Partially-Sampled Random Numbers for Accurate Sampling of the Beta, Exponential, and Other Continuous Distributions

This version of the document is dated 2020-08-13.

Peter Occil

Note: Formerly "Partially Sampled Exponential Random Numbers", due to a merger with "An Exact Beta Generator".

Introduction

This page introduces a Python implementation of *partially-sampled random numbers* (PSRNs). Although structures for PSRNs were largely described before this work, this document unifies the concepts for these kinds of numbers from prior works and shows how they can be used to sample the beta distribution (for most sets of parameters), the exponential distribution (with an arbitrary rate parameter), and other continuous distributions—

- while avoiding floating-point arithmetic, and
- to an arbitrary precision and with user-specified error bounds (and thus in an "exact" manner in the sense defined in (Karney 2014)**(1)**).

For instance, these two points distinguish the beta sampler in this document from any other specially-designed beta sampler I am aware of. As for the exponential distribution, there are papers that discuss generating exponential random numbers using random bits (Flajolet and Saheb 1982)**(2)**, (Karney 2014)**(1)**, (Devroye and Gravel 2015)**(3)**, (Thomas and Luk 2008)**(4)**, but almost all of them that I am aware of don't deal with generating exponential PSRNs using an arbitrary rate, not just 1. (Habibizad Navin et al., 2010)**(5)**, which came to my attention on the afternoon of July 20, after I wrote much of this article, is perhaps an exception; however the approach appears to involve pregenerated tables of digit probabilities.

The samplers discussed here also draw on work dealing with a construct called the *Bernoulli factory* (Keane and O'Brien 1994)**(6)** (Flajolet et al., 2010)**(7)**, which can simulate an arbitrary probability by transforming biased coins to biased coins. One important feature of Bernoulli factories is that they can simulate a given probability *exactly*, without having to calculate that probability manually, which is important if the probability can be an irrational number that no computer can compute exactly (such as `pow(p, 1/2)` or `exp(-2)`).

This page shows **Python code** for these samplers.

About This Document

This is an open-source document; for an updated version, see the source code or its rendering on GitHub. You can send comments on this document either on CodeProject or on the GitHub issues page.

Contents

- **Introduction**
    - **About This Document**
- **Contents**
- **About the Beta Distribution**
- **About the Exponential Distribution**
- **About Partially-Sampled Random Numbers**
    - **Uniform Partially-Sampled Random Numbers**
    - **Exponential Partially-Sampled Random Numbers**
    - **Other Distributions**
    - **Properties**
    - **Comparisons**
    - **Arithmetic**
- **Building Blocks**
- **Algorithms for the Beta and Exponential Distributions**
    - **Beta Distribution**
    - **Exponential Distribution**
    - **Power-of-Uniform Sub-Algorithm**
- **Sampler Code**
    - **Beta Sampler: Known Issues**
    - **Exponential Sampler: Extension**
- **Correctness Testing**
    - **Beta Sampler**
    - **ExpRandFill**
    - **ExpRandLess**
- Accurate Simulation of Continuous Distributions on [0, 1]
    - **An Example: The Continuous Bernoulli Distribution**
- **Complexity**
    - **General Principles**
    - **Complexity of Specific Algorithms**
- **Application to Weighted Reservoir Sampling**
- **Open Questions**
- **Acknowledgments**
- **Other Documents**
- **Notes**
- **Appendix**
    - **SymPy Formula for the algorithm for exp(−*x*/*y*)**
    - **Another Example of an Arbitrary-Precision Sampler**
- **License**

About the Beta Distribution

The **beta distribution** is a bounded-domain probability distribution; its two parameters, `alpha` and `beta`, are both greater than 0 and describe the distribution's shape. Depending on `alpha` and `beta`, the shape can be a smooth peak or a smooth valley. The beta distribution can take on values in the interval [0, 1]. Any value in this interval (`x`) can occur with a probability proportional to—

```
pow(x, alpha - 1) * pow(1 - x, beta - 1).               (1)
```

Although `alpha` and `beta` can each be greater than 0, the sampler presented in this document only works if—

- both parameters are 1 or greater, or
- in the case of base-2 numbers, one parameter equals 1 and the other is greater than 0.

About the Exponential Distribution

The *exponential distribution* takes a parameter λ. Informally speaking, a random number that follows an exponential distribution is the number of units of time between one event and the next, and λ is the expected average number of events per unit of time. Usually, λ is equal to 1.

An exponential random number is commonly generated as follows: `-ln(1 - RNDU01()) / lamda`, where `RNDU01()` is a uniform random number in the interval \0, 1). (This particular formula is not robust, though, for reasons that are outside the scope of this document, but see (Pedersen 2018)[**(8)**.) This page presents an alternative way to sample exponential random numbers.

About Partially-Sampled Random Numbers

In this document, a *partially-sampled random number* (PSRN) is a data structure that allows a random number that exactly follows a continuous distribution to be sampled digit by digit, with arbitrary precision, and without floating-point arithmetic (see “Properties” later in this section). Informally, they represent incomplete real numbers whose contents are sampled only when necessary, but in a way that follows the distribution being sampled.

This section specifies two kinds of PSRNs: uniform and exponential.

Uniform Partially-Sampled Random Numbers

The most trivial example of a PSRN is that of the uniform distribution in [0, 1]. Such a random number can be implemented as a list of items, where each item is either a digit (such as zero or one for binary), or a placeholder value (which represents an unsampled digit), and represents a list of the digits after the radix point, from left to right, of a real number in the interval [0, 1], that is, the number’s *digit expansion* (e.g., *binary expansion* in the case of binary digits). This kind of number is referred to —

- as a *geometric bag* in (Flajolet et al., 2010)**(7)** (but only in the binary case), and
- as a *u-rand* in (Karney 2014)**(1)**.

Each additional digit is sampled simply by setting it to an independent unbiased random digit, an observation that dates from von Neumann (1951)**(9)** in the binary case.

Note that the *u-rand* concept by Karney only contemplates sampling digits from left to right without any gaps, whereas the geometric bag concept is more general in this respect.

Exponential Partially-Sampled Random Numbers

In this document, a exponential PSRN (or *e-rand*, named similarly to Karney’s “u-rands” for partially-sampled uniform random numbers (Karney 2014)**(1)**) samples each bit that, when combined with the existing bits, results in an exponentially-distributed random number of the given rate. Also, because `-ln(1 - RNDU01())` is exponentially distributed, e-rands can also represent the natural logarithm of a partially-sampled uniform random number in (0, 1]. The difference here is that additional bits are sampled not as unbiased random bits, but rather as bits with a vanishing bias.

Algorithms for sampling e-rands are given in the section “Algorithms for the Beta and Exponential Distributions”.

Other Distributions

Partially-sampled numbers of other distributions can be implemented via rejection from the uniform distribution. Examples include the following:

- The beta and continuous Bernoulli distributions, as discussed later in this document.
- The standard normal distribution, as shown in (Karney 2014)**(1)** by running Karney's Algorithm N and filling unsampled digits uniformly at random.
- For uniform distributions in [0, *n*) (not just [0, 1]), a partially-sampled version might be trivial by first ensuring that the first "few" digits are such that the resulting number will be less than *n*, via rejection sampling.

For these distributions (and others that are continuous almost everywhere and bounded from above), Oberhoff (2018)**(10)** proved that unsampled trailing bits of the partially-sampled number converge to the uniform distribution.

Partially-sampled numbers could also be implemented via rejection from the exponential distribution, although no concrete examples are presented here.

