Miscellaneous Observations on Randomization

This version of the document is dated 2021-07-10.

Peter Occil

1 Contents

2 On a Binomial Sampler

Take the following sampler of a binomial(n, 1/2) distribution (where n is even), which is equivalent to the one that appeared in (Bringmann et al. 2014)[(1)], and adapted to be more programmer-friendly.

1. If n is less than 4, generate n unbiased random bits (zeros or ones) and return their sum. Otherwise, if n is odd, set *ret* to the result of this algorithm with $n = n - 1$, then add an unbiased random bit's value to *ret*, then return *ret*.
2. Set m to floor(sqrt(n)) + 1.
3. (First, sample from an envelope of the binomial curve.) Generate unbiased random bits until a zero is generated this way. Set k to the number of ones generated this way.
4. Set s to an integer in 0, m) chosen uniformly at random, then set i to $k*m + s$.
5. Generate an unbiased random bit. If that bit is 0, set *ret* to $(n/2)+i$. Otherwise, set *ret* to $(n/2)-i-1$.
6. (Second, accept or reject *ret*.) If *ret* < 0 or *ret* > n, go to step 3.
7. With probability choose(n, *ret*)$*m*2^{(k-n)+2}$, return *ret*. Otherwise, go to step 3. (Here, choose(n, k) is a *binomial coefficient*, or the number of ways to choose k out of n labeled items.[(2)])

This algorithm has an acceptance rate of 1/16 regardless of the value of n. However, step 7 will generally require a growing amount of storage and time to exactly calculate the given probability as n gets larger, notably due to the inherent factorial in the binomial coefficient. The Bringmann paper suggests approximating this factorial via Spouge's approximation; however, it seems hard to do so without using floating-point arithmetic,

which the paper ultimately resorts to. Alternatively, the logarithm of that probability can be calculated that is much more economical in terms of storage than the full exact probability. Then, an exponential random variate can be generated, negated, and compared with that logarithm to determine whether the step succeeds.

More specifically, step 7 can be changed as follows:

- (7.) Let p be loggamma(n+1)−loggamma(ret+1)−loggamma((n−ret)+1)+ln(m)+ln(2)*((k−n)+2) (where loggamma(x) is the logarithm of the gamma function).
- (7a.) Generate an exponential random variate with rate 1 (which is the negative natural logarithm of a uniform(0,1) random variate). Set h to 0 minus that number.
- (7b.) If h is greater than p, go to step 3. Otherwise, return ret. (This step can be replaced by calculating lower and upper bounds that converge to p. In that case, go to step 3 if h is greater than the upper bound, or return ret if h is less than the lower bound, or compute better bounds and repeat this step otherwise. See also chapter 4 of (Devroye 1986)[3].)

My implementation of loggamma and the natural logarithm (**interval.py**) relies on rational interval arithmetic (Daumas et al. 2007)[4] and a fast converging version of Stirling's formula for the factorial's natural logarithm (Schumacher 2016)[5].

Also, according to the Bringmann paper, m can be set such that m is in the interval [sqrt(n), sqrt(n)+3], so I implement step 1 by starting with $u = 2^{\text{floor}((1+\beta(n))/2)}$, then calculating v = floor((u+floor(n/u))/2), $w = u$, $u = v$ until $v >= w$, then setting m to w + 1. Here, $\beta(n)$ = ceil(ln(n+1)/ln(2)), or alternatively the minimum number of bits needed to store n (with $\beta(0) = 0$).

Notes:

- A binomial(n, 1/2) random variate, where n is odd, can be generated by adding an unbiased random bit's value (either zero or one with equal probability) to a binomial(n−1, 1/2) random variate.
- As pointed out by Farach-Colton and Tsai (2015)[6], a binomial(n, p) random variate, where p is in the interval (0, 1), can be generated using binomial(n, 1/2) numbers using a procedure equivalent to the following:
  1. Set k to 0 and ret to 0.
  2. If the binary digit at position k after the point in p's binary expansion (that is, 0.bbbb... where each b is a zero or one) is 1, add a binomial(n, 1/2) random variate to ret and subtract the same variate from n; otherwise, set n to a binomial(n, 1/2) random variate.
  3. If n is greater than 0, add 1 to k and go to step 2; otherwise, return ret. (Positions start at 0 where 0 is the most significant digit after the point, 1 is the next, etc.)

3 On a Geometric Sampler

The following algorithm is equivalent to the geometric(px/py) sampler that appeared in (Bringmann and Friedrich 2013)[7], but adapted to be more programmer-friendly. As used in that paper, a geometric(p) random variate expresses the number of failing trials before the first success, where each trial is independent and has success probability p. (Note that the terminology "geometric random variate" has conflicting meanings in academic works. Note also that the algorithm uses the rational number px/py, not an arbitrary real number p; some of the notes in this section indicate how to adapt the

algorithm to an arbitrary p.)

1. Set pn to px, k to 0, and d to 0.
2. While pn*2 <= py, add 1 to k and multiply pn by 2. (Equivalent to finding the largest k >= 0 such that p*2^k <= 1. For the case when p need not be rational, enough of its binary expansion can be calculated to carry out this step accurately, but in this case any k such that p is greater than $1/(2^{k+2})$ and less than or equal to $1/(2^k)$ will suffice, as the Bringmann paper points out.)
3. With probability $(1-px/py)^{2^k}$, add 1 to d and repeat this step. (To simulate this probability, the first sub-algorithm below can be used.)
4. Generate a uniform random integer in [0, 2^k), call it m, then with probability $(1-px/py)^m$, return d*2^k+m. Otherwise, repeat this step. (The Bringmann paper, though, suggests to simulate this probability by sampling only as many bits of m as needed to do so, rather than just generating m in one go, then using the first sub-algorithm on m. However, the implementation, given as the second sub-algorithm below, is much more complicated and is not crucial for correctness.)

