Miscellaneous Observations on Randomization

This version of the document is dated 2020-12-07.

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. Set *ret* to either $n/2+i$ or $n/2-i-1$ with equal probability.
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.[(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 large, 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 number 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 number with rate 1 (which is the negative natural logarithm of a uniform(0,1) random number). Set e to 0 minus that number.
- (7b.) If e 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 e is greater than the upper bound, or return ret if e 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 $u = 2^{\text{floor}((1+\text{ceil}(\log 2(n+1)))/2)}$, then calculating v = floor(u+floor(n/u)/2), $w = u$, $u = v$ until $v >= w$, then setting m to w + 1.

Notes:

- A binomial(n, 1/2) random number, where n is odd, can be generated by adding an unbiased random bit (zero or one) to a binomial(n−1, 1/2) random number.
- As pointed out by Farach-Colton and Tsai (2015)[6], a binomial(n, p) random number, 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 number to ret and subtract the same random number from n; otherwise, set n to a binomial(n, 1/2) random number.
  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 number 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 number" 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 (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 either 0 or 2^{k-b} with equal probability, then add that number 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 (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 number is a geometric(p) random number 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 number 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 number 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) is supported on the interval [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 number, 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 number 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) distributions of random numbers, with emphasis on sampling random numbers from them. Some of these families are described in Ahmad et al. (2019)**(8)**.

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 supported on the interval [0, 1] and G is a distribution with an easy-to-compute quantile function (also known as inverse cumulative distribution function or inverse CDF). The following algorithm samples a random number following a distribution from this kind of family:

1. Generate a random number 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 the note at the end of this section.)

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 $\alpha > 0$, and the algorithm for the "X-G" families is used, except step 1 now reads: "Generate a uniform(0, 1) random number 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

number U, then set x to $U^{1/a}$."

- The *transmuted-G* family (described, for example, by Tahir and Cordeiro (2016)[12]). The family uses a shape parameter η in the interval [−1, 1]; step 1 is modified to read: "Generate a piecewise linear random number 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 number. Otherwise, generate two uniform(0, 1) random numbers, set x to the higher of the two, then if η is less than 0, set x to $1-x$.)".

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)[13] 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 with the same support as X and meets certain other conditions. The following algorithm samples a random number from this kind of family:

1. Generate a random number 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 the note at the end of this section.)

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

- The "T-R{Y}" family (Aljarrah et al., 2014)[14], 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 is supported on \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)[15].

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)[16].

A family very similar to the "transformed–transformer" family uses a *decreasing* W. When distribution X is supported on \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)[17], as well as the "generalized gamma-G" family, where "generalized gamma" refers to the Stacy distribution (Boshi et al. 2020)[18]).

A *compound distribution* is simply the minimum of N random variables distributed as X, where $N >= 1$ is an integer distributed as the discrete distribution Y (Tahir and Cordeiro 2016)[12]. For example, the "beta-G-geometric" family represents the minimum of N beta-

G random variables, where N is a random number 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 variables 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)[19].

An *inverse X distribution* (or *inverted X distribution*) is generally the distribution of the reciprocal of a random number distributed as X. But an *inverse exponential distribution* (Keller and Kamath 1982)[20] is distributed as $-\theta/\ln(U)$ where $\theta > 0$ and U is a uniform(0, 1) random number.

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 number 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.

Note: This is a note on quantile generation using uniform **partially-sampled random numbers (PSRNs)**.

A uniform PSRN is ultimately a number that lies in an interval [a, b]. Let G be a distribution for which the quantile is wanted, and let $f(.)$ be a function applied to a or b before calculating the quantile. When a random number x is a uniform PSRN, then to implement this quantile calculation (see (Devroye and Gravel 2020)[21]):

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

The disadvantage is that the desired accuracy has to be made known to the algorithm in advance. To generate a quantile to any accuracy (even if the accuracy is not known in advance), a rejection sampling approach is needed, which requires knowing G's probability density function or a function proportional to it, and that the density function must be continuous almost everywhere and bounded from above (see also (Devroye and Gravel 2020)[21]). This involves calculating lower and upper bounds of the quantiles of $f(a)$ and $f(b)$ (the bounds are [*alow*, *ahigh*] and [*blow*, *bhigh*] respectively) and applying an arbitrary-precision rejection sampler such as Oberhoff's method (described in an **appendix to the PSRN article**) to the distribution G limited to the interval [*alow*, *bhigh*] and accepting the resulting PSRN if it clearly lies in [*ahigh*, *blow*] or rejecting it if it clearly lies outside [*alow*, *bhigh*]. When neither of these is the case, then it gets more complicated; more digits of the input or output PSRN have to be generated (uniformly at random) until it's clear whether to accept or reject the output PSRN.

