Gaussian Random Number Generators

DAVID B. THOMAS and WAYNE LUK

Imperial College

PHILIP H.W. LEONG

The Chinese University of Hong Kong and Imperial College

and

JOHN D. VILLASENOR

University of California, Los Angeles

11

Rapid generation of high quality Gaussian random numbers is a key capability for simulations across a wide range of disciplines. Advances in computing have brought the power to conduct simulations with very large numbers of random numbers and with it, the challenge of meeting increasingly stringent requirements on the quality of Gaussian random number generators (GRNG). This article describes the algorithms underlying various GRNGs, compares their computational requirements, and examines the quality of the random numbers with emphasis on the behaviour in the tail region of the Gaussian probability density function.

Categories and Subject Descriptors: G.3 [**Probability and Statistics**]: *Random number generation*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Random numbers, Gaussian, normal, simulation

ACM Reference Format:

Thomas, D. B., Luk. W., Leong, P. H. W., and Villasenor, J. D. 2007. Gaussian random number generators. *ACM Comput. Surv. 39*, 4, Article 11 (October 2007), 38 pages DOI = 10.1145/1287620.1287622 http://doi.acm.org/10.1145/1287620.1287622

1. INTRODUCTION

Simulations requiring Gaussian random numbers are critical in fields including communications, financial modelling, and many others. A wide range of Gaussian random number generators (GRNGs) have been described in the literature. They all utilize well-understood basic mathematical principles, usually involving transformations of

The support of UK Engineering and Physical Sciences Research Council (Grant EP/D062322/1, EP/D06057/1 and EP/C549481/1), the Hong Kong Research Grants Council (Grant CUHK 4333/02E), the National Science Foundation (Grants CCR-0120778 and CCF-0541453), and the Office of Naval Research (Contract N00014-06-1-0253) is gratefully acknowledged.

Author's email: dt10@doc.ic.ac.uk.

uniform random numbers. Assuming suitably precise arithmetic, the GRNGs can generally be configured to deliver random numbers of sufficient quality to meet the demands of a particular simulation environment.

Recent advances in computing and the increasing demands on simulation environments have made it timely to examine the question of what characterizes "sufficient quality." While the answer depends on the specifics of the simulation environment, it can be bounded by considering the capabilities of modern processors and extrapolating for expected trends. Modern processors programmed to implement a computational process can often reach a rate of 10^8 outputs per second. Dedicating a large computer cluster with 1000 machines to a single simulation for a ten-day period of time would result in a total simulation size of approximately 10^{17}. Adding another two orders of magnitude to allow for technology improvements over the next decade gives an extrapolated total of 10^{19}. Additional factors, such as the use of collaborative Internet-based simulations using significantly larger than 1000 machines could drive this number even higher.

The requirement to generate extremely large numbers of Gaussian random numbers elevates the importance of the quality of the GRNG. For example, while Gaussian random numbers with absolute values greater than 6σ or 7σ rarely occur, it is precisely those extreme events that could contribute disproportionately to certain rare but important system behaviours that the simulation aims to explore. Samples from an ideal GRNG with absolute value exceeding 9σ occur with probability 2.26×10^{-19}. For 10σ, the corresponding probability is 1.52×10^{-23}. Thus, a GRNG accurate in the tails to about 10σ would be sufficient for the largest simulations practical using technology available today and in the foreseeable future. More generally, when running large simulations it is vital to ensure that simulation results measure the performance of the system under study, without contamination due to imperfections in the random number generation process. Thus, the question of random number quality in GRNGs is central to their utility.

This basic question of random number quality has been of interest since the earliest days of computers. The first published survey of this topic appeared in 1959 [Muller 1959], and additional survey papers appeared in the 1960s [Kronmal 1964], 1970s [Ahrens and Dieter 1972], and 1980s [Chen and Burford 1981]. Schollmeyer and Tranter [1991] discussed GRNGs for communications applications in 1991, providing a survey of contemporary methods, and performing a limited number of tests. Their focus was mainly on the pairing of specific uniform random number generators, particularly linear congruential generators (LCGs) [Lehmer 1949] with transformation algorithms, and utilized visual, as opposed to statistical, evaluations of the resulting distributions. An overview of a limited set of GRNGs was provided by Kabal [2000], which compared several of the classic methods for generating Gaussian numbers on modern computers.

Most of the attention to GRNGs in recent years has focused on new generation algorithms as opposed to analysis of existing algorithms. Thus, while the number of algorithms has grown, there has been relatively little published work addressing the universe of GRNGs as a whole. The goals of this article are therefore:

(1) to provide an overview of GRNG methods and algorithms, including a classification of the various techniques,

(2) to present results on the performance and accuracy of the GRNGs that will be useful to practitioners, particularly those working in applications where statistically accurate generation of the "extreme events" noted above is important.

Our discussion also addresses issues that have not previously received significant attention. For instance, to ensure accurate tails, we address the need for careful conversion of uniform integer random numbers to floating-point values.

GRNGs aim to produce random numbers that, to the accuracy necessary for a given application, are statistically indistinguishable from samples of a random variable with an ideal Gaussian distribution. We classify GRNGs into four basic categories: cumulative density function (CDF) inversion, transformation, rejection, and recursive methods. The CDF inversion method simply inverts the CDF to produce a random number from a desired distribution. Transformation methods involve the direct transformation of uniform random numbers to a Gaussian distribution. The third category, rejection, again starts with uniform random numbers and a transformation, but has the additional step of conditionally rejecting some of the transformed values. Recursion, the final category, utilizes linear combinations of previously generated Gaussian numbers to produce new outputs.

An alternative classification is "exact" or "approximate." Exact methods would produce perfect Gaussian random numbers if implemented in an "ideal" environment. For example, the Box-Muller method subjects uniform numbers to various transformations in order to produce Gaussian outputs. If a perfect, and infinitely precise, uniform RNG were used, and if the functions themselves were evaluated with infinite precision, perfect Gaussian random numbers would be produced. Approximate methods, on the other hand, will produce outputs that are approximately Gaussian even if the arithmetic used is perfect. An example of this is the central limit theorem, which is only exact when an infinite number of uniform random numbers are combined and so must be approximate in any practical implementation. In the subsequent discussion of the algorithms, an indication of whether the algorithm is exact or approximate is provided.

Section 2 provides brief descriptions, pseudo code, and references for the GRNGs. Section 3 covers algorithms that focus on the tail region of the Gaussian. Section 4 describes the test parameters and the corresponding results, and Section 5 presents conclusions.

2. ALGORITHMS FOR GAUSSIAN SAMPLES

In the description of different Gaussian random number generator algorithms, we assume the existence of a uniform random number generator (URNG) that can produce random numbers with the uniform distribution over the continuous range $(0, 1)$ (denoted $U(0, 1)$ or U hereafter). Note that the range does not include 0 or 1 since each is possibly an invalid input for a GRNG; for instance, the Box-Muller method requires a non-zero URNG input and CDF inversion must have its URNG input strictly less than 1. Similarly, V is a continuous URNG with outputs in the range $(-1, 1)$ (excluding 0). I is used to denote a discrete uniform integer random number over the range $[0, 2^w - 1]$, where typically w is the machine word-length. Where multiple samples from a uniform random number generator are used within an algorithm, the different samples are identified with subscripts, for example, U_1 and U_2 represent two independent uniform samples in an algorithm. In algorithms with loops, all random numbers within the loop body are freshly generated for each loop iteration.

A Gaussian distribution with mean zero and standard deviation one, often known as a "standard normal" distribution, has the probability density function (PDF):

$$\phi(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}. \tag{1}$$

A plot of $\phi(x)$ versus x gives the familiar bell-curve shape, but does not directly indicate the probability of occurrence of any particular range of values of x. Integrating the PDF

from $-\infty$ to x gives the cumulative distribution function (CDF):

$$\Phi(x) = \int_{-\infty}^{x} \phi(x)dx = \frac{1}{2}\left[1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right]. \quad (2)$$

The CDF $\Phi(x)$ gives the probability that a random sample from the Gaussian distribution will have a value less than x. The CDF can be used to calculate the probability of values occurring within a given range, for example, the probability of a number between a and b (where $a < b$) is $\Phi(b) - \Phi(a)$. There is no closed-form solution for Φ, or for the related function erf, so it must be calculated numerically, or using some form of approximation. A good reference on distributions and random number generation can be found in Devroye [1986] (available for download at the address in the reference).

2.1. The CDF Inversion Method

CDF inversion works by taking a random number α from $U(0, 1)$ and generating a Gaussian random number x through the inversion $x = \Phi^{-1}(\alpha)$. Just as Φ associates Gaussian numbers with a probability value between zero and one, Φ^{-1} maps values between zero and one to Gaussian numbers. While this is conceptually straightforward, and exact if Φ^{-1} is calculated without error, the lack of a closed form solution for Φ^{-1} for the Gaussian distribution necessitates the use of approximations, which in turn affects the quality of the resulting random numbers. Since achieving increased accuracy requires increased complexity, most of the research in this area has focused on improving this trade-off. Numerical integration offers arbitrarily high precision, but at a computational cost that makes it impractical for random number generation, particularly in the tail regions of the Gaussian. As a result, most Gaussian CDF inversion methods utilize polynomial approximations.

One of the earliest approximation efforts was introduced by Muller [1958], who described a fast approximation to the inverse CDF with moderate precision. This method approximates the inverse CDF to within 4×10^{-4} for inputs in the range $[6 \times 10^{-7}, 1 - 6 \times 10^{-7}]$, corresponding to an output range of $\pm 5\sigma$. As the emphasis was on speed rather than accuracy, a simple polynomial approximation scheme was used. The input range was split into 64 pairs of symmetric segments and an interpolating polynomial was associated with each segment. For segments 1..56, linear approximation was sufficient; for 57..62, quadratic polynomials were used, and for segment 63, a quartic polynomial was needed. For the final segment 64, corresponding to the input ranges [0, 1/128] and [127/128, 1], the function becomes difficult to approximate with a single polynomial of reasonable degree. Instead a rational approximation based on a truncated continued fraction expansion was used, with the continued fraction expanded until successive terms differed by less than the target accuracy. A similar approach was used by Gebhardt [1964], though the approximation in the tails was based on iterative refinement of a semiconvergent series rather than a continued fraction. At approximately the same time, Wetherill [1965] proposed another approximate CDF inversion method based on polynomials, but splitting the range into just three sections to reduce the table sizes needed.

More recently, Wichura [1988] described two high precision approximations to the inverse Gaussian CDF using rational polynomials. For inputs x in the range [0.075, 0.925] a rational polynomial in $(x - 0.5)^2$ was used, while for inputs outside this range, one of two rational polynomials in $\sqrt{-\ln x}$ was used. Because most of the inputs fall within the first input range, the square root and logarithm only need to be calculated 15% of the time. The first method, PPND7, gave 7 decimal digits of accuracy in the range $[10^{-316}, 1 - 10^{-316}]$, and the second, PPND16, gave about 16 decimal digits of accuracy

over the same range. The lower precision PPND7 utilized rational polynomials with degree 2 and 3, while PPND16 used rational polynomials with degree 7.

An approximate CDF inversion technique using only one rational polynomial was provided by Hastings [Box and Muller 1958a]. This technique first transforms the input x using $\sqrt{\ln x^{-2}}$, then uses a degree 2 over degree 3 rational polynomial. The cost of having just one polynomial is that the square root and logarithm must be performed every time, rather than only for the tails of the curve as in some of the other CDF inversion methods. In addition, the Hastings technique only works for one side of the input range, so it needs to be slightly modified to allow handling of a full range of inputs. Hardware implementations of CDF inversion techniques have also been developed [Chen et al. 2004; McCollum et al. 2003].

2.2. Transformation Methods

2.2.1. Box-Muller Transform. The Box-Muller transform [Box and Muller 1958b; Pike 1965] is one of the earliest exact transformation methods. It produces a pair of Gaussian random numbers from a pair of uniform numbers. It utilizes the fact that the 2D distribution of two independent zero-mean Gaussian random numbers is radially symmetric if both component Gaussians have the same variance. This can be easily seen by simply multiplying the two 1D distributions $e^{-x^2}e^{-y^2} = e^{-(x^2+y^2)} = e^{-r^2}$. The Box-Muller algorithm can be understood as a method in which the output Gaussian numbers represent coordinates on the two-dimensional plane. The magnitude of the corresponding vector is obtained by transforming a uniform random number; a random phase is then generated by scaling a second uniform random number by 2π. Projections onto the coordinate axes are then performed to give the Gaussian numbers. Algorithm 1 gives pseudo-code for implementing this method. Because the algorithm produces two random numbers each time it is executed, it is common for a generation function to return the first value to the user, and cache the other value for return on the next function call.

Algorithm 1. Box-Muller

1: $a \leftarrow \sqrt{-2\ln U_1}$, $b \leftarrow 2\pi U_2$
2: **return** $(a \sin b, a \cos b)$ {Return pair of independent numbers}

Computation of cosine and sine can often be performed in one step, and highly optimized algorithms based on function evaluation and suitable for fixed-precision hardware implementation have been reported [Lee et al. 2004; Boutillon et al. 2003; Xilinx 2002].