The first sub-algorithm returns 1 with probability $(1-px/py)^n$, assuming that n*px/py <= 1. It implements the approach from the Bringmann paper by rewriting the probability using the binomial theorem. (For the case when p need not be rational, the probability $(1-p)^n$ can be simulated using *Bernoulli factory* algorithms, or by calculating its digit expansion or series expansion and using the appropriate algorithm for **simulating irrational constants**. Run that algorithm n times or until it outputs 1, whichever comes first. This sub-algorithm returns 1 if all the runs return 0, or 1 otherwise.)

1. Set *pnum*, *pden*, and j to 1, then set r to 0, then set *qnum* to px, and *qden* to py, then set i to 2.
2. If j is greater than n, go to step 5.
3. If j is even, set *pnum* to *pnum***qden* + *pden***qnum**choose(n,j). Otherwise, set *pnum* to *pnum***qden* − *pden***qnum**choose(n,j).
4. Multiply *pden* by *qden*, then multiply *qnum* by px, then multiply *qden* by py, then add 1 to j.
5. If j is less than or equal to 2 and less than or equal to n, go to step 2.
6. Multiply r by 2, then add an unbiased random bit's value (either 0 or 1 with equal probability) to r.
7. If r <= floor((*pnum**i)/*pden*) − 2, return 1. If r >= floor((*pnum**i)/*pden*) + 1, return 0. If neither is the case, multiply i by 2 and go to step 2.

The second sub-algorithm returns an integer m in [0, 2^k) with probability $(1-px/py)^m$, or −1 with the opposite probability. It assumes that 2^k*px/py <= 1.

1. Set r and m to 0.
2. Set b to 0, then while b is less than k:
    1. (Sum b+2 summands of the binomial equivalent of the desired probability. First, append an additional bit to m, from most to least significant.) Generate an unbiased random bit (either 0 or 1 with equal probability). If that bit is 1, add 2^{k-b} to m.
    2. (Now build up the binomial probability.) Set *pnum*, *pden*, and j to 1, then set *qnum* to px, and *qden* to py.
    3. If j is greater than m or greater than b + 2, go to the sixth substep.
    4. If j is even, set *pnum* to *pnum***qden* + *pden***qnum**choose(m,j). Otherwise, set *pnum* to *pnum***qden* − *pden***qnum**choose(m,j).
    5. Multiply *pden* by *qden*, then multiply *qnum* by px, then multiply *qden* by py, then add 1 to j, then go to the third substep.

6. (Now check the probability.) Multiply r by 2, then add an unbiased random bit's value (either 0 or 1 with equal probability) to r.
7. If r <= floor(($pnum$*2^b)/$pden$) − 2, add a uniform random integer in [0, 2^{k*b}) to m and return m (and, if requested, the number $k-b-1$). If r >= floor(($pnum$*2^b)/$pden$) + 1, return −1 (and, if requested, an arbitrary value). If neither is the case, add 1 to b.

3. Add an unbiased random bit to m. (At this point, m is fully sampled.)
4. Run the first sub-algorithm with $n = m$, except in step 1 of that sub-algorithm, set r to the value of r built up by this algorithm, rather than 0, and set i to 2^k, rather than 2. If that sub-algorithm returns 1, return m (and, if requested, the number −1). Otherwise, return −1 (and, if requested, an arbitrary value).

As used in the Bringmann paper, a bounded geometric(p, n) random variate is a geometric(p) random variate or n (an integer greater than 0), whichever is less. The following algorithm is equivalent to the algorithm given in that paper, but adapted to be more programmer-friendly.

1. Set pn to px, k to 0, d to 0, and $m2$ to the smallest power of 2 that is greater than n (or equivalently, 2^{bits} where $bits$ is the minimum number of bits needed to store n).
2. While pn*2 <= py, add 1 to k and multiply pn by 2.
3. With probability $(1-px/py)^{2^k}$, add 1 to d and then either return n if d*2^k is greater than or equal to $m2$, or repeat this step if less. (To simulate this probability, the first sub-algorithm above can be used.)
4. Generate a uniform random integer in [0, 2^k), call it m, then with probability $(1-px/py)^m$, return min(n, d*2^k+m). In the Bringmann paper, this step is implemented in a manner equivalent to the following (this alternative implementation, though, is not crucial for correctness):
    1. Run the second sub-algorithm above, except return two values, rather than one, in the situations given in the sub-algorithm. Call these two values m and $mbit$.
    2. If m < 0, go to the first substep.
    3. If $mbit$ >= 0, add 2^{mbit} times an unbiased random bit to m and subtract 1 from $mbit$. If that bit is 1 or $mbit$ < 0, go to the next substep; otherwise, repeat this substep.
    4. Return n if d*2^k is greater than or equal to $m2$.
    5. Add a uniform random integer in [0, 2^{mbit+1}) to m, then return min(n, d*2^k+m).

4 Sampling Unbounded Monotone Density Functions

This section shows a preprocessing algorithm to generate a random variate in [0, 1] from a distribution whose probability density function (PDF)—

- is continuous in the interval [0, 1],
- is monotonically decreasing in [0, 1], and
- has an unbounded peak at 0.

The trick here is to sample the peak in such a way that the result is either forced to be 0 or forced to belong to the bounded part of the PDF. This algorithm does not require the area under the curve of the PDF in [0, 1] to be 1; in other words, this algorithm works even if the PDF is known up to a normalizing constant. The algorithm is as follows.

