Sampling exactly from the normal distribution

Charles F. F. Karney*

SRI International, 201 Washington Rd, Princeton, NJ 08543-5300, USA

(Dated: March 25, 2013)

An algorithm for sampling exactly from the normal distribution is given. The algorithm reads some number of uniformly distributed random digits in a given base and generates an initial portion of the representation of a normal deviate in the same base. Thereafter, uniform random digits are copied directly into the representation of the normal deviate. Thus, in constrast to existing methods, it is possible to generate normal deviates exactly rounded to any precision with a mean cost that scales linearly in the precision. The method performs no arbitrary precision arithmetic, calls no transcendental functions, and, indeed, uses no floating point arithmetic whatsoever; it uses only simple integer operations. The algorithm is inspired by von Neumann's algorithm for sampling from the exponential distribution; an improvement to von Neumann's algorithm is also given.

Keywords: Random deviates, normal distribution, exact sampling

1. INTRODUCTION

Random variables with a normal distribution,

$$\phi(x) = \frac{\exp(-x^2/2)}{\sqrt{2\pi}},$$

are widely used in Monte Carlo simulations. Over the past sixty years, scores of algorithms for generating such normal deviates have been published (Thomas *et al.*, 2007). In this paper, I give another algorithm with the distinguishing feature that, given a source of uniformly distributed random digits in some base b, it generates *exact* normal deviates. In order to make the meaning of "exact" precise, consider Table 1 which illustrates the operation of the algorithm using $b = 10$ (with the implementation given in the appendix). The first column shows the input to the algorithm, which, in this example, is taken from successive lines of the table of random digits produced by the RAND Corporation (1955), beginning at line 13677 (on p. 274); the resulting normal deviates are given in the second column. Referring to the first line of this table, the algorithm reads random decimal digits, $47152\,7$, from the source and produces $+0.2$ as the initial portion of the decimal representation of a normal deviate. Thereafter, random digits can be copied directly from the input to the output, indicated by the ellipses (...) in the table; thus $+0.2\ldots$ represents a uniform random sample in the range $(0.2, 0.3)$. The next digits of the random sequence are $7172\,6817\ldots$; thus the normal deviate can be exactly rounded to 8 decimal digits as $0.27172\,682$. Alternatively, a normal deviate can be exactly compared against a constant 0.5, for example, by adding at most one additional digit to the results in the table. (The results in Table 1 are not "typical," because the starting line in the table of random digits was specifically chosen to limit the number of random digits used.)

It's clear from this example that rounding exactly to an IEEE double precision number is straightforward (particularly if b is a power of two); i.e., the method can be used to generate deviates that satisfy the conditions of "ideal approximation" (Monahan, 1985), namely that the algorithm is equivalent to sampling a real number from the normal distribution and rounding it to the nearest representable floating point number. Other sampling methods are frequently referred to as "exact," for example the polar method (Box and Muller, 1958) and the ratio method (Kinderman and Monahan, 1977); but these are merely "accurate to round off" which, in practice, means only that the accuracy is commensurate with the precision of the floating point number system. Furthermore, for applications requiring high precision normal deviates, the new algorithm offers ideal scaling. There's an amortized constant cost to producing the initial portion of the normal deviate; but, thereafter, the digits can be added to the result at a rate limited only by the cost of producing and copying the random digits.

TABLE 1 Sample input and output for Algorithm N with $b = 10$. The input consists of uniformly distributed random decimal digits and the output gives the resulting normal deviates.

input	output
$47152\,7\ldots$	$+0.2\ldots$
$70314\,19047\,5\ldots$	$+0.\ldots$
$74131\,63178\,62857\ldots$	$+2.2\ldots$
$20147\,9673\ldots$	$-1.6\ldots$
$61921\,13332\,0\ldots$	$-1.132\ldots$

It's not immediately obvious that such an algorithm for exact sampling is possible. However, in the early years of the era of modern computing, von Neumann (1951) presented a remarkably simple algorithm for sampling from the exponential distribution.

Algorithm V (*von Neumann*). Samples E from the exponential distribution e^{-x} for $x > 0$.

V1. [Initialize rejection count.] Set $l \leftarrow 0$.

V2. [Sample fraction.] Set $x \leftarrow U$, where U is a uniform

*Electronic address: charles.karney@sri.com

deviate $U \in (0,1)$.

V3. [Generate a run.] Sample uniform deviates $U_1, U_2, \ldots$ and determine the maximum value $n \geq 0$ such that $x > U_1 > U_2 > \ldots > U_n$.

V4. [Test length of run.] If n is even, set $E \leftarrow l + x$ and return; otherwise set $l \leftarrow l + 1$ and go to step V2. ∎

(Because the algorithm generates continuous random deviates, there's no distinction between the inequalities $x \geq 0$ and $x > 0$ or the intervals $(0,1)$ and $[0,1]$.) According to von Neumann, this algorithm was suggested by the game of Black Jack and this connection is made plain in the slightly different formulation given in Abramowitz and Stegun (1964, §26.8.6.c(2)). We shall see in Sec. 2, that by treating x as a sequence of random bits, Algorithm V can be easily adapted to sample exponential deviates exactly.

Several authors have generalized von Neumann's algorithm (Ahrens and Dieter, 1973; Brent, 1974; Forsythe, 1972; Monahan, 1979). However, these efforts entail using ordinary floating point arithmetic and thus the methods do not generate exact deviates. In this paper, I show that the algorithm can be extended to sample exactly from the unit normal distribution. Although the resulting algorithm is unlikely to displace existing methods for most applications, the algorithm will be useful wherever exactness is required and when using arbitrary precision arithmetic. It is also of theoretical interest as an example of an algorithm where exact transcendental results can be achieved with simple integer arithmetic.

