An Improved Ziggurat Method to Generate Normal Random Samples

JURGEN A DOORNIK
University of Oxford

The ziggurat is an efficient method to generate normal random samples. It is shown that the standard Ziggurat fails a commonly used test. An improved version that passes the test is introduced. Flexibility is enhanced by using a plug-in uniform random number generator. An efficient double-precision version of the ziggurat algorithm is developed that has a very high period.

Categories and Subject Descriptors: G.3 [**Probability and Statistics**]: Random number generation; G.4 [**Mathematical Software**]: Algorithm design and analysis, Efficiency

General Terms: Algorithms, Performance
Additional Key Words and Phrases: Collision test, Random number generation, Standard normal distribution, Ziggurat

1. INTRODUCTION

Because the normal distribution is so fundamental in statistical applications, it is essential to have an efficient and reliable method to generate random numbers from it. Recently, Marsaglia and Tsang [2000b] refined their ziggurat method [Marsaglia and Tsang 1984] to draw from the standard normal distribution. Their refinement simplifies the method, and they show it to be faster than several other efficient methods, so that it may claim to be the current method of choice.

There is something magical about the ziggurat method: it only uses one random number to select a point for the rejection method, while normally two are required to generate an (x, y) coordinate pair. This turns out to be the Achilles heel of the method, potentially affecting the quality of the generated random numbers.

The objective of this note is four-fold. First, to clarify the impact of using a single random number for the accept-reject step. Next, to propose a fix for the identified shortcoming, while maintaining most of the efficiency. Finally, to present a double precision algorithm in such a way that it can be used with different uniform random number generators. Finally, I present optimized and well-structured code, and provide a comparison with some other fast methods.

2. THE ZIGGURAT METHOD

The ziggurat partitions the standard normal density in horizontal blocks of equal area. The standardization can be omitted, using $f(x) = \exp(-x^2/2)$. All blocks

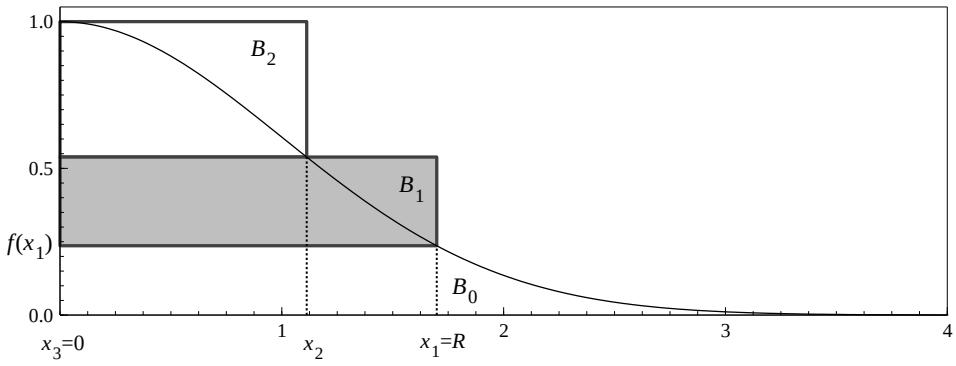

Fig. 1. Example of a three-way ziggurat partitioning of the standard normal density

are rectangular boxes, except the bottom one, which consists of a box joined with the remainder of the density. This is illustrated in Figure 1, using three boxes, labeled from bottom to top as B_0, B_1, B_2. Equal areas of size V implies:

$$x_2 \left[f(x_3) - f(x_2)\right] = x_1 \left[f(x_2) - f(x_1)\right] = x_1 f(x_1) + \int_{x_1}^{\infty} f(x)\mathrm{d}x = V.$$

Note that, working backward along the X-axis, previous values of x can be easily derived. For example, if x_1 is known:

$$x_2 = f^{-1}\left(f(x_1) + V/x_1\right).$$

In general, when dividing in C blocks, the partitioning can be found by solving the non-linear equation:

$$x_{C-1}(r)\left[f(0) - f\left(x_{C-1}(r)\right)\right] - V(r) = 0, \tag{1}$$

where r denotes the right-most x_1, $x_C = 0$, and:

$$V(r) = r f(r) + \int_{r}^{\infty} f(x)\mathrm{d}x.$$

The value of $x_{C-1}(r)$ can be found by subsitution from:

$$x_i = f^{-1}\left(f(x_{i-1}) + V(r)/x_{i-1}\right), \quad i = 2, \ldots, C-1, \quad x_1 = r.$$

Denote the solution of (1) by R, then the complete partitioning has been found.