1. Set i to 1.

2. Calculate the cumulative probability of the interval $[0, 2^{-i}]$ and that of $[0, 2^{-(i-1)}]$, call them p and t, respectively.
3. With probability p/t, add 1 to i and go to step 2. (Alternatively, if i is equal to or higher than the desired number of fractional bits in the result, return 0 instead of adding 1 and going to step 2.)
4. At this point, the PDF at $[2^{-i}, 2^{-(i-1)})$ is bounded from above, so sample a random variate in this interval using any appropriate algorithm, including rejection sampling. Because the PDF is monotonically decreasing, the peak of the PDF at this interval is located at 2^{-i}, so that rejection sampling becomes trivial.

It is relatively straightforward to adapt this algorithm for monotonically increasing PDFs with the unbounded peak at 1, or to PDFs with a different domain than [0, 1].

This algorithm is similar to the "inversion–rejection" algorithm mentioned in section 4.4 of chapter 7 of Devroye's *Non-Uniform Random Variate Generation* (1986)**(3)**. I was unaware of that algorithm at the time I started writing the text that became this section (Jul. 25, 2020). The difference here is that it assumes the whole distribution (including its PDF and cumulative distribution function) can take on any value in the interval [0, 1] and only those values (that is, its *support* is [0, 1]), while the algorithm presented in this article doesn't make that assumption (e.g., the interval [0, 1] can cover only part of the PDF's support).

By the way, this algorithm arose while trying to devise an algorithm that can generate an integer power of a uniform random variate, with arbitrary precision, without actually calculating that power (a naïve calculation that is merely an approximation and usually introduces bias); for more information, see my other article on **partially-sampled random numbers**. Even so, the algorithm I have come up with in this note may be of independent interest.

In the case of powers of a uniform [0, 1] random variate X, namely X^n, the ratio p/t in this algorithm has a very simple form, namely $(1/2)^{1/n}$, which is possible to simulate using a so-called *Bernoulli factory* algorithm without actually having to calculate this ratio. Note that this formula is the same regardless of i. This is found by taking the PDF $f(x) = x^{1/n}/(x * n)$ and finding the appropriate p/t ratios by integrating f over the two intervals mentioned in step 2 of the algorithm.

5 Certain Families of Distributions

This section is a note on certain families of univariate (one-variable) probability distributions, with emphasis on sampling random variates from them. Some of these families are described in Ahmad et al. (2019)**(8)**.

The following definitions are used:

- A distribution's *quantile function* (also known as *inverse cumulative distribution function* or *inverse CDF*) is a nondecreasing function that maps uniform random variates in the closed interval [0, 1] to numbers that follow the distribution.
- A distribution's *support* is the set of values the distribution can take on. For example, the beta distribution's support is the interval [0, 1], and the normal distribution's support is the entire real line.

In general, families of the form "X-G" (such as "beta-G" (Eugene et al., 2002)**(9)**) use two distributions, X and G, where X is a continuous distribution whose support is the interval [0, 1] and G is a distribution with an easy-to-compute quantile function. The following

algorithm samples a random variate following a distribution from this kind of family:

1. Generate a random variate that follows the distribution X. (Or generate a uniform **partially-sampled random number (PSRN)** that follows the distribution X.) Call the number x.
2. Calculate the quantile for G of x, and return that quantile. (If x is a uniform PSRN, see "Random Variate Generation via Quantiles", later.)

Certain special cases of the "X-G" families, such as the following, use a specially designed distribution for X:

- The *alpha power* or *alpha power transformed* family (Mahdavi and Kundu 2017)**(10)**. The family uses a shape parameter $\alpha > 0$; step 1 is modified to read: "Generate a uniform(0, 1) random variate U, then set x to $\ln((\alpha-1)*U + 1)/\ln(\alpha)$ if α != 1, and U otherwise."
- The *exponentiated* family (Mudholkar and Srivastava 1993)**(11)**. The family uses a shape parameter $a > 1$; step 1 is modified to read: "Generate a uniform(0, 1) random variate u, then set x to $u^{1/a}$."
- The *transmuted-G* family (Shaw and Buckley 2007)**(12)**. The family uses a shape parameter η in the interval $[-1, 1]$; step 1 is modified to read: "Generate a piecewise linear random variate in [0, 1] with weight $1-\eta$ at 0 and weight $1+\eta$ at 1, call the number x. (It can be generated as follows, see also (Devroye 1986, p. 71-72)**(3)**: With probability $\min(1-\eta, 1+\eta)$, generate x, a uniform(0, 1) random variate. Otherwise, generate two uniform(0, 1) random variates, set x to the higher of the two, then if η is less than 0, set x to $1-x$.)". ((Granzotto et al. 2017)**(13)** mentions the same distribution, but with parameter $\lambda = \eta + 1$, in the interval [0, 2].)
- A *cubic rank transmuted* distribution (Granzotto et al. 2017)**(13)** uses parameters λ_0 and λ_1 in the interval [0, 1]; step 1 is modified to read: "Generate three uniform(0, 1) random variates, then sort them in ascending order. Then, choose 1, 2, or 3 with probability proportional to these weights: $[\lambda_0, \lambda_1, 3-\lambda_0-\lambda_1]$. Then set x to the first, second, or third variate if 1, 2, or 3 is chosen this way, respectively."

