The Ziggurat Method for Generating Random Variables

George Marsaglia*
The Florida State University
and
Wai Wan Tsang
The University of Hong Kong

Abstract

We provide a new version of our ziggurat method for generating a random variable from a given decreasing density. It is faster and simpler than the original, and will produce, for example, normal or exponential variates at the rate of 15 million per second with a C version on a 400MHz PC. It uses two tables, integers k_i and reals w_i. Some 99% of the time, the required x is produced by: Generate a random 32-bit integer j and let i be the index formed from the rightmost 8 bits of j. If $j < k_i$ return $x = j \times w_i$.

We illustrate with C code that provides for inline generation of both normal and exponential variables, with a short procedure for setting up the necessary tables.

1 Introduction

In the early 1980's we developed the ziggurat method for sampling from decreasing densities, Marsaglia and Tsang [9]. The method was based on covering the target density with a set of horizontal equal-area rectangles, a 'cap' and a tail. The ziggurat appellation came from the appearance of the layered rectangles. A uniform point (x, y) from a randomly chosen rectangle provided the required variate x, most of the time after two table fetches and a test on magnitude.

But the original ziggurat method had a rather complicated procedure for providing the required x for the rare cases when the simple test on magnitude failed. We show here how to form the covering rectangles so that a simpler procedure may be used for the rare cases, and no 'cap' is necessary.

The idea of cutting a density into many small pieces, then choosing and sampling from one of them, goes back to the early 60's, Marsaglia [4]. Particular methods for normal and exponential variates were described in [6,7,8], and they were, for many years, fast and widely used standard generators for many platforms. Many of the details are spelled out in successive editions of Knuth's Volume 2,[2].

Rather than cutting the density into easily handled pieces—in effect, expressing the density as a mixture of very simple densities and a complicated residual density, the ziggurat approach covers the target density with the union of a collection of sets from which it is easy to choose uniform points, then uses the rejection method.

While the method and the set up were complicated, the original ziggurat method [9] led to some of the fastest methods for normal, exponential and other decreasing densities. Our goal here is to provide a version that is faster and simpler. Methods for normal variables by Ahrens and Dieter [1] and Leva [3] are said to be fast; we provide comparisons with those methods.

Details of the new ziggurat method are in Section 2, then Section 3 provides methods for setting up the ziggurat for table size 256 or 128; table size is best chosen a power of 2, to make random selection from the table easy: a 7 or 8-bit segment of a 32-bit random integer.

Section 4 gives specific implementations for normal and exponential variates, Section 5 provides some time comparisons and Section 6 summarizes.

*Research supported by the National Science Foundation

2 The Ziggurat Method

We begin with a description of the basic rejection method: Let $\mathcal{C}$ be the set of points (x, y) under a plot of the curve $y = f(x)$ with finite area. Let $\mathcal{Z}$ be a set of points containing $\mathcal{C} : \mathcal{Z} \supset \mathcal{C}$. The basic idea of the rejection method is: Choose random points (x, y) uniformly from $\mathcal{Z}$ until you get one that falls in $\mathcal{C}$, then return x as the required variate. The density of such an x will be $cf(x)$, with c the normalizing constant that makes $cf(x)$ a proper density. (Dealing with an unscaled f allows us to ignore the nuisance constants that are part of many densities.)

Three of the main criteria for choosing the covering set $\mathcal{Z}$ are:
1) Make it easy and fast to select a random point (x, y) from $\mathcal{Z}$;
2) Make it easy and fast to decide whether the random (x, y) from $\mathcal{Z}$ also falls in $\mathcal{C}$;
3) Make area($\mathcal{C}$)/area($\mathcal{Z}$), the efficiency of the rejection procedure, close to 1.