2.2.2. Central Limit Theorem (Sum-of-uniforms). The PDF describing the sum of multiple uniform random numbers is obtained by convolving the constituent PDFs. Thus, by the central limit theorem, the PDF of the sum of K uniform random numbers $V/2$ each, over the range $(-.5, .5)$, will approximate a Gaussian with zero mean and standard-deviation $\sqrt{\frac{K}{12}}$, with larger values of K providing better approximations. The main disadvantage of this approach is that the convergence to the Gaussian PDF is slow with increasing K. Some intuition can be gained by realizing that the sum is bounded at $-K/2$ and $K/2$, and that the PDF of the sum is composed of segments that are polynomials limited in degree to $K-1$. Thus, the approximation in the tails of the Gaussian is particularly poor. Methods to mitigate this problem by "stretching" the PDF in the tail regions [Teichroew 1953] have used a Chebyshev interpolating polynomial to map the CDF of the distribution for a given K to that of the Gaussian distribution. The polynomial

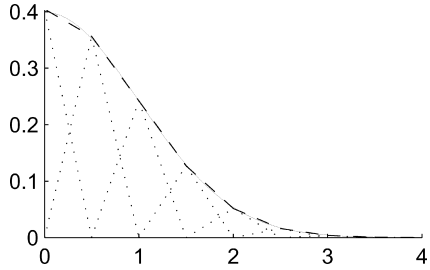

Fig. 1. Approximation to the Gaussian distribution composed of multiple triangle distributions.

will only provide an accurate mapping at a fixed number of finite inputs, based on the polynomial degree, so a trade-off must be made between accuracy and complexity. An example of Teichrow's method given in Muller [1959] uses a 9th degree polynomial on the sum of 12 uniforms.

While this technique improves the resulting distribution, deviations from a true Gaussian PDF remain significant for practical values of K. Additionally, the need to generate and additively combine large numbers of uniform random numbers itself constitutes a computational challenge, so the central limit theorem is rarely used in contemporary GRNGs. However, this approach has been used in hardware implementations as a way of combining two or more lower quality Gaussian numbers to produce one good one [Danger et al. 2000; Lee et al. 2004; Xilinx 2002]. This technique can also be used directly when the fractional accuracy does not need to be large: for example, it has been shown [Andraka and Phelps 1998] that the sum of 128 1-bit variables can provide a useful binomial approximation to the Gaussian distribution. The central limit theorem of course is an example of an "approximate" method—even if perfect arithmetic is used, for finite K the output will not be Gaussian.

2.2.3. Piecewise Linear Approximation using Triangular Distributions. Kabal [2000] describes an approximate method for generating Gaussian random numbers, using a piecewise linear approximation. The Gaussian distribution is decomposed into a set of k basic component triangular distributions $t_1..t_k$, each with the same width $2w$, centered at $c_i = w((k+1)/2 - i)$, and associated with probability q_i. The regular spacing means that each triangle overlaps with one triangle to the left and one triangle to the right, and the sum of the overlaps creates a piecewise linear approximation to the Gaussian PDF, as illustrated in Figure 1 with $w = 0.5$.

Since the component distributions are triangles, only addition and multiplication are needed. Outputs are generated by first probabilistically choosing one of the triangles, and then generating a random number from within the selected triangle distribution. The triangles are selected using Walker's alias method [Walker 1977] for sampling from a discrete distribution using one uniform input; the triangle distributions are then generated using the sum of two more appropriately scaled uniform inputs.

In software this method has the disadvantage of requiring three random numbers per output sample, making it quite computationally expensive to implement. However, in hardware, uniform random numbers are comparatively cheap to generate, while multiplications and other operations are more expensive, so this method is more attractive. By using large numbers of triangles, and by using the central limit theorem to combine multiple random numbers, this method can provide an efficient Gaussian random number generator in hardware [Thomas and Luk 2006].

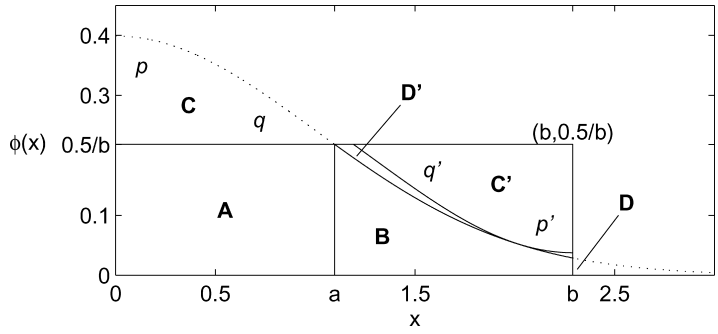

Fig. 2. Packing of the Gaussian distribution into a rectangular area using the Monty Python method.

2.2.4. Monty Python Method. The Monty Python method [Marsaglia and Tsang 1998] relies on a technique of packing the Gaussian distribution into a rectangle, using an exact transform. Figure 2 shows the arrangement graphically, with the desired Gaussian PDF shown as a dashed curve. The central idea is to partition the Gaussian PDF into four disjoint areas, shown as A, B, C, and D. These four areas are designed so that they can be exactly packed into a rectangle, using a transform that leaves the large areas A and B unchanged, maps area C in the Gaussian PDF to area C' in the rectangle through an affine transform, and uses a more complex process to pack the Gaussian tail area D into area D'. Generating a sample using the Monty Python method consists of uniformly generating a random point within the rectangle, identifying which of the areas the point is in, and applying the appropriate unpacking transform for that segment. The advantage of the method is that in the most common cases, areas A and B, the uniform random point can be returned untransformed as a Gaussian sample.

Algorithm 2. Monty Python

1: $s \leftarrow 2\lfloor 2U_1 \rfloor - 1$ {Choose random sign (+1 or −1) for output sample}
2: $x \leftarrow bU_2$ {Horizontal component of uniform 2D random sample}
3: **if** $x < a$ **then** {Check if point is in area A}
4: **return** sx
5: **end if**
6: $y \leftarrow U_3/(2b)$ {Vertical component of uniform 2D random sample}
7: **if** $y < \phi(x)$ **then** {Check if point is under Gaussian PDF in area B}
8: **return** sx
9: **end if**
10: $(x, y) \leftarrow f_C(x, y)$ {Point is in region C', transform it to region C}
11: **if** $y < \phi(x)$ **then** {Check if point is under Gaussian PDF in area C}
12: **return** sx
13: **else**
14: **return** Return x from the tails with $|x| > b$ (see section 3)
15: **end if**

Algorithm 2 provides a simplified description of the sampling process, omitting some optimizations for clarity. The first two conditionals check for points in A and B, returning the horizontal component of the random uniform sample (with attached sign) in either case. If neither case is true then the point is mapped from area C' to area C using a fixed affine mapping f_C. For example, in Figure 2 the two points p' and q' are mapped back to the equivalent points p and q in C. If the transformed point lies

under the Gaussian PDF (third conditional) then the original point was within C', so the transformed horizontal component is returned. Any other points must fall within D', but the mapping from D' to D is nontrivial, so instead a new random value from the tail $|x| > b$ is generated using a method such as those described in Section 3. Note that the area of D' is the same as the area under the tail, as the area of the rectangle is $b\frac{1}{2b} = 0.5 = \Phi(\infty) - \Phi(0)$, and the areas of A, B and C' clearly sum to $\Phi(b) - \Phi(0)$.

The constant b, and the derived constant $a = \phi^{-1}(1/2b)$, determine the overall efficiency of the method. The larger b is made, the smaller the expensive tail area of D. However, b must not be so large that the regions B and C' overlap, as this would distort the shape of the region C. In Figure 2 the value of $b = 2.29$ is used, which requires random numbers from the tail 2.2% of the time. In order to use slightly larger values of b without areas B and C' overlapping, it is possible to apply an area preserving transform to C', stretching horizontally and compressing vertically. This allows $b = \sqrt{2/\pi}$, reducing the number of random numbers taken from the tail to 1.2% [Marsaglia and Tsang 1998].

It should be noted that while Marsaglia originally used a rejection method to sample from the tails, the Monty Python method itself involves the "folding" of the positive Gaussian PDF into the rectangle with width b and height $1/2b$ in Figure 2, and the association of 2D locations in that rectangle with different portions of the Gaussian. Rejection of samples occurring in D' followed by use of a separate tail sampling method (which can be either a direct transform or a rejection method) is one way to implement it, though a direct, computationally impractical, transformation from D' to D does exist. For this reason the Monty Python method is classed as a transformation method, rather than a rejection method.

2.3. Rejection Methods

The rejection method for generating a random number can be described as follows. Let $y = f(x)$ be a function with finite integral, $\mathcal{C}$ be the set of points (x, y) under the curve, and $\mathcal{Z}$ be a finite area superset of $\mathcal{C}$: $\mathcal{Z} \supset \mathcal{C}$. Random points (x, y) are taken uniformly from $\mathcal{Z}$ until $(x, y) \in \mathcal{C}$ and x is returned as the random number [Knuth 1981; Press et al. 1997].

The density of such an x will be $cf(x)$, where c is a normalizing value that makes $cf(x)$ a probability density function ($\int cf(x)dx = 1$).

2.3.1. Polar. The polar method [Bell 1968; Knop 1969] is an exact method related to the Box-Muller transform and has a closely related two-dimensional graphical interpretation, but uses a different method to get the 2D Gaussian distribution. While several different versions of the polar method have been described, we focus on the form by Knop [1969] because it is the most widely used, in part due to its inclusion in Numerical Recipes [Press et al. 1997].

As noted earlier, for the Box-Muller transform, two uniform random numbers are used to generate the magnitude and phase of a vector of which the two Cartesian coordinates are the output Gaussian numbers. In the polar method, two uniform random numbers in the interval $(-1, 1)$ are initially generated and the magnitude of the vector they describe is evaluated. If the magnitude exceeds 1, the uniform numbers are discarded. If the magnitude is less than 1, which occurs with probability $\pi/4$, it is transformed and the result is scaled by each of the two uniform random numbers to give the two Gaussian outputs. This is described in Algorithm 3. In addition to having the conditional step, the polar method differs from the Box-Muller method in that it does not need a sine or cosine calculation, but it does require a division and two additional multiplications. A

fast vectorized implementation that also has the advantage of reducing the number of square root and ln computations has been described in Brent [1993].

Algorithm 3. Polar-Rejection

1: **repeat**
2: $x \leftarrow V_1$, $y \leftarrow V_2$
3: $d \leftarrow x^2 + y^2$
4: **until** $0 < d < 1$
5: $f \leftarrow \sqrt{-2(\ln d)/d}$
6: **return** $(f \times x, f \times y)$

2.3.2. Marsaglia-Bray Rejection Method. The Marsaglia-Bray method [Marsaglia and Bray 1964] is an exact method that uses a combination of four distributions: two direct transformations and one rejection-based distribution are summed to produce outputs in the range $[-3, 3]$, and another rejection-based transformation is used to provide random numbers from the tail regions outside this range. Each distribution has an associated probability, so the overall Gaussian PDF $\phi(x)$ in the range $[-3, 3]$ can be broken into a mixture of two easily generated distributions (g_1 and g_2) plus a more complex residual distribution (g_3):

$$\phi(x) = a_1 g_1(x) + a_2 g_2(x) + a_3 g_3(x) \tag{3}$$
$$g_1(x) = 2(U_1 + U_2 + U_3 - 1.5) \tag{4}$$
$$g_2(x) = 1.5(U_4 + U_5 - 1) \tag{5}$$
$$g_3(x) = \phi(x) - \frac{(a_1 g_1(x) + a_2 g_2(x))}{a_1 + a_2 + a_3} \tag{6}$$

where

$$a_1 = 0.8638 \;\; a_2 = 0.1107 \;\; a_3 = 0.0228002039 \;\; a_4 = 1 - a_1 - a_2 - a_3$$

Outside $[-3, 3]$ a function directly approximating $\phi(x)$ is used (with probability a_4). The top half of Figure 3 shows the three distributions $g_1(x)$, $g_2(x)$, and $g_3(x)$ in the range $[-3, 3]$. Note that a_1 is as large as possible, with g_1 just touching the actual Gaussian PDF at ± 2, so that this case occurs with the highest probability. The more computationally expensive densities, g_3 (the small residual density) and g_4 (the tail distribution outside the $[-3, +3]$ range), occur infrequently. Within the range $[-2, 2]$, g_2 fills in the relatively large gap between g_1 and ϕ, leaving g_3 to compensate for the remaining difference to the Gaussian, as shown using an expanded vertical axis in the lower half of the figure and given by:

$$g_3(x) = \begin{cases} ae^{-x^2/2} - b(3 - x^2) - c(1.5 - |x|) & |x| < 1 \\ ae^{-x^2/2} - d(3 - |x|)^2 - c(1.5 - |x|) & 1 < |x| < 1.5 \\ ae^{-x^2/2} - d(3 - |x|)^2 & 1.5 < |x| < 3 \\ 0 & 3 < |x|. \end{cases} \tag{7}$$

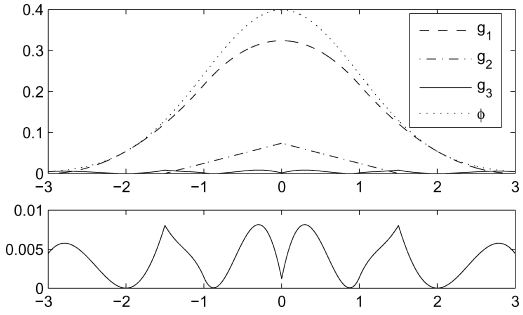

Fig. 3. The Marsaglia-Bray rejection generator relies on the composition of three distributions over the $[-3, +3]$ range. The top graph shows the PDF of the three distributions $g_1..g_3$, along with the Gaussian distribution that they sum to. The lower graph shows the shape of g_3 using an expanded vertical axis, which must be generated through rejection.

where

$$a = 17.49731196 \;\; b = 4.73570326 \;\; c = 2.15787544 \;\; d = 2.36785163$$

Algorithm 4 gives pseudo-code for the generator. The g_3 distribution is generated using a rejection method and g_4 is generated using one of the methods discussed in section 3 for sampling from the tail.

