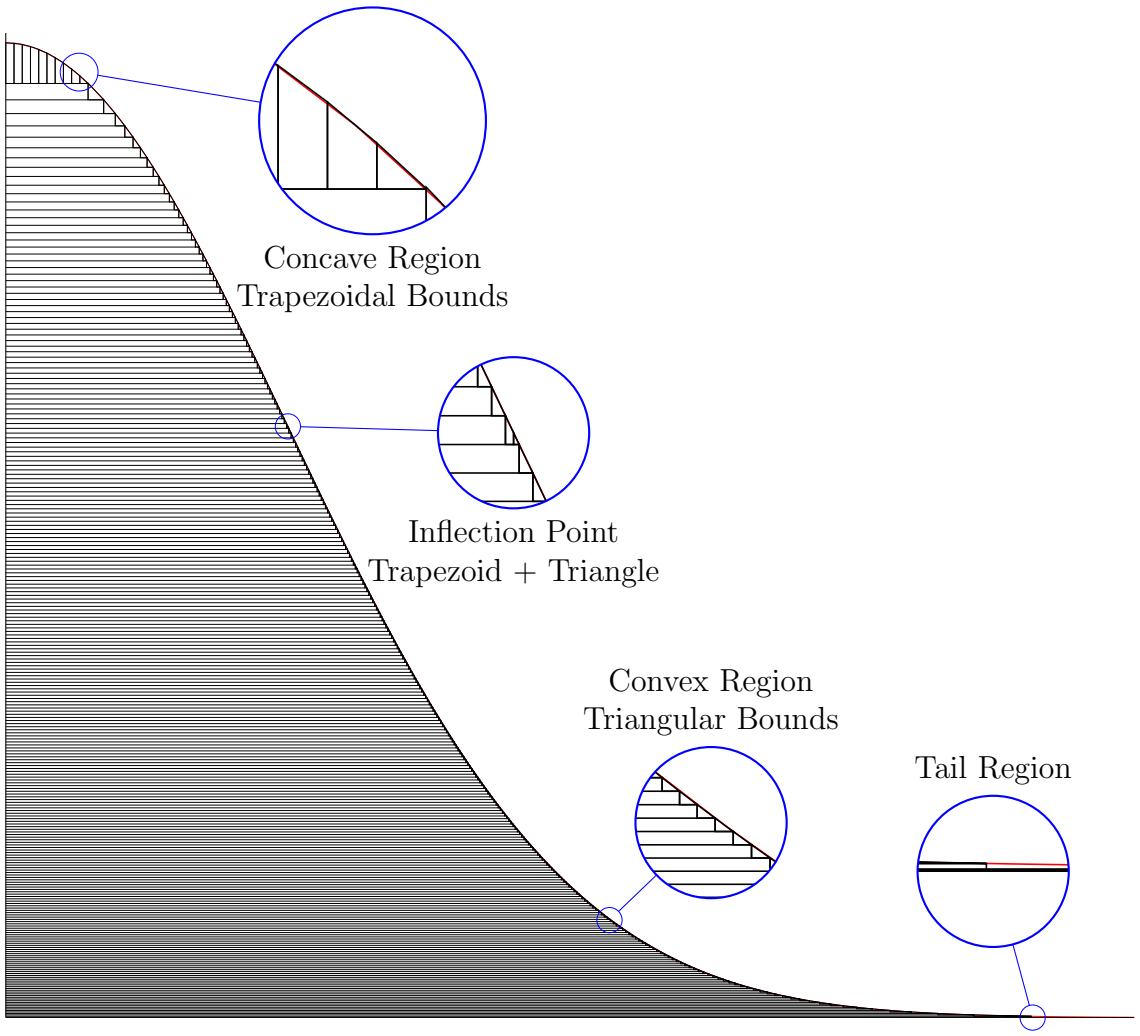
A Trapezoid-Ziggurat Algorithm for Generating Gaussian Pseudorandom Variates

Frank R. Kschischang
Department of Electrical and Computer Engineering
University of Toronto

January 16, 2019



1 Introduction

The ziggurat algorithm is a well-known fast reject sampling method for generating pseudorandom variates [1–4]. In this note we describe our implementation of a version of the modified ziggurat algorithm of McFarland [5] for generating Gaussian pseudorandom variates, in which rectangular ziggurat layers form a subset of the hypograph of the density function, rather than the other way around as in most other implementations. This approach allows *all* points sampled from within a ziggurat layer to be accepted immediately, without need for a comparison, thereby providing a speed advantage. We replace with trapezoids the triangle-shaped regions suggested by McFarland for sampling the ziggurat overhang regions; the trapezoids, however, degenerate to triangles in regions where the density function is convex. *Most* points sampled from within a trapezoid can be accepted by comparison with a linear boundary, obviating with high probability the need for an expensive density-function evaluation. Points in the tail, which arise infrequently, are sampled using a standard technique [6]. Trapezoids and tail regions are selected with the appropriate discrete probability distribution using Walker's alias method [3, 7, 8]. Some use is made of type-punning, taking advantage of the IEEE 754 floating point representation, gaining speed at the expense of portability. The current implementation runs on 32- or 64-bit little endian machines (e.g., machines compatible with Intel x86 architecture), but can easily be modified to run on other architectures. Endianness and floating point representation are checked when the random number generator is initialized. Though compatible with any source of pseudorandom 32-bit unsigned integers, our implementation makes use of O'Neill's PCG family of random number generators [9].