The rejection method for the ziggurat is quite straightforward:

(1) Draw a box at random with probability $1/C$, say B_i.

(2) Draw a random number in the box as $z = U_0 x_i$ for $i > 0$ and $z = U_0 V/f(x_1)$ for $i = 0$.

(3) If $z < x_{i+1}$ accept z.

(4.1) If $i = 0$, accept a v from the tail using [Marsaglia 1964].

(4.2) Else if $i > 0$ and $U_1 [f(x_i) - f(x_{i+1})] < f(z) - f(x_{i+1})$ accept z.

(5) Return to step 1.

U_0 and U_1 denote Uniform$(0, 1)$ random numbers. This method only draws positive numbers. To draw from the entire distribution, replace U_0 in step 2 by $2U_0 - 1$ to give z a sign. Then steps 3 and 4.1 change to:

(3*) If $|z| < x_{i+1}$ accept z.

(4.1*) If $i = 0$, draw v from the tail using [Marsaglia 1964], accept sign$(z)v$.

The efficiency of this rejection algorithm is $0.5 * (2\pi)^{1/2}/(VC)$. Using $C = 3$, as in Figure 1, this is only about 80%. The benefit comes for higher values of C: for $C = 128$ it climbs to 98.78% with average acceptance rate of 98.05% at step 3.

Marsaglia and Tsang [2000b] note that the comparison in step 3a involves x_{i-1}/x_i, which can be precomputed. They implement the first three steps as follows:

(1**) Draw a 32-bit random unsigned integer j, and let i be determined by the p least significant bits, assuming $C = 2^p$.

(2**) Rewrite j as a *signed* integer k.

(3**) Check if k is in the rectangle of box i: if $|k| < 2^{31} x_{i+1}/x_i$ accept z (use $|k| < 2^{31} x_1 f(x_1)/V$ for $i = 0$).

The right-hand side of the comparison in 3** can be stored as an integer, so that the comparison can be done using integer arithmetic.

3. TESTING THE ZIGGURAT METHOD

Testing the normal RNG is perhaps more important than the underlying uniform, because it provides the basis for most statistical simulation experiments. A simple method is to transform the normal random numbers e_i back to uniformity using the normal cdf Φ: $\tilde{u}_i = \Phi(e_i)$. The $\tilde{u}_i$ can then be used in the available testing software, in our case the crush and big crush test suite [L'Ecuyer and Simard 2005].[1] The crush suite has 94 tests in total, using about 2^{35} random numbers, while the big crush involves 60 tests, requiring close to 2^{38} random numbers.

Table I provides the failure count on the crush test for three Ziggurat methods. Two versions of the original RNOR [Marsaglia and Tsang 2000b][2], the original using SHR3, and a modified version using MWC8222 instead of SHR3 to generate the uniform random numbers. The next entry is the new ziggurat, ZIGNOR, using

[1] Of course, when inversion is used, this form of testing is redundant: $\tilde{u}_i = \Phi(\Phi^{-1}(u_i)) = u_i$.

[2] Marsaglia and Tsang [2000a] also list the code for the algorithm, but it contains several syntax errors. It is faster because the `abs` function is used instead of the floating point version `fabs`. The version used here is the former, but using `abs` and changing `float` to `double`.

MWC8222. All of these use $C = 128$. In addition, the test results for the polar transformation are given, also based on MWC8222 for the uniform random numbers. The last line in the table is for the uniform random number generator MWC8222, which is a multiply-with-carry generator working on an array of 256 integers with a period of 2^{8222}, see [Marsaglia 2003].

The crush results in Table I shows that the original ziggurat method, RNOR, always fails one test comprehensively. Indeed, this even happens when using MWC8222, which performs very well as a uniform RNG.

Table I. Number of failures in the crush (94 tests in total) and big crush batteries of tests for some ziggurat normal RNGs as well as MWC8222 polar, and MWC8222 uniform RNG.