Properties

An algorithm that samples from a continuous distribution using PSRNs has the following properties:

1. The algorithm relies only on a source of random bits for randomness, and does not rely on floating-point arithmetic or calculations of irrational or transcendental numbers (other than digit extractions), including when the algorithm samples each digit of a PSRN. The algorithm may use rational arithmetic (such as `Fraction` in Python or `Rational` in Ruby), as long as the arithmetic is exact.
2. If the algorithm outputs a PSRN, the number represented by the sampled digits must follow a distribution that is close to the ideal distribution by a distance of not more than b^{-m}, where *b* is the PSRN's base, or radix (such as 2 for binary), and *m* is the number of sampled digits in the PSRN's fractional part. ((Devroye and Gravel 2015)**(3)** suggests Wasserstein L_∞ distance as the distance to use for this purpose.) The number has to be close this way even if the algorithm's caller later samples unsampled digits of that PSRN at random (e.g., uniformly at random in the case of a uniform PSRN).
3. If the algorithm fills a PSRN's unsampled fractional digits at random (e.g., uniformly at random in the case of a uniform PSRN), so that the number's fractional part has *m* digits, the number's distribution must remain close to the ideal distribution by a distance of not more than b^{-m}.

The concept of *prefix distributions* (Oberhoff 2018)**(10)** comes close to PSRNs, but numbers sampled this way are not PSRNs in the sense used here. This is because the method requires calculating minimums of probabilities and, in practice, requires the use of floating-point arithmetic in most cases (see property 1 above). Moreover, the method samples from a discrete distribution whose progression depends on the value of previously sampled bits, not just on the position of those bits as with the uniform and exponential distributions (see also (Thomas and Luk 2008)**(4)**).

Comparisons

Two PSRNs, each of a different distribution but storing digits of the same base (radix), can be exactly compared to each other using an algorithm similar to the following. The **RandLess** algorithm compares two PSRNs, **a** and **b** (and samples additional bits from them as necessary) and returns `true` if **a** turns out to be less than **b**, or `false` otherwise (see also (Karney 2014)**(1)**)).

1. If **a**'s integer part wasn't sampled yet, sample **a**'s integer part. Do the same for

b.

2. Return `true` if **a**’s integer part is less than **b**’s, or `false` if **a**’s integer part is greater than **b**’s.
3. Set *i* to 0.
4. If **a**’s fractional part has *i* or fewer digits, sample digit *i* of **a** (positions start at 0 where 0 is the most significant digit after the point, 1 is the next, etc.), and append the result to that fractional part’s digit expansion. Do the same for **b**.
5. Return `true` if **a**’s fractional part is less than **b**’s, or `false` if **a**’s fractional part is greater than **b**’s.
6. Add 1 to *i* and go to step 4.

URandLess is a version of **RandLess** that involves two uniform PSRNs. The algorithm for **URandLess**, digits samples digit *i* in step 4 by setting the digit at position *i* to a digit chosen uniformly at random.

Arithmetic

Arithmetic between two PSRNs is not exactly trivial. The naïve approach of adding, multiplying or dividing two PSRNs A and B (see also (Brassard et al., 2019)[(11)]) may result in a partially-sampled number C that is not close to the ideal distribution once additional digits of C are sampled uniformly at random (see properties 2 and 3 above).

On the other hand, partially-sampled-number arithmetic may be possible by relating the relative probabilities of each digit, in the result’s digit expansion, to some kind of formula. (This ignores trivial arithmetic operations, such as addition by half provided the base (radix) is even, or negation — both operations mentioned in (Karney 2014)[(1)] — or operations affecting the integer part only.) For example, I can show empirically that when an exponential(1) random number is multiplied by a uniform [0, 1] random number, the following (approximate) probabilities of 1 occur in the following positions of the result’s binary expansion:

Position after the point	**Approx. prob. of 1**
1st (half)	0.227233
2nd (quarter)	0.321432
3rd	0.388065
4th	0.433957
5th	0.461612

There is previous work that relates continuous distributions to digit probabilities in a similar manner (but only in base 10) (Habibizad Navin et al., 2007)[(12)], (Nezhad et al., 2013)[(13)].

Finally, arithmetic with partially-sampled numbers may be possible if the result of the arithmetic is distributed with a known density function (e.g., one found via Rohatgi’s formula (Rohatgi 1976)[(14)]), allowing for an algorithm that implements rejection from the uniform or exponential distribution. However, that density function may have an unbounded peak, thus ruling out rejection sampling in practice. For example, if X is a uniform PSRN, then X^3 is distributed as `(1/3) / pow(X, 2/3)`, which has an unbounded peak at 0. While this rules out plain rejection samplers for X^3 in practice, it’s still possible to sample powers of uniforms using PSRNs, which will be described later in this article.

Building Blocks

This document relies on several building blocks described in this section.

One of them is the “geometric bag” technique by Flajolet and others (2010)[(7)], which generates heads or tails with a probability that is built up digit by digit. A *geometric bag* was defined earlier.

The algorithm **SampleGeometricBag** is a Bernoulli factory algorithm. For base 2, the algorithm is described as follows (see (Flajolet et al., 2010)**(7)**):

1. Set N to 0.
2. With probability 1/2, go to the next step. Otherwise, add 1 to N and repeat this step.
3. If the item at position N in the geometric bag (positions start at 0) is not set to a digit (e.g., 0 or 1 for base 2), set the item at that position to a digit chosen uniformly at random (e.g., either 0 or 1 for base 2), increasing the geometric bag's capacity as necessary. (As a result of this step, there may be "gaps" in the geometric bag where no digit was sampled yet.)
4. Return the item at position N.

For another base (radix), such as 10 for decimal, this can be implemented as **URandLess**, with **a** being an empty uniform PSRN and **b** being the geometric bag. Return 1 if the algorithm returns `true`, or 0 otherwise.

SampleGeometricBagComplement is the same as the **SampleGeometricBag** algorithm, except the return value is 1 minus the original return value. The result is that if **SampleGeometricBag** outputs 1 with probability U, **SampleGeometricBagComplement** outputs 1 with probability $1 - U$.

FillGeometricBag takes a geometric bag and generates a number whose fractional part has `p` digits as follows:

1. For each position in \0, `p`), if the item at that position is not a digit, set the item there to to a digit chosen uniformly at random (e.g., either 0 or 1 for binary), increasing the geometric bag's capacity as necessary. (See also (Oberhoff 2018, sec. 8)[**(10)**.)
2. Take the first `p` digits of the geometric bag and return $\Sigma_{i=0, ..., \texttt{p}-1}$ bag[i] * b^{-i-1}, where b is the base, or radix. (If it somehow happens that digits beyond `p` are set to 0 or 1, then the implementation could choose instead to fill all unsampled digits between the first and the last set digit and return the full number, optionally rounding it to a number whose fractional part has `p` digits, with a rounding mode of choice.)

The **kthsmallest** method generates the 'k'th smallest 'bitcount'-digit uniform random number out of 'n' of them, is also relied on by this beta sampler. It is used when both `a` and `b` are integers, based on the known property that a beta random variable in this case is the `a`th smallest uniform (0, 1) random number out of `a + b - 1` of them (Devroye 1986, p. 431)**(15)**.

kthsmallest, however, doesn't simply generate 'n' 'bitcount'-digit numbers and then sort them. Rather, it builds up their digit expansions digit by digit, via PSRNs. It uses the observation that (in the binary case) each uniform (0, 1) random number is equally likely to be less than half or greater than half; thus, the number of uniform numbers that are less than half vs. greater than half follows a binomial(n, 1/2) distribution (and of the numbers less than half, say, the less-than-one-quarter vs. greater-than-one-quarter numbers follows the same distribution, and so on). Thanks to this observation, the algorithm can generate a sorted sample "on the fly". A similar observation applies to other bases than base 2 if we use the multinomial distribution instead of the binomial distribution. I am not aware of any other article or paper (besides one by me) that describes the **kthsmallest** algorithm given here.

