purdue University
West Lafayette, IN USA

April 2004

∞

Efficient Generation of Exponential and Normal Deviates

Herman Rubin and Brad C. Johnson

Department of Statistics, Purdue University,
West Lafayette, IN 47907–1399, USA

Abstract

We present efficient procedures for generating random exponential and normal deviates based on the acceptance-complement method (Kronmal & Peterson, 1981). The proposed procedures maintain good precision and compare favorably with the Ziggurat method (Marsaglia & Tsang, 1984, 2000) for efficiency.

Keywords: Exponential deviates; Normal deviates; Acceptance-complement

1 Introduction

Two of the most important continuous distributions in probability and statistics are the exponential and normal distributions. Simulations frequently require large numbers of random deviates from these distributions and efficient methods for generating such deviates are of great interest.

Kronmal and Peterson (1981) introduced the *acceptance-compliment* (AC) method for generating random deviates from specified distributions (see also Kronmal & Peterson, 1984; Devroye, 1986). In the present paper, we show that this method can provide simple and efficient procedures for generating

random deviates with good precision from the standard exponential and normal distributions.

The remainder of this paper is organized as follows. Section 2 gives a brief description of the *acceptance-compliment* method and an implementation for bounded monotone densities. Sections 3 and 4 give the implementation details for the standard exponential and normal distributions respectively with some C code. In section 5 we present some timings and comparisons with the Ziggurat method (Marsaglia & Tsang, 1984, 2000). Section 6 contains some concluding remarks.

2 The Acceptance-Complement Method

The AC method can easily be described as follows. Suppose we wish to generate random deviates X from some distribution with density f with respect to some measure η (usually Lebesgue or counting measure). Decompose f into sub-densities f_1 and f_2 (i.e. $f_i \geq 0$ and $\int f_i \, d\eta > 0$ for $i = 1, 2$) so that $f = f_1 + f_2$ and choose a proper density g dominating f_1 (i.e. $g \geq f_1$ and $\int g \, d\eta = 1$). To generate a deviate with density f we perform the following two steps:

A1. Generate a deviate X with density g and output X with probability $f_1(X)/g(X)$. On rejection, goto **A2**.

A2. Replace X with a deviate from the conditional density $f_2 / \int f_2 \, d\eta$ and output X.

See (Kronmal & Peterson, 1981) for a proof of the validity of this method. Since g is a density, the rejection probability in step **A1** (i.e. the probability step **A2** is required) is simply

$$\int (g - f_1) \, d\eta = \int f_2 \, d\eta.$$

2.1 Bounded Monotone Densities

A frequently encountered special case occurs when f is continuous, bounded and non-increasing with support $[a_0, \infty)$. In this case a suitable g may be constructed as follows: Let n be a given positive integer and, for $i = 0, \ldots, n-1$, let $w_i = [nf(a_i)]^{-1}$ and $a_{i+1} = a_i + w_i$ and define the density

$$g(x) = \sum_{i=0}^{n-1} f(a_i)\mathbf{1}_{[a_i, a_{i+1})}(x). \tag{2.1}$$

Taking $f_1(x) = f(x)\mathbf{1}_{[a_0, a_n)}$ and $f_2(x) = f(x)\mathbf{1}_{[a_n, \infty)}$ we see that g dominates f_1 and $f = f_1 + f_2$ as required. Intuitively, g covers f_1 by a sequence of n equal-area rectangles with heights equal to the value of f_1 at their left end-points. Figure 1 shows the construction for the standard exponential density $f(x) = \exp(-x)\mathbf{1}_{[0,\infty)}(x)$ with $n = 32$.

