Simulation from the Normal Distribution Truncated to an Interval in the Tail

Zdravko Botev
University of New South Wales
High Street, Kensington
Sydney, NSW 2052, Australia
http://web.maths.unsw.edu.au/~zdravkobotev

Pierre L'Ecuyer
DIRO, Université de Montréal
Montréal, Canada;
and Inria–Rennes, France
http://www.iro.umontreal.ca/~lecuyer

ABSTRACT

We study and compare various methods to generate a random variate from the normal distribution truncated to some finite or semi-infinite interval, with special attention to the situation where the interval is far in the tail. This is required in particular for certain applications in Bayesian statistics, such as to perform exact posterior simulations for parameter inference, but could have many other applications as well. We distinguish the case in which inversion is warranted, and that in which a rejection method is also fine. The algorithms are implemented and available in Java, R, and MATLAB, and the software is freely available.

CCS Concepts

•**Mathematics of computing** → **Distribution functions; Probabilistic algorithms; Statistical software;**

Keywords

Truncated normal, normal quantile, random variate generation, inversion, exact sampling

1. INTRODUCTION

We consider the problem of generating (simulating) a standard normal random variable X, conditional on $a \le X \le b$, where $a < b$ are real numbers, and at least one of them is finite. We are particularly interested in the situation where the interval (a, b) is far in one of the tails, i.e., $a \gg 0$ or $b \ll 0$. The standard methods developed for the non-truncated case do not always work well in this case. Moreover, if we insist on using inversion, the standard inversion methods break down when we are far in the tail. Inversion is preferable to a rejection method (in general) in various simulation applications, for example to maintain synchronization and monotonicity when comparing systems with common random numbers, for derivative estimation and optimization, when using quasi-Monte Carlo methods, etc. [2, 11, 12, 13, 14, 16]. For this reason, a good inversion method is needed, even if rejection is faster. We examine both rejection and inversion methods in this paper.

Our motivation for this work stems from applications in Bayesian statistics and computational biology, in which one wishes to generate a random vector $\boldsymbol{X}$ from the multivariate normal or Student-t distribution conditional on $\boldsymbol{a} \le \boldsymbol{X} \le \boldsymbol{b}$ [11], or to accurately estimate the probability $\mathbb{P}[\boldsymbol{a} \le \boldsymbol{X} \le \boldsymbol{b}]$ [9]. for some rectangular box $[\boldsymbol{a}, \boldsymbol{b}]$. These problems occur in particular for the estimation of certain Bayesian regression models and for exact simulation from these models; see [6] and the references given there. More generally, the box can be replaced by a simplex, which can be transformed into a rectangular box by a change of variables (a linear transformation).

Efficient and reliable simulation methods based on importance sampling were developed recently in [4, 5] for exact simulation from such multivariate conditional distributions and to estimate the conditional probability $\mathbb{P}[\boldsymbol{a} \le \boldsymbol{X} \le \boldsymbol{b}]$. Software tools implementing these methods has been made freely available at MATLAB® Central as a toolbox (see www.mathworks.com/matlabcentral/fileexchange/53796) and as an **R** package on CRAN (see cran.r-project.org/web/packages/TruncatedNormal).

The simulation of $\boldsymbol{X}$ from these algorithms requires repeated draws from a standard normal distribution truncated to different intervals, often far in the tail. That is, we need fast and reliable algorithms to generate $X \sim \mathsf{N}(0, 1)$, conditional on $a \le X \le b$, for arbitrary real numbers $a < b$. Various methods have already been proposed to do that; see for example [6, 7, 10, 18, 21, 23]. Some methods work well when the interval $[a, b]$ contains 0 or is not far from it, but not when $a \gg 0$ or $b \ll 0$. Other methods have been designed for the right tail, i.e., when $a \gg 0$ and $b = \infty$, and use rejection. These methods may be adapted in principle to a finite interval $[a, b]$, but they may become inefficient when the interval $[a, b]$ is narrow. We also found no reliable inversion method for an interval far in the tail (say, for $a > 38$; see Section 2). To generate X from a more general normal distribution with mean μ and variance σ^2 truncated to an interval (a', b'), it suffices to apply a simple linear transformation to recover the problem studied here, so there is no loss of generality in assuming a standard normal distribution.

The aim of this paper is to review and compare the most popular methods we know for this task, propose new efficient methods for certain situations, and provide reliable software implementation of these methods. In particular, we propose a new accurate inversion method for arbitrarily large a and

VALUETOOLS 2016 Oct 26–28, Taormina, Italy

DOI:

improvements to commonly used methods. In some of our discussion, we assume that $a > 0$. The case where $b < 0$ is covered by symmetry (just change the sign) and the case where $a \le 0 \le b$ can be handled by standard methods.