The algorithm is as follows:

1. Create `n` empty PSRNs.

2. Set `index` to 1.
3. If `index <= k` and `index + n >= k`:
    1. Generate **v**, a multinomial random vector with b probabilities equal to $1/b$, where b is the base, or radix (for the binary case, b = 2, so this is equivalent to generating `LC` = binomial(`n`, 0.5) and setting **v** to {`LC`, `n - LC`}).
    2. Starting at `index`, append the digit 0 to the first **v**[0] partially-sampled numbers, a 1 digit to the next **v**[1] partially-sampled numbers, and so on to appending a $b - 1$ digit to the last **v**[$b - 1$] partially-sampled numbers (for the binary case, this means appending a 0 bit to the first `LC` u-rands and a 1 bit to the next `n - LC` u-rands).
    3. For each integer i in [0, b): If **v**[i] > 1, repeat step 3 and these substeps with `index` = `index` + $\Sigma_{j=0, ..., i-1}$ **v**[j] and `n` = **v**[i]. (For the binary case, this means: If `LC > 1`, repeat step 3 and these substeps with the same `index` and `n = LC`; then, if `n - LC > 1`, repeat step 3 and these substeps with `index = index + LC`, and `n = n - LC`).
4. Take the `k`th PSRN (starting at 1) and fill it with uniform random digits as necessary to give its fractional part `bitcount` many digits (similarly to **FillGeometricBag** above). Return that number. (An implementation may instead just return the PSRN without filling it this way first, but the beta sampler described later doesn't use this alternative.)

Sampling an e-rand (a exponential PSRN) makes use of two observations (based on the parameter λ of the exponential distribution):

- While a coin flip with probability of heads of exp(-λ) is heads, the exponential random number is increased by 1.
- If a coin flip with probability of heads of $1/(1+\exp(\lambda/2^k))$ is heads, the exponential random number is increased by 2^{-k}, where $k > 0$ is an integer.

(Devroye and Gravel 2015)**[3]** already made these observations in their Appendix, but only for λ = 1.

To implement these probabilities using just random bits, the sampler uses two algorithms:

1. One to simulate a probability of the form `exp(-x/y)` (here, the **algorithm for exp(−*x*/*y*)** described in "**Bernoulli Factory Algorithms**").
2. One to simulate a probability of the form `1/(1+exp(x/(y*pow(2, prec))))` (here, the **LogisticExp** algorithm described in "**Bernoulli Factory Algorithms**").

These two algorithms enable e-rands with rational-valued λ parameters.

Algorithms for the Beta and Exponential Distributions

Beta Distribution

All the building blocks are now in place to describe a *new* algorithm to sample the beta distribution, described as follows. It takes three parameters: a >= 1 and b >= 1 (or one parameter is 1 and the other is greater than 0 in the binary case) are the parameters to the beta distribution, and $p > 0$ is a precision parameter.

1. Special cases:
    - If a = 1 and b = 1, return a uniform random number whose fractional part has p digits (for example, in the binary case, RandomBits(p) / 2^p where `RandomBits(x)` returns an x-bit block of unbiased random bits).

    - If a and b are both integers, return the result of **kthsmallest** with `n = a - b + 1` and `k = a`, and fill it as necessary to give the number a p-digit fractional part (similarly to **FillGeometricBag** above).
    - In the binary case, if a is 1 and b is less than 1, return the result of the **power-of-uniform sub-algorithm** described below, with $px/py = 1/b$, and the *complement* flag set to `true`.
    - In the binary case, if b is 1 and a is less than 1, return the result of the **power-of-uniform sub-algorithm** described below, with $px/py = 1/a$, and the *complement* flag set to `false`.
2. Create an empty list to serve as a “geometric bag”. Create an input coin *geobag* that returns the result of **SampleGeometricBag** using the given geometric bag. Create another input coin *geobagcomp* that returns the result of **SampleGeometricBagComplement** using the given geometric bag.
3. Remove all digits from the geometric bag. This will result in an empty uniform random number, U, for the following steps, which will accept U with probability $U^{a-1}*(1-U)^{b-1}$) (the proportional probability for the beta distribution), as U is built up.
4. Call the **algorithm for $\lambda^{x/y}$**, described in “Bernoulli Factory Algorithms”, using the *geobag* input coin and $x/y = a - 1)/1$ (thus returning with probability U^{a-1}). If the result is 0, go to step 3.
5. Call the same algorithm using the *geobagcomp* input coin and $x/y = (b - 1)/1$ (thus returning 1 with probability $(1-U)^{b-1}$). If the result is 0, go to step 3. (Note that steps 4 and 5 don’t depend on each other and can be done in either order without affecting correctness, and this is taken advantage of in the Python code below.)
6. U was accepted, so return the result of **FillGeometricBag**.

Note that a beta($1/x$, 1) random number is the same as a uniform random number raised to the power of x.

Exponential Distribution

We also have the necessary building blocks to describe how to sample e-rands. As implemented in the Python code, an e-rand consists of five numbers: the first is a multiple of $1/(2^x)$, the second is x, the third is the integer part (initially −1 to indicate the integer part wasn’t sampled yet), and the fourth and fifth are the λ parameter’s numerator and denominator, respectively.

To sample bit k after the binary point of an exponential random number with rate λ (where $k = 1$ means the first digit after the point, $k = 2$ means the second, etc.), call the **LogisticExp** algorithm with x = λ’s numerator, y = λ’s denominator, and *prec* = k.

The **ExpRandLess** algorithm is a special case of the general **RandLess** algorithm given earlier. It compares two e-rands **a** and **b** (and samples additional bits from them as necessary) and returns `true` if **a** turns out to be less than **b**, or `false` otherwise. (Note that **a** and **b** are allowed to have different λ parameters.)

1. If **a**’s integer part wasn’t sampled yet, call the **algorithm for exp(−x/y)** with x = λ’s numerator and y = λ’s denominator, until the call returns 0, then set the integer part to the number of times 1 was returned this way. Do the same for **b**.
2. Return `true` if **a**’s integer part is less than **b**’s, or `false` if **a**’s integer part is greater than **b**’s.
3. Set i to 0.
4. If **a**’s fractional part has i or fewer bits, call the **LogisticExp** algorithm with x = λ’s numerator, y = λ’s denominator, and *prec* = $i + 1$, and append the result to that fractional part’s binary expansion. Do the same for **b**.
5. Return `true` if **a**’s fractional part is less than **b**’s, or `false` if **a**’s fractional part is greater than **b**’s.

6. Add 1 to *i* and go to step 4.

The **ExpRandFill** algorithm takes an e-rand **a** and generates a number whose fractional part has `p` bits as follows:

1. If **a**'s integer part wasn't sampled yet, sample it as given in step 1 of **ExpRandLess**.
2. If **a**'s fractional part has greater than `p` bits, round **a** to a number whose fractional part has `p` bits, and return that number. The rounding can be done, for example, by discarding all bits beyond `p` bits after the place to be rounded, or by rounding to the nearest 2^{-p}, ties-to-up, as done in the sample Python code.
3. While **a**'s fractional part has fewer than `p` bits, call the **LogisticExp** algorithm with x = λ's numerator, y = λ's denominator, and *prec* = *i*, where *i* is 1 plus the number of bits in **a**'s fractional part, and append the result to that fractional part's binary expansion.
4. Return the number represented by **a**.

Power-of-Uniform Sub-Algorithm

The power-of-uniform sub-algorithm is used for certain cases of the beta sampler above. It returns $U^{px/py}$, where U is a uniform random number in the interval [0, 1] and px/py is greater than 1, but unlike the naïve algorithm it supports an arbitrary precision, uses only random bits, and avoids floating-point arithmetic. It also uses a *complement* flag to determine whether to return 1 minus the result.

It makes use of a number of algorithms as follows:

- It uses an algorithm for **sampling unbounded monotone density functions**, which in turn is similar to the inversion-rejection algorithm in (Devroye 1986, ch. 7, sec. 4.4)**(15)**. This is needed because when px/py is greater than 1, $U^{px/py}$ is distributed as `(py/px) / pow(U, 1-py/px)`, which has an unbounded peak at 0.
- It uses a number of Bernoulli factory algorithms, including **SampleGeometricBag** and some algorithms described in "**Bernoulli Factory Algorithms**".

However, this algorithm supports only base 2.