A candidate deviate, X, from density g, is obtained by generating $I \sim \mathrm{DU}\{0, \ldots, n-1\}$ and $D \sim \mathrm{U}(0, w_I)$ and taking $X = a_I + D$. The precision of the resulting deviates (on $[a_0, a_n)$) will be at least $\log_2(2^b/w^*)$ where b is the number of bits used to form D and $w^* = \max_i w_i$. In implementation, it is convenient to take $n = 2^e$ and, from a single 32-bit random integer R, let I be the index formed from the least significant e bits of R and take $D = Rw_I 2^{-32}$.

3 Exponential Deviates

We first apply the AC method to the standard exponential density $f(x) = \exp(-x)\mathbf{1}_{[0,\infty)}(x)$. Let g, f_1 and f_2 be as defined in section 2.1 with $a_0 = 0$ and let $I \sim \mathrm{DU}\{0, \ldots, n-1\}$ and $D \sim \mathrm{U}(0, w_I)$ so that $X = a_I + D$ has density g. Then, if $U \sim \mathrm{U}(0, 1)$, we accept X if

$$U \leq \frac{f_1(X)}{g(X)} = \frac{\exp(-X)}{\exp(-a_I)} = \exp(-D).$$

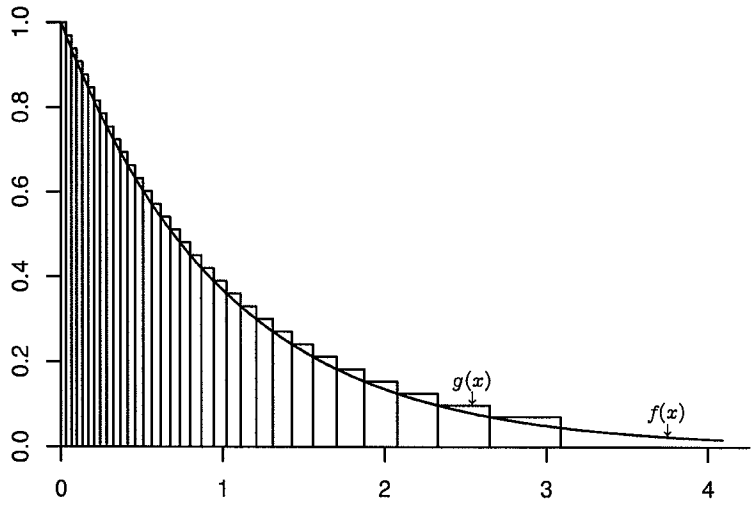

Figure 1: The density g as defined in (2.1) with $n = 32$ and $f(x) = \exp(-x)\mathbf{1}_{[0,\infty)}(x)$.

Taking the logarithm of both sides and negating, this is equivalent to accepting X if $T > D$ where $T \sim \text{Exp}(1)$. Using an exponential random variable for the test eliminates the computation of $f_1(X)/g(X)$. More importantly, when $T > D$, we can avoid the overhead of generating a new test exponential T for the next iteration of the algorithm since $T - D$ is an independent (of D) Exp(1) random deviate and we can set $T = T - D$. When $T \leq D$ (i.e. on rejection) we replace T with a new exponential deviate and generate a deviate in the $[a_n, \infty)$ tail of the exponential distribution. This is accomplished by letting $T = E_1$ and $X = a_n + E_2$ where E_1 and E_2 are obtained from an alternate source of exponential deviates. The entire algorithm is:

Algorithm E For a given positive integer n, Let $a_0 = 0$, for $i = 0, \ldots, n-1$ let $a_{i+1} = a_i + \exp(a_i)/n$ and $w_i = (a_{i+1} - a_i)$. Generate $T \sim \text{Exp}(1)$ from an alternate source of exponential deviates. Each time the following steps are executed a new (random) exponential deviate is generated.

E1. Generate $I \sim \text{DU}\{0, \ldots, n-1\}$ and $D \sim \text{U}(0, w_I)$

E2. [Accept?.] If $T > D$ set $T = T - D$, output $X = a_I + D$ and terminate; otherwise goto **E3**.