Algorithm 4. Marsaglia-Bray Rejection

```
1: s ← U
2: if s < a1 then
3:    return 2(U1 + U2 + U3 − 1.5)    {Sample from g1 with probability a1}
4: else if s < a1 + a2 then
5:    return 1.5(U4 + U5 − 1)    {Sample from g2 with probability a2}
6: else if s < a1 + a2 + a3 then
7:    repeat [Perform rejection step using smallest rectangle fully enclosing g3]
8:       x ← 6U6 − 3,  y ← 0.358U7
9:    until y < g3(x)
10:   return x    {Sample from g3 with probability a3}
11: else
12:   return Return x from the tails with |x| > 3 (see section 3)
13: end if
```

2.3.3. Ratio of Uniforms. Generation of Gaussian random numbers using a ratio of uniform random numbers was originally proposed by Kinderman and Monahan [1977], with enhancements given by Leva [1992a, 1992b]. The ratio of uniforms method has an advantage over the Box-Muller method in that the square root is replaced by a possibly cheaper division, and that the logarithm function, while still present, can in some cases be avoided. A potential disadvantage is that two uniform random numbers are consumed, but at most one Gaussian number is produced. The ratio of uniforms is an exact method.

Figure 4 shows the geometric interpretation, with each of the axes corresponding to one of the input uniform random numbers. Points enclosed by the solid curve $|v| < \sqrt{-4u^2 \ln u}$ need to be retained, while those outside need to be rejected. To avoid unnecessary evaluation of the exact boundary of the acceptance region, most

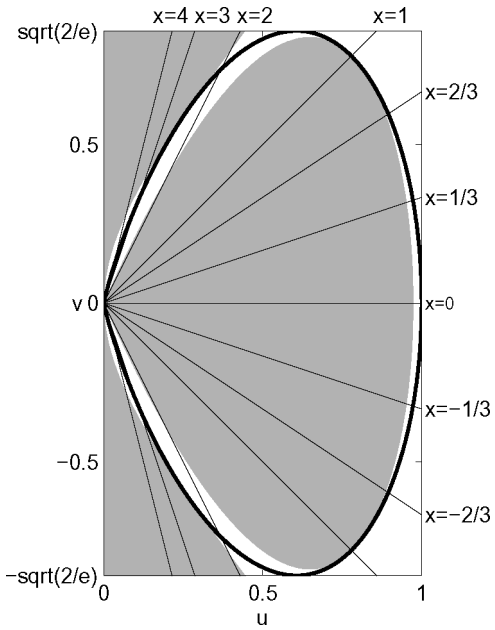

Fig. 4. Acceptance and rejection zones for the ratio of uniforms method. Points in the grey regions are accepted or rejected immediately, while points in the white area must be compared against the exact curve.

implementations of the ratio method approximate it using lower complexity equations that avoid computing the logarithm for many candidate points. The bounds shown in the figure are those suggested in tKinderman and Monahan [1977], and are shown in pseudo-code in Algorithm 5. The central grey region contains points that can be immediately accepted, corresponding to the test in step 2.3.3, while the upper and lower grey regions can be immediately rejected by step 2.3.3. Points in the white region must be tested against the exact curve, shown as a solid line in the figure, and step 2.3.3 in the algorithm. Note that either or both of these quick tests can be eliminated, which may be desirable if the logarithm function is very fast.

Algorithm 5. Ratio-of-Uniforms

1: **loop**
2: $u \leftarrow U_1$
3: $x \leftarrow V_1\sqrt{2/e}/u$
4: **if** $x^2 \leq 5 - 4e^{1/4}u$ **then** {Test for quick accept}
5: **return** x
6: **else if** $x^2 < 4e^{-1.35}/u + 1.4$ **then** {Test for quick accept}
7: **if** $v^2 < -4u^2 \ln u$ **then** {Do full test against exact curve}
8: **return** x
9: **end if**
10: **end if**
11: **end loop**

The bounds shown in Figure 4 are not very tight, and on average still require the full test to be made 0.23 times for each Gaussian number produced. If the logarithm function is very slow it may be worthwhile to use more complex bounds to avoid performing the

exact test. Tighter bounds have been presented [Leva 1992b], where an ellipse is fitted to the curve. This technique reduces the number of full tests per output to 0.012. Another possible advantage for machines with slow division operations is that the division by u only occurs when a point is accepted, rather than for every candidate pair. Pseudo-code for the method is shown in Algorithm 6, which adopts the constants:

$$s = 0.449871 \quad t = -0.386595 \quad r_1 = 0.27597$$
$$a = 0.19600 \quad b = 0.25472 \quad r_2 = 0.27846.$$

Algorithm 6. Leva's Ratio-of-Uniforms

```
1: loop
2:   u ← U1, v ← √(2/e) V1
3:   x ← u − s, y ← |v| − t
4:   Q ← x² + y(ay − bx)
5:   if Q < r1 then
6:     return v/u
7:   else if Q < r2 then
8:     if v² < −4u² ln u then
9:       return v/u
10:    end if
11:  end if
12: end loop
```

Hörmann has noted that when a linear congruential generator (LCG) is used for the uniform random numbers, the relationship between successive LCG values will prevent certain ratios from occurring, leaving gaps in the PDF [Hörmann 1994].

2.3.4. Ahrens-Dieter Table-Free Method. The Ahrens-Dieter Table-Free method is an exact Gaussian generator that transforms a pair of independent exponential and Cauchy random numbers into two independent Gaussian random numbers [Ahrens and Dieter 1988]. This is similar to the idea behind the Box-Muller method, except that instead of applying a complex transform to easily generated uniform random numbers, it applies a simpler transform to two distributions that are more complex to generate. In principle, the exponential and Cauchy distributions could be generated directly, using $-\ln U$ and $\tan(\pi(U - 1/2))$ respectively, which would make this a transform method. However, the only reason this method is feasible is because the authors develop two rejection based algorithms for samples from the exponential and Cauchy distributions. For this reason we have classed this as a rejection algorithm.

Algorithm 7. Ahrens-Dieter Exponential Generator

```
1:  x ← U1, a ← A
2:  while x < 0.5 do
3:    x ← 2x, a ← a + ln 2
4:  end while
5:  x ← x − 1
6:  if x < P then {First branch taken 98% of time.}
7:    return a + B/(C − x)
8:  else
9:    return Return sample using rejection from residual distribution.
10: end if
```

Algorithm 8. Ahrens-Dieter Cauchy Generator

1: $b \leftarrow U_1 - 0.5$
2: $c \leftarrow A - (U_1 - 0.5)^2$
3: **if** $c > 0$ **then** {First branch taken 99.8% of the time.}
4: **return** $b(B/c + C)$
5: **else**
6: **return** Return sample using rejection from residual distribution.
7: **end if**

Algorithm 9. Ahrens-Dieter Table-Free Normal General

1: $s \leftarrow U$
2: **if** $s < 0.5$ **then**
3: $s \leftarrow 1$
4: **else**
5: $s \leftarrow -1$
6: **end if**
7: $x \leftarrow$ Generate Exponential random number.
8: $y \leftarrow$ Generate Cauchy random number.
9: $z \leftarrow \sqrt{2x/1 + y^2}$
10: **return** $(s \times z, y \times z)$ {Return pair of independent Gaussian random numbers.}

The structure of the exponential and Cauchy generation algorithms are shown in Algorithms 7 and 8. The exponential generator uses the memoryless property as an initial range reduction step, allowing exponential generation to be split into the calculation of a geometric random offset in steps 1 to 5 of Algorithm 7, followed by an approximation to the truncated exponential distribution over the range $[0, \ln 2)$ in steps 6 to 10. 98% of the time the method uses the quick return path in step 7, but 2% of the time a rejection process against the residual distribution must be used. Algorithm 8 uses a different rejection method, using steps 1 and 2 to create an approximation to the Cauchy distribution that can be used 99.8% of the time. These two algorithms can then be used by Algorithm 9 to generate pairs of Gaussian random samples.

Only an algorithmic overview of the common execution paths of the method is provided here, as providing the details of the methods would take up too much space. Indeed, one of the drawbacks of this method is that it is complex to understand, must be carefully implemented, and requires many constants. The original paper provides full pseudo-code for the method, along with the required constants, but readers should note that algorithm EA on page 1332 has two errors. First, constant b should be $2+\sqrt{2}$, as defined in Equation 2.4, rather than the value of $\sqrt{2}$ shown at the beginning of the algorithm. Second, the constant called "h" at the beginning of the algorithm should actually be called "H", and is defining the value used in step 7 of the listing.

2.3.5. GRAND. GRAND [Brent 1974] is the best known implementation of a class of exact random number generators known as the odd-even method, first presented by Forsythe [1972], but originally developed by John von Neumann. The odd-even method can be used to produce random numbers from distributions of the form $f(x) = Ke^{-G(x)}$ over a range $[a, b)$, where $a \leq x < b \Rightarrow 0 \leq G(x) \leq 1$, and K is a constant. In order to generate a sample, first a random number $x \sim U[a, b)$ is generated, then $u_0 = G(x)$ is calculated. Next a sequence of $U[0, 1)$ random numbers $u_1, u_2, \ldots$ is generated until

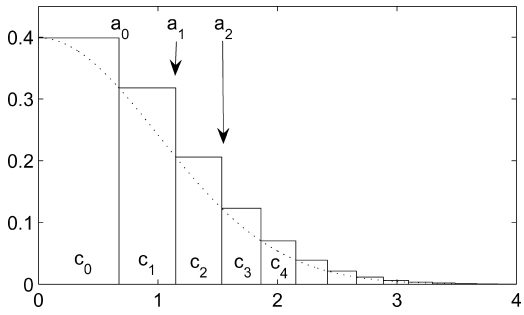

Fig. 5. GRAND division of Gaussian PDF into separate ranges. Range c_i is selected with probability 2^{-i-1} and then used to implement a rejection based selection of a random number within that range.

$u_k > u_{k-1}$. If k is odd then x is returned as a sample (with PDF $f(x)$ in the range $[a, b)$), or if k is even then x is rejected. This process is repeated until some value of x is accepted, which the method guarantees will eventually happen.

For the Gaussian distribution $G(x) = \frac{1}{2}(x^2 - a^2)$, but in order to ensure that $0 \leq G(x) \leq 1$, it is necessary to split the distribution range into a set of contiguous sections. Forsythe used boundaries of the form $a_0 = 0, a_i = \sqrt{2i-1}$, corresponding to $0, 1, 1.73, 2.23 \ldots$, which resulted in an average of 4.04 uniform random numbers consumed per output number, including one uniform random number used to select the section to sample.

The GRAND algorithm, shown in Algorithm 10, uses a different set of boundaries to split the distribution range into sections that increase the candidate acceptance rate, and hence reduces the number of uniform random numbers consumed. First a geometric random index i is generated from a uniform sample, so the probability of using index i is 2^{-i-1}. This index is used to select from within a table of offsets A, where $a_i = \Phi^{-1}(1 - 2^{-i-1})$. Index i is responsible for producing values in the range $[a_i, a_{i+1})$, so the length of the table directly influences the maximum σ that can be achieved. Figure 5 shows how the Gaussian curve is split up into these ranges, where in each rectangle the area under the curve is the acceptance region, and points in the area above the curve are rejected. Moving away from the origin, the area of the acceptance region is exactly half that of the preceding region. Although the odd-even method does not use direct (x, y) rejection, the rectangles give an accurate idea of the accept/reject rate for each point in the range.

Algorithm 10. GRAND method.

```
 1: i ← 0,  x ← U    {Note that 0 < x < 1 according to definition of U}
 2: while x < 0.5 do  {Generate i with geometric distribution}
 3:   x ← 2x,  i ← i + 1
 4: end while
 5: loop  {Now sample within chosen segment using odd-even method}
 6:   u ← (a_{i+1} − a_i)U_1
 7:   v ← u(u/2 + a_i)
 8:   repeat
 9:     if v < U_2 then
10:       if U_3 < 0.5 then
11:         return a_i + u
12:       else
13:         return −a_i − u
```

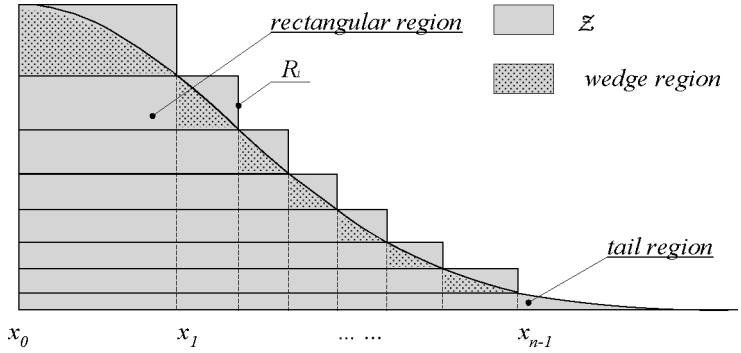

Fig. 6. Diagram showing the Gaussian distribution divided into rectangular, wedge, and tail regions in the Ziggurat method.

14: **end if**
15: **else**
16: $v \leftarrow U_4$
17: **end if**
18: **until** $v < U_5$
19: **end loop**

The algorithm shown here is a simplified version, which uses more uniform inputs than are necessary. A more practical and sophisticated implementation is described by Brent [1974], which recycles uniforms between stages within the algorithm, and between successive calls to the algorithm. This technique reduces the number of uniforms needed per output from 4.04 [Forsythe 1972] to 1.38, at the expense of introducing a division and some extra additions.