The power-of-uniform algorithm is as follows:

1. Set *i* to 1.
2. Call the **algorithm for $(a/b)^{x/y}$** described in "**Bernoulli Factory Algorithms**", with parameters `a = 1, b = 2, x = py, y = px`. If the call returns 1 and *i* is less than *n*, add 1 to *i* and repeat this step. If the call returns 1 and *i* is *n* or greater, return 1 if the *complement* flag is `true` or 0 otherwise (or return a geometric bag filled with exactly *n* ones or zeros, respectively).
3. As a result, we will now sample a number in the interval $[2^{-i}, 2^{-(i-1)})$. We now have to generate a uniform random number X in this interval, then accept it with probability $(py / (px * 2^i)) / X^{1 - py/px}$; the 2^i in this formula is to help avoid very low probabilities for sampling purposes. The following steps will achieve this without having to use floating-point arithmetic.
4. Create an empty list to serve as a geometric bag, then create a *geobag* input coin that returns the result of **SampleGeometricBag** on that geometric bag.
5. Create a *powerbag* input coin that does the following: "Call the **algorithm for $\lambda^{x/y}$**, described in '**Bernoulli Factory Algorithms**', using the *geobag* input coin and with $x/y = 1 - py / px$, and return the result."
6. Append $i - 1$ zero-digits followed by a single one-digit to the geometric bag. This will allow us to sample a uniform random number limited to the interval mentioned earlier.
7. Call the **algorithm for ϵ / λ**, described in "**Bernoulli Factory Algorithms**",

using the *powerbag* input coin (which represents *b*) and with ϵ = *py*/(*px* * 2^i) (which represents *a*), thus returning 1 with probability *a*/*b*. If the call returns 1, the geometric bag was accepted, so do the following:
    1. If the *complement* flag is `true`, make each zero-digit in the geometric bag a one-digit and vice versa.
    2. Either return the geometric bag as is or fill the unsampled digits of the bag with uniform random digits as necessary to give the number an *n*-digit fractional part (similarly to **FillGeometricBag** above), where *n* is a precision parameter, then return the resulting number.
8. If the call to the algorithm for ϵ / λ returns 0, remove all but the first *i* digits from the geometric bag, then go to step 7.

Sampler Code

The following Python code implements the beta sampler just described. It relies on two Python modules I wrote:

- "**bernoulli.py**", which collects a number of Bernoulli factories, some of which are relied on by the code below.
- "**randomgen.py**", which collects a number of random number generation methods, including `kthsmallest`, as well as the `RandomGen` class.

Note that the code uses floating-point arithmetic only to convert the result of the sampler to a convenient form, namely a floating-point number.

This code is far from fast, though, at least in Python.

```
import math
import random
import bernoulli
from randomgen import RandomGen
from fractions import Fraction

def _toreal(ret, precision):
        # NOTE: Although we convert to a floating-point
        # number here, this is not strictly necessary and
        # is merely for convenience.
        return ret*1.0/(1<<precision)

def _power_of_uniform_greaterthan1(bern, power, complement=False, precision=53):
   if power<1:
     raise ValueError("Not supported")
   if power==1:
     bag=[]
     return bern.fill_geometric_bag(bag, precision)
   i=1
   powerfrac=Fraction(power)
   powerrest=Fraction(1) - Fraction(1)/powerfrac
   # Choose an interval
   while bern.zero_or_one_power_ratio(1,2,
         powerfrac.denominator,powerfrac.numerator) == 1:
      if i>=precision:
          # Precision limit reached, so equivalent to endpoint
         return 1.0 if complement else 0.0
      i+=1
   epsdividend = Fraction(1)/(powerfrac * 2**i)
   # -- A choice for epsdividend which makes eps_div
   # -- much faster, but this will require floating-point arithmetic
   # -- to calculate "**powerrest", which is not the focus
   # -- of this article.
   # probx=((2.0**(-i-1))**powerrest)
   # epsdividend=Fraction(probx)*255/256
   bag=[]
   gb=lambda: bern.geometric_bag(bag)
   bf =lambda: bern.power(gb, powerrest.numerator, powerrest.denominator)
   while True:
      # Limit sampling to the chosen interval
      bag.clear()
      for k in range(i-1):
         bag.append(0)
      bag.append(1)
      # Simulate epsdividend / x**(1-1/power)
      if bern.eps_div(bf, epsdividend) == 1:
          # Flip all bits if complement is true
          bag=[x if x==None else 1-x for x in bag] if complement else bag
          ret=bern.fill_geometric_bag(bag, precision)
          return ret

def powerOfUniform(b, px, py, precision=53):
        # Special case of beta, returning power of px/py
        # of a uniform random number, provided px/py
        # is in (0, 1].
        return betadist(b, py, px, 1, 1, precision)

def betadist(b, ax, ay, bx, by, precision=53):
        # Beta distribution for alpha>=1 and beta>=1
        bag=[]
        bpower=Fraction(bx, by)-1
        apower=Fraction(ax, ay)-1
        # Special case for a=b=1
        if bpower==0 and apower==0:
           return _toreal(random.randint(0, (1<<precision)-1), 1<<precision)
        # Special case if a=1
        if apower==0 and bpower<0:
           return _power_of_uniform_greaterthan1(b, Fraction(by, bx), True, precision)
        # Special case if b=1
        if bpower==0 and apower<0:
           return _power_of_uniform_greaterthan1(b, Fraction(ay, ax), False, precision)
        # Special case if a and b are integers
        if int(bpower) == bpower and int(apower) == apower:
           a=int(Fraction(ax, ay))
           b=int(Fraction(bx, by))
           return _toreal(RandomGen().kthsmallest(a+b-1,a, \
                  precision), precision)
        if apower<=-1 or bpower<=-1: raise ValueError
        # Create a "geometric bag" to hold a uniform random
        # number (U), described by Flajolet et al. 2010
        gb=lambda: b.geometric_bag(bag)
        # Complement of "geometric bag"
        gbcomp=lambda: b.geometric_bag(bag)^1
        bPowerBigger=(bpower > apower)
        while True:
           # Create a uniform random number (U) bit-by-bit, and
           # accept it with probability U^(a-1)*(1-U)^(b-1), which
           # is the unnormalized PDF of the beta distribution
           bag.clear()
           r=1
           if bPowerBigger:
             # Produce 1 with probability (1-U)^(b-1)
             r=b.power(gbcomp, bpower)
             # Produce 1 with probability U^(a-1)
             if r==1: r=b.power(gb, apower)
           else:
             # Produce 1 with probability U^(a-1)
             r=b.power(gb, apower)
             # Produce 1 with probability (1-U)^(b-1)
             if r==1: r=b.power(gbcomp, bpower)
           if r == 1:
                 # Accepted, so fill up the "bag" and return the
                 # uniform number
                 ret=_fill_geometric_bag(b, bag, precision)
                 return ret
```

```
def _fill_geometric_bag(b, bag, precision):
        ret=0
        lb=min(len(bag), precision)
        for i in range(lb):
           if i>=len(bag) or bag[i]==None:
              ret=(ret<<1)|b.randbit()
           else:
              ret=(ret<<1)|bag[i]
        if len(bag) < precision:
           diff=precision-len(bag)
           ret=(ret << diff)|random.randint(0,(1 << diff)-1)
        # Now we have a number that is a multiple of
        # 2^-precision.
        return _toreal(ret, precision)
```