The ziggurat method for choosing the covering set $\mathcal{Z}$ comes closer to meeting all of these criteria than does any other general method we are aware of. A crude version of the ziggurat method is pictured in Figure 1:

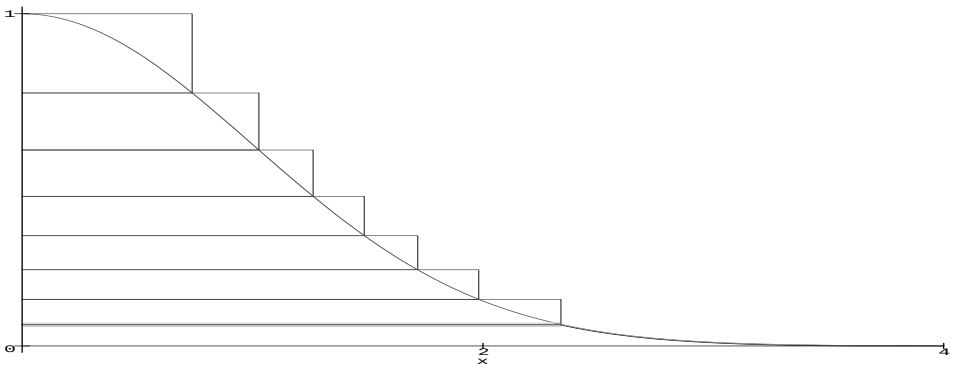

Figure 1: The ziggurat method with 7 rectangles and a bottom, base strip.

Here, $\mathcal{C}$ is the region under the curve $y = f(x) = e^{-x^2/2}, x > 0$, and $\mathcal{Z}$ is the union of 8 sets, 7 rectangles and a bottom strip tailing off to infinity. (We have chosen only 8 sets for clarity; in practice we choose 64, 128 or 256 sets whose union is $\mathcal{Z}$.) All 8 sets in the pictured $\mathcal{Z}$ have the same area, v, so it is easy to choose one of the sets at random. Furthermore, 7 of the 8 sets are rectangles from which it is easy to get a random point (x, y), and further yet, for those rectangles, it is easy to decide if (x, y) is in C: If the rightmost coordinates of the rectangles are $0 = x_0 < x_1 < x_2 < \cdots < x_7$, and rectangle R_i is selected, $i > 0$, then the x-coordinate of a random point in R_i is Ux_i with U uniform (0,1), and if $x < x_{i-1}$ then the random point (x, y) must be in $\mathcal{C}$, confirming the acceptance of x without having to generate y. (Assume rectangle R_0 is empty with right edge $x_0 = 0$, the left edge of R_1.)

And finally, it is easy to get a random point (x, y) from the base strip, as the base is itself a rectangle adjoined to the tail, $x > x_7$. Let r be the rightmost x_i, so that the tail is $r < x$. We may generate from the base strip as follows: generate $x = vU/f(r)$, with U uniform (0,1). If $x < r$ return x, else return an x from the tail. That provides an x from the base rectangle with the required probability: $rf(r)/v$. Note that code to provide from the rectangle part of the base strip can have the same form as that for the other rectangles: Form x and return that x after a succesful test on magnitude.

For the normal tail, the method of Marsaglia [5] provides: generate $x = -\ln(U_1)/r,\ y = -\ln(U_2)$ until $y + y > x \times x$, then return $r + x$. For the exponential tail, of course, $x = r - \ln(U)$.

For our fastest application, we form the ziggurat with layers of 255 rectangles with common area v, and a bottom strip of area v. The rectangles end at $x_1 < x_2 < \cdots < x_{255} = r$. Assume those x's have been determined,as described in the next section.

Since, as in most applications, we provide uniform (0,1) variates U by floating a random integer (32-bit unsigned long), we may save time by incorporating the float operation into the step that forms the x's that are to be returned: For each value of the index i in $1 \le i \le 255$, form the 32-bit integer $k_i = \lfloor 2^{32}(x_{i-1}/x_i) \rfloor$, and set $w_i = .5^{32}x_i$. For the special index $i = 0$, set $k_0 = \lfloor 2^{32}rf(r)/v \rfloor$ and $w_0 = .5^{32}v/f(r)$.