2. SETTING AND BASIC INVERSION

All over the paper, we use ϕ to denote the density of the standard normal distribution (with mean 0 and variance 1), Φ for its cumulative distribution function (cdf), $\overline{\Phi}$ for the complementary cdf, and Φ^{-1} for the inverse cdf defined as

$$\Phi^{-1}(u) = \min\{x \in \mathbb{R} \mid \Phi(x) \ge u\}.$$

Thus, if $X \sim \mathsf{N}(0,1)$,

$$\Phi(x) = \mathbb{P}[X \le x] = \int_{-\infty}^{x} \phi(y)\mathrm{d}y = 1 - \overline{\Phi}(x).$$

Conditional on $a \le X \le b$, X has density

$$\frac{\phi(x)}{\Phi(b) - \Phi(a)} \qquad \text{for } a < x < b, \tag{1}$$

and 0 elsewhere. We denote this truncated normal distribution by $\mathsf{TN}_{a,b}(0,1)$.

It is well known that if $U \sim U(0,1)$, the uniform distribution over the interval $(0,1)$, then

$$X = \Phi^{-1}(\Phi(a) + (\Phi(b) - \Phi(a))U) \tag{2}$$

has exactly the standard normal distribution conditional on $a \le X \le b$. But even though very accurate approximations are available for Φ and Φ^{-1}, (2) is sometimes useless to generate X from the desired conditional distribution.

In particular, recall that whenever computations are made under the IEEE-754 double precision standard (which is typical), any number of the form $1 - \epsilon$ for $0 \le \epsilon < 2 \times 10^{-16}$ (approximately) is identified with 1.0, any positive number smaller than about 10^{-324} cannot be represented at all (it is identified with 0), and numbers smaller than 10^{-308} are represented with less than 52 bits of accuracy. This implies in particular that $\overline{\Phi}(x) = \Phi(-x)$ is identified as 0 whenever $x \ge 39$ and is identified as 1 whenever $-x \ge 8.3$. Thus, (2) cannot work when $a \ge 8.3$. In the latter case, or whenever $a > 0$, it is much better to use the equivalent form:

$$X = -\Phi^{-1}(\overline{\Phi}(a) - (\overline{\Phi}(a) - \overline{\Phi}(b))U), \tag{3}$$

which is accurate for a up to about 37, assuming that we use accurate approximations of $\overline{\Phi}(x)$ for $x > 0$ and of $\Phi^{-1}(u)$ for $u < 1/2$. Such accurate approximations are available for example in [3] for $\Phi^{-1}(u)$ and via the error function `erf` on most computer systems for $\overline{\Phi}(x)$. For larger values of a (and x), a different inversion approach must be developed. We do it in the next section.

3. INVERSION FAR IN THE RIGHT TAIL

When $\overline{\Phi}(x)$ is too small to be represented as a floating-point `double`, we will work instead with the Mills ratio, defined as $q(x) \stackrel{\text{def}}{=} \overline{\Phi}(x)/\phi(x)$, which is the inverse of the hazard rate (or failure rate) evaluated at x. When x is large, this ratio can be approximated by the truncated series (see [1], or [22], page 44):

$$q(x) \approx \frac{1}{x} + \sum_{n=1}^{r} \frac{1 \times 3 \times 5 \times \cdots \times (2n-1)}{(-1)^n x^{2n+1}}. \tag{4}$$

For any x this series diverges when $r \to \infty$ (because the numerator increases faster than exponentially with n), but it gives a lower bound when r is odd and an upper bound when r is even, and the distance between the lowest upper bound and the highest lower bound converges to 0 rapidly when x increases. In our experiments with $x \ge 10$, we compared $r = 5, 6, 7, 8$, and we found no significant difference (up to machine precision) in the approximation of X defined by the inverse cdf in (3), by the method we now describe. In view of (3), we want to find x such that

$$\overline{\Phi}(x) = \Phi(-x) = \overline{\Phi}(a) - (\overline{\Phi}(a) - \overline{\Phi}(b))u,$$

for $0 \le u \le 1$, when a is large. This equation can be rewritten as $h(x) = 0$, where

$$h(x) \stackrel{\text{def}}{=} \overline{\Phi}(a) - \overline{\Phi}(x) + (\overline{\Phi}(b) - \overline{\Phi}(a))u \tag{5}$$