The following Python code implements the exponential sampler described earlier. In the Python code below, note that `zero_or_one` uses `random.randint` which does not necessarily use only random bits, even though it's called only to return either zero or one.

```
import random

def logisticexp(ln, ld, prec):
        """ Returns 1 with probability 1/(1+exp(ln/(ld*2^prec))). """
        denom=ld*2**prec
        while True:
           if zero_or_one(1, 2)==0: return 0
           if zero_or_one_exp_minus(ln, denom) == 1: return 1

def exprandnew(lamdanum=1, lamdaden=1):
     """ Returns an object to serve as a partially-sampled
          exponential random number with the given
          rate 'lamdanum'/'lamdaden'.  The object is a list of five numbers
          as given in the prose.  Default for 'lamdanum'
          and 'lamdaden' is 1.
          The number created by this method will be "empty"
          (no bits sampled yet).
          """
     return [0, 0, -1, lamdanum, lamdaden]

def exprandfill(a, bits):
    """ Fills the unsampled bits of the given exponential random number
           'a' as necessary to make a number whose fractional part
           has 'bits' many bits.  If the number's fractional part already has
           that many bits or more, the number is rounded using the round-to-nearest,
           ties to even rounding rule.  Returns the resulting number as a
           multiple of 2^'bits'. """
    # Fill the integer if necessary.
    if a[2]==-1:
        a[2]=0
        while zero_or_one_exp_minus(a[3], a[4]) == 1:
            a[2]+=1
    if a[1] > bits:
        # Shifting bits beyond the first excess bit.
        aa = a[0] >> (a[1] - bits - 1)
        # Check the excess bit; if odd, round up.
        ret=aa >> 1 if (aa & 1) == 0 else (aa >> 1) + 1
        return ret|(a[2]<<bits)
    # Fill the fractional part if necessary.
    while a[1] < bits:
       index = a[1]
       a[1]+=1
       a[0]=(a[0]<<1)|logisticexp(a[3], a[4], index+1)
    return a[0]|(a[2]<<bits)

def exprandless(a, b):
        """ Determines whether one partially-sampled exponential number
           is less than another; returns
           true if so and false otherwise.  During
           the comparison, additional bits will be sampled in both numbers
           if necessary for the comparison. """
        # Check integer part of exponentials
        if a[2] == -1:
            a[2] = 0
            while zero_or_one_exp_minus(a[3], a[4]) == 1:
                a[2] += 1
        if b[2] == -1:
            b[2] = 0
            while zero_or_one_exp_minus(b[3], b[4]) == 1:
                b[2] += 1
        if a[2] < b[2]:
            return True
        if a[2] > b[2]:
            return False
        index = 0
        while True:
            # Fill with next bit in a's exponential number
            if a[1] < index:
                raise ValueError
            if b[1] < index:
                raise ValueError
            if a[1] <= index:
                a[1] += 1
                a[0] = logisticexp(a[3], a[4], index + 1) | (a[0] << 1)
            # Fill with next bit in b's exponential number
            if b[1] <= index:
                b[1] += 1
                b[0] = logisticexp(b[3], b[4], index + 1) | (b[0] << 1)
            aa = (a[0] >> (a[1] - 1 - index)) & 1
            bb = (b[0] >> (b[1] - 1 - index)) & 1
            if aa < bb:
                return True
            if aa > bb:
                return False
            index += 1

def zero_or_one(px, py):
        """ Returns 1 at probability px/py, 0 otherwise.
            Uses Bernoulli algorithm from Lumbroso appendix B,
            with one exception noted in this code. """
        if py <= 0:
            raise ValueError
        if px == py:
            return 1
        z = px
        while True:
            z = z * 2
            if z >= py:
                if random.randint(0,1) == 0:
                    return 1
                z = z - py
            # Exception: Condition added to help save bits
            elif z == 0: return 0
            else:
                if random.randint(0,1) == 0:
                   return 0

def zero_or_one_exp_minus(x, y):
        """ Generates 1 with probability exp(-px/py); 0 otherwise.
               Reference: Canonne et al. 2020. """
        if y <= 0 or x < 0:
            raise ValueError
        if x==0: return 1
        if x > y:
            xf = int(x / y)  # Get integer part
            x = x % y  # Reduce to fraction
            if x > 0 and zero_or_one_exp_minus(x, y) == 0:
                return 0
            for i in range(xf):
                if zero_or_one_exp_minus(1, 1) == 0:
                    return 0
            return 1
        r = 1
        ii = 1
        while True:
            if zero_or_one(x, y*ii) == 0:
                return r
            r=1-r
            ii += 1

# Example of use
def exprand(lam):
   return exprandfill(exprandnew(lam),53)*1.0/(1<<53)
```

Beta Sampler: Known Issues

In the beta sampler, the bigger `alpha` or `beta` is, the smaller the area of acceptance becomes (and the more likely random numbers get rejected by this method, raising its run-time). This is because `max(u^(alpha-1)*(1-u)^(beta-1))`, the peak of the density, approaches 0 as the parameters get bigger. One idea to solve this issue is to expand the density so that the acceptance rate increases. The following was tried:

- Estimate an upper bound for the peak of the density `peak`, given `alpha` and `beta`.
- Calculate a largest factor `c` such that `peak * c = m < 0.5`.
- Use Huber's `linear_lowprob` Bernoulli factory (implemented in *bernoulli.py*) (Huber 2016)**(16)**, taking the values found for `c` and `m`. Testing shows that the choice of `m` is crucial for performance.

But doing so apparently worsened the performance (in terms of random bits used) compared to the simple rejection approach.

Exponential Sampler: Extension

The code above supports rational-valued λ parameters. It can be extended to support any real-valued λ parameter greater than 0, as long as λ can be rewritten as the sum of one or more components whose fractional parts can each be simulated by a Bernoulli factory algorithm that outputs heads with probability equal to that fractional part.**(17)**.

More specifically:

1. Decompose λ into $n > 0$ positive components that sum to λ. For example, if λ = 3.5, it can be decomposed into only one component, 3.5 (whose fractional part is trivial to simulate), and if λ = π, it can be decomposed into four components that are all (π / 4), which has a not-so-trivial simulation described in "**Bernoulli Factory Algorithms**".
2. For each component $LC[i]$ found this way, let $LI[i]$ be floor($LC[i]$) and let $LF[i]$ be $LC[i]$ − floor($LC[i]$) ($LC[i]$'s fractional part).

The code above can then be modified as follows:

- `exprandnew` is modified so that instead of taking `lamdanum` and `lamdaden`, it takes a list of the components described above. Each component is stored as $LI[i]$ and an algorithm that simulates $LF[i]$.

- `zero_or_one_exp_minus(a, b)` is replaced with the **algorithm for exp(− z)**

described in “**Bernoulli Factory Algorithms**”, where *z* is the real-valued λ parameter.

- `logisticexp(a, b, index+1)` is replaced with the **algorithm for 1 / 1 + exp(*z* / $2^{index+1}$)) (LogisticExp)** described in “**Bernoulli Factory Algorithms**”, where *z* is the real-valued λ parameter.

Correctness Testing

Beta Sampler

To test the correctness of the beta sampler presented in this document, the Kolmogorov–Smirnov test was applied with various values of `alpha` and `beta` and the default precision of 53, using SciPy’s `kstest` method. The code for the test is very simple: `kst = scipy.stats.kstest(ksample, lambda x: scipy.stats.beta.cdf(x, alpha, beta))`, where `ksample` is a sample of random numbers generated using the sampler above. Note that SciPy uses a two-sided Kolmogorov–Smirnov test by default.

See the results of the **correctness testing**. For each pair of parameters, five samples with 50,000 numbers per sample were taken, and results show the lowest and highest Kolmogorov–Smirnov statistics and p-values achieved for the five samples. Note that a p-value extremely close to 0 or 1 strongly indicates that the samples do not come from the corresponding beta distribution.

ExpRandFill

To test the correctness of the `exprandfill` method (which implements the **ExpRandFill** algorithm), the Kolmogorov–Smirnov test was applied with various values of λ and the default precision of 53, using SciPy’s `kstest` method. The code for the test is very simple: `kst = scipy.stats.kstest(ksample, lambda x: scipy.stats.expon.cdf(x, scale=1/lamda))`, where `ksample` is a sample of random numbers generated using the `exprand` method above. Note that SciPy uses a two-sided Kolmogorov–Smirnov test by default.

The table below shows the results of the correctness testing. For each parameter, five samples with 50,000 numbers per sample were taken, and results show the lowest and highest Kolmogorov–Smirnov statistics and p-values achieved for the five samples. Note that a p-value extremely close to 0 or 1 strongly indicates that the samples do not come from the corresponding exponential distribution.