		Crush			Big Crush		
normal	uniform	10^{-300}	10^{-15}	10^{-5}	10^{-300}	10^{-15}	10^{-5}
RNOR: old ziggurat	SHR3	1	0	0	3	1	1
RNOR: old ziggurat	MWC8222	1	0	0	1	0	0
ZIGNOR: new version	MWC8222	0	0	0	0	0	0
polar	MWC8222	0	0	0	0	0	0
—	MWC8222	0	0	0	0	0	0

The test that is failed is the collision test [Knuth 1998]: n balls are thrown randomly in m urns. With $m > n$, less than one ball per urn is expected, and the test counts the number of collisions. The crush version uses $n = 10^7$ and $m = 2^{30}$, amounting to a test for 30 dimensional randomness. The original code provided for RNOR [Marsaglia and Tsang 2000b; 2000a] uses `float`s (32-bit floating-point values) throughout, and it is likely that no method using `float`s will pass the test. The versions of RNOR tested here use `double`s (64-bit floating-point values), and still fail the test.

The cause of this failure is more fundamental: the least significant 7 bits are used in step 1^{**} to determine the box B_i, but these also form part in step 2^{**} of the k that is used. So the x and y coordinate of the accept-reject point are not entirely independent. When using `float`s this overlap will be only in a few bits, but when using `double`s it is all the p least significant bits. As an extreme example, consider decimal arithmetic with two significant digits and ten boxes: then all numbers 0.01, 0.11, ..., 0.91 fall in box one, 0.02, ..., 0.92 in box two, etc. In principle, the random integer can be split in two components, but that would leave only 25 bits for the uniform random number (ignoring the loss of one bit for the sign): not much more than 30 million normal random numbers in total. This takes only a couple of seconds to generate (in contrast, the timings below use 10^9 random numbers).

The additional failures of RNOR-SHR3 in the big crush test are due to deficiencies in the SHR3 RNG. When it is replaced by MWC8222 only the collision test reports

a failure.[3]

4. IMPLEMENTATION

Two new versions of the ziggurat method are created. The first, labelled ZIGNOR, is a straightforward implementation. Correcting the test failure requires the use of two random numbers: a uniform transformed to $U(-1,1)$ and an integer of which the lowest 7 bits are used. Because steps 4.1 and 4.2 are only executed 2% of the time, there is no need to do these very efficiently. So it is unneccesary to precompute $f(x_i)$ in order save computation of one exp function. Moreover, the method is made portable by removing the built-in SHR3 random number generator. Instead, calls are made through a function pointer, which allows any uniform generator to be plugged in. This is, of course, slower than using inline code as in RNOR. As discussed above, ZIGNOR uses MWC8222 with a period of 2^{8222}. MWC8222 uses a 64-bit integer for its calculations, and is particularly well-suited for modern hardware.

The second version, VIZIGNOR, is a highly optimized version. It calls the underlying uniform in three ways: to get a 32-bit integer, to get a vector of 32-bit integers, and to get a uniform$(0,1)$ number. Following Marsaglia and Tsang [2000b], scaling is used to allow the main test of step 3 to be done using integer arithmetic. An additional optimization is that only 1 and a quarter random integers are needed to perform the main test. Finally, these inputs are generated in vectors of 256 elements.

The new ziggurat algorithms ZIGNOR and VIZIGNOR are compared with two other fast methods. The first is VNAL, based on the alias method [Ahrens and Dieter 1988]. This has a similar problem with the collision test in its original form, but VNAL avoids this. The underlying uniform is again MWC8222, and inputs are generated in vectors of 256, as for VIZIGNOR. However, VNAL seems to benefit much less of this. The second is FN3(q) [Wallace 1996][4], which generates normal variates directly. The q parameter is for quality control, and is set to two and three here. FN3(2) and FN3(3) pass the crush test.

To make timing comparisons more robust, I consider a range of computer platforms and compilers, as documented in Table II.

Table III compares the CPU time that is required to generate and sum 10^9 random numbers. The first line is the MWCC8222 uniform generator, the remainder are standard normals. It shows that VIZIGNOR, the new fast ziggurat is significantly faster than the alias-based method VNAL. RNOR-MWC8222 is RNOR with

[3]Leong et al. [2005] also report a deficiency in the original RNOR-SHR3 caused by the use of SHR3. They replace SHR3 by KISS.

[4]Using FastNorm3.c, see ftp.cs.monash.edu.au/pub/csw.