2.3.6. Ziggurat. The Ziggurat method [Marsaglia and Tsang 1984a, 2000] (the second of these two publications is used as the basis for the discussion here) uses an enclosing curve for the positive half of the PDF, which is chosen as the union of n sections, R_i $(1 \leq i \leq n)$, made up of $(n-1)$ rectangles, and the tail region, as illustrated in Figure 6. The rectangles and tail region are chosen so that they are all of equal area, v and their right-hand edges are denoted by x_i. All but one of the rectangles can be further divided into two regions: a "subrectangle" bounded on the right by x_{i-1}, which is completely within the PDF, and to the right of that a wedge shaped region, that includes portions both above and below the PDF. The rectangle bounded by x_1 consists of only a wedge shaped region.

Each time a random number is requested, one of the n sections is randomly (with equal probability) chosen. A uniform sample x is generated and evaluated to see if it lies within the subrectangle of the chosen section that is completely within the PDF. If so, x is output as the Gaussian sample. If not, this means that x lies in the wedge region (unless the tail section is being considered; in that case separate processing occurs), and an appropriately scaled uniform y value is chosen. If the x, y location is below the PDF in the wedge region, then x is output. Otherwise x and y are discarded and the process starts again from the beginning. In the case of the tail section and $x > x_{n-1}$, a value from the tail is chosen using a separate procedure (see Section 3). Provided that the tail sampling method is exact, the Ziggurat method as a whole, is exact. Algorithm 11 gives pseudo-code for the Ziggurat generator, omitting some common optimizations for clarity.

Algorithm 11. The Ziggurat method

```
1: loop
2:   i ← 1 + ⌊nU_1⌋   {Usually n is a binary power: can be done with bitwise mask}
3:   x ← x_i U_2
4:   if |x| < x_{i−1} then
5:     return z   {Point completely within rectangle.}
6:   else if i ≠ n then   {Note that φ(x_{i−1}) and φ(x_i) are table look-ups.}
7:     y ← (φ(x_{i−1}) − φ(x_i))U   {Generate random vertical position.}
8:     if y < (φ(x) − φ(x_i)) then   {Test position against PDF.}
9:       return x   {Point is inside wedge.}
10:    end if
11:  else
12:    return |x| > r from the tail   {see section 3}
13:  end if
14: end loop
```

The values of x_i $(i = 1, 2, \ldots, n)$ are calculated prior to execution, or on program startup, and are determined by equating the area of each of the rectangles with that of the base region. If this area is v, the equations are as follows:

$$v = x_i[\phi(x_{i-1}) - \phi(x_i)] = r\phi(r) + 1\int_r^\infty \phi(x)dx. \tag{8}$$

The value of r can be determined numerically, and can then be used to calculate the values of x_i. More details on the method used to calculated constants, and detailed code for implementing the Ziggurat method can be found in Marsaglia and Tsang [2000]. When $n = 256$ the probability of choosing a rectangular region is 99%.

The Ziggurat method is a refinement of an older method, called the Rectangle-Wedge-Tail Algorithm [Marsaglia et al. 1964], which also uses rectangles in order to provide candidate rejection points, but the rectangles are arranged as adjacent columns, rather than being stacked on their sides. A similar arrangement of quick acceptance of points within the rectangles, with a more complicated accept-reject test for the wedges on top of the columns and the tail is also used. The Ziggurat method improves on this technique by reducing the computation needed to generate a candidate value and increasing the probability of its acceptance. The implementation in the paper also contains a number of improvements which incorporate the conversion from an integer random source to floating point, making the most common code path (where the sample is contained within a rectangle) extremely efficient.

2.4. The Recursive Method (Wallace)

The Wallace random number generator [Wallace 1996] relies on the property that linear combinations of Gaussian distributed random numbers are themselves Gaussian distributed, avoiding the evaluation of elementary functions entirely. Wallace provides several generations of source code referred to as FastNorm1, FastNorm2 and FastNorm3 [Wallace 2005]. Brent has described an optimized implementation on vector processors [Brent 1997] as well as outlined potential problems and remedies for this method [Brent 2003].

The Wallace method uses an initial pool of $N = KL$ independent random numbers from the Gaussian distribution, normalized so that their average squared value is one. In L transformation steps, K numbers are treated as a vector X, and transformed

into K new numbers from the components of the K vector $X' = AX$, where A is an orthogonal matrix. If the original K values are Gaussian distributed, then so are the K new values. The process of generating a new pool of Gaussian distributed random numbers is called a "pass," and R passes are made before the numbers in the pool are used in order to achieve better decorrelation.

The initial values in the pool are normalized so that their average squared value is one. Because A is orthogonal, the subsequent passes do not alter the sum of the squares. This would be a defect, since if $x_1, \ldots, x_N$ are independent samples from the Gaussian distribution, we would expect $\sum_{i=1}^{i=N} x_i^2$ to have a chi-squared distribution χ_N^2. To correct this defect, a random number from the previous pool is used to approximate a random sample S from the χ_N^2 distribution, and all values taken from the pool are scaled by this value before being output. The value used to generate S cannot be further used as a random sample, as it would be correlated with the sum of squares for the next set of output samples, so from each pool of numbers only $N - 1$ are actually returned to be used as normal random numbers.

The pseudo-code is shown in Figure 12. The `generate_addr()` function is used to permute the addresses in a pseudorandom manner, further decorrelating the outputs. Parameter values that provide a good compromise between high statistical quality and performance are $R = 2, L = 1024, K = 4$ as used in FastNorm3.

Algorithm 12. The Wallace method

```
1: for i = 1..R do   {R = retention factor}
2:   for j = 1..L do   {L = N/K}
3:     for z = 1..K do   {K = matrix size}
4:       x[z] ← pool[generate_addr()]
5:     end for      {Apply matrix transformation to the K values}
6:     x' ← transform(x)
7:     for z = 1..K do   {write K values to pool}
8:       pool[generate_addr()] ← x[z]'
9:     end for
10:  end for
11: end for
12: S ← sqrt(pool[N]/N)   {Approximate a χ²_N correction for sum of squares.}
13: return pool[1..(N − 1)] × S   {Return pool with scaled sum of squares.}
```

In Wallace's implementation the orthogonal transform is implemented using a Hadamard matrix. A Hadamard matrix is an orthogonal matrix with the property that all the elements are either $+1$ or -1, making it particularly efficient to implement. With $K = 4$ the following two scaled Hadamard matrices A_1 and A_2 are used in alternating passes:

$$A_1 = \frac{1}{2}\begin{pmatrix} -1 & 1 & 1 & 1 \\ 1 & -1 & 1 & 1 \\ -1 & -1 & 1 & -1 \\ -1 & -1 & -1 & 1 \end{pmatrix} \qquad A_2 = \frac{1}{2}\begin{pmatrix} 1 & -1 & -1 & -1 \\ -1 & 1 & -1 & -1 \\ 1 & 1 & -1 & 1 \\ 1 & 1 & 1 & -1 \end{pmatrix}. \tag{9}$$

Note that $A_2 = -A_1$. For the given set of four values $x[1], x[2], x[3], x[4]$, to be transformed, and with our choice of A_1 and A_2, the new values $x[1]', x[2]', x[3]', x[4]'$; can be

calculated from the old ones as follows:

$$x[1]' = t - x[1];\ x[2]' = t - x[2];\ x[3]' = x[3] - t;\ x[4]' = x[4] - t; \tag{10}$$

and

$$x[1]' = x[1] - t;\ x[2]' = x[2] - t;\ x[3]' = t - x[3];\ x[4]' = t - x[4]; \tag{11}$$

where $t = \frac{1}{2}(x[1] + x[2] + x[3] + x[4])$. This approach, as used in the FastNorm implementations, reduces the number of additions/subtractions required in a matrix-vector multiplication. Orthogonal matrices of size 8 and 16 are obtained by using the property that if H is a Hadamard matrix, then $H'(\begin{smallmatrix} H & H \\ H & -H \end{smallmatrix})$ is also a Hadamard matrix. Appropriate scaling factors should be applied to the Hadamard matrices to preserve a Euclidean norm of 1.

The use of previous outputs to generate future outputs means that the Wallace method is not exact because there will be some correlation between output samples. However, by careful choice of the system parameters the correlation effects can be mitigated to the point where the output Gaussian number quality would be satisfactory for many applications. While the foregoing discussion, and Wallace himself, used Hadamard matrices, other transforms are possible as well (see for example Brent [1997]). The original motivation for using Hadamard was to avoid multiplies, though on machines with dedicated multiply-add instructions, this may not be an important issue.

3. ALGORITHMS FOR GAUSSIAN TAIL SAMPLES

The generation of values from the tails is an important issue, both as a necessary subroutine for some of the previously presented algorithms, and as a means of efficiently testing the distribution of large sigma multiple random numbers. Here we explore techniques that are explicitly designed as algorithms for generating Gaussian random numbers x, with $|x| > r$ for a given value of r. In some cases, it may not be possible to generate these numbers directly. In this case, we generate values of $|x| > q$, where $0 \leq q < r$, and then discard the random numbers until $|x| > r$. We explore how this approach can be followed efficiently for all of the algorithms to be evaluated, with the aim of testing large sigma multiples without requiring the generation of intractably large numbers of random numbers.

All the methods presented here are theoretically exact, but only under the assumption of a source of perfect uniform random numbers and infinite precision arithmetic. The issue of uniform random number generation is considered next, while the effect of finite precision calculations is explored in the evaluation section.

3.1. Accurate Floating Point URNGs

Most methods for generating accurate random numbers from the Gaussian tail distribution rely (either implicitly or explicitly) on the singularity of the logarithmic or division operations for values near zero to transform uniformly distributed numbers to the infinite range required by the Gaussian distribution. The closer the uniform random numbers get to zero, the larger the corresponding output value, although depending on the method, not every such value generated will be used. However, the generation of uniform floating-point values involves subtleties that can significantly affect the accuracy of this method.

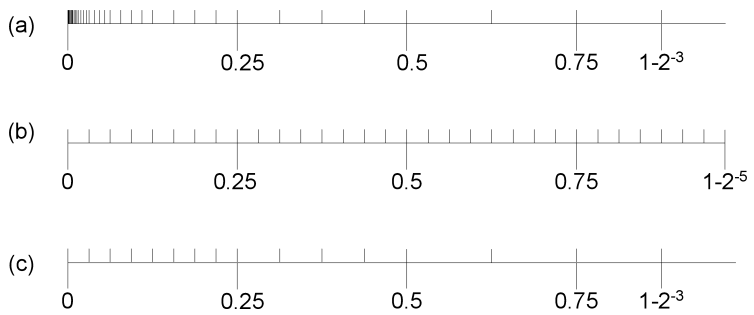

Fig. 7. (a) shows the change in resolution of 4-bit fraction floating-point numbers as the magnitude of the numbers changes, (b) shows the resolution for a 5-bit fixed-point number, such as might be generated by an integer uniform random number generator, (c) shows the results of converting a random number from fixed-point to floating-point. The resulting values inherit both the poor resolution of fixed-point numbers near zero, and the poor resolution of floating-point numbers near one.

Most uniform random number generators produce integer values, while most Gaussian random number generators require floating-point inputs. The standard method for converting a w-bit integer, I, to floating-point is simply to multiply by the floating-point constant 2^{-w}.

Figure 7 demonstrates why this method may lead to problems for GRNGs, particularly near zero. In Figure 7(a) the representable values of 4-bit fraction floating-point numbers are shown. For floating-point, accuracy improves as zero is approached. Figure 7(b) shows the representable numbers for a 5-bit fixed-point value, where the accuracy is the same over the entire range. In Figure 7(c) the result of converting from fixed-point to floating-point is shown, showing how the resulting values inherit the worst of both worlds, with lower precision near zero due to the original fixed-point value, and low precision near one, due to the floating-point representation. An ideal $U(0, 1)$ random number generator should generate every floating-point value with appropriate probability, but if a 32-bit number is converted directly to floating-point through scaling then the smallest number generated is only 2^{-32}. If this were transformed to the Gaussian distribution using the inverse CDF, the maximum value that could be produced is only $\Phi^{-1}(2^{-32}) = -6.2$. Even if 64-bit integers are used, this would only lead to a maximum σ of 9.1, which is still lower than the target of 10σ.

A better method for converting uniform integers to floating-point values would ensure that all representable floating-point numbers in the range $(0, 1)$ could occur. As the density of representable numbers near zero is much higher than near one, the values near zero will need to have a correspondingly lower chance of occurring.

Such a random number generator is used in the Matlab environment, where the rand function is capable of generating any representable double-precision value in the range $(0, 1)$ [Moler 1995, 2004]. This method uses two uniform random number generators, and uses a truncated $\log_2$ operation on one random number to give the required geometric distribution for the uniform random number's floating point exponent, then uses another random number to ensure that the floating-point mantissa is fully-randomized. Even this generator, however, is not without flaws—for example it is relatively slow, and this could be problematic in some applications.

An alternative method is to simulate an infinite precision fixed-point random value between 0 and 1, but to only generate as many of the leading bits as can be represented in a floating-point value. An n-bit uniform random number can be generated by concatenating $\lceil n/w \rceil$ w-bit random numbers. If the floating-point fraction is m bits wide, then half of the time (when the most significant bit of the n-bit wide fixed point number

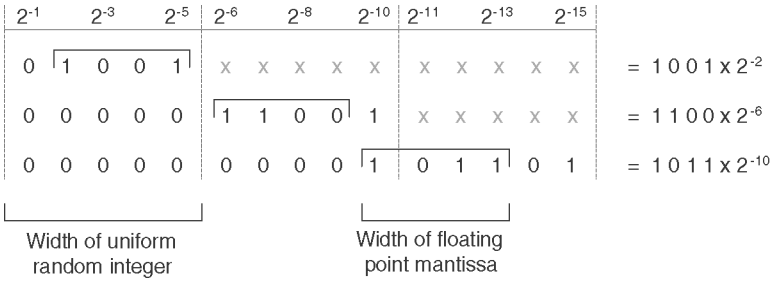

Fig. 8. Extended-precision fixed-point.