In fact, the "X-G" families are a special case of the so-called "transformed–transformer" family of distributions introduced by Alzaatreh et al. (2013)**(14)** that uses two distributions, X and G, where X (the "transformed") is an arbitrary continuous distribution, G (the "transformer") is a distribution with an easy-to-compute quantile function, and W is a nondecreasing function that maps a number in [0, 1] to a number that has the same support as X and meets certain other conditions. The following algorithm samples a random variate from this kind of family:

1. Generate a random variate that follows the distribution X. (Or generate a uniform PSRN that follows X.) Call the number x.
2. Calculate the quantile for G of $W^{-1}(x)$ (where $W^{-1}(.)$ is the inverse of W), and return that quantile. (If x is a uniform PSRN, see "Random Variate Generation via Quantiles", later.)

The following are special cases of the "transformed–transformer" family:

- The "T-R{Y}" family (Aljarrah et al., 2014)**(15)**, in which T is an arbitrary continuous distribution (X in the algorithm above), R is a distribution with an easy-to-compute quantile function (G in the algorithm above), and W is the quantile function for the distribution Y, whose support must be included in the support of T (so that $W^{-1}(x)$ is the CDF for Y).
- Several versions of W have been proposed for the case when distribution X's support

is \0, ∞), such as the Rayleigh and gamma distributions. They include:
    - $W(x) = -\ln(1-x)$ ($W^{-1}(x) = 1-\exp(-x)$). Suggested in the original paper by Alzaatreh et al.
    - $W(x) = x/(1-x)$ ($W^{-1}(x) = x/(1+x)$). Suggested in the original paper by Alzaatreh et al. This choice forms the so-called "odd X G" family, and one example is the "odd log-logistic G" family (Gleaton and Lynch 2006)[**(16)**].

Many special cases of the "transformed–transformer" family have been proposed in many papers, and usually their names suggest the distributions that make up this family. Some members of the "odd X G" family have names that begin with the word "generalized", and in most such cases this corresponds to $W^{-1}(x) = (x/(1+x))^{1/a}$, where $a > 0$ is a shape parameter; examples include the "generalized odd gamma-G" family (Hosseini et al. 2018)**(17)**.

A family very similar to the "transformed–transformer" family uses a *decreasing* W. When distribution X's support is \0, ∞), one such W that has been proposed is $W(x) = -\ln(x)$ ($W^{-1}(x) = \exp(-x)$; examples include the "Rayleigh-G" family or "Rayleigh–Rayleigh" distribution (Al Noor and Assi 2020)[**(18)**, as well as the "generalized gamma-G" family, where "generalized gamma" refers to the Stacy distribution (Boshi et al. 2020)**(19)**).

A *compound distribution* is simply the minimum of N random variates distributed as X, where $N >= 1$ is an integer distributed as the discrete distribution Y (Tahir and Cordeiro 2016)**(20)**. For example, the "beta-G-geometric" family represents the minimum of N beta-G random variates, where N is a random variate expressing 1 plus the number of failures before the first success, with each success having the same probability.

A *complementary compound distribution* is the maximum of N random variates distributed as X, where $N >= 1$ is an integer distributed as the discrete distribution Y. An example is the "geometric zero-truncated Poisson distribution", where X is the distribution of 1 plus the number of failures before the first success, with each success having the same probability, and Y is the zero-truncated Poisson distribution (Akdoğan et al., 2020)**(21)**.

An *inverse X distribution* (or *inverted X distribution*) is generally the distribution of the reciprocal of a random variate distributed as X. For example, an *inverse exponential* random variate (Keller and Kamath 1982)**(22)** is the reciprocal of an exponential random variate with rate 1 (and so is distributed as $-1/\ln(U)$ where U is a uniform(0, 1) random variate) and may be multiplied by a parameter $\theta > 0$.

A *weight-biased X* or *weighted X distribution* uses a distribution X and a weight function $w(x)$ whose values lie in [0, 1] everywhere in X's support. The following algorithm samples from a weighted distribution (see also (Devroye 1986, p. 47)**(3)**):

1. Generate a random variate that follows the distribution X. (Or generate a uniform PSRN that follows X.) Call the number x.
2. With probability $w(x)$, return x. Otherwise, go to step 1.

To generate an *inflated X* (also called *c-inflated X*) random variate with parameters c and α, generate—

- c with probability α, and
- a random variate distributed as X otherwise.

For example, a *zero-inflated beta* random variate is 0 with probability α and a beta random variate otherwise (the parameter c is 0) (Ospina and Ferrari 2010)**(23)** A zero-

and-one inflated X distribution is 0 or 1 with probability α and distributed as X otherwise. For example, to generate a *zero-and-one-inflated unit Lindley* random variate (with parameters α, θ, and p) (Chakraborty and Bhattacharjee 2021)[24]:

1. With probability α, return a number that is 0 with probability p and 1 otherwise.
2. Generate a unit Lindley(θ) random variate, that is, generate $x/(1+x)$ where x is a **Lindley(θ) random variate**.

6 Certain Distributions

In the table below, U is a uniform(0, 1) random variate.

This distribution:	Is distributed as:	And uses these parameters:
Power function(a, c).	$c*U^{1/a}$.	$a > 0$, $c > 0$.
Right-truncated Weibull(a, b, c) (Jodrá 2020)[25].	Minimum of N power function(b, c) random variates, where N is zero-truncated Poisson($a*c^b$).	a, b, $c > 0$.
Lehmann Weibull($a1$, $a2$, β) (Elgohari and Yousof 2020)[26].	$(\ln(1/U)/\beta)^{1/a1}/a2$ or $E^{1/a1}/a2$	$a1$, $a2$, $\beta > 0$. E is exponential with rate β.
Marshall–Olkin(α).	$(1-U)/(U*(\alpha-1) + 1)$.	α in [0, 1].
Lomax(α).	$(-1/(U-1))^{1/\alpha}-1$.	$\alpha > 0$.
Power Lomax(α, β) (Rady et al. 2016)[27].	$L^{1/\beta}$	$\beta > 0$; L is Lomax(α).