E3. [Replace.] Generate $E_1, E_2 \sim \text{Exp}(1)$ from an alternate source of exponential deviates, set $T = E_1$, output $X = a_n + E_2$ and terminate.

Comment: Since, conditional on $T > D$, $T - D$ and $a_I + D$ are independent standard exponential deviates, one could choose to set $X = T - D$, $T = a_I + D$ and output X in step **E2**.

The acceptance probability in the above algorithm is $1 - \exp(-a_n)$. For a

few values of n, these are:

n	8	16	32	64	128	256	512	1024
a_n	2.128	2.593	3.089	3.612	4.155	4.715	5.288	5.872
Pr(accept)	0.881	0.925	0.954	0.973	0.984	0.991	0.995	0.997
Precision	29.77	28.99	28.18	27.35	26.51	25.66	24.79	23.91

The precisions (in bits) reported above assume that I and D are formed from a single 32-bit random integer R by taking the least significant $\log_2 n$ bits for I and setting $D = Rw_I$.

3.1 An implementation

As it stands, algorithm **E** is already fairly efficient. When $n = 256$, for example, we require (on average) about 18 deviates from the alternate source of exponential deviates per 1000 iterations. This being said, the procedure can be significantly improved with the following two refinements:

i) Once T has been replaced in step **3**, the algorithm itself can be used recursively to generate the required tail deviate, thus reducing the number of deviates required from the (presumably slower) alternate source of exponential deviates by a factor of 2.

ii) A second implementation of the algorithm can be used as the source of alternate exponentials for the first, reducing even further the number of deviates required from the alternate source.

Figure 2 shows how these improvements might be implemented in C when $n = 256$. The numbers to the left of the line numbers give the average number of times the line is executed per generated exponential deviate. The following comments refer to the code:

1) The code assumes that `rand_ulong()` returns a random 32-bit unsigned integer. The macro `UNI` transforms these to uniform deviates on $(0, 1]$.

2) The function `rexp_init()` should be called before requesting any deviates. It initializes the array `ae[ne+1]` with the constants $a_0, \ldots, a_{\texttt{ne}}$ and the array `we[ne]` with the constants $w_0/2^{32}, \ldots, w_{\texttt{ne}-1}/2^{32}$. It also initializes the test exponentials `Te` and `T1`.

3) The function `rexp(double *buf, int N)` fills `buf` with `N` exponential deviates and returns a pointer to the last element inserted. When required, it uses `rexp_alt()` for its alternate source of exponentials. By filling a buffer, we avoid the overhead of a function call for each generated deviate and the need to declare variables in the global name space.

4) The function `rexp_alt()` serves as an alternate source of exponentials for `rexp`. When required, it uses the inverse CDF method for replacing the test exponential `T1` and then uses recursion to output a tail deviate.

4 Standard Normal Deviates

Generation of standard normal deviates proceeds similarly. Let ϕ and Φ denote the density and CDF of the standard normal distribution respectively and, for $\xi \in \mathbb{R}$, let

$$\phi_\xi(x) = \frac{\phi(x)}{1 - \Phi(\xi)} \mathbf{1}_{[\xi,\infty)}(x)$$

denote the density of X given $X > \xi$ where X is standard normal. Thus, ϕ_0 is the positive standard half-normal distribution. Taking g as defined in section 2.1 with $a_0 = 0$ and $f(x) = f_1(x) + f_2(x)$ where $f_1(x) = \phi_0(x)$ and $f_2(x) = \phi_{a_n}(x)$, let $I \sim \mathrm{DU}\{0, \ldots, n-1\}$ and $D \sim \mathrm{U}(0, w_I)$ so that $X = a_I + D$ has density g. Then, if $U \sim \mathrm{U}(0,1)$, we accept X if