All of the following is well known, and is included here only for completeness. An excellent reference on the general topic of algorithms that transform a source of uniformly distributed random numbers to variates with a given non-uniform probability distribution is the book of Devroye [10].

2 Sampling the Hypograph of a Density Function

The basic idea of the ziggurat algorithm is to return the x-coordinate of a point uniformly sampled from the hypograph of the desired probability density function, in this case the standard normal. The (truncated) hypograph[1] associated with a non-negative function $g(x)$ is the subset

$$\mathcal{R}_g = \{(x, y) \in \mathbb{R}^2 : 0 \leq y \leq g(x)\}$$

of the Cartesian plane. Suppose that $g(x) = Af_Z(x)$, where $f(x)$ is a probability density function for a random variable Z and A is a positive constant. A uniform density over $\mathcal{R}_g$

[1] Hypograph = hypo (beneath, below from the Greek *hupo* 'under') + graph (something drawn or written, from the Greek *graphos*).

is then well defined: the density simply takes the value $1/A$ at all points of $\mathcal{R}_g$. Sampling from this density (i.e., choosing points from $\mathcal{R}_g$ uniformly at random) results in the random variable (X, Y), where X denotes the x-coordinate of the sampled point and Y denotes its y-coordinate. The cumulative distribution function (cdf) for X is easily obtained as the fraction of the total area of the hypograph on or to the left of a given x, i.e.,

$$F_X(x) = P[X \le x] = \frac{\int_{-\infty}^{x} g(t)\,\mathrm{d}t}{\int_{-\infty}^{\infty} g(t)\,\mathrm{d}t} = \frac{AF_Z(x)}{A} = F_Z(x),$$

where $F_Z(x)$ denotes the cdf of Z. In other words, the X coordinate of samples obtained uniformly from $\mathcal{R}_{Af_Z}$ has the distribution of Z.

Since the standard normal is symmetric, it is standard practice to sample only from the positive half of the distribution, and then attach a uniformly distributed sign. Thus we take $g(x) = \exp(-x^2/2)$, $x \ge 0$, as sketched on the cover of this report. Note that $g(x)$ has an inflection point at $x = 1$, being concave over the interval $[0, 1]$ and convex over $[1, \infty)$.

Generating a uniform distribution over an axis-aligned rectangle is easy. Generating a uniform distribution over a non-rectangle can be accomplished by enclosing the region in a tight bounding rectangle, generating points uniformly in the bounding rectangle, rejecting any points that occur outside of the given region. The number of trials before a point is accepted is a geometrically distributed random variable with expected value given by the ratio of the area of the bounding rectangle to that of the given region.

We follow the approach of McFarland [5] and use inscribed ziggurat rectangles, each selected to have 2^{-m} of the total area. The widest of these determines the start of the tail. The parameters of each subsequent layer can be determined in terms of the previous layer. Let R denote the maximum number of rectangles that can be so inscribed. When $m = 8$, for example, we can $R = 253$ rectangles, accounting for $253/256 = 98.8\%$ of the total probability, as illustrated on the cover of this report.

The remaining $(2^m - R)/2^m$ of the area is divided among the tail and R "overhang regions." As described below, in regions where $g(x)$ is concave, the overhang regions can be enclosed in a bounding trapezoid, while in regions where $g(x)$ is convex, a bounding triangle (which may be regarded as a degenerate trapezoid with a zero-height side) can be used. One of the overhang regions will generally straddle the inflection point at $x = 1$; as illustrated on the cover of this report, this region is subdivided and bounded to the left of $x = 1$ with a trapezoid and to the right of $x = 1$ with a triangle. We also subdivide the upper-most overhang region into 10 subregions. With the tail, this gives a total of $R+11$ non-rectangular regions.

The overall sampling procedure is then described as follows:

generate an integer U uniformly distributed in $\{0, 1, \ldots, 2^m - 1\}$

```
if (U < R)
    sample the x coordinate uniformly from the Uth rectangle
else
    select a non-rectangular region T with the appropriate nonuniform distribution
    if T is a trapezoid or triangle
        sample x use rejection sampling from the Tth region
    else
        sample x from the tail
    endif
endif
return x with a random sign attached
```

Sampling from a discrete nonuniform distribution with a relatively small number of probability masses can be accomplished efficiently using the alias method described in Section 3. Sampling from a trapezoid can be accomplished using the method described in Section 4. Sampling from the Gaussian tail can be accomplished as described in Section 5. We provide a few programming notes in Section 6. Results of testing the implemented Gaussian random number generator are given in Section 7.

3 The Alias Method

The alias method provides a means of sampling from a discrete probability distribution on $\{1, 2, \ldots, M\}$ in $\mathcal{O}(1)$ time, using $\mathcal{O}(M)$ space.

Two vectors must be constructed:

$$r = (r_1, r_2, \ldots, r_M),\ 0 \leq r_i \leq 1$$
$$a = (a_1, a_2, \ldots, a_M),\ a_i \in \{1, 2, \ldots, M\}.$$

Each r_i is referred to as a *rest probability* and each a_i is referred to as an *alias*. The sampling procedure is as follows:

```
generate an integer i uniformly distributed in {1, 2, ..., M}
generate a real number U uniformly distributed in [0, 1).
if (U < r_i)
    return i
else
    return a_i
endif
```

Evidently, given that a particular index i is chosen in the first step, the value i is returned with probability r_i and the value a_i is returned with probability $1 - r_i$. Define, for every

element $j \in \{1, 2, \ldots, M\}$ the set $I(j) = \{i \in \{1, \ldots, M\} \setminus \{j\} : a_i = j\}$ of (other) indices having alias j, and note that $I(j)$ may possibly be empty. The probability that the algorithm produces output $j \in \{1, 2, \ldots, M\}$ is then given by

$$p(j) = \frac{1}{M} \left(r_j + \sum_{i \in I(j)} (1 - r_i) \right). \tag{1}$$

By an appropriate choice of r and a, any desired probability mass function may be obtained.