7 Batching Random Samples via Randomness Extraction

Devroye and Gravel (2020)[28] suggest the following randomness extractor to reduce the number of random bits needed to produce a batch of samples by a sampling algorithm. The extractor works based on the probability that the algorithm consumes X random bits to produce a specific output Y (or $P(X \mid Y)$ for short):

1. Start with the interval [0, 1].
2. For each pair (X, Y) in the batch, the interval shrinks from below by $P(X-1 \mid Y)$ and from above by $P(X \mid Y)$. (For example, if [0.2, 0.8] (range 0.6) shrinks from below by 0.1 and from above by 0.8, the new interval is [0.2+0.1*0.6, 0.2+0.8*0.6] = [0.26, 0.68]. For correctness, though, the interval is not allowed to shrink to a single point, since otherwise step 3 would run forever.)
3. Extract the bits, starting from the binary point, that the final interval's lower and upper bound have in common (or 0 bits if the upper bound is 1). (For example, if the final interval is [0.101010..., 0.101110...] in binary, the bits 1, 0, 1 are extracted, since the common bits starting from the point are 101.)

After a sampling method produces an output Y, both X (the number of random bits the sampler consumed) and Y (the output) are added to the batch and fed to the extractor, and new bits extracted this way are added to a queue for the sampling method to use to produce future outputs. (Notice that the number of bits extracted by the algorithm above

grows as the batch grows, so only the new bits extracted this way are added to the queue this way.)

The issue of finding $P(X \mid Y)$ is now discussed. Generally, if the sampling method implements a random walk on a binary tree that is driven by unbiased random bits and has leaves labeled with one outcome each (Knuth and Yao 1976)[29], $P(X \mid Y)$ is found as follows (and Claude Gravel clarified to me that this is the intention of the extractor algorithm): Take a weighted count of all leaves labeled Y up to depth X (where the weight for depth z is $1/2^z$), then divide it by a weighted count of all leaves labeled Y at all depths (for instance, if the tree has two leaves labeled Y at z=2, three at z=3, and three at z=4, and X is 3, then $P(X \mid Y)$ is $(2/2^2+3/2^3) / (2/2^2+3/2^3+3/2^4)$). In the special case where the tree has at most 1 leaf labeled Y at every depth, this is implemented by finding $P(Y)$, or the probability to output Y, then chopping $P(Y)$ up to the X^{th} binary digit after the point and dividing by the original $P(Y)$ (for instance, if X is 4 and P(Y) is 0.101011..., then $P(X \mid Y)$ is 0.1010 / 0.101011...).

Unfortunately, $P(X \mid Y)$ is not easy to calculate when the number of values Y can take on is large or even unbounded. In this case, I can suggest the following ad hoc algorithm, which uses a randomness extractor that takes *bits* as input, such as the von Neumann, Peres, or Zhou–Bruck extractor (see "**Notes on Randomness Extraction**"). The algorithm counts the number of bits it consumes (X) to produce an output, then feeds X to the extractor as follows.

1. Let z be abs($X-lastX$), where $lastX$ is either the last value of X fed to this extractor for this batch or 0 if there is no such value.
2. If z is greater than 0, feed the bits of z from most significant to least significant to a queue of extractor inputs.
3. Now, when the sampler consumes a random bit, it checks the input queue. As long as 64 bits or more are in the input queue, the sampler dequeues 64 bits from it, runs the extractor on those bits, and adds the extracted bits to an output queue. (The number 64 can instead be any even number greater than 2.) Then, if the output queue is not empty, the sampler dequeues a bit from that queue and uses that bit; otherwise it generates an unbiased random bit as usual.

8 Random Variate Generation via Quantiles

This note is about generating random variates from a continuous distribution via inverse transform sampling (or via quantiles), using uniform **partially-sampled random numbers (PSRNs)**. See "Certain Families of Distributions" for a definition of quantile functions. A *uniform PSRN* is ultimately a number that lies in an interval; it contains a sign, an integer part, and a fractional part made up of digits sampled on demand.

Take the following situation:

- Let $f(.)$ be a function applied to a or b before calculating the quantile.
- Let $Q(z)$ be the quantile function for the desired distribution.
- Let x be a random variate in the form of a uniform PSRN, so that this PSRN will lie in the interval $[a, b]$. If $f(t) = t$, the PSRN x must have a positive sign and an integer part of 0, so that the interval $[a, b]$ is either the interval [0, 1] or a closed interval in [0, 1], depending on the PSRN's fractional part.
- Let β be the digit base of digits in x's fractional part (such as 2 for binary).

Then the following algorithm transforms that number to a random variate for the distribution associated with Q, with a desired error tolerance of ε with probability 1 (see

(Devroye and Gravel 2020)[28]):

1. Generate additional digits of x uniformly at random—thus shortening the interval [a, b]—until a lower bound of $Q(f(a))$ and an upper bound of $Q(f(b))$ differ by no more than $2*\varepsilon$. Call the two bounds *low* and *high*, respectively.
2. Return $low+(high-low)/2$.

In some cases, it may be possible to calculate the needed digit size in advance.

As one example, if $f(t) = t$ and the quantile function is *Lipschitz continuous* on the interval [a, b], which roughly means that it's a continuous function with no vertical slope on that interval, then the following algorithm generates a quantile with error tolerance ε:

1. Let d be ceil((ln(max(1,L)) − ln(ε)) / ln(β)), where L is an upper bound of the quantile function's maximum slope (also known as the *Lipschitz constant*). For each digit among the first d digits in x's fractional part, if that digit is unsampled, set it to a digit chosen uniformly at random.
2. The PSRN x now lies in the interval [a, b]. Calculate lower and upper bounds of $Q(a)$ and $Q(b)$, respectively, that are within $\varepsilon/2$ of the true quantiles, call the bounds *low* and *high*, respectively.
3. Return $low+(high-low)/2$.