2. VON NEUMANN'S ALGORITHM

I start with a brief proof of von Neumann's algorithm and an improvement to it. The crucial step is step V3. The probability that $U_1, \ldots, U_n$ are all less than x is x^n (provided that $x \in [0,1]$). The probability that, in addition, they are in descending order (one of the possible $n!$ permutations) is $x^n/n!$. For the condition to hold for a sequence of $n+1$ numbers, it must hold for the first n of them; therefore the probability that the length of the longest decreasing sequence is n is $x^n/n! - x^{n+1}/(n+1)!$. For a given x, the probability that n is even is

$$1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \ldots = e^{-x},$$

while the probability that n is odd (averaged over x) is $1 - \int_0^1 e^{-x}\,\mathrm{d}x = e^{-1}$. Thus the probability that the algorithm terminates with a particular value of l and x is $\exp\bigl(-(l+x)\bigr)$ as required.

On average, this algorithm requires $e^2/(e-1) \approx 4.30$ uniform deviates; in effect, the algorithm sums all the terms in the Taylor series for e^{-x} in a finite mean time. Conventionally, U would be sampled from the subset of reals which are representable as double precision floating point numbers; in this case, the results would be only approximately equivalent to sampling exactly from the exponential distribution and rounding the results to the closest floating point number. However, because the only operation performed on the uniform deviates is a comparison, the uniform deviate can be generated one bit at a time. Typically, the comparisons can be decided by generating only a few bits of each operand. At the completion of the algorithm, l and some initial set of the bits in the fraction of x are known. The rest of x can be filled in with uniform random bits. In this way, the algorithm generates exponential deviates which are exact in the sense discussed in connection with Table 1 in Sec. 1. This was the insight provided by Knuth and Yao (1976) and the method was extensively analyzed by Flajolet and Saheb (1986). They showed that the mean number of bits needed to generate m bits of the result (coded in Knuth and Yao's unary-binary notation) is $m + \gamma$ where $\gamma \approx 5.680$ is the "balance" of the algorithm. Here I generalize this methodology by representing the uniform deviates in some base b with digits uniformly distributed in $[0,b)$ being added as needed. Interesting values of b are typically 2 (adding a bit at a time) and 2^{32} (adding a random "word" at a time).

It's possible to make von Neumann's algorithm slightly more efficient by using early rejection.

Algorithm E (*improved von Neumann*). Improved algorithm for sampling E from e^{-x} for $x > 0$.

E1. [Initialize rejection count.] Set $l \leftarrow 0$.

E2. [Sample fraction.] Set $x \leftarrow U$, where U is a uniform deviate $U \in (0,1)$.

E3. [Early rejection.] If $x > \frac{1}{2}$, set $l \leftarrow l + 1$ and go to step E2.

E4. von Neumann's step V3.

E5. [Test length of run.] If n is even, set $E \leftarrow \frac{1}{2}l + x$ and return; otherwise set $l \leftarrow l + 1$ and go to step E2. ∎

The early rejection step results in lowering the mean number of uniform deviates required to $e/(\sqrt{e}-1) \approx 4.19$. The balance is reduced to about 3.906.

Von Neumann's algorithm can be adapted to generate a Bernoulli trial with probability $1/\sqrt{e}$, as follows.

Algorithm H (*a half-exponential Bernoulli trial*). A Bernoulli trial H which is true with probability $1/\sqrt{e}$.

H1. [Generate a run.] Sample uniform deviates $U_1, U_2, \ldots$ and determine the maximum value $n \geq 0$ such that $\frac{1}{2} > U_1 > U_2 > \ldots > U_n$.

H2. [Test length of run.] Set $H \leftarrow (n \text{ is even})$. ∎

On average, the algorithm uses $\frac{1}{2}e/(\sqrt{e}-1)$ (resp. $\frac{1}{2}e$) uniform deviates if the result is false (resp. true); the overall weighted average is $\sqrt{e}$. If $b = 2$, the test uses about 3.786 (resp. 2.236) bits on average if the result is false (resp. true), i.e., 2.846 bits overall. With probability $\frac{1}{2}$, this algorithm exits after sampling just one bit (i.e., $U_1 > \frac{1}{2}$ and $n = 0$); this accounts for the relative efficiency of Algorithm E compared to Algorithm V. Algorithm H will be used for sampling from the normal distribution.

3. SAMPLING FROM THE NORMAL DISTRIBUTION

Here I tackle the problem of sampling normal deviates using only comparisons of uniform deviates and simple integer

arithmetic, so that exact samples can be obtained. First of all, it is clear that determining the integer part of the exponential deviate in von Neumann's algorithm from the number of rejections depends on the property of exponentials, $\exp(-(k+x)) = \exp(-k)\exp(-x)$, which does not hold for other distributions. This suggests that the sign and integer part of the normal deviate be separately sampled which leads to the following skeleton of an algorithm.

Algorithm N (*normal sampling*). Sample N from a unit normal distribution $\phi(x)$ using a rejection method.

N1. [Sample integer part of deviate k.] Select integer $k \geq 0$ with probability $\exp(-\frac{1}{2}k)(1-1/\sqrt{e})$.

N2. [Adjust relative probability of k by rejection.] Accept k with probability $\exp\bigl(-\frac{1}{2}k(k-1)\bigr)$; otherwise go to step N1.

N3. [Sample fractional part of deviate x.] Set $x \leftarrow U$, where U is a uniform deviate $U \in (0,1)$.

N4. [Adjust relative probability of x by rejection.] Accept x with probability $\exp\bigl(-\frac{1}{2}x(2k+x)\bigr)$; otherwise go to step N1.

N5. [Combine integer and fraction.] Set $x \leftarrow k + x$.

N6. [Assign a sign.] With probability $\frac{1}{2}$, set $x \leftarrow -x$.