To solve $h(x) = 0$, we will start by finding an approximate solution and then refine this approximation via Newton iterations. We detail how this is achieved. To find an approximate solution, we replace the normal cdf Φ in (3) by the standard Rayleigh distribution, whose complementary cdf and density are given by $\overline{F}(x) = \exp(-x^2/2)$ and $f(x) = x\exp(-x^2/2)$ for $x > 0$. Its inverse cdf can be written explicitly as $F^{-1}(u) = (-2\ln(1-u))^{1/2}$. This choice of approximation of Φ^{-1} in the tail has been used before (see for example [3] and Section 4). It is motivated by the facts that $F^{-1}(u)$ is easy to compute and that $\bar{\Phi}(x)/\bar{F}(x) \to 1$ rapidly when $x \to \infty$. By plugging $\overline{F}$ and F^{-1} in place of $\overline{\Phi}$ and Φ^{-1} in (3), and solving for x, we find the approximate root

$$x \approx (a^2 - 2\ln\left(1 - u + u\exp\left((a^2 - b^2)/2\right)\right))^{1/2}, \tag{6}$$

which is simply the u-th quantile of the standard Rayleigh distribution truncated over (a,b), with density

$$f(x) = \frac{x\exp(-(x^2 - a^2)/2)}{1 - \exp(-(b^2 - a^2)/2)} \qquad \text{for } a < x < b. \tag{7}$$

The next step is to improve the approximation (6) by applying Newton's method to Equation (5). For this, it is convenient to make the change of variable $x = \xi(z)$, where $\xi(z) \stackrel{\text{def}}{=} \sqrt{a^2 - 2\ln(z)}$ and $z = \xi^{-1}(x) = \exp((a^2 - x^2)/2)$, and apply Newton's method to $g(z) \stackrel{\text{def}}{=} h(\xi(z))$. Newton's iteration for solving $g(z) = 0$ has the form

$$z_{\text{new}} = z - g(z)/g'(z),$$

where

$$\begin{aligned}
\frac{g(z)}{g'(z)} &= \frac{h(\xi(z))}{h'(\xi(z))} \cdot \frac{1}{\xi'(z)}, \quad \text{(by the chain rule)} \\
&= -z\xi(z)\frac{\overline{\Phi}(a) - \overline{\Phi}(\xi(z)) + u(\overline{\Phi}(b) - \overline{\Phi}(a))}{\phi(\xi(z))} \\
&= z\xi(z)\frac{\overline{\Phi}(\xi(z)) - \overline{\Phi}(a) + u(\overline{\Phi}(a) - \overline{\Phi}(b))}{\phi(\xi(z))} \\
&= z\xi(z)\Big(q(\xi(z)) - q(a)(1-u)\exp(\tfrac{\xi(z)^2 - a^2}{2}) - \\
&\qquad - q(b)u\exp\left(\tfrac{\xi(z)^2 - b^2}{2}\right)\Big) \\
&= x\Big(zq(x) - q(a)(1-u) - q(b)u\exp\left(\tfrac{a^2 - b^2}{2}\right)\Big),
\end{aligned}$$

where the identity $x = \xi(z)$ was used for the last equality. A key observation here is that, thanks to the replacement of

$\overline{\Phi}$ by q, the computation of $g(z)/g'(z)$ does not involve extremely small quantities that can cause numerical underflow, even for extremely large a. The resulting Newton algorithm converges rapidly whenever a is large (say, $a \geq 10$).

The complete procedure is summarized in Algorithm 1, which we have implemented in Java, MATLAB®, and **R**. According to our experiments, the larger a the faster the convergence. Figure 1 shows the required number of Newton iterations to have $\delta_x \leq \delta^* = 10^{-10}$, as a function of a, where δ_x represents the relative change in x in the last iteration.

Algorithm 1 : Returns the u-quantile of $\mathsf{TN}_{a,b}(0,1)$

Require: Input $u \in (0,1)$, δ^*
  $q_a \leftarrow q(a)$
  $q_b \leftarrow q(b)$
  $c \leftarrow q_a(1-u) + q_b u \exp(\frac{a^2-b^2}{2})$
  $\delta_x \leftarrow \infty$
  $z \leftarrow 1 - u + u\exp(\frac{a^2-b^2}{2})$
  $x \leftarrow \sqrt{a^2 - 2\ln(z)}$
  repeat
    $z \leftarrow z - x(zq(x) - c)$
    $x_{\text{new}} \leftarrow \sqrt{a^2 - 2\ln(z)}$
    $\delta_x \leftarrow |x_{\text{new}} - x|/x$
    $x \leftarrow x_{\text{new}}$
  until $\delta_x \leq \delta^*$
  return Quantile x

Figure 1: Number of Newton iterations necessary to achieve $\delta_x < \delta^* = 10^{-10}$ for the median of $\mathsf{TN}_{a,\infty}(0,1)$, as a function of a.