6 Certain Distributions

A *power function(a, c) distribution* is distributed as $c*U^{1/a}$, where U is a uniform(0,1) random number, $a > 0$, and c is a scale parameter greater than 0.

A *right-truncated Weibull(a, b, c) distribution* (truncated at c) is distributed as the minimum of N power function(b, c) random variables, where N is distributed as the zero-truncated Poisson($a*c^b$) distribution. (Jodrá 2020)**(22)**.

A *Lehmann Weibull*($a1$, $a2$, β) random number (Elgohari and Yousof 2020)**(23)** is distributed as $(\ln(1/U)/\beta)^{1/a1}/a2$, where $a1$, $a2$, and β are greater than 0, and U is a uniform(0, 1) random number. Alternatively, distributed as $E^{1/a1}/a2$ where E is an exponential(β) random number.

A *Marshall–Olkin*(α) random number is distributed as $(1-U)/(U*(\alpha-1) + 1)$, where α is in the interval [0, 1], and U is a uniform(0, 1) random number.

7 Batching Random Samples via Randomness Extraction

Devroye and Gravel (2020)**(24)** 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 $P(X \mid Y)$ and from above by $P(X-1 \mid Y)$. (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]. Note, though, that for correctness, 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. (Note 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.)

Now we discuss the issue of finding $P(X \mid Y)$. 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)**(25)**, $P(X \mid Y)$ 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 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) = n!/(k! * (n − k)!) is a binomial coefficient. It can be calculated, for example, by calculating i/($n-i+1$) for each integer i in [$n-k+1$, 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 all m>0, choose(m, 0) = choose(m, m) = 1 and choose(m, 1) = choose(m, $m-1$) = m.
- (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) Tahir, M.H., Cordeiro, G.M., "Compounding of distributions: a survey and new generalized classes", *Journal of Statistical Distributions and Applications* 3(13), 2016.
- (13) Alzaatreh, A., Famoye, F., Lee, C., "A new method for generating families of continuous distributions", *Metron* 71:63–79 (2013).
- (14) 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.
- (15) Gleaton, J.U., Lynch, J. D., "Properties of generalized log-logistic families of lifetime distributions", *Journal of Probability and Statistical Science* 4(1), 2006.
- (16) Hosseini, B., Afshari, M., "The Generalized Odd Gamma-G Family of Distributions: Properties and Application", *Austrian Journal of Statistics* vol. 47, Feb. 2018.
- (17) 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/θ^2) = sqrt(1/λ^2). The first Rayleigh distribution uses θ and the second, λ.
- (18) Boshi, M.A.A., et al., "Generalized Gamma – Generalized Gompertz Distribution", *Journal of Physics: Conference Series* 1591, 012043 (2020).
- (19) Akdoğan, Y., Kus, C., et al., "Geometric-Zero Truncated Poisson Distribution: Properties and Applications", *Gazi University Journal of Science* 32(4), 2019.
- (20) Keller, A.Z., Kamath A.R., "Reliability analysis of CNC machine tools", *Reliability Engineering* 3 (1982).
- (21) Devroye, L., Gravel, C., "**Random variate generation using only finitely many unbiased, independently and identically distributed random bits**", arXiv:1502.02539v6 [cs.IT], 2020.
- (22) Jodrá, P., "A note on the right truncated Weibull distribution and the minimum of power function distributions", 2020.
- (23) Elgohari, Hanaa, and Haitham Yousof. "New Extension of Weibull Distribution: Copula, Mathematical Properties and Data Modeling." Stat., Optim. Inf. Comput., Vol.8, December 2020.
- (24) Devroye, L., Gravel, C., "**Random variate generation using only finitely many unbiased, independently and identically distributed random bits**", arXiv:1502.02539v6 [cs.IT], 2020.
- (25) 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.

9 License