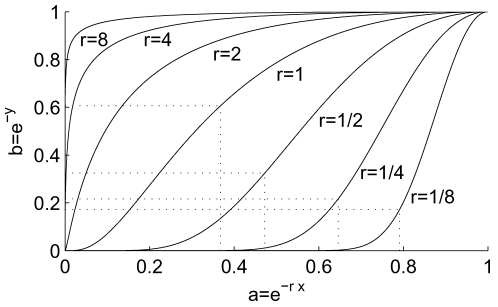

Fig. 9. Graphical representation of the acceptance regions for Algorithm 16 with different values of r. The x and y axes are the absolute input values, and areas under the curve are acceptance regions. The dotted boxes show the restricted region that must be sampled to produce values greater than 2.2.

is one) only the first m bits are needed, a quarter of the time only the first $m + 1$ bits are needed, and in general the probability of needing b or fewer bits is $1 - 0.5^{b-m}$. We can take advantage of this observation by expanding the fixed-point number until the most-significant non-zero bit is seen, to provide the floating-point fraction.

Figure 8 demonstrates this technique using the same 4-bit fraction floating-point and 5-bit fixed-point system used earlier. Each group of five bits represents a uniform random number, and as soon as the first one, and trailing $m - 1$ digits, have been determined, the following bits are not needed (indicated by grey crosses). Simplified pseudo-code for the case where $m < w$ is shown in Algorithm 13, but in practice the code can be simplified using bitwise operations and machine-specific instructions. In particular the most common case, where a one is found within the first random integer, can be optimized, costing just one extra comparison per uniform random number (which replaces an equivalent comparison to check for zero, which is no longer needed). In the common case of IEEE single precision, where $m = 24$ and $w = 32$, the extra code is only needed for 1 in every 256 calls.

A potential advantage of this method over the Matlab technique is that on average, only slightly more than one integer uniform sample is needed to generate each floating-point output sample and hence may be faster, depending on the relative speed of the underlying operations. If the integer random numbers are of a different overall width than the floating-point format then more integer samples may be needed; for example, two 32-bit integers will be needed for each 64-bit double-precision sample.

Algorithm 13. Method for producing floating-point numbers with fully-random fractions (where $w > m$)

```
 1: c = 1    {Sets maximum value that can be generated}
 2: repeat
 3:   x ← I_1, c ← 2^{-w} c
 4: until x ≠ 0    {Loop until first one is found}
 5: t ← w    {Number of random bits left in x}
 6: while x < 2^{w-1} do
 7:   t ← t − 1, x ← 2x, c ← c/2    {Shift first one to MSB}
 8: end while
 9: if t < m then    {Add more random less significant bits if necessary}
10:   x ← x + 2^{-t} I_2    {Right shift new value into place}
11: end if
12: return c x    {Convert to floating-point}
```

Signed floating-point numbers can be generated in the same way, with the sign determined by an extra bit from the URNG.

3.2. CDF Inversion

To generate $|x| > r$ from an approximation to the inverse CDF $G(u) = \Phi^{-1}(u)$, one can simply restrict the inputs of the generators to uniform values in the ranges $(0, \Phi(-r)]$ and $[\Phi(r), 1)$. However, the asymmetric accuracy of floating-point representation over the range $(0, 1)$, shown in Figure 7(c), means that although negative output values can be accurately reproduced, as input values very close to zero can be represented, positive values cannot, due to the lower accuracy of floating-point numbers near 1. This asymmetry leads to a potentially large asymmetry between the negative and positive tails.

For example, in single-precision IEEE the smallest number greater than zero is roughly 10^{-44}, while the largest number less than one is about $1 - 10^{-7}$. It is thus possible to represent numbers much closer (more than 30 orders of magnitude) to zero than to one. This means that the largest possible Gaussian number that can be produced by CDF inversion is around $+5$, while the smallest value is -14. Even worse, the large value will occur with much higher probability, as the corresponding input value covers a larger segment of the uniform range.

A solution to this problem is to only apply the inverse CDF approximation to values less then 0.5 and to attach the sign afterwards. Internally many of the CDF inversion techniques already perform this step to take advantage of the Gaussian distribution's symmetry, since this calculation may be achieved at little or no cost. Pseudo-code for generating values from the tails is show in Algorithm 14, although this organization should be applied to any uses of the CDF inversion technique for Gaussian random number generation, not just when tail values are the focus.

3.3. Marsaglia Tail Algorithm

Marsaglia proposed an algorithm specifically for sampling from the tails [Marsaglia 1964], and it was used to produce random numbers from the tails in the Marsaglia-Bray rejection method from Section 2.3.2 [Marsaglia and Bray 1964] and the Rectangle-Wedge-Tail method mentioned at the end of Section 2.3.6 [Marsaglia et al. 1964]. The

Algorithm 14. Sampling From the Tails Through CDF Inversion

1: $a \leftarrow U$
2: **if** $a < \frac{1}{2}$ **then** {Extract random sign from uniform sample}
3: $s \leftarrow 1,\ a \leftarrow 2a$
4: **else**
5: $s \leftarrow -1,\ a \leftarrow 2a + 1$
6: **end if**
7: $a \leftarrow \Phi(-r)\frac{1}{2}a$ {Scale uniform sample a to smaller range.}
8: $x \leftarrow G(|a|)$ {Where $G(u) \approx \Phi^{-1}(u)$}
9: **return** sx {Attach random sign.}

algorithm is based on the Polar method, first generating two uniform values whose sum-of-squares is less than 1, then performing a transformation biased to produce values over a threshold. Algorithm 15 gives pseudo-code for the method, an obvious difference from the polar method is that two rejection steps are required rather than just one, as even after selecting a suitable pair of uniform values there is no guarantee that either will be larger than $|r|$.

Algorithm 15. Original Marsaglia Tail Method

1: **loop**
2: **repeat**
3: $a \leftarrow V_1, b \leftarrow V_2$
4: $d \leftarrow a^2 + b^2$
5: **until** $0 < d < 1$
6: $t \leftarrow \sqrt{\frac{r^2 - 2\ln d}{d}}$
7: $x \leftarrow ta,\ y \leftarrow tb$
8: **if** $|x| > r$ **then**
9: **return** x
10: **else if** $|y| > r$ **then**
11: **return** y
12: **end if**
13: **end loop**

In his more recent work, Marsaglia introduced [Marsaglia and Tsang 1984b] and used [Marsaglia and Tsang 1998, 2000] a different version of the tail algorithm. The algorithm is shown as Algorithm 16, and requires only one loop and fewer operations than the original method, although it requires two logarithms per iteration rather than just one.

Algorithm 16. New Marsaglia Tail Method

1: **repeat**
2: $a \leftarrow V_1, b \leftarrow U_2$
3: $x \leftarrow -\frac{1}{r}\ln|a|,\ y \leftarrow -\ln b$
4: **until** $2y > x^2$
5: **return** $a > 0\ ?\ r + x\ :\ -r - x$

3.4. Box-Muller

With reference to Algorithm 1, it is clear that the magnitude of the outputs is bounded by a since the subsequent steps multiply a by a value between -1 and 1. Thus, in order to generate all values above some threshold r, cases where:

$$u_1 > e^{-\frac{1}{2}r^2} \tag{12}$$

can be ignored.

3.5. Polar-Rejection

As with Box-Muller, the Polar method described in Algorithm 3 is bounded by f, which provides its magnitude. In this case it is derived from both inputs:

$$r = \sqrt{-2\frac{\ln d}{d}},\ d = x^2 + y^2. \tag{13}$$

The maximum value of x that needs to be considered happens when $y = 0$, and vice versa, so the equation can be simplified to $d = x^2$. This gives

$$x, y < \sqrt{e^{-\frac{1}{2}r^2}}, \tag{14}$$

which, unsurprisingly, is the square root of the Box-Muller limit. This method is also closely related to Marsaglia's original tail method.

3.6. GRAND

The odd-even method can be used as a tail production method simply by altering the table of constants. To produce values above q the first table entry needs to be set to $A[0] = q$, then the rest of the entries can be calculated using the recurrence $A[i] = \Phi^{-1}(1 - \Phi(-A[i-1])/2)$. As with the full Gaussian GRAND generator, the maximum sigma-multiple that can be achieved in the tails is limited by the size of the table.

3.7. Ratio-of-Uniforms

Large values in the ratio-of-uniforms method are produced when small values of the denominator u occur, corresponding to values very close to the origin of Figure 4. To limit generation to values above r, we first determine the upper limit for the u axis, u_f, and then select the minimum threshold v_f for the v axis that encloses that area:

$$v^2 = -4u_f^2 \ln u_f \qquad \frac{v}{u_f} = r \tag{15}$$

$$u_f = \exp\left(\frac{1}{4}r^2\right) \tag{16}$$

$$v_f = \begin{cases} \sqrt{-4u^2 \ln u_f}, & \text{if } u_f < e^{-1/2} \\ \sqrt{2/e}, & \text{otherwise} \end{cases}. \tag{17}$$

Unfortunately, this method results in rapidly decreasing acceptance ratios as r is increased, with 33% of points accepted at $r = 1$, 10% at $r = 2$, and 3% at $r = 4$. Due

to these poor acceptance rates, the ratio-of-uniforms method is unlikely to be a good candidate for generating samples from the tail distribution.

3.8. Ahrens-Dieter

In the Ahrens-Dieter generator the exponential and Cauchy distributions interact to determine the magnitude of the output sample, but it is the exponential component that determines the magnitude. For samples x and y, from the exponential and Cauchy standard distributions respectively, the maximum output magnitude is given by:

$$\max\left(y\sqrt{2x/(1+y^2)}, \sqrt{2x/(1+y^2)}\right). \tag{18}$$

This equation is maximized at $y = 0$ and $y = \infty$, leading to the minimum value of x that should be generated for a threshold r of:

$$x_f = r^2/2. \tag{19}$$

This can be implemented in Algorithm 7 simply by initializing a to $A + x_f$ in step 1.

4. TESTS, TEST PARAMETERS AND RESULTS

Much has been written on the general issue of testing for randomness and the specific issue of testing for goodness of fit. Good overall discussions for testing of uniform random number generators can be found [L'Ecuyer 1992, 2001]. Sophisticated test suites and procedures are available from the U.S. NIST [Rukhin et al. 2001] and from the "Diehard" tests developed by Marsaglia [1997]. Another comprehensive test suite for randomness is TestU01 [L'Ecuyer and Simard 2005], which provides a wide selection of paramaterizable tests, as well as some predefined test suites such as Crush [L'Ecuyer 2001]. Specific attention to GRNGs can also be found [Molle et al. 1992].

4.1. Testing Methodology

The main uniform random number generator used in the tests presented here is the MT19937 Mersenne Twister [Matsumoto and Nishimura 1998], which is among the best quality generators commonly in use. As well as passing all common empirical tests for randomness, the generator also has a number of theoretically determined qualities such as a very long period and good equidistribution. Although not the fastest generator available, it provides a good trade-off between speed and quality, and so it is used as the generator in all tests for statistical quality. It is also used as the main generator in performance tests, although additional results using alternate uniform generators are also explored.

4.1.1. Goodness-of-Fit Tests. Both the standard and tail generation algorithms are evaluated using the χ^2 test. In the classic χ^2 test, a set of observed samples is compared against the expected distribution. This requires that a histogram be constructed and the frequency compared with the expected number. The number and allocation of histogram "buckets" is an area of substantial flexibility. Using more buckets gives higher resolution with respect to different input values, but reduces the expected number in each bin. Another choice is whether to use regularly-spaced bucket boundaries or equal-probability buckets. In order to achieve maximum sensitivity an equal-probability bin arrangement is used in preference to one with regular spacing, as it allows more bins to

Table I. Number of bins k (according to Equation 20) and expected number of samples bucket E used in χ^2 tests for different numbers of samples n.

n	2^{10}	2^{20}	2^{30}	2^{36}
k	64	4096	2^{18}	3.17×10^6
E	16	256	4096	21619

be used without the minimum expected count in any bin falling below the level at which the χ^2 assumptions break down. In the results presented here, the number of buckets, k, used in a test, is determined from the number of samples, n, to be accumulated, through the following formula:

$$k = \lceil n^{3/5} \rceil. \tag{20}$$

The choice of $3/5$ as the exponent is somewhat arbitrary, and the trade-off is between avoiding overly wide buckets, which inhibits the ability to identify local inaccuracies in the PDF, and overly narrow buckets, which can lead to too few samples per bucket for statistical significance. Often, an exponent of $1/2$ is used, though in our experiments we found this produced too few bins.

Table I gives the number of bins, k, and expected frequency, E, for different numbers of samples, n, under this approach. Each generator is tested with successively larger sample sizes, starting from 2^{10} and doubling in size up to 2^{36}. After processing each batch the p-value is calculated: a p-value greater than 0.1 is considered a pass, and the next batch size will be tested; a p-value less than 10^{-6} is an immediate fail, and the generator will be considered to have failed at that sample size; or if the p-value is in between, another batch of the same size is tested, until the geometric mean of all the p-values at a given batch size rises above or falls below the pass or fail threshold. The same χ^2 test is applied to both Gaussian generators and tail generators, with the appropriate distribution used in each case in order to ensure equal probability buckets.

The χ^2 test is the only goodness-of-fit used here to test the Gaussian distribution (as opposed to the Gaussian tail distribution), as it is found that in this context the Empirical-Distribution-Function (EDF) tests, such as the Anderson-Darling and Kolmogorov-Smirnov tests, do not have significantly better analytical power. The chosen bucketing strategy provides a relatively fine-grain analysis of the structure, and using an equal-probability scheme means that the coverage across the distribution is good even for tail values, removing the need for EDF tests. It is also extremely difficult to apply EDF tests to large numbers of samples, due to the need to store and sort the entire sample before calculating the test statistic. However, the EDF tests are used in the high sigma-multiple tests described next.