N7. [Return result.] Set $N \leftarrow x$. ∎

The analysis of this algorithm is simple. After step N2, the relative probability of k is $\exp(-\frac{1}{2}k) \times \exp\bigl(-\frac{1}{2}k(k-1)\bigr) = \exp(-\frac{1}{2}k^2)$ for $k \geq 0$; after step N4, the relative probability of $[k,x]$ is $\exp(-\frac{1}{2}k^2) \times \exp\bigl(-\frac{1}{2}x(2k+x)\bigr) = \exp\bigl(-\frac{1}{2}(k+x)^2\bigr)$ for $k \geq 0$ and $x \in (0,1)$. From this, it follows that the returned value of x has a Gaussian distribution, $\phi(x)$. Step N2 always succeeds for $k = 0$ and 1, the two most common cases. Overall, the probability that step N2 succeeds is $(1-1/\sqrt{e})G \approx 0.690$ where $G = \sum_{k=0}^{\infty}\exp(-\frac{1}{2}k^2) \approx 1.75$. Similarly, step N4 succeeds with probability $\sqrt{\pi/2}/G \approx 0.715$. Thus, step N1 is executed $\sqrt{2/\pi}/(1-1/\sqrt{e}) \approx 2.03$ times on average.

Steps N1 and N2 can be expressed in terms of half-exponential Bernoulli trials H with

Steps N1 and N2 in terms of H.

N1. [Test H until failure.] Generate a sequence of Bernoulli deviates $H_1, H_2, \ldots$ and determine the largest $k \geq 0$ such that $H_1, H_2, \ldots, H_k$ are all true.

N2. [Make $k(k-1)$ tests of H.] Set $k' \leftarrow k(k-1)$ and generate up to k' Bernoulli deviates $H_1, H_2, \ldots, H_{k'}$. Accept k if H_i is true for all $i \in [1,k']$; otherwise go to step N1.

Steps N3, N5, and N6 are simple to implement (for details, see Sec. 4). This just leaves step N4 which is changed to

Rewriting step N4.

N4. [Break N4 into $k+1$ steps.] Perform up to $k+1$ Bernoulli trials, $B_1, B_2, \ldots, B_{k+1}$, each with probability $\exp\bigl(-x(2k+x)/(2k+2)\bigr)$. Accept x if B_i is true for all $i \in [1,k+1]$; otherwise go to step N1.

This transformation of N4 is motivated by the requirement in the proof of von Neumann's method that $x \in [0,1]$. Repeating the trial $k+1$ times means that the argument to the exponential in the original step N4 is divided by $k+1$; note that the maximum value of $x(2k+x)/(2k+2)$ (as x is varied) is $(2k+1)/(2k+2) < 1$.

In order to carry out a Bernoulli trial B, I generalize von Neumann's procedure.

Algorithm B (*generalizing von Neumann's step V3*). A Bernoulli trial with probability $\exp\bigl(-x(2k+x)/(2k+2)\bigr)$. Sample two sets of uniform deviates $U_1, U_2, \ldots$ and $V_1, V_2, \ldots$ and determine the maximum value $n \geq 0$ such that $x > U_1 > U_2 > \ldots > U_n$ and $V_i < (2k+x)/(2k+2)$ for all $i \in [1,n]$.

B1. [Initialize loop.] Set $p \leftarrow x$, $n \leftarrow 0$.

B2. [Generate and test next samples.]
(i) Sample $q \leftarrow U$; go to step B4, unless $q < p$.
(ii) Sample $r \leftarrow U$; go to step B4, unless $r < (2k+x)/(2k+2)$.

B3. [Increment loop counter and repeat.] Set $p \leftarrow q$, $n \leftarrow n+1$; go to step B2.

B4. [Test length of runs.] Set $B \leftarrow (n \text{ is even})$. ∎

Without step B2(ii), steps B1 to B3 are just step V3 of von Neumann's algorithm. Because of the additional test B2(ii), the probability that the nth trip through the loop succeeds is $x^n/n! \times \bigl((2k+x)/(2k+2)\bigr)^n$. The requirement that n be even means that B succeeds with probability

$$1 - x\frac{2k+x}{2k+2} + \frac{x^2}{2!}\left(\frac{2k+x}{2k+2}\right)^2 - \frac{x^3}{3!}\left(\frac{2k+x}{2k+2}\right)^3 + \ldots = \exp\left(-x\frac{2k+x}{2k+2}\right).$$

In order to avoid performing arithmetic on uniform deviates in step B2(ii), we note that as x varies in $(0,1)$ the right side of the inequality varies from $2k/(2k+2)$ to $(2k+1)/(2k+2)$. Thus, regardless of the values of x and r, the test will succeed with probability $2k/(2k+2)$ and fail with probability $1/(2k+2)$. The remaining probability, $1/(2k+2)$, is divided between success and failure according to $r < x$. Thus the test $r < (2k+x)/(2k+2)$ can be replaced with

Algorithm T (*the test in B2(ii)*). Perform test $T = \bigl(r < (2k+x)/(2k+2)\bigr)$ without doing arithmetic on real numbers.

T1. [Sample a selector f.] Set $f \leftarrow C(2k+2)$ where $C(m)$ is -1 with probability $1-2/m$, 0 with probability $1/m$, and 1 with probability $1/m$.

T2. [Act on the value of f.] If $f < 0$, set $T \leftarrow$ false; else if $f > 0$, set $T \leftarrow$ true; otherwise ($f = 0$), set $T \leftarrow (r < x)$. ∎

Finally, $C(m)$ can be computed with

Algorithm C (*the 3-way selector*). The choice $C(m)$, $(-1,0,1)$ with probabilities $(1/m, 1/m, 1-2/m)$, implemented as the test $w < n/m$ where w is a uniform deviate in $(0,1)$ and $n = m-2$ or $n = m-1$. For each successive

digit d of w, substitute $w = (d + w')/b$ so that the test becomes $w' < n'/m$, where $n' = bn - dm$, and exit as soon as the n' is outside the range $(0, m)$.