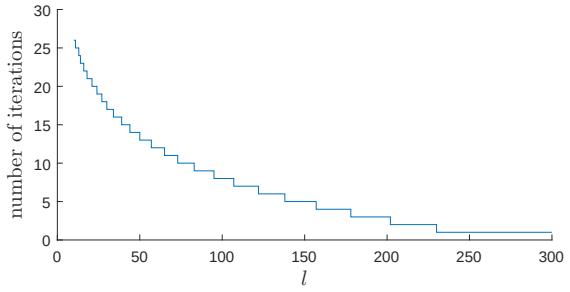

Table 1: Inversion using (3) vs using Algorithm 1: a comparison for some values of a, b, and u, with $r = 5$ and $\delta^* = 10^{-14}$.

a	b	u	using (3)	using Algo. 1
10.0	12.0	0.99	10.446272896499	10.446272896855
10.0	12.0	0.30	10.035260039588	10.035260039626
20.0	22.0	0.99	20.228389499595	20.228389499595
20.0	22.0	0.30	20.017781627473	20.017781627473
30.0	32.0	0.99	30.152946658582	30.152946658582
30.0	32.0	0.30	30.011873653870	30.011873653867
40.0	42.0	0.99	—	40.114892634811
40.0	42.0	0.30	—	40.008910319783
50.0	52.0	0.99	—	50.091982066969
50.0	52.0	0.30	—	50.007130140913

We note that for an interval $[a,b] = [a, a+w]$ of fixed length w, when a increases the conditional density concentrates closer to a. As an illustration, Table 2 gives the conditional probability $\mathbb{P}[X > a+1 \mid X > a] = \overline{\Phi}(a+1)/\overline{\Phi}(a)$ for a few values of a. The third column in the table reports the approximation (4) with $w = 1$,

$$\frac{\overline{\Phi}(a+w)}{\overline{\Phi}(a)} = \frac{q(a+w)\phi(a+w)}{q(a)\phi(a)} \approx \frac{a\exp[-w^2/2 - wa]}{a+w},$$

which shows that this conditional probability decreases as $\exp(-aw)$ when $a \to \infty$. We see that there is practically no difference between generating X conditional on $a \leq X \leq a+1$ and conditional on $X \geq a$ when $a \geq 30$, but there can be a significant difference for small a.

Table 2: Conditional probability $\overline{\Phi}(a+1)/\overline{\Phi}(a)$ and its Mill's ratio approximation for some values of a.

a	$\mathbb{P}[X > a+1 \mid X > a]$	$\frac{a}{a+1}\exp(-a-1/2)$
2	5.93×10^{-2}	5.47×10^{-2}
10	2.51×10^{-5}	2.50×10^{-5}
20	1.19×10^{-9}	1.19×10^{-9}
30	5.49×10^{-14}	5.49×10^{-14}

4. REJECTION METHODS

We now examine *rejection* (or *acceptance-rejection*) methods, which can be faster than inversion. A large collection of rejection-based generation methods for the normal distribution have been proposed over the years; see [6, 7, 10, 23] for surveys, discussions, comparisons, and tests. Most of them (the fastest ones) use a change of variable and/or precomputed tables to speedup the computations. In its most elementary form, a rejection method to generate from some density f uses a hat function $h \geq f$ and rescales h vertically to a probability density $g = h/\int_{-\infty}^{\infty} h(y)\mathrm{d}y$, often called the proposal density. A random variate X is generated from g, is accepted with probability $f(X)/h(X)$, is rejected otherwise, and the procedure is repeated until X is accepted as the retained realization. In practice, more elaborate versions are used that incorporate transformations and partitions of the area under h.

Any of these proposed rejection methods can be applied easily if $\Phi(b) - \Phi(a)$ is large enough, just by adding a rejection step to reject any value that falls outside $[a,b]$. The acceptance probability for this step is $\Phi(b) - \Phi(a)$. When this probability is too small, this becomes too inefficient and something else must be done. One way is to define a proposal g whose support is exactly $[a,b]$, but this could be inefficient (too much overhead) when a and b change very often. Chopin [6] developed a rejection method specially adapted to this situation. It is based on a hat function defined by juxtaposing a large number of vertical rectangles of different heights but equal surface over some finite interval $[a_{\min}, a_{\max}]$, and use an exponential proposal with rate $a = a_{\max}$ (the RejectTail variant of Algorithms 2 below) for the tail above $a_{\max}$ or when $a > a'_{\max}$. The fastest implementation uses 4000 rectangles, $a_{\max} \approx 3.486$, $a'_{\max} \approx 2.605$. This method is fast, although it requires the storage of very large precomputed tables, which could actually slow down computations on certain type of hardware for which memory is limited, like GPUs.