4.1.2. High Sigma-Multiple Tests. One of the goals of our tests is to assess the performance of the GRNGs in the tails. This goal poses an additional challenge in that, on the one hand, the randomness tests require large numbers of samples to achieve appropriate sensitivity, but samples in the tails occur infrequently. If, for example, one million samples from the tail region beyond $|x| > 10\sigma$ are desired, it is impractical to run an unmodified GRNG long enough to accumulate the desired number of samples.

Our approach, therefore, is to modify the GRNGs to force the generation of random numbers with large σ multiples. Here we examine how the different ways in which values from the tails can be generated, as well as how each method could be forced to produce values over a given threshold. These forcing techniques are used with special attention to ensure that the algorithms are not changed from their standard forms. The

only change made is to modify the input uniform random numbers that the algorithms consume, with all other constants remaining the same, in order to guarantee that the large sigma-multiple random numbers generated accurately characterize the original generators.

In order to avoid introducing bias, the uniform random numbers are filtered in their original integer form before conversion to floating-point representation. To filter for values below a certain threshold, the uniform random numbers are first reduced to the nearest power of two range using a bitwise mask, then the final range reduction to the desired threshold is performed through rejection.

As noted further below, there is no clear way to force the Wallace method to generate high sigma-multiple values. Thus the brute force approach of generating a large number of samples and selecting all values over a certain threshold is applied.

The aim of the high-sigma multiple tests is to determine at what point a generator starts to deviate significantly from the Gaussian distribution. The approach used here is to determine for each generator a threshold value q, such that a set of n samples above this threshold will not conform to the Gaussian tail distribution. Although this threshold will clearly depend on the value of n and the method used to detect goodness-of-fit, it allows meaningful comparisons between generators to be made, and gives a general idea of the maximum absolute value (and hence total number of samples) that a generator can accurately produce.

When testing the performance of a generator at forced high sigma-multiples, it is necessary to reject many candidate samples, even when manipulating the source random number generator. Combined with the inherent inefficiency of many methods at high sigma-multiples, it is computationally infeasible to generate 2^{36} random numbers (as was used with the basic goodness-of-fit test described above), particularly as many different thresholds must be tested to determine that failure point. Instead, a pool of n samples is maintained, and as the threshold is raised, all samples below the threshold are discarded and replaced with new samples above the threshold. Because many samples in the pool will not change as the threshold is raised, there will be correlations between goodness-of-fit statistics at successive thresholds, which tends to lead to a gradual decrease in p-values. This issue is taken into account by reporting the highest threshold at which a "good" p-value is seen, rather than the first threshold at which a "bad" p-value is seen.

The algorithm for this testing procedure is shown in Algorithm 17. The pool size, n, is chosen to be 100000, while Δt is 0.1. The goodness-of-fit algorithm applied in step 4.1.2 uses the EDF based Anderson-Darling and Kolmogorov-Smirnov tests, returning whichever of the two p-values is lower.

When we apply the high sigma-multiple tests, it is infeasible to run each generator until sufficient samples have been gathered. To test the distribution of a generator above 8σ using brute-force, over 1.6×10^{20} values, would need to be generated. Instead the generators are adjusted such that given a target sigma q, the generator is guaranteed to generate all possible values greater than q, but is no longer guaranteed to generate samples below q. This adjustment is achieved by limiting the range of the uniform samples generated for use within the generator, and the generator algorithm itself remains unchanged.

The method for constraining uniform random number generation must be calculated for each generator type. In many cases the methods described in Section 3 can be used unchanged, but further constraints are required for some of the algorithms.

Old Marsaglia Tail Method. When forcing the production of values above a threshold q for an old-style Marsaglia Tail generator, the existing threshold parameter r must be taken into consideration (where $q > r$). The correct constraints are achieved by only choosing values for the uniform random numbers a and b that produce values of d

Algorithm 17. High Sigma-Multiple test algorithm

```
1: S ← ∅, q ← 0, g ← 0
2: loop
3:   S ← S/{s ∈ S : |s| < q}    {Remove values below current threshold}
4:   while |S| < n do    {Replace any discarded samples}
5:     x ← Generate()
6:     if |x| > q then
7:       S ← S ∪ {x}
8:     end if
9:   end while
10:  p ← EDF(S)    {Apply EDF tests to get p-value for sample}
11:  if p > 0.01 then
12:    g ← r    {Record last-known-good point}
13:  else if p < 10^−6 then
14:    return g    {On failed p-value return last-known-good point}
15:  end if
16:  q ← q + Δq
17: end loop
```

below a certain threshold d_q:

$$q = \sqrt{d_q}\sqrt{\frac{r^2 - 2\ln d_q}{d_q}} \tag{21}$$

$$d_q = \sqrt{\exp(r^2 - q^2)}. \tag{22}$$

Hence only values of $|a|$ and $|b|$ less than d_q need be generated in order to force values larger than r.

New Marsaglia Tail Method. This method is not symmetric, so the two values a and b must be limited separately. From inspection, it is clear that a directly controls the magnitude of the output, while b is only used for rejection. We first limit the range of a by choosing $a < a_q$ where a_q defines the maximum value for a, and then calculate a corresponding limit b_q for b that reduces the rejection probability as much as possible:

$$a_q = \exp(r^2 - qr) \tag{23}$$

$$b_q = \exp\left(\frac{-\ln^2 a_q}{2r^2}\right). \tag{24}$$

GRAND. Forcing values above a threshold q for an existing generator can be achieved by ignoring sections of the table that produce lower values. So find an index i_q, such that $A[i_q] < q < A[i_q + 1]$, and only generate table indices greater or equal to i_q in Algorithm 10. Because the area in sector i is equal to the area in all sectors at higher indices, at least half the time the generated values will be above q, although this could be slightly improved by restricting the values of w generated in step 2.3.5 of Algorithm 10.

Wallace. Generating or forcing high σ multiple outputs with the Wallace method is difficult. As noted earlier, the Wallace method utilizes linear transformations of

previous outputs. One can insert one or more large values into a pool and be confident that the subsequent pool will be more likely to contain larger values as a result. However, by definition, this approach intentionally creates and utilizes inter-pool correlations, and the degree of "randomness" has been substantially reduced. The alternative, while computationally expensive but certainly cleaner, is to simply run the generator for long enough (fortunately it is very fast) to accumulate the desired number of high σ multiple outputs.

4.1.3. Conversion to a Uniform Distribution. In order to test for statistical randomness, we use the Crush battery which is part of the TestU01 suite. The Crush battery applies 94 separate tests for uniform randomness, consuming a total of about 2^{35} inputs. The input random numbers can be provided as double-precision floating-point or 32-bit integer values, as long as the numbers contain at least 30 "random" bits.

The Gaussian distributed samples output by the generator under test are mapped to the uniform distribution by applying the Gaussian CDF to each random number in the sample. The mapping process is performed using a double-precision Gaussian CDF approximation [Marsaglia 2004] with absolute error less than 10^{-15}, and so will provide more than the required 30 bits of accuracy when applied to double-precision Gaussian samples.

The tests for statistical randomness assume that the inputs will be 32-bit uniform random integers, and if there are less than 30 bits, some tests will always fail (although TestU01 does support parameterization for different numbers of random bits, the predefined battery Crush does not support this). However, if a Gaussian single-precision floating-point value is transformed to a uniform 32-bit integer then only a subset of integers can be produced, due to the limited accuracy of the floating-point source value. For example, consider the Gaussian values 1 and $1 + \epsilon$, the next largest representable number. In single precision $\epsilon = 1.192092896 \times 10^{-7}$, so $2^{32}(\Phi(1 + \epsilon) - \Phi(1)) = 124$. This means that there are 123 integers that cannot occur after the transformation, and over the entire range there are thousands of values that cannot occur. The effect of this is to interfere with the randomness of the low-order bits of the generated numbers, effectively reducing the number of random bits to a value less than 32.

To allow the existing tests to be used without modifying them for fewer bits, we retain the n random most-significant bits and drop the $32 - n$ low bits, replacing them with bits from another uniform random number generator. The generator supplying the additional low bits is known to pass the test suites, and so if the combination of Gaussian-derived high bits and additional low bits also passes the tests, then we can say with some confidence that the Gaussian generator provides *at least* n random bits, although it may provide more.

In the studies reported here, we retain 23 bits of precision after the transformation, motivated in part by the fact that the fraction in IEEE 754 single precision floating-point arithmetic utilizes 24 bits. The choice of 23 bits was made as a compromise that allows one bit of "spare" precision in the single-precision Gaussian representation, and two bits in the integer uniform representation. No detailed analysis of the maximum number of bits that could be retained is made, but we note that at least some of the generators pass the test suites with 23 bits, showing evidence that this does not exceed the maximum number of bits that can be retained. Also, as we mention earlier, for Gaussian values near 1 there are gaps of at least 123 between the possible integer values, suggesting that at best $32 - \lceil \log_2 123 \rceil = 25$ bits could be used. However, this is an upper bound, and the non-exact transform from the Gaussian distribution to the uniform distribution could possibly further reduce the number of bits that can be safely used.

4.1.4. Test for Interblock Correlations. The Wallace random number generator has a defect whereby large output samples bias the distribution of the nearby samples in the sequence. To detect such biases it is necessary to wait for samples that exceed a selected trigger threshold, then test the distribution of the samples following the trigger sample (not including the trigger sample itself). The distribution of each sample should be independent of any preceding samples, but if the defect exists then the distribution of samples closely following large values will be biased away from the Gaussian distribution towards large values.

The test used here is to choose a trigger threshold t, then to generate blocks of k samples $S_i = x_{ki} \ldots x_{ki+k-1}$. A block S_i, which contains a value with absolute value greater than k, acts as a trigger block, and the following block S_{i+1} is then added to the set F of samples to be tested. If F has not yet reached a target size, then the process continues by examining block S_{i+2}.

Once the size of F has reached a target size n, its distribution is investigated by using a χ^2 test with 16 equal probability (under the expected Gaussian PDF) buckets. If the resulting p-value is greater than a "good" threshold p_g then n is recorded as the last known good sample n_g, while if the p-value is less than a failure threshold p_f the test is reported as failed with a sample count of $n_g = n_g + 1$ and the test is finished. If the test is not failed then the target count n is doubled, and more blocks are added to F until either the test fails or n exceeds a maximum value.

Our tests are performed for $n = 2^{14} \ldots 2^{32}$, with a block size of $k = 2^{10}$, the last known good threshold $p_g = 0.01$, and the failure threshold $p_f = 10^{-6}$. The test could be made more sensitive by changing these parameters.

4.2. Results: Gaussian Generators

Table II gives the relative speed and operation count for each of the algorithms. The speed is expressed relative to that of the Polar Rejection method, as this is a simple and commonly used method and so can be considered as a baseline for performance. The underlying absolute speed is calculated as the geometric average of the measured speed on four different platforms (described shortly), using the Mersenne Twister [Matsumoto and Nishimura 1998] as the source generator. In all cases, direct implementation of the algorithms using the C++ programming language is used, with no explicit attempt to perform processor-specific optimization. The table also contains a full breakdown of the operation counts. For operations that occur only conditionally, average numbers derived either analytically or based on simulation are presented.

The Wallace algorithm provides the highest performance, but only when the quality parameter is at its lowest setting. The Ziggurat, while not as fast as the Wallace method, has better statistical properties with respect to correlation. Table III provides a break-down of the Ziggurat and Mersenne Twister combination's speed across the four platforms used for benchmarking. These consist of two Intel and two AMD processors, using versions of either Microsoft Visual Studio (*msvc*) or the GNU Compiler Collection (*gcc*) to compile and link the executables. The peak speed (measured in millions of generated samples per second) of 56.96 MSamples/s is achieved using the Pentium-4, which is also the highest clock rate processor tested. If the generator sample rate is scaled by the processor clock rate, then the Pentium-4 actually provides the worst performance per processor cycle, and the Opteron the best.

Figure 10 shows the performance (not adjusted for clock rate) for a reduced set of generators, relative to the geometric mean across all platforms. Except for the Ziggurat method, the Opteron is the fastest for all other generators (including those not shown in the chart), it just happens that the Pentium-4 is fastest for the most important generator. In some cases, for example, Box-Muller, the Pentium-4 is actually slower

Table II. Table showing speed relative to the polar rejection algorithm, and operation counts per generated random number. U is the number of input uniform numbers, and C is the number of constants used in the implementation. All calculations are performed in single precision, with full-fraction floating point uniform inputs.

	Speed	U	+	×	÷	Cmp	$\sqrt{x}$	Ln, Exp, Trig	C
Wallace (qual = 1) [1996]	6.41	0.001	10.02	1.50	ϵ	1.51	ϵ		9
Ziggurat [2000]	4.29	1.04	1.10	1.07		2.07		0.001, 0.03, 0	388
Wallace (qual = 4) [1996]	2.48	0.003	37.07	3.01	ϵ	3.04	ϵ		9
Monty Python [1998]	1.61	1.30	0.88	1.96		2.57		0.03, 0, 0	16
PPND7 (ICDF) [1988]	1.16	1	8.15	7.40	1	1.45	0.15	0.15, 0, 0	26
Mixture-of-Triangles [2000]	1.14	3	3	2	1	1			122
Polar [1969]	1.00	1.27	1.91	3.27	1	1.27	1	1, 0, 0	4
Leva (Ratio) [1992b]	0.98	2.74	6.84	6.89	1	3.12		0.01, 0, 0	9
Marsaglia-Bray [1964]	0.94	3.92	3.22	1.36	0.01	1.42	0.006	0.01, 0.05, 0	15
GRAND [1974]	0.92	1.38	8.65	6.49	1.16	4.88			27
Box-Muller [1958b]	0.81	1		2		0	0.5	0.5, 0, 1	2
Ahrens-Dieter [1988]	0.78	1.02	4.55	4.04	1.5	4.51	0.5	0, 0.01, 0	20
Kinderman (Ratio) [1977]	0.76	2.74	3.20	4.34	1.84	3.44		0.23, 0, 0	6
Hastings (ICDF) [1959]	0.62	1	8	7	2	1	1	1, 0, 0	7
PPND16 (ICDF) [1988]	0.55	1	14.45	14.85	1	1.45	0.15	0.15, 0, 0	52
Central-Limit (n = 12)	0.39	12	12						1
CLT-Stretched [1959]	0.35	12	17	8					5

Table III. Performance comparison of the ziggurat generator across four different platforms, using the mersenne twister generator as the source for uniform random numbers. Performance is measured using millions of generated samples per second (MSamples/s), and relative performance is in comparison to the geometric mean of the four platforms.