λ	**Statistic**	***p*-value**
1/10	0.00233-0.00435	0.29954-0.94867
1/4	0.00254-0.00738	0.00864-0.90282
1/2	0.00195-0.00521	0.13238-0.99139
2/3	0.00295-0.00457	0.24659-0.77715
3/4	0.00190-0.00636	0.03514-0.99381
9/10	0.00226-0.00474	0.21032-0.96029
1	0.00267-0.00601	0.05389-0.86676
2	0.00293-0.00684	0.01870-0.78310
3	0.00284-0.00675	0.02091-0.81589
5	0.00256-0.00546	0.10130-0.89935
10	0.00279-0.00528	0.12358-0.82974

ExpRandLess

To test the correctness of `exprandless`, a two-independent-sample T-test was applied to scores involving e-rands and scores involving the Python `random.expovariate` method. Specifically, the score is calculated as the number of times one exponential number compares as less than another; for the same λ this event should ideally be as likely as the event that it compares as greater. The Python code that follows the table calculates this score for e-rands and `expovariate`. Even here, the code for the test is very simple: `kst = scipy.stats.ttest_ind(exppyscores, exprandscores)`, where `exppyscores` and `exprandscores` are each lists of 20 results from `exppyscore` or `exprandscore`, respectively, and the results contained in `exppyscores` and `exprandscores` were generated independently of each other.

The table below shows the results of the correctness testing. For each pair of parameters, results show the lowest and highest T-test statistics and p-values achieved for the 20 results. Note that a p-value extremely close to 0 or 1 strongly indicates that exponential random numbers are not compared as less or greater with the expected probability.

Left λ	Right λ	Statistic	*p*-value
1/10	1/10	-1.21015 – 0.93682	0.23369 – 0.75610
1/10	1/2	-1.25248 – 3.56291	0.00101 – 0.39963
1/10	1	-0.76586 – 1.07628	0.28859 – 0.94709
1/10	2	-1.80624 – 1.58347	0.07881 – 0.90802
1/10	5	-0.16197 – 1.78700	0.08192 – 0.87219
1/2	1/10	-1.46973 – 1.40308	0.14987 – 0.74549
1/2	1/2	-0.79555 – 1.21538	0.23172 – 0.93613
1/2	1	-0.90496 – 0.11113	0.37119 – 0.91210
1/2	2	-1.32157 – -0.07066	0.19421 – 0.94404
1/2	5	-0.55135 – 1.85604	0.07122 – 0.76994
1	1/10	-1.27023 – 0.73501	0.21173 – 0.87314
1	1/2	-2.33246 – 0.66827	0.02507 – 0.58741
1	1	-1.24446 – 0.84555	0.22095 – 0.90587
1	2	-1.13643 – 0.84148	0.26289 – 0.95717
1	5	-0.70037 – 1.46778	0.15039 – 0.86996
2	1/10	-0.77675 – 1.15350	0.25591 – 0.97870
2	1/2	-0.23122 – 1.20764	0.23465 – 0.91855
2	1	-0.92273 – -0.05904	0.36197 – 0.95323
2	2	-1.88150 – 0.64096	0.06758 – 0.73056
2	5	-0.08315 – 1.01951	0.31441 – 0.93417
5	1/10	-0.60921 – 1.54606	0.13038 – 0.91563
5	1/2	-1.30038 – 1.43602	0.15918 – 0.86349
5	1	-1.22803 – 1.35380	0.18380 – 0.64158
5	2	-1.83124 – 1.40222	0.07491 – 0.66075
5	5	-0.97110 – 2.00904	0.05168 – 0.74398

```
def exppyscore(ln,ld,ln2,ld2):
        return sum(1 if random.expovariate(ln*1.0/ld)<random.expovariate(ln2*1.0/ld2) \
              else 0 for i in range(1000))

def exprandscore(ln,ld,ln2,ld2):
        return sum(1 if exprandless(exprandnew(ln,ld), exprandnew(ln2,ld2)) \
              else 0 for i in range(1000))
```

Accurate Simulation of Continuous Distributions on [0, 1]

The beta sampler in this document shows one case of a general approach to simulating a wide class of continuous distributions supported on [0, 1], thanks to Bernoulli factories. This general approach can sample a number that follows one of these distributions, using the algorithm below. The algorithm allows any arbitrary base (or radix) *b* (such as 2 for binary).

1. Create an "empty" uniform PSRN (or "geometric bag"). Create a **SampleGeometricBag** Bernoulli factory that uses that geometric bag.
2. As the geometric bag builds up a uniform random number, accept the number with a probability that can be represented by a Bernoulli factory (that takes the **SampleGeometricBag** factory from step 1 as part of its input), or reject it otherwise. Let $f(U)$ be the probability function modeled by this Bernoulli factory, where U is the uniform random number built up by the geometric bag. f is a multiple of the density function for the underlying continuous distribution (as a result, this algorithm can be used even if the distribution's density function is only known up to a normalization constant). As shown by Keane and O'Brien **(6)**, however, this step works if and only if $f(\lambda)$, in a given interval in [0, 1]—
    - is continuous everywhere, and
    - either returns a constant value in [0, 1] everywhere, or returns a value in [0, 1] at each of the points 0 and 1 and a value in (0, 1) at each other point,

   and they give the example of $2 * \lambda$ as a probability function that cannot be represented by a Bernoulli factory. In the case of constants, the Bernoulli factory can represent them by a geometric bag—
    - that is prefilled with the digit expansion of the constant in question, or
    - that uses a modified **SampleGeometricBag** algorithm in which the constant's digit expansion's digits are not sampled at random, but rather calculated "on the fly" and as necessary.
3. If the geometric bag is accepted, either return the bag as is or fill the unsampled digits of the bag with uniform random digits as necessary to give the number an *n*-digit fractional part (similarly to **FillGeometricBag** above), where *n* is a precision parameter, then return the resulting number.

However, the speed of this algorithm depends crucially on the mode (highest point) of f in [0, 1]. As that mode approaches 0, the average rejection rate increases. Effectively, this step generates a point uniformly at random in a 1×1 area in space. If that mode is close to 0, f will cover only a tiny portion of this area, so that the chance is high that the generated point will fall outside the area of f and have to be rejected.

The beta distribution's probability function at (1) fits the requirements of Keane and O'Brien (for `alpha` and `beta` both greater than 1), thus it can be simulated by Bernoulli factories and is covered by this general algorithm.

This algorithm can be modified to produce random numbers in the interval $[m, m + b^i]$ (where b is the base, or radix, and i and m are integers), rather than [0, 1], as follows:

1. Apply the algorithm above, except a modified probability function $f'(x) = f(x * b^i + m)$ is used rather than f.
2. Multiply the resulting random number or geometric bag by b^i, then add m (this step is relatively trivial given that the geometric bag stores a base-b fractional part).
3. If the random number (rather than its geometric bag) will be returned, and the number's fractional part now has fewer than n digits due to step 2, re-fill the number as necessary to give the fractional part n digits.

Note that here, the probability function f' must meet the requirements of Keane and O'Brien. (For example, take the probability function `sqrt((x - 4) / 2)`, which isn't a Bernoulli factory function. If we now seek to sample from the interval $[4, 4+2^1] = [4, 6]$, the f used in step 2 is now `sqrt(x)`, which *is* a Bernoulli factory function so that we can apply this algorithm.)

On the other hand, modifying this algorithm to produce random numbers in any other interval is non-trivial, since it often requires relating digit probabilities to some kind of formula (see "About Partially-Sampled Random Numbers", above).

An Example: The Continuous Bernoulli Distribution

The continuous Bernoulli distribution (Loaiza-Ganem and Cunningham 2019)**(18)** was designed to considerably improve performance of variational autoencoders (a machine learning model) in modeling continuous data that takes values in the interval [0, 1], including "almost-binary" image data.

The continous Bernoulli distribution takes one parameter `lamda` (a number in [0, 1]), and takes on values in the interval [0, 1] with a probability proportional to—

`pow(lamda, x) * pow(1 - lamda, 1 - x)`.

Again, this function meets the requirements stated by Keane and O'Brien, so it can be simulated via Bernoulli factories. Thus, this distribution can be simulated in Python as described below.

The algorithm for sampling the continuous Bernoulli distribution follows. It uses an input coin that returns 1 with probability `lamda`.