C1. [Set the numerators of the fractions.] Set $n_1 \leftarrow m - 2$ and $n_2 \leftarrow m - 1$.

C2. [Sample the next digit of w, d.] Sample $d \leftarrow D$ where D is a uniformly distributed integer in $[0, b)$.

C3. [Multiply inequalities by bm.] Set $n_1 \leftarrow bn_1 - dm$ and $n_2 \leftarrow bn_2 - dm$.

C4. [Test the new numerators.] If $n_1 \geq m$, set $C(m) \leftarrow 1$ and return; else if $n_2 \leq 0$, set $C(m) \leftarrow -1$ and return; else if $n_1 \leq 0$ and $n_2 \geq m$, set $C(m) \leftarrow 0$ and return; otherwise, go to step C2. ∎

Step B2 can now be written as

Step B2 incorporating algorithm T.

B2. [Generate and test next samples.]
(a) Sample $q \leftarrow U$; go to step B4, unless $q < p$.
(b) Set $f \leftarrow C(2k + 2)$; if $f < 0$, go to step B4.
(c) If $f = 0$, sample $r \leftarrow U$ and go to step B4, unless $r < x$.

4. IMPLEMENTATION

The appendix includes an implementation of Algorithm N in C++ together with a simple test program and its output. The class `unit_normal_dist` generates the normal deviates closely following the description in Sec. 3; it uses a class `rand_digit` which produces pseudo random digits in a base b and a class `rand_number` which holds a random deviate. The latter class is key to generating exact deviates; a random deviate is represented as $[s, n, L, d_0, d_1, \ldots, d_{L-1}]$ which corresponds to the uniform deviate

$$x = s\Big(n + \sum_{i=0}^{L-1} d_i b^{-i-1} + b^{-L}U\Big),$$

where $s = \pm 1$, n and L are non-negative integers, and $d_i \in [0, b)$. Here, as above, U represents a uniform deviate $U \in (0, 1)$. Implicit in this representation is a mechanism by which L may be increased and additional digits d_i can be generated. In this implementation, operators which manipulate a `rand_number` and may need to generate additional bits are passed a `rand_digit` for this purpose. The operation $x \leftarrow U$ corresponds to $s \leftarrow 1$, $n \leftarrow 0$, and $L \leftarrow 0$ and is performed by `init`. Setting the integer part of a deviate (step N5) and switching its sign (step N6) are performed by `set_integer` and `negate`. Comparing two random deviates $x < t$ (needed in Algorithms H and B) is performed by `less_than`. If the sign and integer parts of x and t match, then successive digits of the fractions are compared until the digits are unequal, increasing L for x or t as necessary. Finally `less_than_half` tests $x < \frac{1}{2}$ (in Algorithm H and in determining the sign in step N6); if b is even this reduces to $d_0 < b/2$ and the generalization to all b is straightforward.

Because comparisons may cause digits to be generated, care must be taken to avoid copying random deviates. For this reason, the assignment $p \leftarrow x$ in step B1 is incorporated into step B2. The order of the comparisons in step B2, at the end of Sec. 3, is somewhat arbitrary; in this implementation, the number of random digits needed is reduced by reversing the order of B2(a) and B2(b) whenever $k = 0$. The normal deviates produced by `unit_normal_dist` are instances of `rand_number` with enough digits added to the fraction to allow Algorithm N to complete. Digits can be added to the normal deviate as illustrated in the test program. Table 1 shows sample normal deviates produced by this implementation with `rand_digit` modified to use the tabulated random digits.

In examining the properties of Algorithm N, let us start with the case $b = 2$. Measuring the number of bits used on each invocation of the algorithm, I find that the frequency decays exponentially with a e-folding constant of about 29.9 bits. (This exponential behavior is expected given the nature of the algorithm.) Thus the mean running time is finite—in fact, this implementation uses about 30.000 random digits on average and the average number of bits in the fraction is $\langle L\rangle = 1.556$. From these two quantities we can obtain the balance of the algorithm. For this purpose, the normal deviate is represented in a unary-binary format using $n + L + 2$ bits (this includes 1 bit for the sign and 1 bit to separate the integer and fraction). For Gaussians, the mean value of n is $P = \langle\lfloor|x|\rfloor\rangle \approx 0.365$, and the balance is $30.000 - (P + 1.556 + 2) \approx 28.079$. Another useful metric is the number of bits needed to generate a correctly rounded result in a floating point representation with p bits in the floating point fraction. The mean floating point exponent of a normal deviate is $Q = \langle\lfloor\log_2|x|\rfloor\rangle + 1 \approx -0.416$, and the average number of random bits needed to produce a rounded floating point result is then $30.000-(Q+1.556)+p+1 \approx 29.861+p$. (The addition of 1 on the left side of this expression accounts for the extra bit needed for rounding the result.) For IEEE double precision floating point numbers, we have $p = 53$, and, on average, 82.861 random bits are required per rounded normal deviate.

Algorithm N effectively decomposes $\phi(x)$ into a set of uniform distributions which are shown in Fig. 1 for the case $b = 2$. The right half of this figure, $x > 0$, shows the decomposition given by the implementation in the appendix. The left side, $x < 0$, shows the decomposition given by the slightly different implementation of step B2 provided by the `ExactNormal` class in RandomLib version 1.6 (Karney, 2012). In this version, the comparisons are reordered for $n = 0$ in order to minimize the number of bits added to x. This reduces $\langle L\rangle$ to 1.187; however, the average number of random bits used increases to 30.104 and thus, overall, this variant is more expensive by about 0.47 bits. Unless the specific goal is to minimize the number of bits in the result, the method given in the appendix is preferred.