Simple rejection methods for the standard normal truncated to $[a, \infty)$, for $a \geq 0$, have been proposed long ago. Marsaglia [19] proposed a method that uses for g the standard Rayleigh distribution truncated over $[a, \infty)$. An efficient implementation is given in [7, page 381]. Devroye [7, page 382] also gives an algorithm that uses for g an exponential density of rate a shifted by a. There two methods have exactly the same acceptance probability,

$$\alpha(a) = a\sqrt{2\pi}\exp(a^2/2)\overline{\Phi}(a),$$

which converges to 1 when $a \to \infty$. Geweke [8] and Robert [21] optimized the acceptance probability to

$$\beta(a) = \lambda\sqrt{2\pi}\exp\left(a\lambda - \lambda^2/2\right)\ \overline{\Phi}(a)$$

by taking the rate $\lambda = (a + \sqrt{a^2+4})/2 > a$ for the shifted exponential proposal. However, the gain with respect to Devroye's method is small and can be wiped out easily by a larger computing time per step. Table 3 compares these two acceptance probabilities for some values of a. For large a, both are very close to 1 and there is not much difference between them.

Table 3: Acceptance probabilities $\alpha(a)$ and $\beta(a)$ for some values of a.

a	$\alpha(a)$	$\beta(a)$
2	0.84273845	0.93364532
10	0.99028596	0.99520084
20	0.99751852	0.99876308
30	0.99889257	0.99944705

We will compare two ways of adapting these methods to a truncation over a finite interval $[a, b]$. The first one is to keep the same proposal g which is positive over the interval $[a, \infty)$ and reject any value generated above b. The second one truncates and rescales the proposal to $[a, b]$ and applies rejection with the truncated proposal. We label them by *RejectTail* and *TruncTail*, respectively. TruncTail has a smaller rejection probability, by the factor $1 - \overline{\Phi}(a)/\overline{\Phi}(b)$, but also entails additional overhead to properly truncate the proposal. Typically, it is worthwhile only if this additional overhead is small and/or the interval $[a, b]$ is very narrow, so it improves the rejection probability significantly. Our experiments will confirm this.

Algorithms 2, 3, 4, state the rejection methods for the TruncTail case with the exponential proposal with rate a [7], with the rate λ proposed in [21], and with the standard Rayleigh distribution, respectively, extended to the case of a finite interval $[a, b]$. For the RejectTail variant, one would remove the computation of q, replace $\ln(1 - qU)$ by $\ln U$, and add $X \leq b$ to the acceptance condition. Algorithm 5 gives this variant for the Rayleigh proposal.

Algorithm 2 : $X \sim \mathsf{TN}_{a,b}(0, 1)$ with exponential proposal with rate a, truncated

$K_a \leftarrow 2a^2$
$q \leftarrow 1 - \exp(-(b-a)a)$
repeat
  Generate $U, V \sim \mathsf{U}(0, 1)$, independent
  $X \leftarrow -\ln(1 - qU)$
  $E \leftarrow -\ln(V)$
until $X^2 \leq K_a V$
return $a + X/a$

Algorithm 3 : $X \sim \mathsf{TN}_{a,b}(0, 1)$ with exponential proposal with rate λ, truncated

$\lambda \leftarrow (a + \sqrt{a^2+4})/2$
$q \leftarrow 1 - \exp(-(b-a)\lambda)$
repeat
  Generate $U, V \sim \mathsf{U}(0, 1)$, independent
  $X \leftarrow a - \ln(1 - qU)/\lambda$
until $V \leq \exp((X-\lambda)^2/2)$
return $a + X/a$

Algorithm 4 : $X \sim \mathsf{TN}_{a,b}(0, 1)$ with Rayleigh proposal, truncated

$c \leftarrow a^2/2$
$q \leftarrow 1 - \exp(c - b^2/2)$
repeat
  Simulate $U, V \sim \mathsf{U}(0, 1)$, independently.
  $X \leftarrow c - \ln(1 - qU)$
until $V^2 X \leq a$
return $X \leftarrow \sqrt{2X}$

Algorithm 5 : $X \sim \mathsf{TN}_{a,b}(0, 1)$ with Rayleigh proposal and RejectTail

$c \leftarrow a^2/2$
repeat
  Simulate $U, V \sim \mathsf{U}(0, 1)$, independently.
  $X \leftarrow c - \ln(U)$
until $V^2 X \leq a$ and $2X \leq b * b$
return $\sqrt{2X}$

When the interval $[a, b]$ is very narrow, it makes sense to just use the uniform distribution over this interval for the proposal g. This is suggested in [21] and shown in Algorithm 6. Generating from the proposal is then very fast. On the other hand, the acceptance probability may become very small if the interval is far in the tail and $b - a$ is not extremely small. Indeed, the acceptance probability in this case is:

$$\frac{\sqrt{2\pi}\exp(a^2/2)(\overline{\Phi}(a) - \overline{\Phi}(b))}{b-a} = \frac{q(a) - q(b)\exp((a^2-b^2)/2)}{b-a},$$

which decays at a rate of $1/a$ when $a \to \infty$ while $(b - a)$ remains constant.