This algorithm chooses a random interval of size equal to β^d, and because the quantile function is Lipschitz continuous, the values at the interval's bounds are guaranteed to vary by no more than $2*\varepsilon$ (actually ε, but the calculation in step 2 adds an additional error of at most ε), which is needed to meet the tolerance ε (see also Devroye and Gravel 2020[28]).

A similar algorithm can exist even if the quantile function Q is not Lipschitz continuous on the interval [a, b].

Specifically, if—

- $f(t) = t$,
- Q on the interval [a, b] is continuous and has a minimum and maximum, and
- Q on [a, b] admits a continuous and monotone increasing function $\omega(\delta)$ as a *modulus of continuity*,

then d in step 1 above can be calculated as—

$$\max(0, \operatorname{ceil}(-\ln(\omega^{-1}(\varepsilon))/\ln(\beta))),$$

where $\omega^{-1}(\varepsilon)$ is the inverse of the modulus of continuity. (Loosely speaking, a modulus of continuity $\omega(\delta)$ gives the quantile function's maximum range in a window of size δ, and the inverse modulus $\omega^{-1}(\varepsilon)$ finds a window small enough that the quantile function differs by no more than ε in the window.[30]).[31]

For example—

- if *Q is Lipschitz continuous with Lipschitz constant L on [a, b], then the function is no "steeper" than $\omega(\delta) = L*\delta$, so $\omega^{-1}(\varepsilon\_) = \varepsilon/L$*, and
- if *Q is α-Hölder continuous with Hölder constant M on that interval, then the function is no "steeper" than $\omega(\delta) = M*\delta\alpha$, so $\omega^{-1}(\varepsilon) = (\varepsilon/M)^{1/}\alpha\_$*.

The algorithms given earlier in this section have a disadvantage: the desired error tolerance has to be made known to the algorithm in advance. To generate a quantile to any error tolerance (even if the tolerance is not known in advance), a rejection sampling approach is needed. For this to work:

- The target distribution's probability density function, or a function proportional to it, must be known. This is called the density function in the rest of this section.
- The density function must be continuous almost everywhere and bounded from above (see also (Devroye and Gravel 2020)[28]).

Here is a sketch of how this rejection sampler might work:

1. After using one of the algorithms given earlier in this section to sample digits of x as needed, let a and b be x's upper and lower bounds. Calculate lower and upper bounds of the quantiles of $f(a)$ and $f(b)$ (the bounds are [$alow$, $ahigh$] and [$blow$, $bhigh$] respectively).
2. Given the density function, sample a uniform PSRN, y, in the interval [$alow$, $bhigh$] using an arbitrary-precision rejection sampler such as Oberhoff's method (described in an **appendix to the PSRN article**).
3. Accept y (and return it) if it clearly lies in [$ahigh$, $blow$]. Reject y (and go to the previous step) if it clearly lies outside [$alow$, $bhigh$]. If y clearly lies in [$alow$, $ahigh$] or in [$blow$, $bhigh$], generate more digits of x, uniformly at random, and go to the first step.
4. If y doesn't clearly fall in any of the cases in the previous step, generate more digits of y, uniformly at random, and go to the previous step.

9 ExpoExact

This algorithm `ExpoExact`, samples an exponential random variate given the rate `rx`/`ry` with an error tolerance of $2^{-precision}$; for more information, see "**Partially-Sampled Random Numbers**"; see also Morina et al. (2019)[32]; Canonne et al. (2020)[33]. In this section, `RNDINT(1)` generates an independent unbiased random bit.

```
METHOD ZeroOrOneExpMinus(x, y)
  if y <= 0 or x<0: return error
  if x==0: return 1 // exp(0) = 1
  if x > y
    x = rem(x, y)
    if x>0 and ZeroOrOneExpMinus(x, y) == 0: return 0
    for i in 0...floor(x/y): if ZeroOrOneExpMinus(1,1) == 0: return 0
    return 1
  end
  r = 1
  oy = y
  while true
    if ZeroOrOne(x, y) == 0: return r
    r=1-r; y = y + oy
  end
END METHOD

METHOD ExpoExact(rx, ry, precision)
   ret=0
   for i in 1..precision
    // This loop adds to ret with probability 1/(exp(2^-prec)+1).
    // References: Alg. 6 of Morina et al. 2019; Canonne et al. 2020.
    denom=pow(2,i)*ry
    while true
       if RNDINT(1)==0: break
       if ZeroOrOneExpMinus(rx, denom) == 1:
         ret=ret+MakeRatio(1,pow(2,i))
    end
   end
```

```
   while ZeroOrOneExpMinus(rx,ry)==1: ret=ret+1
   return ret
END METHOD
```

> **Note:** After `ExpoExact` is used to generate a random variate, an application can append additional binary digits (such as `RNDINT(1)`) to the end of that number while remaining accurate to the given precision (Karney 2014)[34].

10 A sampler for distributions with nonincreasing or nondecreasing weights

An algorithm for sampling an integer in the interval [a, b) with probability proportional to weights listed in *nonincreasing* order (example: [10, 3, 2, 1, 1] when $a = 0$ and $b = 5$) can be implemented as follows (Chewi et al. 2021)[35]. It has a logarithmic time complexity in terms of setup and sampling.