Turning now to the opposite limit, consider the case $b = 2^{32}$. Table 2 shows some comparative timings for producing normal deviates with a precision of p bits in either the IEEE floating point format or the floating point representation of MPFR (2011), a library for arbitrary precision arith-

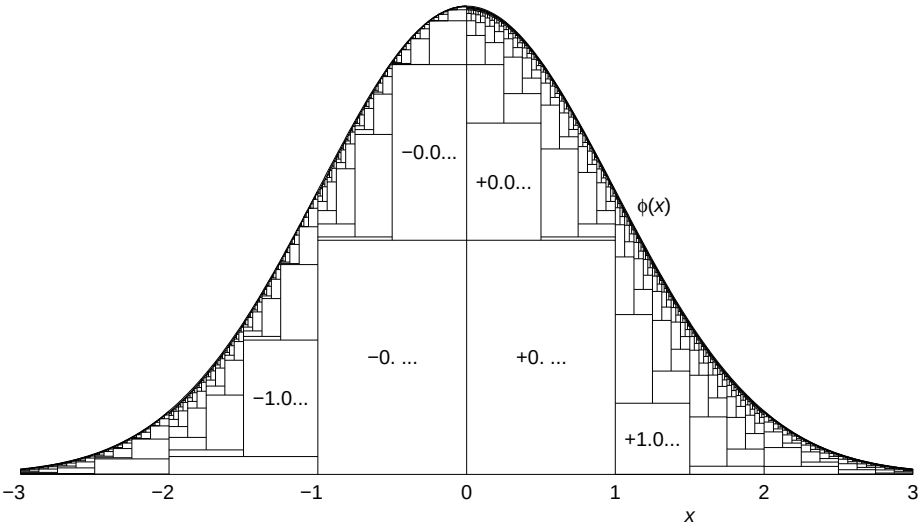

FIG. 1 Algorithm N's decomposition of the normal distribution into a set of uniform distributions with $b = 2$. For example, the frequency with which $+0.0\ldots$ is returned is equal to the relative area of the rectangle spanning $x \in (0, \frac{1}{2})$ which is $\frac{1}{16}\sqrt{2/\pi} \approx 5\%$. The left and right portions of this figure correspond to two different implementations of Algorithm N (see the text). On the left (resp. right) half of the figure, the minimum range of the uniform distributions shown is 2^{-7} (resp. 2^{-8}).

metic (Fousse *et al.*, 2007). Column A uses the ratio method, with the optimizations given by Leva (1992), and ordinary floating point arithmetic. Columns B and C use Algorithm N with the results rounded to IEEE floating point numbers (column B) or to MPFR's `mpfr_t` (column C). Column D uses the routine, `mpfr_grandom`, from MPFR version 3.1.0, for sampling normal deviates. The source of uniform random numbers for these methods is the SFMT19937 algorithm (Saito and Matsumoto, 2008) included in RandomLib for columns A and B, and the `mpz_urandomb` routine in Gnu MP library for columns C and D. The tests were carried out on a Fedora Linux system running on an Intel 2.66 GHz CPU. As expected, the scaling of the time for Algorithm N in column C is offset linear, approximately $(1 + 300\,p/2^{20})\,\mu$s; this makes Algorithm N among the least expensive operations for an arbitrary precision library.

TABLE 2 Times (in μs) for sampling from the normal distribution. p is the number of bits in the fraction of the rounded floating point samples. Columns A–D use different methods as explained in the text.

p	A	B	C	D
24	0.034	0.38	0.96	3.5
32			1.0	3.7
53	0.041	0.44	1.0	3.9
64	0.044	0.45	1.0	4.2
128			1.0	5.7
256			1.1	8.5
2^{10}			1.3	26
2^{12}			2.2	150
2^{14}			5.7	1400
2^{16}			20	15000
2^{18}			79	120000
2^{20}			300	820000

5. DISCUSSION

I consider three criteria by which Algorithm N might be judged, the complexity of the algorithm, its speed, and its accuracy. The core of Algorithm N, the class `unit_normal_dist`, is about a page of code. It is roughly 10 times longer than many classical methods of sampling normal deviates and about 2–3 times longer that some of the faster methods, e.g., Marsaglia and Tsang (2000). On the other hand, it is shorter than converting one of the standard methods to yield exactly rounded results, epitomized by `mpfr_grandom` which uses the polar method, because of the extra code needed to guarantee that an accurate result is returned. Of course, in some respects, Algorithm N can be regarded as the simplest of the available algorithms; because it uses only integer operations, you can compute accurate normal deviates by tossing a coin and using just pencil and paper!

Algorithm N is an order of magnitude slower than conventional methods for sampling normal deviates (compare columns A and B in Table 2), which is hardly surprising given its low level representation of random deviates. However it is considerably faster than `mpfr_grandom` at exact sampling, particularly at high precision (compare columns C and D in Table 2). If the source of randomness is a slow hardware generator, then the timing comparison reduces to measuring the use of random bits. By this metric, the best conventional methods (Box and Muller, 1958; Marsaglia and Tsang, 2000) consume one uniform deviate for each normal deviate, typically requiring 53 random bits for each result at double precision. In contrast, Algorithm N (with $b = 2$) uses 83 bits

which would be 60% slower; however, Algorithm N delivers accurate bounds on the the deviate using just 30 or so bits and thus it's possible that in some applications Algorithm N will be faster.