We may then describe the entire procedure with these simple steps:

The Ziggurat Algorithm

1. Generate a random 32-bit integer j and let i be the index provided by the rightmost 8 bits of j.
2. Set $x = jw_i$. If $j < k_i$ return x.
3. If $i = 0$ return an x from the tail.
4. If $[f(x_{i-1}) - f(x_i)]U < f(x) - f(x_i)$, return x.
5. Go to step 1.

3 Setting up the ziggurat

Given the (unscaled) target density $f(x), x \geq 0$, we want to find 255 (or 127, 63, etc.) equal-area rectangles, such as pictured in Figure 1, so that the covering set $\mathcal{Z}$ is very close to $\mathcal{C}$.

A picture of a ziggurat with some of its 255 rectangles is in Figure 2, for the exponential density. The rectangles are so closely layered that images blur in the lower part if all 255 are shown. So, after the first 20, only rectangles $R_{30}, R_{40}, \ldots, R_{250}$ and the final R_{255} are shown. Those few will give an idea of how closely $\mathcal{Z}$ covers $\mathcal{C}$, and the closeness of x_{i-1}/x_i to unity for almost all of the rectangles.

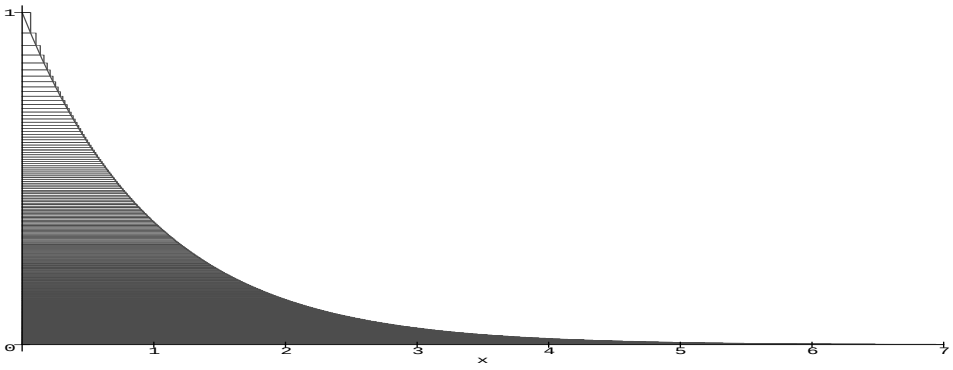

Figure 2: The ziggurat, showing every 10′th after the first 20 of 255 rectangles.

The rectangles must be chosen so that their common area, say v, has the same value as that of the final, base, region. How is this done?

The right edges of the rectangles are at $0 = x_0 < x_1 < x_2 < \cdots < x_{255}$. From the figure, it is clear that we must have, using r to designate the rightmost end point, $r = x_{255}$:

$$x_i[f(x_{i-1}) - f(x_i)] = v, \text{ for } i = 1, 2, \ldots, 255 \text{ with } v = rf(r) + \int_r^\infty f(x)\,dx.$$

Even if v is given, it is not easy to find the necessary x's. Instead, a feasible procedure is this: define a function of r, the rightmost end point, say $z(r)$, by a sequence of Maple-like commands:

$z(r)$: $x_{255} = r$; $v = rf(r) + \int_r^\infty f(x)\,dx$;

for i **from** 254 **by** -1 **to** 1 **do** $x_i = f^{-1}(v/x_{i+1} + f(x_{i+1}))$ **od** ; **return** $(v - x_1 + x_1 f(x_1))$;

Then the problem is to find the value of r that makes $z(r) = 0$. For $f(x) = e^{-x^2/2}$ the choice $r = 3.6541528853610088$ will make $z(r) = 0$, and then from that value for x_{255} the other $x's$ may be found, from the relation $x_i = f^{-1}(v/x_{i+1} + f(x_{i+1}))$, a procedure that requires inverting f.

The common area of the rectangles and the base turns out to be $v = .00492867323399$, making the efficiency of the rejection procedure 99.33%.

For $f(x) = e^{-x}$, the exponential density, $r = 7.69711747013104972$ makes $z(r) = 0$ and will assign the proper value to x_{255}, from which the other x's follow. The common area of the rectangles turns out to be $v = .003949659822581557 1993$, making the efficiency of the rejection procedure 98.9%.