$$U < \frac{f_1(X)}{g(X)} = \frac{\exp(X^2/2)}{\exp(-a_I^2/2)} = \exp(-X^2/2 + a_I^2/2).$$

```
         1  #include <math.h>
         2  #define two32i  0.23283064365386962890E-9   /* 2^(-32) as double */
         3  #define UNI     (two32i*rand_ulong()+two32i) /* Uniform(0,1]      */
         4  #define ne      256                          /* table size        */
         5  #define ne_mask 255                          /* index mask        */
         6  static double Te, T1, we[ne], ae[ne+1];
         7  double rexp_alt(void) {
         8    register unsigned long R,I; register double D;
0.01809  9    R=rand_ulong(); I=R&ne_mask; D=we[I]*R;
0.01809 10    if(T1<D)
0.00016 11      { T1=-log(UNI); return ae[ne]+rexp_alt(); }
        12    else
0.01793 13      { T1-=D; return ae[I]+D; }
        14  }
        15  double *rexp( double *buf, int N ) {
        16    register int j; register unsigned long R,I; register double T=Te;
        17    for(j=0; j<N; j++) {
1.00000 18      R=rand_ulong(); I=R&ne_mask; buf[j]=we[I]*R;
1.00000 19      if(T<buf[j])
0.00896 20        { T=rexp_alt(); buf[j] = ae[ne]+rexp_alt(); }
        21      else
0.99104 22        { T-=buf[j]; buf[j]+=ae[I]; }
        23    }
        24    Te = T;
        25    return &buf[N-1];
        26  }
        27  void rexp_init( void ) {
        28    register int i;
        29    for(ae[0]=0.0, i=0; i < ne; i++) {
        30      ae[i+1] = ae[i] + exp( ae[i] )/ne;
        31      we[i] = (ae[i+1] - ae[i])*two32i;
        32    }
        33    T1 = -log(UNI);
        34    Te = rexp_alt();
        35  }
```

Figure 2: A possible C implementation of algorithm **E** with improvements

If $T \sim \text{Exp}(2)$, this is equivalent to accepting X if $T > X^2 - a_I^2$. As before, on the event $T > X^2 - a_I^2$ we have $T - X^2 + a_I^2 \sim \text{Exp}(2)$ and a new exponential test deviate need not be generated on each iteration of the algorithm. On the event $T \leq X^2 - a_I^2$ we replace T with a new Exp(2) deviate and generate a deviate with density ϕ_{a_n}. The algorithm becomes:

Algorithm N For a given positive integer n, let $a_0 = 0$ and, for $k = 0, \ldots, n-1$, let $a_{k+1} = a_k + 1/n\phi_0(a_k)$ and $w_k = a_{k+1} - a_k$. Generate $T \sim \text{Exp}(2)$.

N1. Generate $I \sim \text{DU}\{0, \ldots, n-1\}$, $D \sim U(0, w_I)$ and $S \sim \text{DU}\{-1, 1\}$. Set $X = a_I + D$ and $T = T - X^2 + a_I^2$.

N2. [Accept?] If $T > 0$ output $S \cdot X$ and terminate, otherwise goto **N3**.

N3. [Replace T.] Generate a new $T \sim \text{Exp}(2)$.

N4. [Generate Tail Deviate.] Generate a deviate X with density ϕ_{a_n} and output $S \cdot X$.

The acceptance probability in the above algorithm can be calculated as $1 - 2\Phi(-a_n)$ and, for a few values of n, these are:

n	8	16	32	64	128	256	512	1024
a_n	1.746	1.940	2.135	2.328	2.518	2.703	2.883	3.058
Pr(accept)	0.919	0.948	0.967	0.980	0.988	0.993	0.996	0.998
Precision	29.35	28.74	28.08	27.38	26.66	25.91	25.14	24.35

The precision reported is the precision (in bits) of the deviates accepted in step **N2** of algorithm **N** (i.e. on $(-a_n, a_n)$). As we will see below, the precision of the tail deviates will be at least as good as the precision of the exponential deviates.