Given a desired list of probabilities $\{p_1, p_2, \ldots, p_M\}$ summing to one, let us now see how to construct r and a. We do so in Robin Hood fashion, by taking from the "rich" and giving to the "poor." Initially we set $r_i = Mp_i$ and $a_i = i$. We also associate a Boolean flag f_i with each index i to indicate whether that index is "active" or not. Indices i where $r_i > 1$ are called "rich" and indices i where $r_i < 1$ are called "poor" and their flag f_i is set to TRUE. Indices i where $r_i = 1$ (neither rich nor poor) have their flag f_i set to FALSE. Necessarily, if some index i is rich, some other index j must be poor. In this case we can remove a mass $1 - r_j$ from index i be assigning $r_i := r_i - (1 - r_j)$ and "donate" this mass to index j, remembering the source of the donation by setting $a_j = i$. We remove index j from further consideration by setting f_j to FALSE. If it should happen that $r_i = 1$, then f_i is set to FALSE also. The process continues while there remain active indices.

In summary, given a list $p_1, \ldots, p_M$ of probabilities summing to one, we construct r and a as follows:

initialize: $a_i = i$, $i \in \{1, 2, \ldots, M\}$
initialize: $r_i = Mp_i$, $i \in \{1, 2, \ldots, M\}$
initialize: f_i to TRUE if $r_i = 1$, FALSE otherwise
while f_i are not all FALSE
    find a "rich" index i with $r_i > 1$
    find a "poor" index j with $r_j < 1$
    set $a_j = i$
    set f_j to FALSE
    set r_i to $r_i - (1 - r_j)$
    if $r_i = 1$ then set f_i to FALSE
endwhile

Note that the number of active indices reduces by at least one in each step of the algorithm, thus the algorithm must eventually terminate. Note also that every "poor" index is donated to exactly once, although "rich" indices might indeed make several donations (and become "poor" in the process). In our implementation we always take from the "richest" to give the "poorest."

For example, given $(p_1, \ldots, p_4) = (0.6, 0.2, 0.1, 0.1)$ we would proceed as follows:

$$\begin{bmatrix} r \\ a \\ f \end{bmatrix} = \begin{bmatrix} 2.4 & 0.8 & 0.4 & 0.4 \\ 1 & 2 & 3 & 4 \\ \text{T} & \text{T} & \text{T} & \text{T} \end{bmatrix} \mapsto \begin{bmatrix} 1.8 & 0.8 & 0.4 & 0.4 \\ 1 & 2 & 3 & 1 \\ \text{T} & \text{T} & \text{T} & \text{F} \end{bmatrix} \mapsto \begin{bmatrix} 1.2 & 0.8 & 0.4 & 0.4 \\ 1 & 2 & 1 & 1 \\ \text{T} & \text{T} & \text{F} & \text{F} \end{bmatrix}$$
$$\mapsto \begin{bmatrix} 1.0 & 0.8 & 0.4 & 0.4 \\ 1 & 1 & 1 & 1 \\ \text{F} & \text{F} & \text{F} & \text{F} \end{bmatrix}.$$

From (1) we see that

1. $I(1) = \{2, 3, 4\}$ and $p(1) = \frac{1}{4}(1 + 0.2 + 0.6 + 0.6) = 0.6$,
2. $I(2) = \emptyset$ and $p(2) = \frac{1}{4}(0.8) = 0.2$,
3. $I(3) = \emptyset$ and $p(3) = \frac{1}{4}(0.4) = 0.1$,
4. $I(4) = \emptyset$ and $p(4) = \frac{1}{4}(0.4) = 0.1$,

as desired.

It is convenient to pad M to a power of two by introducing sufficiently many zero-probability dummy symbols. Indeed, such dummy symbols have a zero rest probability, obviating the need to generate the uniform floating point random variate U, at the expense of a comparison (to check if an index corresponds to a dummy symbol). Such a scheme is called the alias-urn method [10, Section III.4] [11]. When the number of dummy symbols is relatively large, some speed-up can be obtained.

4 Sampling Uniformly from a Trapezoid

Consider the problem of sampling uniformly from the region bounded by a right trapezoid with width w, left height h_1, and right height h_2, oriented as shown in Fig. 1(a). This can be accomplished by sampling uniformly from the $w \times (h_1 + h_2)$ rectangle shown in Fig. 1(b), mapping any point p that lands above the sloped dividing line to a corresponding point p' below the dividing line, as illustrated.

Suppose the mid-point of the sloped dividing line is given coordinates $(0, 0)$. Denote the slope $(h_2 - h_1)/w$ of the dividing line as m. Then a point (X, Y) uniformly distributed over the lower trapezoid can be accomplished as follows.