For those who may want to use a table of 127 x's: for the (half-) normal density, the value $x_{127} = 3.442619855899$ will serve to find v and the other x's; the efficiency is then 98.78%. For the exponential, $x_{127} = 6.898315116616$, for an efficiency of 97.98%. With memory so readily available, there seems little reason for choosing 127 rather than 255 x's, but we will use 127 for the normal distribution in the implementation below, because we have to accomodate $x < 0$ as well as $x > 0$.

Furthermore, with memory so cheap, it is feasible to speed up the generating process even more: form an auxiliary table of integers, $k_i = \lfloor 2^{32}(x_{i-1}/x_i) \rfloor$, and, rather than storing a table of x's, store w_i as $.5^{32}x_i$. Then the fast part of the generating procedure is: Generate j. Form i from the last 8 bits of j. If $j < k_i$, return $x = jw_i$. (Special values $k_0 = \lfloor 2^{32}rf(r)/v \rfloor$ and $w_0 = .5^{32}v/f(r)$ provide for the fast part when $i = 0$.)

4 Implementation

It turns out that storing a third table, $f_i = f(x_i)$ is no more costly in terms of the final size of the compiled program than is a more complicated version that computes the $f(x_i)$'s. Assuming that table, the essential part of the generating procedure in C looks like this, assuming the variables have been declared, static tables k[256],w[256] and f[256] are setup, and `SHR3, UNI` are inline generators of unsigned longs or uniform (0,1) variates, using the `#define` statements given below. The result is a remarkably simple generating procedure, using a 32-bit integer generator such as SHR3, described below:

```
for(;;){
j=SHR3; i=(j&255);
x=j*w[i]; if(j<k[i]) return x;
if(i==0) return x from the tail.
if( UNI*(f[i-1]-f[i]) < f(x)-f[i] ) return x;
      }
```

The infinite '`for`' is executed until one of the three `return`'s provides the required x (better than 99% of the time from the first `return`). The method works for any decreasing density, conveniently scaled as $f(x)$, for which a tail method can be provided. For symmetric densities, the right half is used and a random $\pm$ attached. The procedures differ only in that different f's are used in the third return; three tables `k[256],w[256],f[256]` need be constructed from that function and different tail methods need be provided.

Finally, before giving a C implementation that combines both an exponential and a normal generator, we point out a feature of C, the `#define` statement, that provides inline access to a sequence of expression evaluations, saving the overhead costs of calls to a procedure. For the ziggurat method, as we have outlined it, the required variate is returned some 99% of the time after two table fetches and a test on magnitude. If these steps could be done inline, the overall average running time would be reduced. This can be done—if appropriate `#define` statements are used.

First, with all integers 32-bit unsigned long, one needs a `#define` statement that provides an inline random number generator:

```
#define SHR3 (jz=jsr, jsr^=(jsr<<13), jsr^=(jsr>>17), jsr^=(jsr<<5),jz+jsr)
```

then the following `#define` statement provides inline generation of the fast part of the ziggurat method for exponential variates:

```
#define REXP (j=SHR3, i=(j&255), (j<kz[i] ? j*wz[i] : efix()))
```

Here `efix` needs to be a procedure that returns a variate from the tail if `i=0`, or generates y in $f(x_{i-1}) < y < f(x_i)$ and returns $x = j * wz[i]$ if $y < f(x)$, or else starts all over. And `jsr,i,j` need to be `static` variables. To avoid possible confusion with the commonly used variable names `i` and `j`, we designate them `iz,jz` in the code below.

We now give a single C program for the ziggurat method. It provides inline generation of either exponential (REXP) or normal (RNOR) random variables. To do this, we must have seperate arrays for exponentials and normals: `ke[256],we[256],fe[256],kn[128],wn[128],fn[128]`, initialized in a single procedure, zigset, and seperate 'fix' procedures that provide for generation when that 99%, the inline part, does not provide an immediate value.