- Setup: Let $w[i]$ be the weight for integer i (with i starting at a).
  1. (Envelope weights.) Build a list q as follows: The first item is $w[a]$, then set j to 1, then while $j < b-a$, append $w[a + j]$ and multiply j by 2. The list q's items should be rational numbers that equal the true values, if possible, or overestimate them if not.
  2. (Envelope chunk weights.) Build a list r as follows: The first item is $q[0]$, then set j to 1 and m to 1, then while $j < b-a$, append $q[m]$*min($(b-a) - j$, j) and multiply j by 2 and add 1 to m.
  3. (Start and end points of each chunk.) Build a list D as follows: The first item is the list [a, a+1], then set j to 1, then while $j < n$, append the list [j, j + min($(b-a) - j$, j)] and multiply j by 2.
- Sampling:
  1. Choose an integer in [0, s) with probability proportional to the weights in r, where s is the number of items in r. Call the chosen integer k.
  2. Set x to an integer chosen uniformly at random in the half-open interval [$D[k]$[0], $D[k]$[1]).
  3. With probability $w[x]$ / $q[k]$, return x. Otherwise, go to step 1.

For *nondecreasing* rather than nonincreasing weights, the algorithm is as follows instead:

- Setup: Let $w[i]$ be the weight for integer i (with i starting at a).
  1. (Envelope weights.) Build a list q as follows: The first item is $w[b-1]$, then set j to 1, then while $j < (b-a)$, append $w[b-1-j]$ and multiply j by 2. The list q's items should be rational numbers that equal the true values, if possible, or overestimate them if not.
  2. (Envelope chunk weights.) Build a list r as given in step 2 of the previous algorithm's setup.
  3. (Start and end points of each chunk.) Build a list D as follows: The first item is the list [$b-1$, b], then set j to 1, then while $j < (b-a)$, append the list [$(b-j) -$ min($(b-a) - j$, j), $b-j$] and multiply j by 2.
- The sampling is the same as for the previous algorithm.

> **Note:** The weights can be base-β logarithms, especially since logarithms preserve order, but in this case the algorithm requires changes. In the setup step 2, replace "$q[m]$*min($(b-a)$" with "$q[m]$+ln(min($(b-a)$)/ln(β)" (which is generally inexact unless β is 2); in sampling step 1, use an algorithm that takes base-β logarithms as weights; and replace sampling step 3 with "Generate an exponential random variate with rate ln(β). If that variate is greater than $q[k]$

minus $w[x]$, return x. Otherwise, go to step 1." These modifications can introduce numerical errors unless care is taken, such as by using partially-sampled random numbers (PSRNs).

11 A sampler for unimodal distributions of weights

The following is an algorithm for sampling an integer in the interval _a_, b) with probability proportional to a *unimodal distribution* of weights (that is, nondecreasing on the left and nonincreasing on the right) (Chewi et al. 2021)[(35)]. It assumes the mode (the point with the highest weight) is known. An example is [1, 3, 9, 4, 4] when a = 0 and b = 5, and the *mode* is 2, which corresponds to the weight 9. It has a logarithmic time complexity in terms of setup and sampling.

- Setup:
  1. Find the point with the highest weight, such as via binary search. Call this point *mode*.
  2. Run the setup for *nondecreasing* weights on the interval [a, *mode*), then run the setup for *nonincreasing* weights on the interval [*mode*, b). Both setups are described in the previous section. Then, concatenate the two q lists into one, the two r lists into one, and the two D lists into one.
- The sampling is the same as for the algorithms in the previous section.

12 Notes

- (1) K. Bringmann, F. Kuhn, et al., “Internal DLA: Efficient Simulation of a Physical Growth Model.” In: *Proc. 41st International Colloquium on Automata, Languages, and Programming (ICALP'14)*, 2014.
- (2) choose(n, k) = (1*2*3*...*n)/((1*...*k)*(1*...*(n−k))) = n!/(k! * (n − k)!) is a *binomial coefficient*, or the number of ways to choose k out of n labeled items. It can be calculated, for example, by calculating i/(n−i+1) for each integer i in the interval [n−k+1, n], then multiplying the results (Yannis Manolopoulos. 2002. "**Binomial coefficient computation: recursion or iteration?**", SIGCSE Bull. 34, 4 (December 2002), 65–67). Note that for every m>0, choose(m, 0) = choose(m, m) = 1 and choose(m, 1) = choose(m, m−1) = m; also, in this document, choose(n, k) is 0 when k is less than 0 or greater than n.
- (3) Devroye, L., ***Non-Uniform Random Variate Generation***, 1986.
- (4) Daumas, M., Lester, D., Muñoz, C., "**Verified Real Number Calculations: A Library for Interval Arithmetic**", arXiv:0708.3721 [cs.MS], 2007.
- (5) R. Schumacher, "**Rapidly Convergent Summation Formulas involving Stirling Series**", arXiv:1602.00336v1 [math.NT], 2016.
- (6) Farach-Colton, M. and Tsai, M.T., 2015. Exact sublinear binomial sampling. *Algorithmica* 73(4), pp. 637-651.
- (7) Bringmann, K., and Friedrich, T., 2013, July. Exact and efficient generation of geometric random variates and random graphs, in *International Colloquium on Automata, Languages, and Programming* (pp. 267-278).
- (8) Ahmad, Z. et al. "Recent Developments in Distribution Theory: A Brief Survey and Some New Generalized Classes of distributions." Pakistan Journal of Statistics and Operation Research 15 (2019): 87-110.
- (9) Eugene, N., Lee, C., Famoye, F., "Beta-normal distribution and its applications", *Commun. Stat. Theory Methods* 31, 2002.
- (10) Mahdavi, Abbas, and Debasis Kundu. "A new method for generating distributions with an application to exponential distribution." *Communications in Statistics -- Theory and Methods* 46, no. 13

(2017): 6543-6557.