4.1 Generating tail deviates.

In order to complete the algorithm, we need an efficient method for generating deviates with density ϕ_ξ. We now describe an accept-reject algorithm for this purpose.

Let $Y \sim \text{Exp}(1)$, let $q = (\xi/2)+\sqrt{(\xi/2)^2+1}$ and consider the transformation $X = q + (Y-1)/q$ so that X has density

$$g(x) = q\exp(q^2 - qx - 1)\mathbf{1}_{[\xi,\infty)}(x).$$

Then

$$\frac{\phi_\xi(x)}{g(x)} = c\exp\left(-\frac{(x-q)^2}{2}\right) \quad \text{where} \quad c = \frac{\exp(1-q^2/2)}{q\Phi(-\xi)\sqrt{2\pi}}.$$

Whence, $\phi_\xi(x) \le cg(x)$ (and $\phi_\xi(q) = cg(q)$) and, by the same arguments as before, we let $T \sim \text{Exp}(2)$ and accept X if

$$T > (X-q)^2 = \left(\frac{Y-1}{q}\right)^2.$$

This leads to the following algorithm for generating a deviate from the density $\phi_\xi(x)$:

Algorithm TN Generate $T \sim \text{Exp}(2)$ and let $q = (\xi/2) + \sqrt{(\xi/2)^2+1}$.

TN1. Generate $Y \sim \text{Exp}(1)$ and set $U = (Y-1)/q$.

TN2. [Accept?] If $T > U^2$ set $T = T - U^2$, output $X = q + U$ and terminate.

TN3. [Reject] Generate $T \sim \text{Exp}(2)$ and goto **TN1**.

For a given ξ, the acceptance probability is given by $p_\xi = c^{-1}$ and the average number of exponential deviates required per iteration is $\mu_\xi = 2c - 1$. For step

N4 of algorithm **N** we use $\xi = a_n$ and for, for a few values of n we have

n	1	4	8	16	32	64	128	256	512
ξ	1.253	1.559	1.746	1.940	2.135	2.328	2.518	2.703	2.883
p_ξ	0.895	0.914	0.923	0.931	0.939	0.945	0.950	0.955	0.959
μ_ξ	1.234	1.189	1.167	1.148	1.131	1.117	1.105	1.095	1.086

Since $Y \sim \text{Exp}(1)$ and $q \geq 1$ the tail deviates will have at least the same precision as Y.

4.2 Implementation

As in the exponential case, the implementation is straight forward. Figure 3 shows one possible implementation which makes use of `rexp()` from Figure 2 as the source of exponential deviates. The following comments refer to the code:

1) The function `rnorm_init` must be called before requesting any deviates. This initializes the arrays `an[nn+1]` with the constants $a_0, \ldots, a_{\texttt{nn}}$; the array `an2[nn]` with the constants $a_0^2, \ldots, a_{\texttt{nn}-1}^2$; and the array `wn[nn]` with the constants $w_0/2^{32}, \ldots, w_{\texttt{nn}-1}/2^{32}$. It also initializes the test exponential `Tn` and the constants `qn` $= q$ and `qn_inv` $= q^{-1}$.

2) The function `rnorm_tail` implements the algorithm **T** given the constants q (`qn`) and q^{-1} (`qn_inv`).

3) The function `rnorm` fills the buffer `buf` with N normal deviates using algorithm **N**.