Here is C code that provides for both exponential and normal generators by means of inline REXP's and RNOR's:

```
#define SHR3 (jz=jsr, jsr^=(jsr<<13), jsr^=(jsr>>17), jsr^=(jsr<<5),jz+jsr)
#define UNI (.5 + (signed) SHR3 * .2328306e-9)
#define RNOR (hz=SHR3, iz=hz&127, (abs(hz)<kn[iz])? hz*wn[iz] : nfix())
#define REXP (jz=SHR3, iz=jz&255, (    jz <ke[iz])? jz*we[iz] : efix())

static unsigned long iz,jz,jsr=123456789,kn[128],ke[256];
static long hz; static float wn[128],fn[128], we[256],fe[256];

float nfix(void) {       /*provides RNOR if #define cannot */
  const float r = 3.442620f; static float x, y;
  for(;;){ x=hz*wn[iz];
     if(iz==0){ do{x=-log(UNI)*0.2904764; y=-log(UNI);} while(y+y<x*x);
     return (hz>0)? r+x : -r-x;
              }
  if( fn[iz]+UNI*(fn[iz-1]-fn[iz]) < exp(-.5*x*x) ) return x;
  hz=SHR3; iz=hz&127;if(abs(hz)<kn[iz]) return (hz*wn[iz]);
        }       }

float efix(void)   {     /*provides REXP if #define cannot */
float x; for(;;){
    if(iz==0) return (7.69711-log(UNI));
    x=jz*we[iz];
    if( fe[iz]+UNI*(fe[iz-1]-fe[iz]) < exp(-x) ) return (x);
    jz=SHR3; iz=(jz&255);
    if(jz<ke[iz]) return (jz*we[iz]);
                }  }
/*--------This procedure sets the seed and creates the tables------*/
void zigset(unsigned long jsrseed)                 {
        const double m1 = 2147483648.0, m2 = 4294967296.;
      double dn=3.442619855899,tn=dn,vn=9.91256303526217e-3, q;
      double de=7.697117470131487, te=de, ve=3.949659822581572e-3;
        int i;  jsr=jsrseed;
/* Tables for RNOR: */   q=vn/exp(-.5*dn*dn);
        kn[0]=(dn/q)*m1; kn[1]=0;
        wn[0]=q/m1;      wn[127]=dn/m1;
        fn[0]=1.;        fn[127]=exp(-.5*dn*dn);
        for(i=126;i>=1;i--)                  {
          dn=sqrt(-2.*log(vn/dn+exp(-.5*dn*dn)));
          kn[i+1]=(dn/tn)*m1;   tn=dn;
          fn[i]=exp(-.5*dn*dn); wn[i]=dn/m1; }
/* Tables for REXP */  q = ve/exp(-de);
        ke[0]=(de/q)*m2; ke[1]=0;
        we[0]=q/m2;      we[255]=de/m2;
        fe[0]=1.;        fe[255]=exp(-de);
        for(i=254;i>=1;i--)                    {
         de=-log(ve/de+exp(-de));
         ke[i+1]= (de/te)*m2; te=de;
         fe[i]=exp(-de);      we[i]=de/m2;     }   }
\begin{verbatim}
```

These comments apply to use of the above code:

- A main program must be provided, in which use of REXP or RNOR in an expression will provide the required variate.
- Before such uses, the tables must be set up by means of a statement such as zigset(17235321); with any (non-zero) unsigned long argument.

- SHR3 uses an inline 3-shift shift register generator, and UNI floats it to (0,1). SHR3 is very fast, has period $2^{32}-1$ and does very well in tests of randomness—in particular for sequences made from the last 8 bits.

  SHR3 adds two successive terms of the sequence $y_{n+1} = y_n(I+L^{13})(I+R^{17})(I+L^5)$, with the $y's$ viewed as 1×32 binary vectors, L the 32×32 matrix that causes a left-shift of 1, and R its transpose. The resulting matrix $T = (I+L^{13})(I+R^{17})(I+L^5)$ has order $2^{32}-1$ in the group of non-singular 32×32 binary matrices, ensuring that period for SHR3. (See Marsaglia and Tsay, [10].)