- (11) Mudholkar, G. S., Srivastava, D. K., "Exponentiated Weibull family for analyzing bathtub failure-rate data", *IEEE Transactions on Reliability* 42(2), 299-302, 1993.
- (12) Shaw, W.T., Buckley, I.R.C., "The alchemy of probability distributions: Beyond Gram-Charlier expansions, and a skew-kurtotic-normal distribution from a rank transmutation map", 2007.
- (13) Granzotto, D.C.T., Louzada, F., et al., "Cubic rank transmuted distributions: inferential issues and applications", *Journal of Statistical Computation and Simulation*, 2017.
- (14) Alzaatreh, A., Famoye, F., Lee, C., "A new method for generating families of continuous distributions", *Metron* 71:63–79 (2013).
- (15) Aljarrah, M.A., Lee, C. and Famoye, F., "On generating T-X family of distributions using quantile functions", Journal of Statistical Distributions and Applications,1(2), 2014.
- (16) Gleaton, J.U., Lynch, J. D., "Properties of generalized log-logistic families of lifetime distributions", *Journal of Probability and Statistical Science* 4(1), 2006.
- (17) Hosseini, B., Afshari, M., "The Generalized Odd Gamma-G Family of Distributions: Properties and Application", *Austrian Journal of Statistics* vol. 47, Feb. 2018.
- (18) N.H. Al Noor and N.K. Assi, "Rayleigh-Rayleigh Distribution: Properties and Applications", *Journal of Physics: Conference Series* 1591, 012038 (2020). The underlying Rayleigh distribution uses a parameter θ (or λ), which is different from *Mathematica*'s parameterization with σ = sqrt($1/\theta^2$) = sqrt($1/\lambda^2$). The first Rayleigh distribution uses θ and the second, λ.
- (19) Boshi, M.A.A., et al., "Generalized Gamma – Generalized Gompertz Distribution", *Journal of Physics: Conference Series* 1591, 012043 (2020).
- (20) Tahir, M.H., Cordeiro, G.M., "Compounding of distributions: a survey and new generalized classes", *Journal of Statistical Distributions and Applications* 3(13), 2016.
- (21) Akdoğan, Y., Kus, C., et al., "Geometric-Zero Truncated Poisson Distribution: Properties and Applications", *Gazi University Journal of Science* 32(4), 2019.
- (22) Keller, A.Z., Kamath A.R., "Reliability analysis of CNC machine tools", *Reliability Engineering* 3 (1982).
- (23) Ospina, R., Ferrari, S.L.P., "Inflated Beta Distributions", 2010.
- (24) Chakraborty, S., Bhattacharjee, S., "**Modeling proportion of success in high school leaving examination- A comparative study of Inflated Unit Lindley and Inflated Beta distribution**", arXiv:2103.08916 [stat.ME], 2021.
- (25) Jodrá, P., "A note on the right truncated Weibull distribution and the minimum of power function distributions", 2020.
- (26) Elgohari, Hanaa, and Haitham Yousof. "New Extension of Weibull Distribution: Copula, Mathematical Properties and Data Modeling." Stat., Optim. Inf. Comput., Vol.8, December 2020.
- (27) Rady, E.H.A., Hassanein, W.A., Elhaddad, T.A., "The power Lomax distribution with an application to bladder cancer data", (2016).
- (28) Devroye, L., Gravel, C., "**Random variate generation using only finitely many unbiased, independently and identically distributed random bits**", arXiv:1502.02539v6 [cs.IT], 2020.
- (29) Knuth, Donald E. and Andrew Chi-Chih Yao. "The complexity of nonuniform random number generation", in *Algorithms and Complexity: New Directions and Recent Results*, 1976.
- (30) Ker-I Ko makes heavy use of the inverse modulus of continuity in his complexity theory, e.g., "Computational complexity of roots of real functions." In *30th Annual Symposium on Foundations of Computer Science*, pp. 204-209. IEEE Computer Society, 1989.
- (31) Here is a sketch of the proof: Because the quantile function $Q(x)$ is continuous on a closed interval, it's uniformly continuous there. For this reason, there is a positive function $\omega^{-1}(\varepsilon)$ such that $Q(x)$ is less than ε-away from $Q(y)$ whenever x is less than $\omega^{-1}(\varepsilon)$-away from y, for every $\varepsilon>0$ and for any x and y in that interval. The inverse modulus of continuity is one such function, which is formed by inverting a modulus of continuity admitted by Q, as long as that modulus is continuous and monotone increasing on that interval to make that modulus invertible. Finally, max(0, ceil($-\ln(z)/\ln(\beta)$)) is an upper bound on the number of base-β fractional digits needed to store $1/z$ with an error of at most ε.
- (32) Morina, G., Łatuszyński, K., et al., "**From the Bernoulli Factory to a Dice Enterprise via Perfect Sampling of Markov Chains**", arXiv:1912.09229v1 [math.PR], 2019.
- (33) Canonne, C., Kamath, G., Steinke, T., "**The Discrete Gaussian for Differential Privacy**",

arXiv:2004.00010 [cs.DS], 2020.

- (34) Karney, C.F.F., "**Sampling exactly from the normal distribution**", arXiv:1303.6257v2 [physics.comp-ph], 2014.
- (35) Chewi, S., Gerber, P., et al., "**Rejection sampling from shape-constrained distributions in sublinear time**", arXiv:2105.14166, 2021

13 License