4.3 Variation of algorithm N

It turns out that algorithm **N** is quite efficient even when $n = 1$ since the many of the memory accesses and operations may be avoided. The following variation of algorithm **N** shows how this is accomplished.

```
#include <math.h>
#include "rexp.h" /* include rexp interface */
#define two32i 0.23283064365386962890625E-9  /* 2^(-32) as double */
#define phi0   0.79788456080286535588        /* sqrt(2/pi)        */
#define nn     256
#define nn2    255
static double Tn, qn, qn_inv, wn[nn], an2[nn], an[nn+1];
double rnorm_tail( register double qn, register double qn_inv ) {
   register double X; double E[2];
   do { (void) rexp_fill(E,2);
     X = qn_inv*E[0] - qn_inv; Tn = 2.*E[1] - X*X; } while( Tn<0 );
   return X+=qn;
}
double *rnorm( double *buf, int N) {
  register int i;
  register unsigned long R,I; register double D;
  for(i=0; i<N; i++) {
     R=rand_ulong(); I=R&nn2; buf[i]=wn[I]*R+an[I]; D=buf[i]*buf[i]-an2[I];
     if( Tn<D ) buf[i]=rnorm_tail(qn, qn_inv); else Tn-=D;
     if( R&nn ) buf[i] = -buf[i];
  }
  return &buf[N-1];
}
void rnorm_init( void ) { /* Initialize tables for normal */
  register int i;  double m = 1.0/(phi0*nn);
  an[0] = an2[0] = 0;
  for(i=0; i < nn; i++) {
    an2[i]  = an[i]*an[i];
    an[i+1] = an[i] + exp(.5*an2[i] )*m;
    wn[i]   = (an[i+1]-an[i])*two32i;
  }
  qn = an[nn]/2. + sqrt(an[nn]*an[nn]/4.+1.);
  qn_inv = 1.0/qn;
  Tn = 2.*rexp_alt();
}
```

Figure 3: A possible C implementation of algorithm N

Algorithm NV Define $q = \sqrt{\pi/8} + \sqrt{\pi/8 + 1}$, $v = 1/q$ and $u = \sqrt{\pi/2} * 2^{-32}$ and generate $T \sim \text{Exp}(2)$.

NV1. Generate a random 32-bit integer R and let $S = \pm 1$ depending on the least significant bit of R. Set $X = u * R$ and $T = T - X * X$.

NV2. [Accept?.] If $T > 0$ output $S * X$ and terminate; otherwise goto **NV3**.

NV3. [Tail.] Generate $E_1, E_2 \sim \text{Exp}(1)$, set $X = v * E_1 - v$ and $T = 2 * E_2 - X * X$.

NV4. If $T > 0$ output $S(X + q)$ and terminate; otherwise goto step **NV3.**.

5 Timings

The time it takes for a particular algorithm to produce deviates from a given distribution is one measure of its usefulness (others include: ease of implementation, precision, and memory requirements). In general, these timings can be quite sensitive to the specifics of implementation, including the amount of code and the memory requirements of the calling program(s). This being said, we compare the timings for the exponential and normal generators proposed in this paper (`rexp` and `rnorm` respectively) with similarly implemented versions of the Ziggurat generators proposed in (Marsaglia & Tsang, 2000) (`zexp` and `znorm`). For comparison, we also provide the timings for the inverse CDF exponential generator ($-\ln(U)$). Each of these generators were tested with 3 different underlying pseudo-random number generators: the `SHR3` generator found in (Marsaglia & Tsang, 2000); the `XORWOW` generator proposed in (Marsaglia, 2003); and the Mersenne Twister generator proposed in (Matsumoto & Nishimura, 1998).

Each generator was tested on two platforms: an IBM F50 server with 4 326MHz R6000 CPU's running AIX, and a 933Mhz Intel Mobile Pentium III

Table 1: Comparative generator timings (in seconds)

	Uniform Generator		
Platform/Generator	SHR3	XORWOW	Mersenne Twister
Windows/GCC			
rexp	40.3	48.2	69.6
zexp	41.1	45.2	69.3
rnorm	61.1	64.6	93.9
znorm	39.4	40.8	75.1
$-\ln(U)$	261.5	261.5	282.7
AIX/GCC			
rexp	103.2	117.8	196.3
zexp	99.0	133.0	195.0
rnorm	134.4	163.1	240.3
znorm	103.6	139.4	203.6
$-\ln(U)$	404.1	413.5	501.9
AIX/CC			
rexp	104.9	119.0	190.5
zexp	100.0	155.7	189.6
rnorm	128.7	138.7	212.8
znorm	103.7	134.0	197.7
$-\ln(U)$	384.9	392.4	491.5

running Microsoft Windows XP. Compilers used on the AIX system included both the GNU `gcc` and the AIX `cc` compilers. The GNU/Cygwin `gcc` compiler was used on the Windows system. The following compiler flags were used:

```
cc -O3 -qinline

gcc -O3 -finline-functions
```