1. Generate W uniformly distributed over $[-w/2, w/2)$.

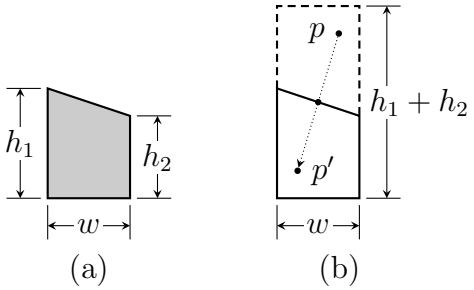


Figure 1: Uniform sampling from the trapezoid (a) can be accomplished by uniform sampling from the rectangle (b), flipping points that arise above the sloped dividing line to corresponding points below the sloped dividing line.

2. Generate Z uniformly distributed over $[-(h_1 + h_2)/2, (h_1 + h_2)/2)$.

3. If $Z \leq mW$ (i.e., if (W, Z) already falls in the lower trapezoid) then set $X = W$ and $Y = Z$; otherwise set $X = -W$ and $Y = -Z$ (i.e., reflect points in the upper trapezoid through the origin to the corresponding point in the lower trapezoid).

In this coordinate system, the lower left corner of the lower trapezoid is located at $(-w/2, -(h_1 + h_2)/2)$. This can be translated to any arbitrary location in the plane by adding an appropriate offset in each coordinate.

Note that the boundary points of the trapezoid occur with half the probability density of interior points. This bug becomes a useful feature when sampling uniformly from a larger region that is decomposed into trapezoids overlapping along their boundaries, though special care may need to be taken at (corner) points that fall into more than two trapezoids. In most applications, such corner cases occur sufficiently rarely that they can safely be ignored.

In applications involving the accept-reject algorithm, one would like to select a trapezoid of minimum area that contains a given region. Typically such regions are defined by a function $f(x)$ and an interval $[x_0, x_1)$, with the region given as $\{(x, y) : x_0 \leq x < x_1, 0 \leq y < f(x)\}$.

In case the function is convex over the given interval, then, as illustrated in Fig. 2(a), the top of the trapezoid should be chosen as the line joining $(x_0, f(x_0))$ with $(x_1, f(x_1))$.

In case the function is concave over the given interval, then, as illustrated in Fig. 2(b), the top of the trapezoid should be chosen tangent to the function $f(x)$ at some point t between x_0 and x_1. This line is given as $y(x) = f'(t)(x - t) + f(t)$, where f' denotes the derivative of f. To minimize the area of the given trapezoid, it suffices to minimize the sum of the heights $S = y(x_0) + y(x_1)$ at the endpoints of the line. We have

$$\begin{aligned} S(t) &= f'(t)(x_0 - t) + f(t) + f'(t)(x_1 - t) + f(t) \\ &= f'(t)(x_0 + x_1) + 2f(t) - 2tf'(t). \end{aligned}$$

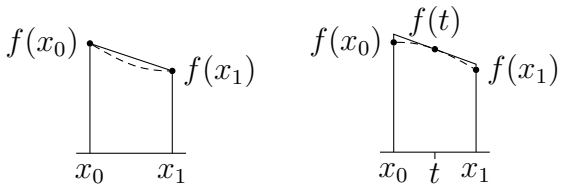


Figure 2: Fitting a tight trapezoid when (a) $f(x)$ is convex over $[x_0, x_1)$, (b) $f(x)$ is concave over $[x_0, x_1)$.

At an extremum,

$$\begin{aligned} 0 &= S'(t) \\ &= f''(t)(x_0 + x_1) + 2f'(t) - 2f'(t) - 2tf''(t) \\ &= f''(t)(x_0 + x_1 - 2t). \end{aligned}$$

Since $f''(t) < 0$, we find that the optimum choice for t is the midpoint $\frac{1}{2}(x_0 + x_1)$. That this choice is indeed a minimizer of S, we note that

$$S''(t) = f'''(t)(x_0 + x_1 - 2t) - 2f''(t),$$

which takes on a positive value when $t = (x_0 + x_1)/2$.

In case the function $f(x)$ is neither concave nor convex over the interval $[x_0, x_1)$, then in most cases of practical interest the interval can be refined, i.e., broken into smaller subintervals divided at the points of inflection of $f(x)$, so that in each resulting subinterval the function is indeed either concave or convex.

5 Sampling From the Tail of the Gaussian Distribution

Recall the general accept-reject algorithm for drawing pseudorandom variates. Let h and g be two probability density functions such that $h(x) \leq Mg(x)$ for every x in the support of h. A pseudorandom variate X with density h can be generated as follows.

1. Draw Z according to density g; this constitutes a "trial."
2. Accept Z with probability $h(Z)/Mg(Z)$, setting $X = Z$; otherwise reject Z, returning to step 1.

Note that the probability of accepting Z in any single trial is

$$P[\text{accept}] = \int P[\text{accept} \mid Z = x]g(x)\,\mathrm{d}x = \int \frac{h(x)}{Mg(x)}g(x)\,\mathrm{d}x \;= \frac{1}{M}.$$

The number of trials needed until generating a Z that is accepted is geometrically distributed with an expected value of M. Clearly, small values of M are preferred.

The right Gaussian tail, taken from a zero-mean unit-variance standard normal, and supported on $[a, \infty)$, has density function

$$h(x) = \begin{cases} \frac{1}{K(a)} \exp\left(-\frac{x^2}{2}\right) & x \geq a, \\ 0 & \text{otherwise;} \end{cases}$$

where

$$K(a) = \sqrt{\frac{\pi}{2}} \operatorname{erfc}\left(\frac{a}{\sqrt{2}}\right)$$

is the appropriate normalizing constant.