Table II. Computer platforms and compilers used to time random number generators

Label	Hardware	Operating system
A	AMD Athlon 64 3400 (2.4Ghz, Socket 754)	Linux Fedora Core 3 64-bit
I	Intel Pentium 4 (2.8GHz, 800FSB)	Microsoft Windows XP SP2
Label	**Compiler**	**Settings**
1	Microsoft Visual C++ 6 SP6	/O2
2	GCC 3.4.2	-O2 -m32 (32-bit executable)
3	GCC 3.4.2	-O2 -m64 (64-bit executable)

inline calls to MWC8222. This is faster still, but fails the collision test. FN3(2) is competitive with VIZIGNOR on some platforms, but this algorithm is much less understood than a rejection method that transforms uniforms to normals.

To put the computational speed in perspective, Table III also gives timings for a straightforward implementation of the polar method. VIZIGNOR is between three and four times faster.[5]

Table III. Speed comparison: seconds of CPU time to generate and sum 10^9 standard normal random numbers (MWC8222: $U(0,1)$ uniform)

	I-1	I-2	A-2	A-3
	Using MWC8222			
MWC8222 uniform	17.7	14.6	9.8	8.1
VIZIGNOR	33.6	40.0	21.6	17.2
ZIGNOR	49.2	52.0	32.8	26.8
VNAL	54.3	64.7	39.6	36.7
RNOR-MWC8222	23.5	25.1	16.6	12.7
Polar	85.7	177.8	83.7	79.6
	Other			
FN3(2)	43.6	35.3	15.3	15.0
FN3(3)	57.4	43.4	19.7	19.7

The precise form of the ziggurat will depend on the context. For example, as argued elsewhere [Doornik 2005], it would be desirable to return 52 bits of precision instead of 32. ZIGNOR is easily adapted by using an appropriate underlying higher precision uniform RNG (but a more efficient implementation would use two integers, using 7 bits for the box, leaving enough for the higher precision uniform). Alternatively, when vectors of random numbers are required, the implementation

[5]The results of Doornik [2005] are relevant here: often uniform RNGs are unnecessarily penalized by up to 50% through inefficient conversion to floating point. That is avoided here. As a first consequence, the uniform MWC8222 always clearly outperforms any standard normal RNG. Secondly, the gap between the polar method and the fast ziggurat method is somewhat smaller: the fast ziggurat returns a signed number, and avoids the speed penalty which occurs twice in the simple polar method. These two artefacts are usually present in other published benchmarks.

will be closer to VIZIGNOR. This is more difficult to extend to 52-bit precision, though. The resulting performance should be expected to lie somewhere between ZIGNOR and VIZIGNOR.

5. CONCLUSION

An improved ziggurat to generate standard normal random numbers was created. It is portable, allowing the use of very-high period random number generators, and passes the collision test. The ziggurat is an elegant method that can be easily understood and implemented. VIZIGNOR is probably the fastest general purpose normal RNG, and the generator of choice when only 32-bit precision is required. Indeed, the ziggurat method is increasingly finding its way into commonly used software, so it is important that the improved version is used instead of the original. ZIGNOR is less efficient, but still between two and three times faster than the polar method. It could be preferred when more flexibility is required. The code for ZIGNOR is listed in the Appendix; the complete source code can be downloaded from www.doornik.com/research. The original RNOR should only be considered if speed is much more important than quality and high periodicity.

These results may also impact on the generation of Gamma variables from a normal and uniform [Marsaglia and Tsang 2000a]: the corrected ziggurat should be used to avoid carrying over deficiencies to the gamma. Because the speed gap between uniform and normal has grown to twofold (the improved ziggurat is somewhat slower, and the uniform has been made faster [Doornik 2005]), it remains to be seen which gamma method is faster.