1. Create an empty list to serve as a "geometric bag".
2. Create a **complementary lambda Bernoulli factory** that returns 1 minus the result of the input coin.
3. Remove all digits from the geometric bag. This will result in an empty uniform random number, *U*, for the following steps, which will accept *U* with probability $\texttt{lamda}^{U}*(1-\texttt{lamda})^{1-U}$) (the proportional probability for the beta distribution), as *U* is built up.
4. Call the **algorithm for λ^μ** described in "**Bernoulli Factory Algorithms**", using the input coin as the λ-coin, and **SampleGeometricBag** as the µ-coin (which will return 1 with probability $\texttt{lamda}^{U}$). If the result is 0, go to step 3.
5. Call the **algorithm for λ^μ** using the **complementary lambda Bernoulli factory** as the λ-coin and **SampleGeometricBagComplement** algorithm as the µ-coin (which will return 1 with probability $(1\text{-}\texttt{lamda})^{1-U}$). If the result is 0, go to step 3. (Note that steps 4 and 5 don't depend on each other and can be done in either order without affecting correctness.)
6. *U* was accepted, so return the result of **FillGeometricBag**.

The Python code that samples the continuous Bernoulli distribution follows.

```
def _twofacpower(b, fbase, fexponent):
    """ Bernoulli factory B(p, q) => B(p^q).
           - fbase, fexponent: Functions that return 1 if heads and 0 if tails.
             The first is the base, the second is the exponent.
             """
    i = 1
    while True:
        if fbase() == 1:
            return 1
        if fexponent() == 1 and \
            b.zero_or_one(1, i) == 1:
            return 0
        i = i + 1

def contbernoullidist(b, lamda, precision=53):
    # Continuous Bernoulli distribution
    bag=[]
    lamda=Fraction(lamda)
    gb=lambda: b.geometric_bag(bag)
    # Complement of "geometric bag"
    gbcomp=lambda: b.geometric_bag(bag)^1
    fcoin=b.coin(lamda)
    lamdab=lambda: fcoin()
    # Complement of "lambda coin"
    lamdabcomp=lambda: fcoin()^1
    acc=0
    while True:
       # Create a uniform random number (U) bit-by-bit, and
       # accept it with probability lamda^U*(1-lamda)^(1-U), which
       # is the unnormalized PDF of the beta distribution
       bag.clear()
       # Produce 1 with probability lamda^U
       r=_twofacpower(b, lamdab, gb)
       # Produce 1 with probability (1-lamda)^(1-U)
       if r==1: r=_twofacpower(b, lamdabcomp, gbcomp)
       if r == 1:
             # Accepted, so fill up the "bag" and return the
             # uniform number
             ret=_fill_geometric_bag(b, bag, precision)
             return ret
       acc+=1
```

Complexity

The *bit complexity* of an algorithm that generates random numbers is measured as the number of random bits that algorithm uses on average.

General Principles

Existing work shows how to calculate the bit complexity for any distribution of random numbers:

- For a 1-dimensional continuous distribution, the bit complexity is bounded from below by `DE + prec - 1` random bits, where `DE` is the differential entropy for the distribution and *prec* is the number of bits in the random number's fractional part (Devroye and Gravel 2015)**(3)**.
- For a discrete distribution (a distribution of random integers with separate probabilities of occurring), the bit complexity is bounded from below by the binary entropies of all the probabilities involved, summed together (Knuth and Yao 1976)**(19)**. (For a given probability *p*, the binary entropy is `p*log2(1/p)`.) An optimal algorithm will come within 2 bits of this lower bound on average.

For example, in the case of the exponential distribution, `DE` is log2(exp(1)/λ), so the minimum bit complexity for this distribution is log2(exp(1)/λ) + *prec* − 1, so that if *prec* = 20, this minimum is about 20.443 bits when λ = 1, decreases when λ goes up, and increases when λ goes down. In the case of any other continuous distribution, `DE` is the integral of `f(x) * log2(1/f(x))` over all valid values `x`, where `f` is the distribution's density function.

Although existing work shows lower bounds on the number of random bits an algorithm will need on average, most algorithms will generally not achieve these lower bounds in practice.

In general, if an algorithm calls other algorithms that generate random numbers, the total expected bit complexity is—

- the expected number of calls to each of those other algorithms, times
- the bit complexity for each such call.

Complexity of Specific Algorithms

The beta and exponential samplers given here will generally use many more bits on average than the lower bounds on bit complexity, especially since they generate a PSRN one digit at a time.

The `zero_or_one` method generally uses 2 random bits on average, due to its nature as a Bernoulli trial involving random bits, see also (Lumbroso 2013, Appendix B)**(20)**. However, it uses no random bits if both its parameters are the same.

For **SampleGeometricBag** with base 2, the bit complexity has two components.

- One component comes from sampling a geometric (1/2) random number, as follows:
    - Optimal lower bound: Since the binary entropy of the random number is 2, the optimal lower bound is 2 bits.
    - Optimal upper bound: 4 bits.
- The other component comes from filling the geometric bag with random bits. The complexity here depends on the number of times **SampleGeometricBag** is called for the same bag, call it `n`. Then the expected number of bits is the expected number of bit positions filled this way after `n` calls.

SampleGeometricBagComplement has the same bit complexity as **SampleGeometricBag**.

FillGeometricBag's bit complexity is rather easy to find. For base 2, it uses only one bit to sample each unfilled digit at positions less than `p`. (For bases other than 2, sampling *each* digit this way might not be optimal, since the digits are generated one at a time and random bits are not recycled over several digits.) As a result, for an algorithm that uses both **SampleGeometricBag** and **FillGeometricBag** with `p` bits, these two contribute, on average, anywhere from `p + g * 2` to `p + g * 4` bits to the complexity, where `g` is the number of calls to **SampleGeometricBag**. (This complexity could be increased by 1 bit if **FillGeometricBag** is implemented with a rounding mechanism other than simple truncation.)

The complexity of the **algorithm for exp(−*x*/*y*)** (which outputs 1 with probability exp(−*x*/*y*)) was discussed in some detail by (Canonne et al. 2020)**(21)**, but not in terms of its bit complexity. The special case of γ =*x*/*y* = 0 requires no bits. If γ is an integer greater than 1, then the bit complexity is the same as that of sampling a geometric(exp(−1)) random number, but truncated to [0, γ]. (In this document, the geometric(`n`) distribution has the density function `pow(x, n) * (1 - x)`.)

- Optimal lower bound: Has a complicated formula for general γ, but approaches `log2(exp(1)-(exp(1)+1)*ln(exp(1)-1))` = 2.579730853... bits with increasing γ.
- Optimal upper bound: Optimal lower bound plus 2.
- The actual implementation's average bit complexity is generally—
    - the expected number of calls to the **algorithm for exp(−*x*/*y*)** (with γ = 1), which is the expected value of the truncated geometric distribution described above, times
    - the bit complexity for each such call.

If γ is 1 or less, the optimal bit complexity is determined as the complexity of sampling a random integer *k* with probability function—

- $P(k) = \gamma^k/k! - \gamma^{k+1}/(k+1)!$,

and the optimal lower bound is found by taking the binary entropy of each probability (`P(k)/log2(1/P(k))`) and summing them all.

- Optimal lower bound: Again, this has a complicated formula (see the appendix for SymPy code), but it appears to be highest at about 1.85 bits, which is reached when γ is about 0.848.
- Optimal upper bound: Optimal lower bound plus 2.
- The actual implementation's average bit complexity is generally—

    - the expected number of calls to `zero_or_one`, which was determined to be exp(γ) in (Canonne et al. 2020)[21], times
    - the bit complexity for each such call (which is generally 2, but is lower in the case of γ = 1, which involves `zero_or_one(1, 1)` that uses no random bits).

If γ is a non-integer greater than 1, the bit complexity is the sum of the bit complexities for its integer part and for its fractional part.