Suppose we take g as the tail of an exponential density with parameter $\lambda > 0$, so that

$$g(x) = \begin{cases} \lambda \exp\left(-\lambda(x-a)\right) & x \geq a, \\ 0 & \text{otherwise.} \end{cases}$$

Let

$$\begin{aligned} f(x) &= \frac{\exp(-x^2/2)\exp(\lambda(x-a))}{\lambda K(a)} \\ &= \frac{1}{\lambda K(a)} \exp\left(-\frac{1}{2}\left(x^2 - 2\lambda x + 2\lambda a\right)\right) \\ &= \frac{1}{\lambda K(a)} \exp\left(-\frac{1}{2}(x-\lambda)^2\right) \exp\left(\frac{\lambda}{2}(\lambda - 2a)\right). \end{aligned}$$

For $x \geq a$, $f(x)$ agrees with $h(x)/g(x)$ and, over this interval we require that $M \geq f(x)$. For any fixed a and any fixed $\lambda > 0$, we see that $f(x)$ is "bell-shaped" with a peak at $x = \lambda$, as shown in Fig. 3.

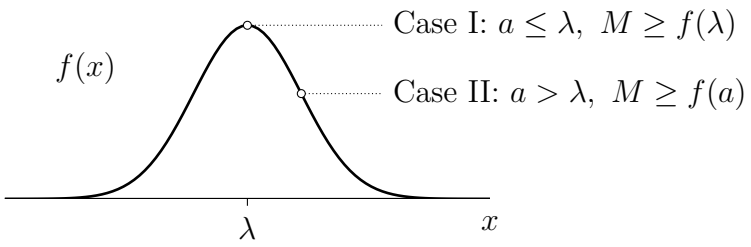


Figure 3: Since we require $M \geq f(x)$ for all $x \geq a$, two cases arise as illustrated.

We must choose λ and M. If we choose $\lambda \geq a$, then we must chose $M \geq f(\lambda)$; this is Case I as illustrated in Fig. 3. If we choose $\lambda \leq a$, then it suffices to have $M \geq f(a)$; this is Case II as illustrated in Fig. 3. Note that Cases I and II overlap when $\lambda = a$.

In Case II ($\lambda \leq a$), we would choose

$$M = f(a) = \frac{1}{\lambda K(a)} \exp(-a^2/2).$$

To minimize M, we would then choose λ as large as possible, namely $\lambda = a$, the point of overlap with Case I. Thus choosing $\lambda < a$ is strictly suboptimal.

In Case I ($\lambda \geq a$), we choose

$$M = f(\lambda) = \frac{1}{\lambda K(a)} \exp\left(\frac{\lambda}{2}(\lambda - 2a)\right).$$

Now since the first derivative of $f(\lambda)$ with respect to λ is

$$\frac{\mathrm{d}}{\mathrm{d}\lambda} f(\lambda) = f(\lambda) \frac{\lambda^2 - a\lambda - 1}{\lambda},$$

we expect to achieve minimum M when $\lambda^2 - a\lambda - 1 = 0$, i.e., when

$$\lambda = \lambda^* = \frac{a + \sqrt{a^2 + 4}}{2}.$$

To see that this stationary point of f is a minimum, we note that the stationary point is in the interior of the region of optimization, and the second derivative of $f(\lambda)$ with respect to λ, given by

$$\frac{\mathrm{d}^2}{\mathrm{d}\lambda^2} f(\lambda) = f(\lambda) \frac{(\lambda^2 - a\lambda - 1)^2 + \lambda^2 + 1}{\lambda^2},$$

is indeed positive, so the function f is convex.

Table 1 gives the value M as a function of a when choosing $\lambda = \lambda^*$, compared with several suboptimal choices of choosing λ. When a is 3 or larger, there is very little practical difference, among these possibilities, in the expected number of trials.

Table 1: Expected number of trials M for different values of a and λ

$a =$	0	0.5	1	1.5	2	2.5	3	3.5
$\lambda = \lambda^*$	1.315	1.208	1.141	1.098	1.071	1.053	1.041	1.032
$\lambda = a + 1/a$	—	3.373	1.257	1.117	1.076	1.054	1.041	1.032
$\lambda = a + 1/2$	1.808	1.293	1.152	1.098	1.076	1.066	1.063	1.063
$\lambda = a$	—	2.282	1.525	1.292	1.187	1.129	1.094	1.072

Whatever value of $\lambda \geq a$ is chosen, we take $M = f(\lambda)$, so that the accept probability when $Z = x$ is given by

$$\begin{aligned} \frac{h(x)}{Mg(x)} &= \frac{\exp(-x^2/2)}{K(a)} \frac{\exp(\lambda(x-a))}{\lambda} \lambda K(a) \exp\left(\frac{\lambda}{2}(2a - \lambda)\right) \\ &= \exp\left(-\frac{1}{2}(x - \lambda)^2\right). \end{aligned}$$

Suppose that $\lambda = a + \Delta$. Equivalent to generating a random variate Z with density g is generating an exponential random variable Y with parameter λ, and then setting $Z = Y + a$. Then

$$Z - \lambda = Y + a - (a + \Delta) = Y - \Delta.$$

The accept-reject procedure then becomes:

1. Draw Y according to an exponential density with parameter $\lambda = a + \Delta$.
2. Accept Y with probability $\exp(-(Y - \Delta)^2/2)$, setting $X = Y + a$; otherwise reject Y, returning to step 1.