For each test a 10^3 element buffer was filled 10^6 times for a total of 10^9 deviates generated per test. Table 1 presents the average timings (in seconds) of five test runs for each generator/platform/compiler combination.

As expected, the AC and Ziggurat generators (both exponential and normal) ran significantly faster than the inverse CDF generator. The AC exponential generator (`rexp`) and the Ziggurat exponential generator (`zexp`) are fairly even in terms of timings. The `rnorm` generator was slower than the `znorm` generator

on all platform/uniform generator combinations ranging from only 3% slower on the AIX/GCC platform using the `XORWOW` generator to about 58% slower on the Windows/GCC platform using the `XORWOW` generator.

6 Conclusions

The AC algorithm, combined with the reuse of exponential deviates and an efficient dominating density results in very efficient procedures for generating exponential and normal deviates which are easy to set up and implement. The proposed generators also maintain good precision over the entire supports of the densities in question (> 25 bits as implemented here). When greater precision is required one may use two 32-bit integers, say R_1 and R_2, form I from the least significant bits of R_2 and take $D = R_1/2^{32} + R_2/2^{64}$ (of course, one could exploit the floating point representation used by the compiler/architecture to accomplish this efficiently).

These methods can also be used for vector and parallel machines. There are many different types of vector machines, on some of which different "tricks" can be used, so let us look at the procedure up to the point of acceptance-rejection for a vector machine. We have a vector of input indices $\mathbf{I}$ and a vector of input uniform random variables $\mathbf{U}$ (these can be obtained from an original vector by decomposition at run time), and the jth candidate random variable is $\mathbf{A}[I[j]] + \mathbf{U}[j] * \mathbf{W}[I[j]]$. We also have a vector of test exponentials $\mathbf{T}$, and the test is to form $\mathbf{T}[i] - f(\mathbf{A}[\mathbf{I}[j]], \mathbf{U}[j] * \mathbf{W}[\mathbf{I}[j]])$, and set this to be the new value of $\mathbf{T}[j]$. If $\mathbf{T}[j] < 0$, this leads to rejection.

At this point, one method (perhaps the best for SIMD machines) is to just store the j's for which rejection occurs in an array. One can then use the correction methods for the elements pointed to by that array and store the results in the appropriate places.

References

Devroye, L. (1986). *Non-uniform random variate generation.* New York, New York: Springer-Verlag.

Kronmal, R. A., & Peterson, A. V. (1981). A variant of the accept-rejection method for computer generation of random variables. *Journal of the American Statistical Association, 76*, 446–451.

Kronmal, R. A., & Peterson, A. V. (1984). An acceptance-complement analogue of the mixture-plus-acceptance-rejection method for generating random variables. *ACM Transactions on Mathematical Software, 10*, 271–281.

Marsaglia, G. (2003). Xorshift RNGs. *Journal of Statistical Software, 8*, issue 14, 1–6.

Marsaglia, G., & Tsang, W. W. (1984). A fast, easily implemented method for sampling from decreasing or symmetric unimodal density functions. *SIAM Journ. Scient. and Statis. Computing, 5*, 349–359.

Marsaglia, G., & Tsang, W. W. (2000). The ziggurat method for generating random variables. *Journal of Statistical Software, 5*, issue 8, 1–7.

Matsumoto, M., & Nishimura, T. (1998). Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation, 8*(1), 3–30.