Application to Weighted Reservoir Sampling

Weighted reservoir sampling (choosing an item at random from a list of unknown size) is often implemented by—

- assigning each item a *weight* (an integer 0 or greater) as it's encountered, call it *w*,
- giving each item an exponential random number with λ = *w*, call it a key, and
- choosing the item with the smallest key

(see also (Efraimidis 2015)[22]). However, using fully-sampled exponential random numbers as keys (such as the naïve idiom `-ln(1-RNDU01())/w` in common floating-point arithmetic) can lead to inexact sampling, since the keys have a limited precision, it's possible for multiple items to have the same random key (which can make sampling those items depend on their order rather than on randomness), and the maximum weight is unknown. Partially-sampled e-rands, as given in this document, eliminate the problem of inexact sampling. This is notably because the `exprandless` method returns one of only two answers—either "less" or "greater"—and samples from both e-rands as necessary so that they will differ from each other by the end of the operation. (This is not a problem because randomly generated real numbers are expected to differ from each other almost surely.) Another reason is that partially-sampled e-rands have potentially arbitrary precision.

Open Questions

There are some open questions on PSRNs:

1. Are there constructions for partially-sampled normal random numbers with a standard deviation other than 1 and/or a mean other than an integer?
2. Are there constructions for PSRNs other than for cases given earlier in this document?
3. What are exact formulas for the digit probabilities when arithmetic is carried out between two PSRNs (such as addition, multiplication, division, and powering)?

Acknowledgments

I acknowledge Claude Gravel who reviewed a previous version of this article.

Other Documents

The following are some additional articles I have written on the topic of random and pseudorandom number generation. All of them are open-source.

- **Random Number Generator Recommendations for Applications**
- **Randomization and Sampling Methods**
- **More Random Number Sampling Methods**
- **Code Generator for Discrete Distributions**
- **The Most Common Topics Involving Randomization**
- **Bernoulli Factory Algorithms**
- **Testing PRNGs for High-Quality Randomness**

- **Examples of High-Quality PRNGs**

Notes

[1] Karney, C.F.F., "**Sampling exactly from the normal distribution**", arXiv:1303.6257v2 [physics.comp-ph], 2014.

[2] Philippe Flajolet, Nasser Saheb. The complexity of generating an exponentially distributed variate. [Research Report] RR-0159, INRIA. 1982. inria-00076400.

[3] Devroye, L., Gravel, C., "**Sampling with arbitrary precision**", arXiv:1502.02539v5 [cs.IT], 2015.

[4] Thomas, D.B. and Luk, W., 2008, September. Sampling from the exponential distribution using independent bernoulli variates. In 2008 International Conference on Field Programmable Logic and Applications (pp. 239-244). IEEE.

[5] A. Habibizad Navin, R. Olfatkhah and M. K. Mirnia, "A data-oriented model of exponential random variable," 2010 2nd International Conference on Advanced Computer Control, Shenyang, 2010, pp. 603-607, doi: 10.1109/ICACC.2010.5487128.

[6] Keane, M. S., and O'Brien, G. L., "A Bernoulli factory", *ACM Transactions on Modeling and Computer Simulation* 4(2), 1994.

[7] Flajolet, P., Pelletier, M., Soria, M., "**On Buffon machines and numbers**", arXiv:0906.5560v2 [math.PR], 2010.

[8] Pedersen, K., "**Reconditioning your quantile function**", arXiv:1704.07949v3 [stat.CO], 2018.

[9] von Neumann, J., "Various techniques used in connection with random digits", 1951.

[10] Oberhoff, Sebastian, "**Exact Sampling and Prefix Distributions**", *Theses and Dissertations*, University of Wisconsin Milwaukee, 2018.

[11] Brassard, G., Devroye, L., Gravel, C., "Remote Sampling with Applications to General Entanglement Simulation", *Entropy* 2019(21)(92), doi:10.3390/e21010092.

[12] A. Habibizad Navin, Fesharaki, M.N., Teshnelab, M. and Mirnia, M., 2007. "Data oriented modeling of uniform random variable: Applied approach". *World Academy Science Engineering Technology*, 21, pp.382-385.

[13] Nezhad, R.F., Effatparvar, M., Rahimzadeh, M., 2013. "Designing a Universal Data-Oriented Random Number Generator", *International Journal of Modern Education and Computer Science* 2013(2), pp. 19-24.

[14] Rohatgi, V.K., 1976. An Introduction to Probability Theory Mathematical Statistics.

[15] Devroye, L., ***Non-Uniform Random Variate Generation***, 1986.

[16] Huber, M., "**Optimal linear Bernoulli factories for small mean problems**", arXiv:1507.00843v2 [math.PR], 2016.

[17] In fact, thanks to the "geometric bag" technique of Flajolet et al. (2010), that fractional part can even be a uniform random number in [0, 1] whose contents are built up digit by digit.

[18] Loaiza-Ganem, G., Cunningham, J.P., "**The continuous Bernoulli: fixing a pervasive error in variational autoencoders**", arXiv:1907.06845v5 [stat.ML], 2019.

[19] 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.

[20] Lumbroso, J., "**Optimal Discrete Uniform Generation from Coin Flips, and Applications**", arXiv:1304.1916 [cs.DS].

[21] Canonne, C., Kamath, G., Steinke, T., "**The Discrete Gaussian for Differential Privacy**", arXiv:2004.00010v2 [cs.DS], 2020.

[22] Efraimidis, P. "**Weighted Random Sampling over Data Streams**", arXiv:1012.0256v2 [cs.DS], 2015.

Appendix

SymPy Formula for the algorithm for exp(−*x*/*y*)

The following Python code uses SymPy to plot the bit complexity lower bound for the **algorithm for exp(−*x*/*y*)** when γ is 1 or less:

```
def ent(p):
   return p*log(1/p,2)

def expminusformula():
   i=symbols('i',integer=True)
   x=symbols('x',real=True)
   # Approximation for k = [0, 6]; the result is little different
   # for k = [0, infinity]
   return summation(ent(x**i/factorial(i) - \
      x**(i+1)/factorial(i+1)), (i,0,6))

plot(expminusformula(), xlim=(0,1), ylim=(0,2))
```

Another Example of an Arbitrary-Precision Sampler

As an additional example of how PSRNs can be useful, here we reimplement an example from Devroye's book *Non-Uniform Random Variate Generation* (Devroye 1986, pp. 128–129)**[15]**. The following algorithm generates a random number from a distribution with the following cumulative distribution function: `1 - cos(pi*x/2)`. The random number will be in the interval [0, 1]. What is notable about this algorithm is that it's an arbitrary-precision algorithm that avoids floating-point arithmetic. Note that the result is the same as applying acos(*U*)*2/π, where *U* is a uniform [0, 1] random number, as pointed out by Devroye. The algorithm follows.

1. Call the **kthsmallest** algorithm with `n = 2` and `k = 2`, but without filling it with digits at the last step. Let *ret* be the result.
2. Set *m* to 1.
3. Call the **kthsmallest** algorithm with `n = 2` and `k = 2`, but without filling it with digits at the last step. Let *u* be the result.
4. With probability 4/(4**m***m* + 2**m*), call the **URandLess** algorithm with parameters *u* and *ret* in that order, and if that call returns `true`, call the **algorithm for π / 4**, described in "**Bernoulli Factory Algorithms**", twice, and if both of these calls return 1, add 1 to *m* and go to step 3. (Here, we incorporate an erratum in the algorithm on page 129 of the book.)
5. If *m* is odd, fill *ret* with uniform random digits as necessary to give its fractional part the desired number of digits (similarly to **FillGeometricBag**), and return *ret*.
6. If *m* is even, go to step 1.

And here is Python code that implements this algorithm. Note again that it uses floating-point arithmetic only at the end, to convert the result to a convenient form, and that it relies on methods from *randomgen.py* and *bernoulli.py*.

```
def example_4_2_1(rg, bern, precision=53):
    while True:
       ret=rg.kthsmallest_urand(2,2)
       k=1
       while True:
```

```
u=rg.kthsmallest_urand(2,2)
kden=4*k*k+2*k # erratum incorporated
if randomgen.urandless(rg,u, ret) and \
   rg.zero_or_one(4, kden)==1 and \
   bern.zero_or_one_pi_div_4()==1 and \
   bern.zero_or_one_pi_div_4()==1:
   k+=1
elif (k&1)==1:
   return randomgen.urandfill(rg,ret,precision)/(1<<precision)
else: break
```

License