6 Programming Notes

6.1 Type-punning

We use C's `union` construct to access the internal bit representations of floating-point numbers. The IEEE 754 standard represents a nonzero number x using a sign bit s—assumed to be the most significant bit (msb)—a binary exponent e, an integer mantissa m of p bits (an integer between 0 and $2^p - 1$), so that

$$x = (-1)^s(1 + m \times 2^{-p})2^e$$

For example, 32-bit `float`s have $p = 23$ bits while 64-bit `double`s have $p = 52$.

This representation allows us to quickly attach a sign to a floating point number (by twiddling the msb). For example, in the case of a `float`, if s is either `0` or `1<<31`, then writing `x.i^=s;return x.r` seems to compile to faster code than writing `return s?-x:x` (here `x.i` is the integer alias for `x`, and `x.r` is the floating point alias for `x`).

From a source of random integers m of appropriate bit-length, we can quickly generate a floating-point number uniformly distributed over all floating point numbers in $[1, 2)$, simply by choosing $s = 0$, and an exponent corresponding to $e = 0$. For `float`s, $s = 0$ and $e = 0$ corresponds to setting the most significant 9 of 32 bits to `001111111`, while for `double`s, $s = 0$ and $e = 0$ means setting the most significant 12 of 64 bits to `001111111111`. This approach seems to be significantly faster than `ldexp(u,-32)`, though the latter gives a number in the range $[0, 1)$ and is itself obtained more quickly than by subtracting 1.0 from a number in $[1, 2)$.

6.2 Rest Probabilities

Recall that in the alias method, we must "rest" (return i when the ith bin is selected) with probability r_i. For speed (to permit an unsigned integer comparison), all such rest probabilities are rounded to an integer multiple of 2^{-32}. We assume that all zero rest-probabilities occur only at the end of the table, and therefore if the bin index i is above a threshold we return the corresponding alias value a_i with probability one (and without need to generate a random variate). Accordingly, only nonzero rest probabilities need to be represented in the table. We use an off-by-one representation, i.e., if $r_i = m_i \times 2^{-32}$, we store $m_i - 1$ in the ith table entry. If we generate a uniformly distributed integer u in the range $[0, 2^{32} - 1]$, then we would return the alias value a_i only if u strictly exceeds the stored value $m_i - 1$ (or, equivalently, we "rest," returning value i, if $u \leq m_i - 1$).

6.3 Random Multipliers

For efficiency, we try to minimize the number of calls to the random number generator. We generate a random integer and use its least significant bits m bits to generate the index of a ziggurat rectangle, we extract a sign bit (and shift it to the msb position), and we use the remaining bits (shifted appropriately) as a multiplier for a suitably scaled rectangle width. In our implementation we maintain a minimum of 23 bits for this multiplier; in case $m + 1 > 9$, we require another call to the random number generator. Since m is known at compile-time, we use conditional compilation (i.e., a `#if` C-preprocessor directive) to achieve the desired behaviour.

6.4 Source of Uniform Random Numbers

We assume the availability of a good generator of uniformly distributed unsigned 32-bit pseudorandom integers. The current implementation builds on O'Neill's PCG family [9]. In the source code this dependence is encoded symbolically via appropriate `typedef` and `#define` statements, which can easily be modified should some other random number generator be preferred.

6.5 Speed Tests

We provide two implementations: a single-precision floating-point version `zmgf` and a double-precision version `zmgd`.

The graph of Fig. 4 shows the (approximate) time required to generate 5×10^9 random variates using the two implementations on my (current) laptop: (Intel Core i7-7500, 2.70GHz). The

number of ziggurat rectangles (nearly 2^m) was varied. Note the jump in time for `zmgf` that occurs when $m = 9$; as noted above, for $m \geq 9$ an additional call to the random number generator is required. In light of these timings, we have chosen $m = 8$ for both implementations.