Algorithm 6 : $X \sim \mathsf{TN}_{a,b}(0,1)$ with uniform proposal, truncated

repeat
  Simulate $U, V \sim \mathsf{U}(0,1)$, independently.
  $X \leftarrow a + (b-a)U$
until $2 \ln V \le a^2 - X^2$
return X

Another choice that the user can have with those generators (and for any variate generator that depends on some distribution parameters) is to either *precompute* various constants that depend on the parameters and store them in some "distribution" object with fixed parameter values, or to *recompute* these parameter-dependent constants each time a new variate is generated. This type of alternative is common in modern variate generation software [15, 17]. The first approach is worthwhile if the time to compute the relevant constants is significant and several random variates are to be generated with exactly the same distribution parameters. For the applications in Bayesian statistics mentioned earlier, it is typical that the parameters a and b change each time a new variate is generated [6]. But there can be applications in which a large number of variates are generated with the same a and b.

For one-sided intervals $[a, \infty)$, the algorithms can be simplified. One can use the RejectTail framework and since $b = \infty$, there is no need to check if $X \le b$. When reporting our test results, we label this the *OneSide* case.

Note that computing an exponential is typically more costly than computing a log (by a factor of 2 or 3 for negative exponents and 10 for large exponents, in our experiments) and the latter is more costly than computing a square root (also by a factor of 10). This means significant speedups could be obtained by avoiding to recompute the exponential each time at the beginning of Algorithms 2, 3, and 4. This is possible if the same parameter b is used several times, or if $b = \infty$, or if we use RejectTail instead of TruncTail.

5. SPEED COMPARISONS

We report a representative subset of results of speed tests made with the different methods, for some pairs (a, b). In each case, we generated 10^8 (100 millions) truncated normal variates, added them up, printed the CPU time required to do that, and printed the sum for verification. The experiments were made in Java using the SSJ library [15], under Eclipse and Windows 10, on a Lenovo X1 Carbon Thinkpad with an Intel Core(TM) i7-5600U (single) processor running at 2.60 GHz. All programs were executed in a single thread and the CPU times were measured using the stopwatch facilities in class `Chrono` of SSJ, which relies on the `getThreadCpuTime` method from the Java class `ThreadMXBean` to obtain the CPU time consumed so far by a single thread, and subtracts to obtain the CPU time consumed between any two instructions. The measurements were repeated a few times to verify consistency and varied by about 1 to 2 percent at most. The compile times are negligible relative to the reported times. Of course, these timings depend on CPU and memory usage by other processes on the computer, and they are likely to change if we move to a different platform, but on standard processors the relative timings should remain roughly the same. They provide a good idea of what is most efficient to do.

Table 4: Time to generate $n = 10^8$ random variates for $[a, b] = [3.0, 3.1]$.

Method	CPU time (seconds)	
	recompute	precompute
Generation in $[a, b)$		
ExponD	6.46	6.22
ExponDRejectTail	23.04	23.20
ExponR	16.63	9.92
ExponRRejectTail	32.40	32.40
ExponRRejectTailLog	25.10	25.30
Rayleigh	10.29	4.60
RayleighRejectTail	15.23	15.33
Uniform	4.26	4.34
InverseSSJ	15.14	8.14
InverseQuickSSJ	18.80	3.31
InverseRightTail	31.12	7.66
Generation in $[a, \infty)$		
ExponDOneSide	6.43	6.46
ExponROneSideLog	7.05	6.99
RayleighOneSide	4.07	4.41
InverseSSJOneSide	18.81	8.20
InverseRightTailOneSide	18.72	7.64

Table 5: Time to generate $n = 10^8$ random variates for $[a, b] = [7.0, 8.0]$.

Method	CPU time	
	recompute	precompute
Generation in $[a, b)$		
ExponD	11.70	6.16
ExponDRejectTail	6.04	6.08
ExponR	15.96	8.98
ExponRRejectTail	9.20	9.09
ExponRRejectTailLog	7.03	7.02
Rayleigh	9.86	4.27
RayleighRejectTail	3.91	3.99
Uniform	25.40	25.68
InverseSSJ	30.67	8.14
InverseRightTail	31.12	7.70
Generation in $[a, \infty)$		
ExponDOneSide	5.90	5.96
ExponROneSideLog	6.80	6.71
RayleighOneSide	3.74	4.05
InverseSSJOneSide	19.00	8.19
InverseRightTailOneSide	18.76	7.59

Table 6: Time to generate $n = 10^8$ random variates for $[a, b] = [100.0, 102.0]$.