Algorithm N provides a novel way of exactly sampling normal deviates. Several classical methods can be modified to achieve the same results by using arbitrary precision arithmetic; however we saw from column D of Table 2 that this can be very slow. One method for sampling normal deviates which can be made exact with similar performance to Algorithm N is given by Kahn (1956, p. 41); see also Abramowitz and Stegun (1964, §26.8.6.a(4)). The algorithm is: sample x and x' from the exponential distribution and, if $(x-1)^2 < 2x'$, attach a random sign to x and return the result. Unfortunately, the comparison entails performing arithmetic on x which requires the use of arbitrary precision arithmetic. On the other hand, the exponential deviates may be sampled using Algorithm E and the comparison can be accomplished adding only a few digits (on average) to x and x'. This means that it shares with Algorithm N the ideal scaling of cost with precision. I have implemented this algorithm using MPFR and find that it is more bit efficient than Algorithm N; despite this, the constant term in the expression for the time per normal deviate is slightly larger than Algorithm N. (Incidentally, the first two steps of Algorithm N are essentially an integer version of Kahn's method.)

Many Monte Carlo simulations make heavy use of normal deviates. However such simulations would have to be extraordinarily long to distinguish between a conventional double precision version of the ratio method, for example, and Algorithm N with the results rounded to double precision. Given that it is about 10 times slower, there is little reason to prefer Algorithm N in most such simulations. However in some specialized applications, e.g., cryptography, exact sampling is required (Granboulan and Pornin, 2007; Micciancio and Regev, 2009), and then Algorithm N offers a good solution (often in conjunction with a library such as MPFR).

This paper presents an algorithm for exactly sampling normal deviates. This means that the three major maximum entropy distributions, uniform (trivially), exponential (via Algorithm E), and normal (via Algorithm N), can be sampled exactly with perfect asymptotic scaling. This makes them ideal for libraries for arbitrary precision arithmetic, such as MPFR, because the results can be exactly rounded to a precision of p bits with a cost which is $O(p)$. However, it might often be advantageous to delay the extension and rounding of the random deviates (with the associated costs and loss of accuracy). The output of these algorithms gives the first few digits of the random deviate and a rule for generating additional digits. As such, it occupies $O(1)$ storage and is still exact. Furthermore, certain operations can be performed on such random deviates at $O(1)$ cost, an example being the comparison between exponential deviates in Kahn's algorithm; it would be of interest to find more operations on random deviates that can be cheaply and exactly carried out. The resulting "lazy evaluation" framework would, in principal, require less storage, be faster, and be exact. (It's worth reiterating here that the exactness of these algorithms depend on the availability of an ideal source of random digits.)

Algorithm N relies on von Neumann's algorithm in the rejection methods used to select the integer and fractional parts of the normal deviate. The new techniques introduced are breaking step N4 into $k+1$ tests, to reduce of the argument of the exponential, and adding a second set of tests, in step B2(ii), to compute a more complex exponential probability. Presumably algorithms similar to Algorithm E and N can be found for other distributions although, as yet, there is no systematic approach to finding such algorithms. Related work by Flajolet *et al.* (2011) discusses several interesting methods for sampling discrete distributions and considers ways in which they can be combined. For example, the authors give an algorithm for a Bernoulli trial with probability $1/\pi$, based on a formula given by Ramanujan (1914, Eq. (28)), which requires just under 10 bits, on average. It's probable that some of their techniques will be useful in finding algorithms for sampling from continuous distributions; they might also lead to improvements to Algorithm N for normal deviates.

Acknowledgment

I would like to thank Damien Stehlé for pointing out the possibility of using Kahn's algorithm for sampling exactly from the normal distribution.

References

M. Abramowitz and I. A. Stegun, 1964, *Handbook of Mathematical Functions* (NBS, Washington, DC), http://www.cs.bham.ac.uk/~aps/research/projects/as.

J. H. L. Ahrens and U. O. Dieter, 1973, *Extensions of Forsythe's method for random sampling from the normal distribution*, Math. Comp., **27**(124), 927–937, doi:10.1090/S0025-5718-1973-0329190-8.

G. E. P. Box and M. E. Muller, 1958, *A note on the generation of random normal deviates*, Ann. Math. Stat., **29**(2), 610–611, doi:10.1214/aoms/1177706645.

R. P. Brent, 1974, *Algorithm 488: A Gaussian pseudo-random number generator*, Comm. ACM, **17**(12), 704–706, doi:10.1145/361604.361629.

P. Flajolet, M. Pelletier, and M. Soria, 2011, *On Buffon machines and numbers*, in D. Randall, editor, *Proc. 22nd ACM-SIAM Symposium on Discrete Algorithms*, pp. 172–183 (SIAM, Philadelphia), http://www.siam.org/proceedings/soda/2011/SODA11_015_flajoletp.pdf.

P. Flajolet and N. Saheb, 1986, *The complexity of generating an exponentially distributed variate*, J. Algorithms, **7**(4), 463–488, doi:10.1016/0196-6774(86)90014-3.

G. E. Forsythe, 1972, *Von Neumann's comparison method for random sampling from the normal and other distributions*, Math. Comp., **26**(120), 817–826, doi:10.1090/S0025-5718-1972-0315863-9.

L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, 2007, *MPFR: A multiple-precision binary floating-point library with correct rounding*, ACM TOMS, **33**(2), 13:1–15, doi:10.1145/1236463.1236468.

L. Granboulan and T. Pornin, 2007, *Perfect block ciphers with small blocks*, in A. Biryukov, editor, *Fast Software Encryp-*

tion, volume 4593 of *Lecture Notes in Computer Science*, pp. 452–465 (Springer, Berlin), doi:10.1007/978-3-540-74619-5_28, http://www.iacr.org/archive/fse2007/45930457/45930457.pdf.

H. Kahn, 1956, *Applications of Monte Carlo*, Technical Report RM-1237-AEC, RAND Corp., Santa Monica, CA, http://www.rand.org/pubs/research_memoranda/RM1237.html.

C. F. F. Karney, 2012, *RandomLib, version 1.6*, http://randomlib.sf.net.

A. J. Kinderman and J. F. Monahan, 1977, *Computer generation of random variables using the ratio of uniform deviates*, ACM TOMS, **3**(3), 257–260, doi:10.1145/355744.355750.