APPENDIX

A.1 ZIGNOR

`DRanU()` returns a uniform random number, $U(0,1)$, and `IRanU()` returns 32-bit unsigned random integer. `DRanNormalTail` is implemented as a separate function: it gets called only rarely, so that efficiency does not matter. For the same reason, it is not necessary to avoid a call to `exp()` when checking for the wedges (this could be achieved by precomputing the function values $f(x_i)$.

```
static double DRanNormalTail(double dMin, int iNegative)
{
    double x, y;
    do
    {   x = log(DRanU()) / dMin;
        y = log(DRanU());
    } while (-2 * y < x * x);
    return iNegative ? x - dMin : dMin - x;
}
```

```
#define ZIGNOR_C 128                    /* number of blocks */
#define ZIGNOR_R 3.442619855899 /* start of the right tail */
                   /* (R * phi(R) + Pr(X>=R)) * sqrt(2\pi) */
#define ZIGNOR_V 9.91256303526217e-3

/* s_adZigX holds coordinates, such that each rectangle has*/
/* same area; s_adZigR holds s_adZigX[i + 1] / s_adZigX[i] */
static double s_adZigX[ZIGNOR_C + 1], s_adZigR[ZIGNOR_C];

static void zigNorInit(int iC, double dR, double dV)
{
    int i;  double f;

    f = exp(-0.5 * dR * dR);
    s_adZigX[0] = dV / f; /* [0] is bottom block: V / f(R) */
    s_adZigX[1] = dR;
    s_adZigX[iC] = 0;

    for (i = 2; i < iC; ++i)
    {
        s_adZigX[i] = sqrt(-2 * log(dV / s_adZigX[i - 1] + f));
        f = exp(-0.5 * s_adZigX[i] * s_adZigX[i]);
    }
    for (i = 0; i < iC; ++i)
        s_adZigR[i] = s_adZigX[i + 1] / s_adZigX[i];
}
double  DRanNormalZig(void)
{
    unsigned int i;
    double x, u, f0, f1;

    for (;;)
    {
        u = 2 * DRanU() - 1;
        i = IRanU() & 0x7F;
        /* first try the rectangular boxes */
        if (fabs(u) < s_adZigR[i])
            return u * s_adZigX[i];
        /* bottom box: sample from the tail */
        if (i == 0)
            return DRanNormalTail(ZIGNOR_R, u < 0);
        /* is this a sample from the wedges? */
        x = u * s_adZigX[i];
        f0 = exp(-0.5 * (s_adZigX[i] * s_adZigX[i] - x * x) );
        f1 = exp(-0.5 * (s_adZigX[i+1] * s_adZigX[i+1] - x * x) );
```

```
        if (f1 + DRanU() * (f0 - f1) < 1.0)
            return x;
    }
}
```

A.2 VIZIGNOR

The efficient version relies on the fact that `IRanU()` returns an unsigned 32-bit integer, while `RanVecIntU` fills a vector with unsigned 32-bit integers. Taking into account that the final result will be signed, x_i is now stored as $x_i 2^{-31}$ and the ratio as $2^{-31} x_{i+1}/x_i$. A candidate is drawn as a 32-bit unsigned integer, cast to a signed integer. This should be scaled by 2^{-31} to obtain a uniform between -1 and 1. The code draws random integers and boxes in chunks of 256, refilling when empty.

REFERENCES

Ahrens, J. H. and Dieter, U. 1988. An alias method for sampling from the normal distribution. *Computing 42*, 159–170.

Doornik, J. A. 2005. Conversion of high-period random numbers to floating point. Mimeo, Nuffield College.

Knuth, D. E. 1998. *The Art of Computer Programming. Volume 2: Seminumerical Algorithms*, 3rd ed. Addison Wesley, Reading, MA.

L'Ecuyer, P. and Simard, R. 2005. Testu01. A software library in ANSI C for empirical testing of random number generators. Mimeo, University of Montreal, Canada.

Leong, P. H. W., Zhang, G., Lee, D.-U., Luk, W., and Villasenor, J. D. 2005. A comment on the implementation of the ziggurat method. *Journal of Statistical Software 12*, 1–4. www.jstatsoft.org.

Marsaglia, G. 1964. Generating a variable from the tail of the normal distribution. *Technometrics 6*, 101–102.

Marsaglia, G. 2003. Re: good C random number generator. Posting, Usenet newsgroup sci.lang.c. 13-May-2003.

Marsaglia, G. and Tsang, W. W. 1984. A fast, easily implemented method for sampling from decreasing or symmetric unimodal density functions. *SIAM Journal on Scientific and Statistical Computing 5*, 349–359.

Marsaglia, G. and Tsang, W. W. 2000a. A simple method for generating gamma variables. *ACM Transactions on Mathematical Software 26*, 363–372.

Marsaglia, G. and Tsang, W. W. 2000b. The ziggurat method for generating random variables. *Journal of Statistical Software 5*, 1–7. www.jstatsoft.org.

Wallace, C. S. 1996. Fast pseudorandom generators for normal and exponential variates. *ACM Transactions on Mathematical Software 22*, 119–127.