Method	CPU time	
	recompute	precompute
Generation in $[a, b)$		
ExponD	11.68	6.01
ExponDRejectTail	5.88	5.91
ExponR	15.79	8.86
ExponRRejectTail	9.13	9.02
ExponRRejectTailLog	6.93	6.96
Rayleigh	9.97	4.16
RayleighRejectTail	3.84	3.90
Uniform	650.62	656.42
InverseMillsRatio	22.31	15.97
Generation in $[a, \infty)$		
ExponDOneSide	5.77	5.82
ExponROneSideLog	6.72	6.63
RayleighOneSide	3.67	3.96
InverseMillsRatioOneSide	15.62	15.84

Table 7: Time to generate $n = 10^8$ random variates for $[a, b] = [100.0, 100.0001]$.

Method	CPU time	
	recompute	precompute
Generation in $[a, b)$		
ExponD	12.31	6.83
ExponDRejectTail	543.80	546.58
ExponR	16.47	10.65
ExponRRejectTail	865.24	865.34
ExponRRejectTailLog	651.19	648.99
Rayleigh	10.59	5.07
RayleighRejectTail	323.08	322.41
Uniform	3.59	3.62
InverseMillsRatio	18.03	12.12
Generation in $[a, \infty)$		
ExponDOneSide	5.79	5.83
ExponROneSideLog	6.74	6.63
RayleighOneSide	3.66	3.99
InverseMillsRatioOneSide	15.67	15.84

Tables 4 to 7 report the timings, in seconds. The two columns "recompute" and "precompute" are for the cases where the constants that depend on a and b are recomputed each time a random variate is generated or are precomputed once for all, respectively, as discussed earlier.

ExponD, ExponR, and Rayleigh refer to the TruncTail versions of Algorithms 2, 3, and 4, respectively. We add "RejectTail" to the name for the RejectTail versions. For ExponRRejectTailLog, we took the log on both sides of the inequality to remove the exponential in the "until" condition. Uniform refers to Algorithm 6. InversionSSJ refers to the default inversion method implemented in SSJ, which uses [3] and gives at least 15 decimal digits of relative precision, combined with a generic (two-sided) "truncated distribution" class also offered in SSJ. InverseQuickSSJ is a faster but much less accurate version based on a cruder approximation of $\overline{\Phi}$ from [20] based on table lookups, which returns about 6 decimal digits of precision. We do not recommend it, due to its low accuracy. Moreover, the implementation we used does not handle well values larger than about 5 in the right tail, so we report results only for small a. InverseRightTail uses the accurate approximation of $\overline{\Phi}$ together with (3). InverseMillsRatio is our new inversion method based on Mills ratio, with $\delta^* = 10^{-10}$. This method is designed for the case where a is large, and our implementation is designed to be accurate for $a \geq 10$, so we do not report results for it in Tables 4 and 5. For all the methods, we add "OneSide" for the simplified OneSide versions, for which $b = \infty$.

For the OneSide case, i.e., $b = \infty$, the Rayleigh proposal gives the fastest method in all cases, and there is no significant gain in precomputing and storing the constant $c = a^2/2$.

For finite intervals $[a, b]$, when $b-a$ is very small so $\overline{\Phi}(b)/\overline{\Phi}(a)$ is close to 1, the uniform proposal wins and the RejectTail variants are very slow. See Table 7. Precomputing the constants is also not useful for the uniform proposal. For larger intervals in the tail, $\overline{\Phi}(x)$ decreases quickly at the beginning of the interval and this leads to very low acceptance ratios; see Tables 5 and 6. A Rayleigh proposal with the RejectTail option is usually the fastest method in this case. Precomputing and storing the constants is also not very useful for this option. For intervals closer to the center, as in Table 4, the uniform proposal performs well for larger (but not too large) intervals, and the RejectTail option becomes slower unless $[a, b]$ is very wide. The reason is that for a fixed $w > 0$, $\overline{\Phi}(a + w)/\overline{\Phi}(a)$ is larger (closer to 1) when $a > 0$ is closer to 0.

6. CONCLUSION

We have proposed and tested both inversion and rejection methods to generate a standard normal truncated to an interval $[a, b]$, when $a \gg 0$.

In general, inversion is slower than the fastest rejection method. But as mentioned in the introduction, there are many situations where inversion is required. The new Mills ratio technique is useful for those situations when a is large (say, $a \geq 10$). For a not too large (say, $a \leq 30$), the accurate approximation of [3] implemented in InversionSSJ works well.