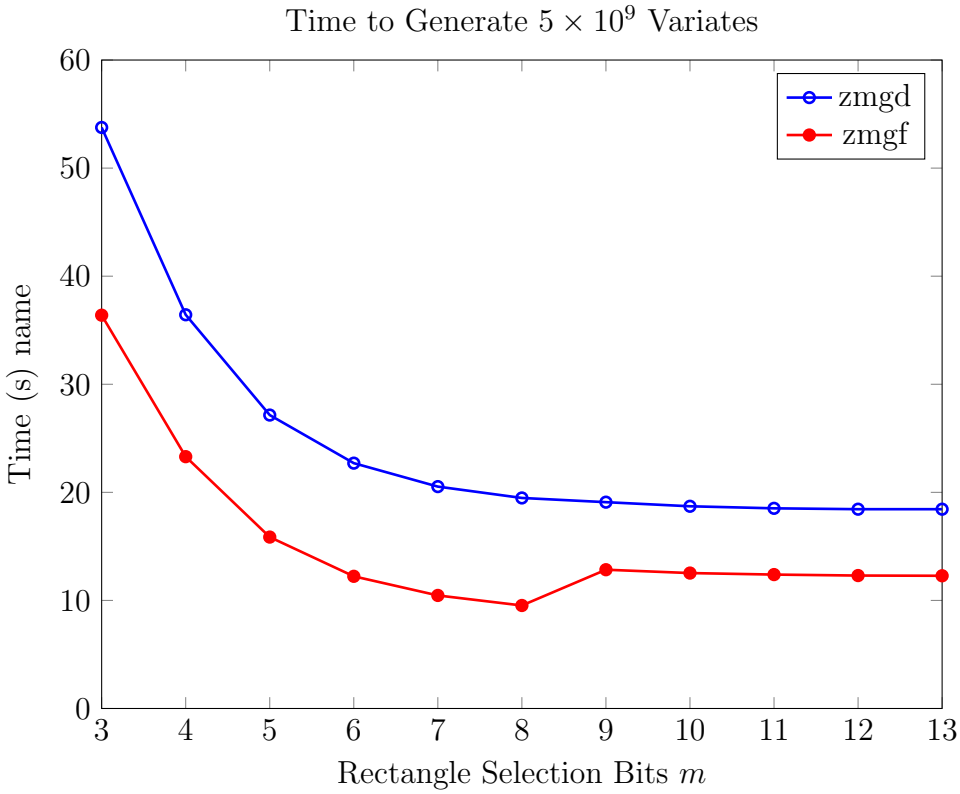


Figure 4: Execution time as a function of the number of rectangle-selection bits

.

In comparison, the ziggurat implementation `gsl_ran_gaussian_ziggurat` in the GNU Scientific Library required 60.9s to generate the same number of variates (using an underlying Mersenne Twister random number generator) and the Box-Muller implementation of `gsl_ran_ugaussian` required 288s. A single-precision version of Box-Muller using the PCG random number generator was implemented; it required 39.41s. The corresponding double-precision version required 104.9s.

Among all of these cases, the `zmgf` implementation, which required just 9.53s, was clearly the fastest.

7 Results

Fig. 5 shows a histogram generated from 5×10^9 samples from `zmgf`, compared with the histogram expected from a Gaussian distribution. Running `zmgd` gives similar results.

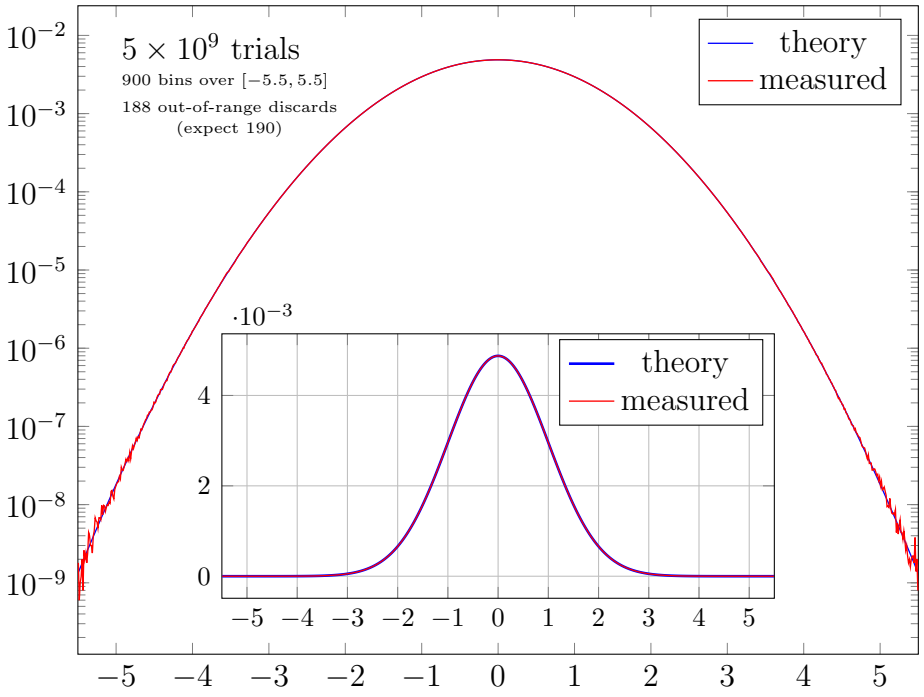


Figure 5: Histogram from `zmgf` (measured), compared with the histogram expected from a Gaussian distribution (theory). Simulation parameters are given in the plot.

In another trial of 5×10^9 samples from `zmgf`, the sample moments (1st moment up to 8th moment) shown in Table 2 were obtained. Similar results are obtained from `zmgd`.

Table 2: mth Moments

m	1	2	3	4	5	6	7	8
measured	−0.000022	0.999998	−0.000061	3.000149	−0.000236	15.002819	−0.000915	105.043475
expected	0	1	0	3	0	15	0	105

Finally, Fig. 6 shows a *normal plot* obtained from 20 order-statistic trials of `zmgf.d`, each of size 200 samples. In each order-statistic trial, a vector of length 200 samples is generated and then sorted, resulting in a vector $(y_1, \ldots, y_{200})$ with $y_1 \leq y_2 \leq \cdots \leq y_{200}$. Denote the expected value of the ith order statistic (called a *rankit*) as x_i. Shown in the figure is a scatter plot of the resulting (x_i, y_i) pairs obtained from 20 trials. Also shown in the figure are coloured "bands" corresponding to the ϵ and $1 - \epsilon$ percentiles for each order statistic, where $\epsilon \in \{2.5\%, 0.5\%, 0.05\%\}$. In a large number of trials, we would expect 95% of the

samples to fall within in the first (inner) band, 99% to fall within the first or second bands, and 99.9% to fall within the first, second, or third bands.