			Observed Performance		Adjusted for Clock Rate	
Processor	GHz	Compiler	MSamples/s	Relative	MSamples/s/GHz	Relative
Pentium-M	1.73	msvc 2005	37.11	0.78	21.45	1.02
Athlon-MP	2.13	msvc 2003	46.14	0.97	21.66	1.03
Pentium-4	3.20	gcc 3.4.3	56.94	1.20	17.79	0.84
Opteron	2.20	gcc 3.4.5	52.72	1.11	23.96	1.14

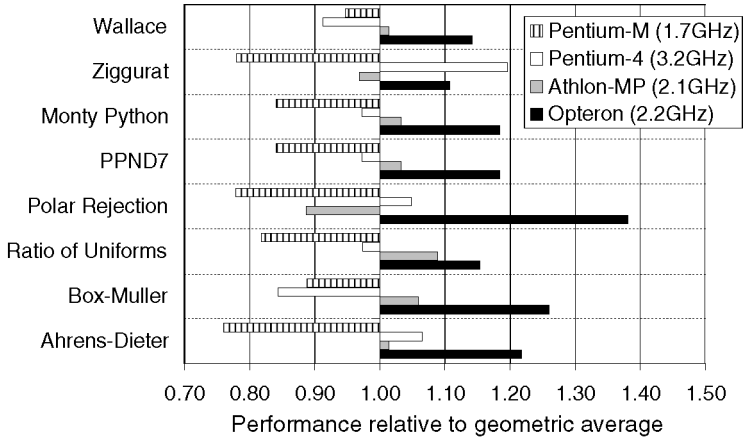

Fig. 10. Performance for generators on different platforms, relative to the geometric mean performance across platforms.

than the Pentium-M, even though it is running at almost twice the clock rate (possibly due to superior support for floating-point intrinsics in the Microsoft compiler). However, even with this significant variation between platforms, the relative ordering of the fastest four generators always followed that shown in Table II.

This evaluation used the Mersenne Twister as the source of uniform random numbers, as it is a well established and widely used high quality generator. However in certain situations it may be acceptable to degrade the quality of the uniform random numbers in favor of speed, or a platform may provide an instruction for fast hardware random number generation. Figure 11 compares the absolute performance of a subset of the Gaussian generators using three different uniform sources on the Opteron 2.2GHz test platform. A less complex Combined Tausworthe generator (Taus88) is used, which provides higher speed but lower statistical quality [L'Ecuyer 1996] as well as the Mersenne Twister. An even higher speed uniform generator is provided using a "Quick and Dirty" Linear Congruential generator [Press et al. 1997], which requires just one addition and one multiplication per output sample. However, it has significant statistical defects, and is only used here to provide the simplest possible generator that will allow the Gaussian generators to function correctly. In most of the cases in Figure 11, the variation in speed is small, even when moving from the complex Mersenne Twister to the extremely simple Quick and Dirty generator. The difference is most noticeable in the Ziggurat method, where performance is more than doubled by using an extremely fast uniform random number generator.

One aspect of the generator algorithms we have not considered in this article is the possible vectorization of algorithms. For obvious reasons, this has the potential

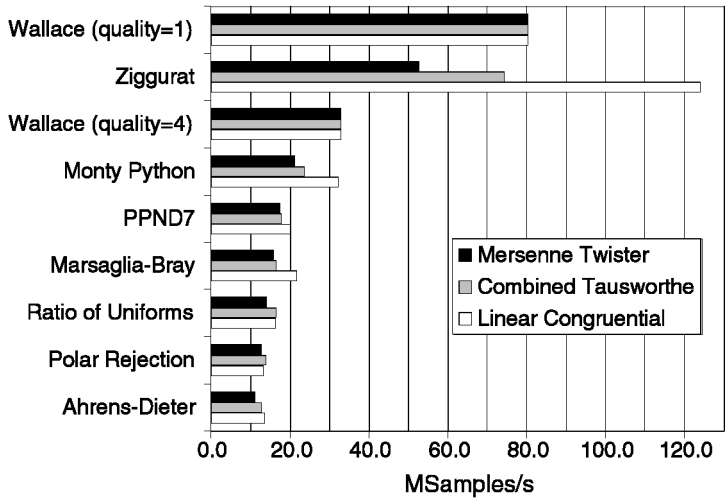

Fig. 11. Performance of selected Gaussian generators using different uniform random number sources on an Opteron 2.2GHz.

Table IV. Statistical quality of generators as measured by the χ^2 and high sigma-multiple tests for single-precision generators, using standard integer to floating-point conversion (*standard*) and fully-random fraction (*Full-Fraction*) conversion. Generators passing the χ^2 test for more than 2^{36} samples are shown using "+." Where high sigma testing becomes computationally infeasible before generator failure, the point at which testing stopped is suffixed with "+." An entry of "n/a" indicates that the test or parametrisation do not apply to that particular generator.

	χ^2 Test ($\log_2(n)$)		High Sigma Test	
	Standard	Full-Fraction	Standard	Full-Fraction
Wallace (qual = 1) [1996]	+	n/a	6+	n/a
Ziggurat [2000]	+	+	8.15	17.4
Wallace (qual = 4) [1996]	+	n/a	n/a	n/a
Monty Python [1998]	34	n/a	8.27	14.88
PPND7 (ICDF) [1988]	34	34	4.11	12.64
Ahrens-Dieter [1988]	15	+	17.3	17.3
Mixture-of-Triangles [2000]	26	n/a	n/a	n/a
Polar [1969]	36	+	8.09	11.59
GRAND [1974]	36	+	9.2	17+
Hastings (ICDF) [1959]	29	30	5.25	12.64
Leva (Ratio) [1992b]	+	+	7.91	17+
PPND16 (ICDF) [1988]	35	+	4.11	13.7
Marsagla-Bray [1964]	35	+	8.35	15.78
Box-Muller [1958b]	26	35	5.57	13.96
Kinderman (Ratio) [1977]	+	+	7.91	17+
Central-Limit (n = 12)	20	n/a	0.99	n/a
CLT-Stretched [1959]	28	n/a	2.84	n/a

to greatly speed up execution. The challenge is that it is difficult to make any general statements about vector performance in light of the many differences between different vector and SIMD architectures and in the possible ways to exploit these. The issue of vectorised random number generators has been addressed in Brent [1993] and Brent [1997], where the performance of the Box-Muller, Polar, Ratio-of-Uniforms and Wallace algorithms is considered. Of these the Wallace appears to offer the best performance.

Table IV shows the χ^2 goodness-of-fit results for the Gaussian generators as well as the tests for high sigma-multiple correctness. The χ^2 test results are presented either as

Table V. Statistical quality of generators as measured by crush for single-precision and double-precision generators. All generators were tested, but only generators that failed at least one test are shown here.

Generator	Single-Precision		Double-Precision	
	Failures	Classes	Failures	Classes
Ziggurat [2000]	0	0	1	1 (COLL)
Mixture-of-Triangles [2000]	3	3 (MOT,SP,WD)	3	3 (MOT,SP,WD)
GRAND [1974]	2	1 (MOT)	0	0
Ahrens-Dieter [1988]	3	3 (COLL,BDAY,PIS)	3	3 (COLL,BDAY,PIS)
Hastings (ICDF) [1959]	2	1 (MOT)	2	1 (MOT)
Central-Limit (n = 12)	19	12	19	12
CLT-Stretched [1959]	3	2 (COLL,MOT)	3	2 (COLL,MOT)

the + symbol, indicating that the generator does not fail the tests for samples sizes less than or equal to 2^{36}, as an integer, representing the binary power at which the test failed, or as not applicable ("n/a") if the tests do not apply. The high sigma-multiple tests are shown as a number, indicating the point (sigma-multiple) above which the tests fail, or if suffixed with + then the point at which testing is stopped due to excessive computation time. The tests are applied in single-precision using both direct conversion from integer to floating-point random numbers, and using the method for fully-random fractions. The χ^2 tests are also applied using double-precision arithmetic, but the results are not shown here, as the only observed differences from the single-precision full-fraction results are that both the Box-Muller and PPND7 pass the χ^2 tests.

Table V shows the results of applying the Crush battery to the generators. Only generators that produce failures are shown. Single-precision results are collected by masking in 10 bits of "good" randomness after transforming to uniform as explained earlier, while double-precision tests are transformed directly. The tests were performed using both standard integer to floating-point conversion, and fully-random fraction conversion, but this was not found to change the results of the tests. The *Failures* column indicates the total number of failed tests, while the *Classes* indicates how many different types of test fail, since some classes of statistical tests are applied with different parameters. Where only a few tests fail, the specific cases are identified using the key: COLL = Collisions, MOT = Max-Of-T, SP = Sample-Products, WD = Weight-Distribution, BDAY = Birthday, PIS = Period-in-Strings.

The most commonly failed test is Max-Of-T, which collects groups of samples and examines the statistics of the maximum element in each group. Generators that produce a poor Gaussian curve fail this test due to a poor distribution in the near tails. The Ziggurat method passes all tests in single-precision, but in double-precision, fails a single test, the Collisions test. This is because in the published version of the Ziggurat algorithm [Marsaglia and Tsang 2000] the same random value is used both to select the rectangle and to provide the sample, so there is a correlation between the low bits of the sample used to select the rectangle, and the absolute magnitude of the sample. Using an independent random number generator to select the rectangle fixes this minor problem. For example, the eight least significant bits of the floating point fraction can be replaced using bit-wise operations, requiring one extra 32-bit uniform random integer for every four generated Gaussian random samples.

The test for inter-block correlations is applied to the Ziggurat and the Monty Python generators, and to the Wallace generator with quality levels (number of pool transformations per output) of 1 and 4. The slower generators are not tested due to the large numbers of samples that must be generated in these tests when the generator passes, and it is expected that all will pass apart from Wallace. Initially the test is applied to all generators for triggering thresholds from 1 to 5. The Ziggurat and Monty Python generators passed all tests.

Table VI. Number of samples ($\log_2$) before bias is noticeable in wallace algorithm for increasing trigger thresholds (+ means no bias detected).

	Trigger Threshold					
Iterations	1	2	3	4	5	6
1	+	+	+	26	24	22
4	+	+	+	+	32	26
8	+	+	+	+	+	+

Table VII. Generation rate and uniform samples per output tail sample.

	Algorithm Generation Rate					Uniform Samples per Output Sample				
r	0	1	2	3	4	0	1	2	3	4
GRAND [1974]	8.80	8.09	7.85	7.67	7.62	2.47	2.76	2.91	2.98	3.02
Box-Muller [1958b]	8.69	4.50	2.97	2.15	1.69	1.00	1.91	2.97	4.11	5.30
New-Marsaglia [1984b]		3.65	4.69	5.08	5.26		3.05	2.37	2.19	2.11
Old-Marsaglia [1964]	10.71	5.48	3.60	2.64	2.08	2.55	2.89	3.88	5.25	6.74
Polar [1969]	10.43	4.96	3.17	2.34	1.84	2.55	2.89	3.88	5.25	6.74
PPND7 [1988]	15.41	15.41	15.40	15.40	15.3	1	1	1	1	1
Ahrens-Dieter [1988]	4.62	3.91	2.92	2.18	1.70	1.34	2.57	4.00	5.53	7.11

The low-quality Wallace fails with a trigger threshold of 4, while the higher quality version fails with a trigger threshold of 5. Table VI shows the results of the tests on the Wallace generators, including the results for a trigger threshold of 6 (calculated using parallel generators). Increasing the number of iterations from 1 to 4, reduces the effect of correlations, and the table shows that if the number of iterations is increased to 8 then the correlations are no longer detectable using this test.

The results described convey several messages. First, some of the algorithms are more resilient than others when used with single-precision uniform random numbers derived directly from 32-bit integers. For example, the Box-Muller method fails beyond 5.6σ while the Ziggurat method does not fail until 8.1σ. That said, since values with magnitude exceeding 8σ occur fewer than one time out of 10^{14}, using a uniform random number generator with period $\approx 10^9$ would be a very poor idea if random numbers in the 8σ region are desired. Provided that input single-precision uniform random numbers with fully-random fractions are used, all of the tested algorithms deliver good performance to at least 11σ, and in many cases well beyond. Since fewer than one in 10^{27} Gaussian numbers can be expected to have magnitude exceeding 11σ, it could be argued that differences among GRNGs in terms of where failure occurs in the region above 11σ are less important. That said, it is nonetheless noteworthy that the Ziggurat and Marsaglia-Bray methods extended significantly further than the others, failing at 17.4σ and 15.8σ respectively.