When inversion is not needed, the rejection method with the Rayleigh proposal is usually the fastest when a is large enough, especially if a large number of variates must be generated for the same interval $[a, b]$, in which case the cost of precomputing the constants used in the algorithm can be amortized over many calls. The RejectTail variant is usually the fastest, unless $\overline{\Phi}(b)/\overline{\Phi}(a)$ is far from 0, which happens when the interval $[a, b]$ is very narrow or a is not large (say $a \leq 5$). It is interesting to see that using the Rayleigh proposal is faster than using an exponential proposal as in the popular methods of [6, 8, 21].

7. ACKNOWLEDGMENTS

Zdravko Botev has been supported by the *Australian Research Council Discovery Early Career Researcher Award* DE140100993. Pierre L'Ecuyer received support from an NSERC-Canada Discovery Grant, a Canada Research Chair, and an Inria International Chair.

8. REFERENCES

[1] M. Abramowitz and I. A. Stegun. *Handbook of Mathematical Functions*. Dover, New York, 1970.

[2] S. Asmussen and P. W. Glynn. *Stochastic Simulation*. Springer-Verlag, New York, 2007.

[3] J. M. Blair, C. A. Edwards, and J. H. Johnson. Rational Chebyshev approximations for the inverse of the error function. *Mathematics of Computation*, 30:827–830, 1976.

[4] Z. I. Botev. The normal law under linear restrictions: simulation and estimation via minimax tilting. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 2016. doi: 10.1111/rssb.12162.

[5] Z. I. Botev and P. L'Ecuyer. Efficient estimation and simulation of the truncated multivariate Student-t distribution. In *Proceedings of the 2015 Winter Simulation Conference*, pages 380–391. IEEE Press, 2015.

[6] N. Chopin. Fast simulation of truncated Gaussian distributions. *Statistics and Computing*, 21(2):275–288, 2011.

[7] L. Devroye. *Non-Uniform Random Variate Generation*. Springer-Verlag, New York, NY, 1986.

[8] J. Geweke. Efficient simulation of the multivariate normal and Student-t distributions subject to linear constraints and the evaluation of constraint probabilities. In *Computing science and statistics: Proceedings of the 23rd symposium on the interface*, pages 571–578, Fairfax, Virginia, 1991.

[9] C. Hans. Model uncertainty and variable selection in Bayesian lasso regression. *Statistics and Computing*, 20(2):221–229, 2010.

[10] W. Hörmann, J. Leydold, and G. Derflinger. *Automatic Nonuniform Random Variate Generation*. Springer-Verlag, Berlin, 2004.

[11] D. P. Kroese, T. Taimre, and Z. I. Botev. *Handbook of Monte Carlo Methods*. John Wiley and Sons, New York, 2011.

[12] P. L'Ecuyer. Variance reduction's greatest hits. In *Proceedings of the 2007 European Simulation and Modeling Conference*, pages 5–12, Ghent, Belgium, 2007. EUROSIS.

[13] P. L'Ecuyer. Quasi-Monte Carlo methods with applications in finance. *Finance and Stochastics*, 13(3):307–349, 2009.

[14] P. L'Ecuyer. Random number generation with multiple streams for sequential and parallel computers. In *Proceedings of the 2015 Winter Simulation Conference*, pages 31–44. IEEE Press, 2015.

[15] P. L'Ecuyer. SSJ: Stochastic simulation in Java, software library, 2016. http://simul.iro.umontreal.ca/ssj/.

[16] P. L'Ecuyer and G. Perron. On the convergence rates of IPA and FDC derivative estimators. *Operations Research*, 42(4):643–656, 1994.

[17] J. Leydold. *UNU.RAN—Universal Non-Uniform RANdom number generators*, 2009. Available at http://statmath.wu.ac.at/unuran/.

[18] G. Marsaglia. Generating a variable from the tail of the normal distribution. *Technometrics*, 6(1):101–102, 1964.

[19] G. Marsaglia and T. A. Bray. A convenient method for generating normal variables. *SIAM Review*, 6:260–264, 1964.

[20] G. Marsaglia, A. Zaman, and J. C. W. Marsaglia. Rapid evaluation of the inverse normal distribution function. *Statistics and Probability Letters*, 19:259–266, 1994.

[21] C. P. Robert. Simulation of truncated normal variables. *Statistics and computing*, 5(2):121–125, 1995.

[22] C. G. Small. *Expansions and Asymptotics for Statistics*. Number 115 in Monographs on Statistics and Applied Probability. CRC Press, 2010.

[23] D. B. Thomas, W. Luk, P. H. Leong, and J. D. Villasenor. Gaussian random number generators. *ACM Computing Surveys*, 39(4):Article 11, Nov. 2007.