D. E. Knuth and A. C. Yao, 1976, *The complexity of nonuniform random number generation*, in J. F. Traub, editor, *Algorithms and Complexity*, pp. 357–428 (Academic Press, New York).

J. L. Leva, 1992, *A fast normal random number generator*, ACM TOMS, **18**(4), 449–453, doi:10.1145/138351.138364.

G. Marsaglia and W. W. Tsang, 2000, *The ziggurat method for generating random variables*, J. Stat. Software, **5**(8), 1–7, http://www.jstatsoft.org/v05/i08.

M. Matsumoto and T. Nishimura, 1998, *Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator*, ACM TOMACS, **8**(1), 3–30, doi:10.1145/272991.272995.

D. Micciancio and O. Regev, 2009, *Lattice-based cryptography*, in D. J. Bernstein, J. Buchmann, and E. Dahmen, editors, *Post-Quantum Cryptography*, pp. 147–191 (Springer, Berlin), doi:10.1007/978-3-540-88702-7_5, http://www.cims.nyu.edu/~regev/papers/pqc.pdf.

J. F. Monahan, 1979, *Extensions of von Neumann's method for generating random variables*, Math. Comp., **33**(147), 1065–1069, doi:10.1090/S0025-5718-1979-0528058-7.

—, 1985, *Accuracy in random number generation*, Math. Comp., **45**(172), 559–568, doi:10.1090/S0025-5718-1985-0804945-X.

MPFR, 2011, *A multiple-precision binary floating-point library with correct rounding, version 3.1.0*, http://www.mpfr.org.

S. Ramanujan, 1914, *Modular equations and approximations to π*, Quart. J. Pure App. Math., **45**, 350–372, http://books.google.com/books?id=oSioAM4wORMC&pg=PA23.

RAND Corporation, 1955, *A Million Random Digits with 100,000 Normal Deviates* (The Free Press, Glencoe, IL), http://www.rand.org/pubs/monograph_reports/MR1418.html.

M. Saito and M. Matsumoto, 2008, *SIMD-oriented fast Mersenne twister: A 128-bit pseudorandom number generator*, in A. Keller, S. Heinrich, and H. Niederreiter, editors, *Monte Carlo and Quasi-Monte Carlo Methods 2006*, pp. 607–622 (Springer, Berlin), doi:10.1007/978-3-540-74496-2_36, http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/ARTICLES/sfmt.pdf.

D. B. Thomas, W. Luk, P. H. W. Leong, and J. D. Villasenor, 2007, *Gaussian random number generators*, ACM Comput. Surv., **39**(4), 11:1–38, doi:10.1145/1287620.1287622.

J. von Neumann, 1951, *Various techniques used in connection with random digits*, in A. S. Householder, G. E. Forsythe, and H. H. Germond, editors, *Monte Carlo Method*, number 12 in Applied Mathematics Series, pp. 36–38 (NBS, Washington, DC), proceedings of a symposium held June 29–July 1, 1949, in Los Angeles.

Appendix A: Sample implementation

Here I give listings of `exact_normal_dist.hpp` which implements Algorithm N and of a test program `exact_normal_test.cpp`. Also shown is some sample output.

The implementation is via a header file, which assumes a standard C++ environment. Lines 9–17 define a class `rand_digit` to produce random digits in a base b. Lines 19–70 define a class `rand_number` with realizes a random deviate with an arbitrary number of digits. Finally, lines 72–133 define the class `unit_normal_dist` which samples from the normal distribution. In order to simplify the exposition, `rand_digit` uses `std::rand` which, typically, has a short period; in "production" use, this should be replaced by a generator with a much longer period, for example, the Mersenne Twister (Matsumoto and Nishimura, 1998), which is included in the standard library for C++11, or by a source of truly random digits. Also, in order to avoid overflow in Algorithm C, the base b needs to be small enough that $(2k+2)b$ does not overflow; this limitation is easily removed.

The test program computes and prints 50 normal deviates exactly rounded to 10 decimal digits. This is achieved by generating 11 digits in the fraction and rounding the mid-point of the resulting range.

NOTE: The following files can be obtained by visiting http://arxiv.org/a/karney_c_1, selecting the "other" option for this paper, and clicking on "Download source." Compilation instructions are given in `00README.txt` in the resulting archive.

