Miscellaneous Observations on Randomization

This version of the document is dated 2020-11-16.

Peter Occil

1 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. Set m to floor(sqrt(n)) + 1.
2. (First, sample from an envelope of the binomial curve.) Generate unbiased random bits (zeros or ones) until a zero is generated this way. Set k to the number of ones generated this way.
3. Set s to an integer in 0, m) chosen uniformly at random, then set i to $k*m + s$.
4. Set *ret* to either $n/2+i$ or $n/2-i-1$ with equal probability.
5. (Second, accept or reject *ret*.) If *ret* < 0 or *ret* > n, go to step 2.
6. With probability choose(n, *ret*)$*m*2^{k-(n+2)}$, return *ret*. Otherwise, go to step 2. (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 6 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 6 can be changed as follows:

- (6.) Let p be $\text{loggamma}(n+1)-\text{loggamma}(k+1)-\text{loggamma}((n-k)+1)+\ln(m)+\ln(2)*k-(n+2)$ (where loggamma(x) is the logarithm of the gamma function).
- (6a.) 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.
- (6b.) If e is greater than p, go to step 2. Otherwise, return *ret*. (This step can be replaced by calculating lower and upper bounds that converge to p. In that case, go to step 2 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.

Note that 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. Note also that 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.)

2 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 $d*2^k+m$. (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*, $d*2^k+m$).

3 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.

4 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.

The "odd X Y" family uses two distributions, X and Y, where X is an arbitrary continuous distribution and Y is a distribution with an easy-to-sample quantile function (also known as inverse cumulative distribution function or inverse CDF). The following algorithm samples a random number following a distribution from this family:

1. Generate a random number that follows the distribution X, call it x.
2. Calculate the quantile for Y of $x/(1+x)$, and return that quantile.

Examples of this family include the "odd log-logistic G" family (where "G" or "generated" corresponds to Y) (Gleaton and Lynch 2006)**(8)** and the "generalized odd Weibull generated" family (where X is the Weibull distribution and Y is arbitrary) (Korkmaz et al. 2018)**(9)**. Many special cases of this family have been proposed in many papers, and usually their names suggest the distributions that make up this family. Some of these members have names that begin with the word "generalized", and in most such cases the quantile in step 2 should be calculated as $(x/(1+x))^{1/a}$, where a is a shape parameter greater than 0; an example is the "generalized odd gamma-G" family (Hosseini et al. 2018)**(10)**.

A *compound distribution* is simply the minimum of N random variables distributed as X, where N is distributed as the discrete distribution Y (Tahir and Cordeiro 2016)**(11)**. For example, the "beta-G-geometric" family represents the minimum of N beta-G random variables (beta-G is the quantile of a beta-distributed random number, where the quantile comes from an arbitrary distribution (Eugene et al., 2002)**(12)**), 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 is 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)**(13)**. An *inverse X distribution* (or *inverted X distribution*) is the distribution of the reciprocal of a random number distributed as X.

5 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)**(14)**.

6 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] Gleaton, J.U., Lynch, J. D., "Properties of generalized log-logistic families of lifetime distributions", *Journal of Probability and Statistical Science* 4(1), 2006.
- [9] Korkmaz, M.Ç., Alizadeh, M., et al., "The Generalized Odd Weibull Generated Family of Distributions: Statistical Properties and Applications", *Pak. J. Stat. Oper. Res.* XIV(3), 2018.
- [10] Hosseini, B., Afshari, M., "The Generalized Odd Gamma-G Family of Distributions: Properties and Application", *Austrian Journal of Statistics* vol. 47, Feb. 2018.
- [11] Tahir, M.H., Cordeiro, G.M., "Compounding of distributions: a survey and new generalized classes", *Journal of Statistical Distributions and Applications* 3(13), 2016.
- [12] Eugene, N., Lee, C., Famoye, F., "Beta-normal distribution and its applications", *Commun. Stat. Theory Methods* 31, 2002.
- [13] Akdoğan, Y., Kus, C., et al., "Geometric-Zero Truncated Poisson Distribution: Properties and Applications", *Gazi University Journal of Science* 32(4), 2019.
- [14] Jodrá, P., "A note on the right truncated Weibull distribution and the minimum of power function distributions", 2020.

7 License