In terms of speed, Table II shows that the Ziggurat and Wallace methods are the fastest. The Wallace method, however, is recursive, utilizing transformations of previous outputs to generate new ones. The resulting inevitable correlations are seen in Table VI. This can be mitigated by increasing the pool size and mixing in Wallace, but care must be taken so that the pool size doesn't become so large that speed, the Wallace method's most significant advantage, is sacrificed.

4.3. Results: Gaussian Tail Methods

The Gaussian tail methods all use a parameter r, which controls the minimum absolute value that will be produced. As r is varied, the behavior of the methods varies, altering the acceptance ratios and computation per output random number, as well as magnitude and accuracy of the numbers used in calculations. For this reason, the tests are performed with different values of r.

Table VIII. Number of sample ($\log_2$) before failure of χ^2 test for gaussian tail sampling methods.

r	1	2	3	4
Box-Muller [1958b]	33	32	32	30
Polar [1969]	36	+	+	+
Old-Marsaglia [1964]	36	36	+	+
New-Marsaglia [1984b]	33	34	34	34
GRAND [1974]	36	36	36	36
PPND7(Std) [1988]	30	29	29	28
Ahrens-Dieter [1988]	+	+	36	35
PPND7(Full) [1988]	+	36	34	33

Table IX. Results of high sigma-multiple tests for tail generation methods using single precision floating-point.

	Standard				Full-Fraction			
	1	2	3	4	1	2	3	4
GRAND [1974]	10.2	11.1	11.7	11.9	17+	17+	17+	17+
Box-Muller [1958b]	5.9	6.6	7.2	7.5	14.2	14.9	15.4	15.6
New-Marsaglia [1964]	6.6	7.7	8	7.9	14.8	15.9	17.1	17.4
Old-Marsaglia [1984b]	7.9	8	8.3	8.6	14.8	15.9	17.1	17.4
Polar [1969]	8.2	8.5	8.9	9.1	12.8	13.2	13.5	13.7
PPND7 [1988]	4.8	5.7	6.5	7.1	12.1	12.3	12.3	12.2

4.3.1. χ^2 Tests. Although the tests are performed on standard and fully-random fraction floating-point numbers (using single-precision), only the standard results are shown for the majority of the tests, since there is almost no difference observed between the two. The only cases where a difference is seen are for the Box-Muller method with $r = 1$, where the fully-random fraction version fails at 2^{34}. Interestingly, the Box-Muller method is also the only generator that degrades when higher thresholds are used: all the other generators either maintain quality or improve.

4.3.2. High Sigma-Multiple Tests. Table IX shows the results of the high sigma-multiple test when applied to the methods for tail sample generation. In all cases the single-precision full-fraction versions perform significantly better than the standard single-precision cases. The Box-Muller method performs particularly badly when the standard uniform generation method is used, and when generating samples above a threshold of one, it is only accurate out to 5.9σ. These results suggest that this generator can only be used to generate $\Phi(1)/\Phi(5.9) \approx 7.7 \times 10^7$ samples.

5. CONCLUSION

This article presents a survey and a classification of Gaussian random number generators. We describe a comprehensive test methodology to determine the statistical quality of the different methods, particularly with regards to the distribution in the tails. This testing has demonstrated that single-precision calculations are usually sufficient, even for applications requiring good coverage in the tails of the Gaussian distribution, as long as care is taken when converting uniform random integers to floating-point random numbers. It is shown that the Wallace method is the fastest, but can suffer from correlation problems; the Ziggurat method, the second in speed, is about 33% slower than the Wallace method but does not suffer from correlation problems. Thus, when maintaining extremely high statistical quality is the first priority, and subject to that constraint, speed is also desired, the Ziggurat method will often be the most appropriate choice. If the quality requirements are not so stringent but speed is essential then Wallace may be appropriate. One disadvantage of Ziggurat lies in the large number

of constants (388 for single-precision), so in environments where that is problematic simpler methods such as polar or GRAND may also be appropriate.

REFERENCES

AHRENS, J. H. AND DIETER, U. 1972. Computer methods for sampling from the exponential and normal distributions. *Comm. ACM 15*, 10, 873–882.

AHRENS, J. H. AND DIETER, U. 1988. Efficient table-free sampling methods for the exponential, Cauchy, and normal distributions. *Comm. ACM 31*, 11, 1330–1337.

ANDRAKA, R. AND PHELPS, R. 1998. An FPGA based processor yields a real time high fidelity radar environment simulator. In *Military and Aerospace Applications of Programmable Devices and Technologies Conference*.

BELL, J. R. 1968. Algorithm 334: Normal random deviates. *Comm. ACM 11*, 7, 498.

BOUTILLON, E., DANGER, J.-L., AND GHAZEL, A. 2003. Design of high speed AWGN communication channel emulator. *Analog Integr. Circ. Sig. Proc. 34*, 2, 133–142.

BOX, G. E. P. AND MULLER, M. E. 1958a. A note on the generation of random normal deviates. *Annals Math. Stat. 29*, 610–611.

BOX, G. E. P. AND MULLER, M. E. 1958b. A note on the generation of random normal deviates. *Annals Math. Stat. 29*, 2, 610–611.

BRENT, R. P. 1974. Algorithm 488: A Gaussian pseudo-random number generator. *Comm. ACM 17*, 12, 704–706.

BRENT, R. P. 1993. Fast normal random number generators on vector processors. Tech. Rep. TR-CS-93-04, Department of Computer Science, The Australian National University, Canberra 0200 ACT, Australia.

BRENT, R. P. 1997. A fast vectorised implementation of Wallace's normal random number generator. Tech. Rep. TR-CS-97-07, Department of Computer Science, The Australian National University, Canberra 0200 ACT, Australia.

BRENT, R. P. 2003. Some comments on C. S. Wallace's random number generators. *Comput. J., to appear*.

CHEN, J., MOON, J., AND BAZARGAN, K. 2004. Reconfigurable readback-signal generator based on a field-programmable gate array. *IEEE Trans. Magn. 40*, 3, 1744–1750.

CHEN, W. AND BURFORD, R. L. 1981. Quality evaluation of some combinations of unit uniform random number generators and unit normal transformation algorithms. In *ANSS '81: Proceedings of the 14th Annual Symposium on Simulation*. IEEE Press, Piscataway, NJ. 129–149.

DANGER, J.-L., GHAZEL, A., BOUTILLON, E., AND LAAMARI, H. 2000. Efficient FPGA implementation of Gaussian noise generator for communication channel emulation. In *ICECS'2000*. IEEE, Jounieh, Lebanon.

DEVROYE, L. 1986. *Non-Uniform Random Variate Generation*. Springer-Verlag, `http://cg.scs.carleton.ca/~luc/rnbookindex.html`, New York.

FORSYTHE, G. E. 1972. Von Neumann's comparison method for random sampling from the normal and other distributions. *Math. Computation 26*, 120, 817–826.

GEBHARDT, F. 1964. Generating normally distributed random numbers by inverting the normal distribution function. *Math. Computation 18*, 86, 302–306.

HÖRMANN, W. 1994. A note on the quality of random variates generated by the ratio of uniforms method. *ACM Trans. Model. Comput. Simul. 4*, 1, 96–106.

KABAL, P. 2000. Generating Gaussian pseudo-random deviates. Tech. Rep., Department of Electrical and Computer Engineering, McGill University.

KINDERMAN, A. J. AND MONAHAN, J. F. 1977. Computer generation of random variables using the ratio of uniform deviates. *ACM Trans. Math. Softw. 3*, 3, 257–260.

KNOP, R. 1969. Remark on Algorithm 334 [g5]: normal random deviates. *Comm. ACM 12*, 5, 281.

KNUTH, D. E. 1981. *Seminumerical Algorithms*, Second ed. The Art of Computer Programming, vol. 2. Addison-Wesley, Reading, Massachusetts.

KRONMAL, R. 1964. Evaluation of a pseudorandom normal number generator. *J. ACM 11*, 3, 357–363.

L'ECUYER, P. 1992. Testing random number generators. In *WSC '92: Proceedings of the 24th Conference on Winter Simulation*. ACM Press, New York, NY. 305–313.

L'ECUYER, P. 1996. Maximally equidistributed combined Tausworthe generators. *Math. Computation 65*, 213, 203–213.

L'ECUYER, P. 2001. Software for uniform random number generation: distinguishing the good and the bad. In *WSC '01: Proceedings of the 33nd Conference on Winter Simulation*. IEEE Computer Society, Washington, DC. 95–105.

L'ECUYER, P. AND SIMARD, R. 2005. TestU01. http://www.iro.umontreal.ca/~simardr/indexe.html.

LEE, D.-U., LUK, W., VILLASENOR, J. D., AND CHEUNG, P. Y. 2004. A Gaussian noise generator for hardware-based simulations. *IEEE Trans. Comput. 53*, 12 (Dec.), 1523–1534.

LEHMER, D. H. 1949. Mathematical methods in large-scale computing units. In *Proceedings of the 2nd Symposium on Large-Scale Digital Calculating Machinery*. Harvard University Press, Cambridge, Massachusetts, 141–146.

LEVA, J. L. 1992a. Algorithm 712; a normal random number generator. *ACM Trans. Math. Softw. 18*, 4, 454–455.

LEVA, J. L. 1992b. A fast normal random number generator. *ACM Trans. Math. Softw. 18*, 4, 449–453.

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

MARSAGLIA, G. 1997. The Diehard random number test suite. http://stat.fsu.edu/pub/diehard/.

MARSAGLIA, G. 2004. Evaluating the normal distribution. *J. Statis. Softw. 11*, 4, 1–11.

MARSAGLIA, G. AND BRAY, T. A. 1964. A convenient method for generating normal variables. *SIAM Rev. 6*, 3, 260–264.

MARSAGLIA, G., MACLAREN, M. D., AND BRAY, T. A. 1964. A fast procedure for generating normal random variables. *Comm. ACM 7*, 1, 4–10.

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

MARSAGLIA, G. AND TSANG, W. W. 1984b. A fast, easily implemented method for sampling from decreasing or symmetric unimodal density functions. *SIAM J. Sci. Stat. Comput. 5*, 2 (June), 349–359.

MARSAGLIA, G. AND TSANG, W. W. 1998. The Monty Python method for generating random variables. *ACM Trans. Math. Softw. 24*, 3, 341–350.

MARSAGLIA, G. AND TSANG, W. W. 2000. The ziggurat method for generating random variables. *J. Statis. Softw. 5*, 8, 1–7.

MATSUMOTO, M. AND NISHIMURA, T. 1998. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul. 8*, 1 (Jan.), 3–30.

MCCOLLUM, J. M., LANCASTER, J. M., BOULDIN, D. W., AND PETERSON, G. D. 2003. Hardware acceleration of pseudo-random number generation for simulation applications. In *IEEE Southeastern Symposium on System Theory*. IEEE, Morgantown, WV.

MOLER, C. 1995. Random thoughts : 10^{435} years is a very long time. http://www.mathworks.com/company/newsletters/news_notes/pdf/Cleve.pdf.

MOLER, C. B. 2004. *Numerical Computing in Matlab*. Society for Industrial and Applied Mathematics, USA.

MOLLE, J. W. D., HINICH, M. J., AND MORRICE, D. J. 1992. Higher-order cumulant spectral-based statistical tests of pseudo-random variate generators. In *WSC '92: Proceedings of the 24th conference on Winter simulation*. ACM Press, New York, NY. 618–625.

MULLER, M. E. 1958. An inverse method for the generation of random normal deviates on large-scale computers. *Mathematical Tables and Other Aids to Computation 12*, 63, 167–174.

MULLER, M. E. 1959. A comparison of methods for generating normal deviates on digital computers. *J. ACM 6*, 3, 376–383.

PIKE, M. C. 1965. Algorithm 267: random normal deviate [g5]. *Comm. ACM 8*, 10, 606.

PRESS, W. H., TEUKOLSKY, S. A., VETTERLING, W. T., AND FLANNERY, B. P. 1997. *Numerical Recipes in C*, Second ed. Cambridge University Press, Cambridge.

RUKHIN, A., SOTO, J., NECHVATAL, J., SMID, M., BARKER, E., LEIGH, S., LEVENSON, M., VANGEL, M., BANKS, D., HECKERT, A., DRAY, J., AND VO, S. 2001. A statistical test suite for random and pseudorandom number generators for cryptographic applications. NIST special publication 800-22, National Institute of Standards and Technology (NIST), Gaithersburg, Maryland See http://csrc.nist.gov/rng/.

SCHOLLMEYER, M. F. AND TRANTER, W. H. 1991. Noise generators for the simulation of digital communication systems. In *ANSS '91: Proceedings of the 24th Annual Symposium on Simulation*. IEEE Computer Society Press, Los Alamitos, CA, 264–275.

TEICHROEW, D. 1953. Distribution sampling with high speed computers. (PhD thesis).

THOMAS, D. B. AND LUK, W. 2006. Non-uniform random number generation through piecewise linear approximations. In *International Conference on Field Programmable Logic and Applications*.

WALKER, A. J. 1977. An efficient method for generation discrete random variables with general distribution. *ACM Trans. Math. Softw. 3*, 253–256.

WALLACE, C. 2005. Random number generators. Tech. rep., Monash University. See `http://www.datamining.monash.edu.au/software/random/`.

WALLACE, C. S. 1996. Fast pseudorandom generators for normal and exponential variates. *ACM Trans. Math. Softw. 22*, 1, 119–127.

WETHERILL, G. B. 1965. An approximation to the inverse normal function suitable for the generation of random normal deviates on electronic computers. *Appl. Statis. 14*, 2, 201–205.

WICHURA, M. J. 1988. Algorithm AS 241: The percentage points of the normal distribution. *Appl. Statis. 37*, 3, 477–484.

XILINX. 2002. Additive white Gaussian noise (AWGN) core. CoreGen documentation file.

Received March 2006; revised November 2006; accepted March 2007