5 Timings

We find that the ziggurat algorithm can provide RNOR or REXP, normal or exponential variates, at about 15 million per second for CPU's operating at 400MHz. Exact timings vary, of course, with the platform used and the compiler. We compared times for two other methods said to be fast: Leva [3] and Ahrens-Dieter [1]. We used a 400 MHz Pentium II PC with two different compilers: Microsoft Visual C++ (MS) and GNU gcc with -O3 optimization. To provide a fair comparison, we used the fast inline SHR3 or UNI wherever random integers or reals were needed. Times are in nanoseconds.

Method	400MHz, MS	400MHz, gcc
Leva [3]	307	384
Ahrens-Dieter [1]	161	193
RNOR	55	65
REXP	77	40

Speed is one of the criteria by which random variate generators are judged, but not the only one. However, if a method is a record for speed, and at the same time comes from a simple generating procedure, then it is worth considering among the many that have been put forward; we think the ziggurat method is such a one.

6 Summary

Extremely fast and simple sampling from decreasing densities can be provided by covering, as in Figures 1 and 2, the decreasing density $f(x)$ with layers of 255 equal-area rectangles such that the base, made up of a smaller rectangle with the tail attached, has the same area, v, as each of the 255 rectangles. The rectangle R_i has range $0 < x < x_i$, with $0 = x_0 < x_1 < \cdots < x_{255}$, and with vertical range $f(x_i) < y < f(x_{i-1})$.

The x's are related by

$$x_i[f(x_{i-1}) - f(x_i)] = v, \text{ and } rf(r) + \int_r^\infty f(x)\,dx = v, \text{ with } r = x_{255}.$$

The x's may be found by successively trying values for $r = x_{255}$ until the process

$$v = rf(r) + \int_r^\infty f(x)\,dx \text{ and } x_i[f(x_{i-1}) - f(x_i)] = v, \text{ for } i = 254, \ldots, 2, 1,$$

yields an x_1 for which $x_1(f(0) - f(x_1)) = v$.

Implementation is made faster by using tabled values $w_i = x_i/2^{32}$, $k_i = \lfloor 2^{32}(x_{i-1}/x_i) \rfloor$ and $f_i = f(x_i)$. The fast part, used about 99% of the time, can be implemented inline, for speed and for the convenience of being able to produce a required variate by merely placing, for example, REXP or RNOR in any expressions where such random variables are required.

References

[1] Ahrens, J.H. and Dieter, U, (1989), An alias method for sampling from the normal distribution, *Computing*, **42**, 159-170.

[2] Knuth, Donald E. (1998) *The Art of Computer Programming, Volume II*, 3rd Ed., Addison Wesley, Reading, Mass.

[3] Leva, J.L., (1992), A fast normal random number generator,
ACM Transactions Mathematical Sofware, **18**, 454-455.

[4] Marsaglia, George, (1961), Expressing a random variable in terms of uniform random variables, *Annals of Math. Statis.*, **32**, 894–898.

[5] Marsaglia, George, (1964), Generating a variable from the tail of the normal distribution, *Technometrics*, **6**, 101–102.

[6] Marsaglia, George, MacLaren, M.D. and Bray, T. A., (1964), A fast procedure for generating normal random variables, *Comm. of the ACM*, **7**, 3–10.

[7] Marsaglia, George and Maclaren, M.D., (1964), A fast procedure for generating exponential random variables, *Comm. of the ACM*, **7**, 298–300.

[8] Marsaglia, George, Ananthanarayan, K., and Paul, N. J., (1976), Improvements on fast methods for generating normal random variables, *Information Processing Letters*, **5**, 27–30.

[9] Marsaglia, George and Tsang, Wai Wan (1984), A fast, easily implemented method for sampling from decreasing or symmetric unimodal density functions, *SIAM Journ. Scient. and Statis. Computing*, **5**, 349–359.

[10] Marsaglia, George and Tsay, L.H. (1985), Matrices and the structure of random number sequences, *Linear Algebra and its Applications*, **67**, 147–156.