1. Header File, `exact_normal_dist.hpp`

```
#if !defined(EXACT_NORMAL_DIST_HPP)
#define EXACT_NORMAL_DIST_HPP 1

#include <vector>               // for fraction in rand_number
#include <algorithm>            // for std::swap, std::min, std::max
#include <utility>              // for std::pair
#include <cstdlib>              // for std::rand
```

```
// a class to produce random digits in [0,b)
template<int b> class rand_digit {
  static const int _div = RAND_MAX / b, _max = _div * b;
public:
  rand_digit(unsigned s)        // constructor takes a seed
  { std::srand(s); }
  int operator()()              // a random digit
  { int r; while ((r = std::rand()) >= _max); return r / _div; }
};

// an arbitrary precision random number
template<int b> class rand_number {
  int _s, _n;                   // sign and integer part; N.B. _n >= 0
  std::vector<int> _d;          // the digits in the fraction
public:
  rand_number() : _s(1), _n(0) {}             // construct as (0,1)
  void init() { _s = 1; _n = 0; _d.clear(); } // reset to (0,1)
  void swap(rand_number& t) {
    if (this == &t) return;
    std::swap(_s, t._s); std::swap(_n, t._n); _d.swap(t._d);
  }
  rand_number& operator=(const rand_number& t) // safe copy assignment
  { rand_number r(t); swap(r); return *this; }
  int sign() const { return _s; }
  void negate() { _s = -_s; }
  int integer() const { return _n; }
  void set_integer(int n) { _n = n; } // require n >= 0
  int ndigits() const { return _d.size(); }
  // return digit in fraction generating it if necessary
  int digit(rand_digit<b>& D, int k) {
    for (int i = ndigits(); i <= k; ++i) _d.push_back(D());
    return _d[k];
  }
  // test *this < t
  bool less_than(rand_digit<b>& D, rand_number& t) {
    if (this == &t) return false;
    if (_s != t._s) return _s < t._s;
    if (_n != t._n) return (_s < 0) ^ (_n < t._n);
    for (int k = 0;; ++k) {
      int x = digit(D, k), y = t.digit(D, k);
      if (x != y) return (_s < 0) ^ (x < y);
    }
  }
  // test *this < 1/2
  bool less_than_half(rand_digit<b>& D) {
    if (_s < 0) return true;
    if (_n > 0) return false;
    for (int k = 0;; ++k) {
      int d = digit(D, k);
      if (d < b / 2) return true;
      if (d >= (b + 1) / 2) return false;
    }
  }
  // the uniform range represented by the number
  std::pair<double, double> range() const {
    double x = 0, d = 1;
    for (int k = _d.size(); k--;) { x = (_d[k] + x) / b; d /= b; }
    x += _n;
    return std::pair<double, double>(_s > 0 ? x : -(x + d),
                                     _s > 0 ? (x + d) : -x);
  }
};
```

```
// sample exactly from the normal distribution
template<int b> class unit_normal_dist {
public:

  rand_number<b> operator()(rand_digit<b>& D) const {
    for (;;) {
      int k = G(D);                                    // step 1
      if (!P(D, k * (k - 1))) continue;                // step 2
      _x.init();                                       // step 3
      int j = k + 1; while (j-- && B(D, k, _x));       // step 4
      if (!(j < 0)) continue;
      _x.set_integer(k);                               // step 5
      if (_p.init(), _p.less_than_half(D)) _x.negate(); // step 6
      return _x;                                       // step 7
    }
  }

private:
  mutable rand_number<b> _p, _q, _x; // temporary storage

  // Algorithm H: true with probability exp(-1/2).
  bool H(rand_digit<b>& D) const {
    if (_p.init(), !_p.less_than_half(D)) return true;
    for (;;) {
      if (_q.init(), !_q.less_than(D, _p)) return false;
      if (_p.init(), !_p.less_than(D, _q)) return true;
    }
  }

  // Step N1: return n >= 0 with prob. exp(-n/2) * (1 - exp(-1/2)).
  int G(rand_digit<b>& D) const
  { int n = 0; while (H(D)) ++n; return n; }

  // Step N2: true with probability exp(-n/2).
  bool P(rand_digit<b>& D, int n) const
  { while (n-- && H(D)); return n < 0; }

  // Algorithm C: return (-1, 0, 1) with prob (1/m, 1/m, 1-2/m).
  static int C(rand_digit<b>& D, int m) {
    int n1 = m - 2, n2 = m - 1;
    for (;;) {
      int d = D();
      if ((n1 = std::max(0, n1 * b - d * m)) >= m) return 1;
      if ((n2 = std::min(m, n2 * b - d * m)) <= 0) return -1;
      if (n1 <= 0 && n2 >= m) return 0;
    }
  }

  // Algorithm B: true with prob exp(-x * (2*k + x) / (2*k + 2)).
  bool B(rand_digit<b>& D, int k, rand_number<b>& x) const {
    int n = 0, m = 2 * k + 2, f;
    for (;; ++n) {
      if ( ((f = k ? 0 : C(D, m)) < 0) ||
           (_q.init(), !_q.less_than(D, n ? _p : x)) ||
           ((f = k ? C(D, m) : f) < 0) ||
           (f == 0 && (_p.init(), !_p.less_than(D, x))) ) break;
      _p.swap(_q);              // an efficient way of doing p = q
    }
    return (n % 2) == 0;
  }

};

#endif
```

2. Test Program, `exact_normal_test.cpp`

```
#include <ctime>                // for std::time
#include <iostream>             // for std::cout
#include <iomanip>              // for std::fixed, std::setprecision
#include "exact_normal_dist.hpp"

int main() {
  const int b = 10, p = 10, n = 50;
  unsigned s = std::time(0);
  rand_digit<b> D(s); rand_number<b> x; unit_normal_dist<b> N;
  std::cout << n << " normal deviates rounded to "
            << p << " decimal digits (seed = " << s << ")\n\n"
            << std::fixed << std::setprecision(p) << std::showpos;
  for (int i = 1; i <= n; ++i) {
    x = N(D);                   // generate normal deviate
    x.digit(D, p);              // generate p+1 digits in the fraction
    std::pair<double, double> r = x.range();
    std::cout << (r.first + r.second) / 2 << (i % 5 ? ' ' : '\n');
  }
  return 0;
}
```

3. Sample Output

```
50 normal deviates rounded to 10 decimal digits (seed = 1363577353)

+0.5900895626 -2.8475038632 -1.2625957742 +0.0490725275 -0.7481913286
+0.3615463565 +0.2476988696 +1.5432699093 +0.6572955312 +2.0787863832
-1.0229481519 -0.4487782598 +1.0177729117 -1.8864255023 -0.3178039375
-0.5808291519 -0.9202656614 -0.2466482059 -0.8982221741 +0.2979553171
+1.1007225649 -0.0181433259 +0.1360926971 +0.3408369630 -0.6025914923
-2.3302632498 +0.2586801142 +0.5558464123 +1.0995017431 -0.7912363548
-0.9253888913 +0.5205200343 +0.1950752917 -0.4549750103 +1.5675432394
+0.6598556315 +0.3984283840 +1.1167816249 -0.1015855019 +1.3171410458
-0.3099378684 +0.2188000064 -1.3685027317 -0.4542858594 -0.0287872347
-0.5041824162 -0.0543469225 -0.8011012065 -0.5274086369 +0.2397280757
```