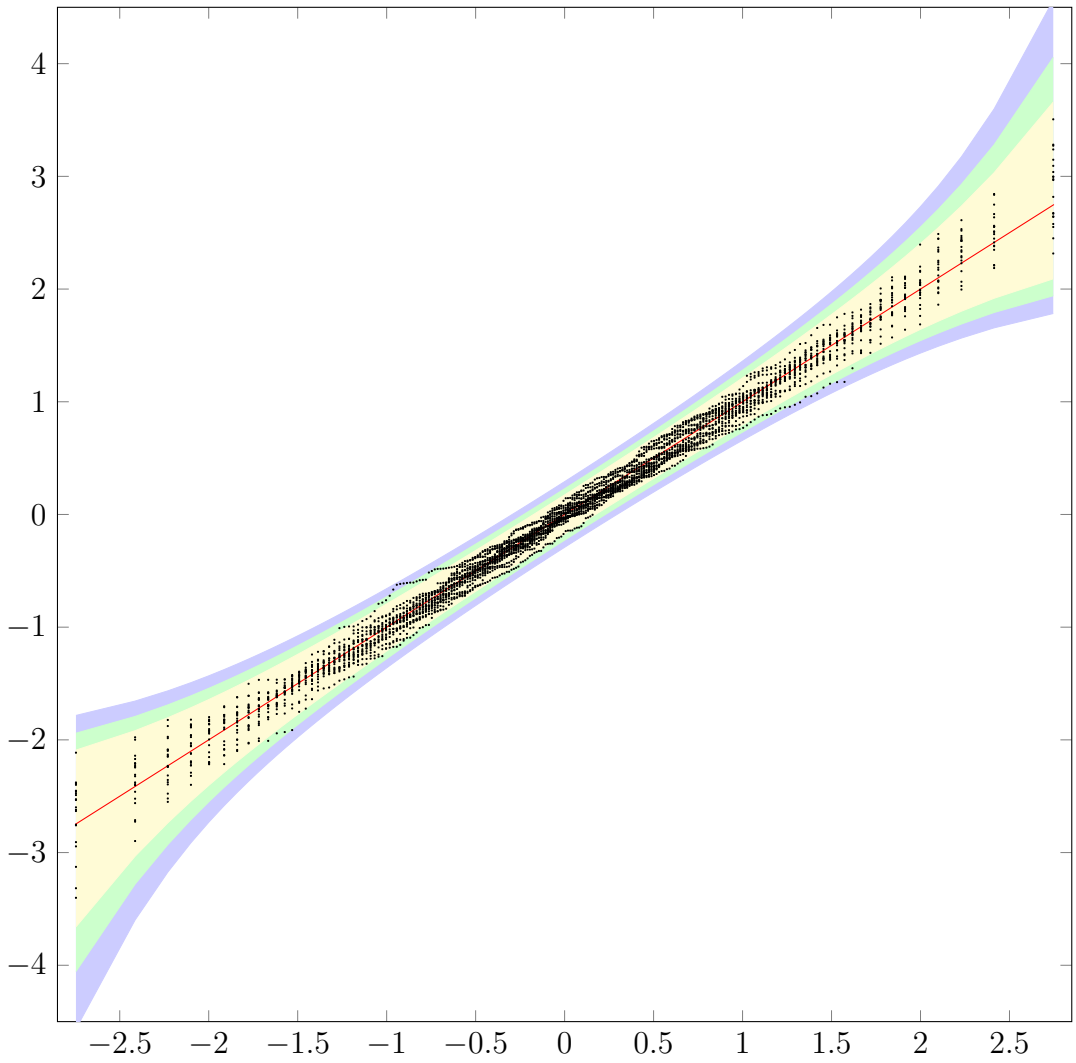


Figure 6: A normal plot showing 20 trials of size 200. Also shown are order-statistic ranges corresponding to the ϵ and $1 - \epsilon$ percentiles for each order statistic, with $\epsilon \in \{2.5\%, 0.5\%, 0.05\%\}$.

These results provide strong confidence that the samples produced by `zmgf` and `zmgd` do appear to be consistent with a standard normal distribution.

References

[1] G. Marsaglia and W. W. Tsang, "A fast, easily implemented method for sampling from decreasing or symmetric unimodal density functions," *SIAM J. Scient. Statist. Comput.*, vol. 5, pp. 349–359, 1984.

[2] G. Marsaglia and W. W. Tsang, "The ziggurat method for generating random variables," *J. Statist. Software*, vol. 5, pp. 1–7, 2000.

[3] D. E. Knuth, *The Art of Computer Programming, Vol. II*, 3rd Ed., Addison Wesley, 1998.

[4] D. Eddelbuettel, "Ziggurat revisited," University of Illinois at Urbana-Champaign, June 2018.

[5] C. D. McFarland, "A modified ziggurat algorithm for generating exponentially- and normally-distributed pseudorandom numbers," *J. Statist. Comput. Simul.*, vol. 86, pp. 1281–1294, 2016.

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

[7] A. J. Walker, "New fast method for generating discrete random numbers with arbitrary frequency distributions," *Electronics Letters*, vol. 10, p. 127, Apr. 1974.

[8] A. J. Walker, "An efficient method for generating discrete random variables with general distributions," *ACM Trans. Mathem. Software*, vol. 3, pp. 253–256, Sep. 1977.

[9] M. E. O'Neill, "PCG: a family of simple fast space-efficient statistically good algorithms for random number generation," Technical Report HMC-CS-2014-0905, Harvey Mudd College, 2014, paper and source code available online at `http://www.pcg-random.org`.

[10] L. Devroye, *Non-Uniform Random Variate Generation*, Springer Verlag, 1986. Available online: `http://www.nrbook.com/devroye`.

[11] A. V. Peterson and R. A. Kronmal, "On mixture methods for the computer generation of random variables," *The American Statistician*, vol. 36, pp. 184–191, 1982.