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

This version of the document is dated 2020-10-16.

Peter Occil

1 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.

1.1 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.

2 Contents

3 Notation

In this document, `RNDINT(x)` is a uniformly-distributed random integer in the interval [0, x], `RNDINTEXC(x)` is a uniformly-distributed random integer in the interval [0, x), and `RNDU01OneExc()` is a uniformly-distributed random real number in the interval [0, 1).

4 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.

5 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 - RNDU01OneExc()) / lamda`. (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.

6 About Partially-Sampled Random Numbers

In this document, a *partially-sampled random number* (PSRN) is a data structure that stores a real number of unlimited precision, but whose contents are sampled only when necessary. PSRNs open the door to algorithms that sample a random number that "exactly" follows a continuous distribution, *with arbitrary precision*, and *without floating-point arithmetic* (see "Properties" later in this section).

PSRNs specified here consist of the following three things:

- A *fractional part* with an arbitrary number of digits. This can be implemented as an array of digits or as a packed integer containing all the digits. Some algorithms care whether those digits were *sampled* or *unsampled*; in that case, if a digit is unsampled, its unsampled status can be noted in a way that distinguishes it from sampled digits (e.g., by using the `None` keyword in Python, or the number −1, or by storing a separate bit array indicating which bits are sampled and unsampled). The

base in which all the digits are stored (such as base 10 for decimal or base 2 for binary) is arbitrary. The fractional part's digits form a so-called *digit expansion* (e.g., *binary expansion* in the case of binary or base-2 digits). Digits beyond those stored in the fractional part are unsampled.

For example, if the fractional part stores the base-10 digits [1, 3, 5], in that order, then it represents a random number in the interval [0.135, 0.136], reflecting the fact that the digits between 0.135 and 0.136 are unknown.

- An optional *integer part* (more specifically, the integer part of the number's absolute value, that is, `floor(abs(x))`).
- An optional *sign* (positive or negative).

If an implementation cares only about PSRNs in the interval [0, 1], it can store only a fractional part; in this case, the unstored integer part and sign are assumed to be 0 and positive, respectively.

PSRNs ultimately represent a random number between two others; one of the number's two bounds has the following form: sign * (integer part + fractional part), which is a lower bound if the PSRN is positive, or an upper bound if it's negative. For example, if the PSRN stores a positive sign, the integer 3, and the fractional part [3, 5, 6] (in base 10), then the PSRN represents a random number in the interval [3.356, 3.357]. Here, one of the bounds is built using the PSRN's sign, integer part, and fractional part, and because the PSRN is positive, this is a lower bound.

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

6.1 Uniform Partially-Sampled Random Numbers

The most trivial example of a PSRN is that of the uniform distribution.

- Flajolet et al. (2010)**(7)** use the term *geometric bag* to refer to a uniform PSRN in the interval [0, 1] that stores binary (base-2) digits, some of which may be unsampled. In this case, the PSRN can consist of just a fractional part, which can be implemented as described earlier.
- (Karney 2014)**(1)** uses the term *u-rand* to refer to uniform PSRNs that can store a sign, integer part, and a fractional part, where the base of the fractional part's digits is arbitrary, but Karney's concept only contemplates sampling digits from left to right without any gaps.

Each additional digit of a uniform PSRN's fractional part is sampled simply by setting it to an independent uniform random digit, an observation that dates from von Neumann (1951)**(9)** in the binary case.**(10)** A PSRN with this property is called a **uniform PSRN** in this document, even if was generated using a non-uniform random sampling algorithm (such as Karney's algorithm for the normal distribution). (This is notably because, in general, this kind of PSRN represents a uniformly-distributed random number in a given interval. For example, if the PSRN is 3.356..., then it represents a random number that is uniformly distributed in the interval [3.356, 3.357].)

6.2 Exponential Partially-Sampled Random Numbers

In this document, an **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. (More specifically, an exponential PSRN generally represents an exponentially-distributed random number in a given interval.)

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

6.3 Other Distributions

PSRNs 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, or as shown in an improved version of that algorithm by Du et al. (2020)**(11)**.
- Sampling uniform distributions in [0, n) (not just [0, 1]), is described later in "**Sampling Uniform PSRNs**".)

For these distributions (and others that are continuous almost everywhere and bounded from above), Oberhoff (2018)**(12)** proved that unsampled trailing bits of the PSRN converge to the uniform distribution (see also (Kakutani 1948)**(13)**).

PSRNs could also be implemented via rejection from the exponential distribution, although no concrete examples are presented here.

6.4 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.
2. The algorithm 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.
3. The algorithm may use rational arithmetic (such as `Fraction` in Python or `Rational` in Ruby), as long as the arithmetic is exact.
4. 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 position, starting from 1, of the rightmost sampled digit of the PSRN's fractional part. ((Devroye and Gravel 2015)**(3)** suggests Wasserstein distance, or "earth-mover 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).
5. 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}.

   Note: The *exact rejection sampling* algorithm described by Oberhoff (2018)**(12)**

produces samples that act like PSRNs; however, the algorithm doesn't have the properties described in this section. 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 2 above). Moreover, the algorithm's 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]). For completeness, Oberhoff's method appears in the appendix.

6.5 Comparisons

Two PSRNs, each of a different distribution but storing digits of the same base (radix), can be exactly compared to each other using algorithms similar to those in this section.

The **RandLess** algorithm compares two PSRNs, **a** and **b** (and samples additional bits from them as necessary) and returns 1 if **a** turns out to be less than **b** almost surely, or 0 otherwise (see also (Karney 2014)[1]).

1. If **a**'s integer part wasn't sampled yet, sample **a**'s integer part according to the kind of PSRN **a** is. Do the same for **b**.
2. If **a**'s sign is different from **b**'s sign, return 1 if **a** is negative and 0 if non-negative. If **a** is non-negative, return 1 if **a**'s integer part is less than **b**'s, or 0 if greater. If **a** is negative, return 0 if **a**'s integer part is less than **b**'s, or 1 if greater.
3. Set *i* to 0.
4. If the digit at position *i* of **a**'s fractional part is unsampled, set the digit at that position according to the kind of PSRN **a** is. (Positions start at 0 where 0 is the most significant digit after the point, 1 is the next, etc.) Do the same for **b**.
5. Let *da* be the digit at position *i* of **a**'s fractional part, and let *db* be **b**'s corresponding digit.
6. If **a** is non-negative, return 1 if *da* is less than *db*, or 0 if *da* is greater than *db*.
7. If **a** is negative, return 0 if *da* is less than *db*, or 1 if *da* is greater than *db*.
8. Add 1 to *i* and go to step 4.

URandLess is a version of **RandLess** that involves two uniform PSRNs. The algorithm for **URandLess** samples digit *i* in step 4 by setting the digit at position *i* to a digit chosen uniformly at random. (For example, if **a** is a uniform PSRN that stores base-2 or binary digits, this can be done by setting the digit at that position to `RNDINTEXC(2)`.)

The **RandLessThanReal** algorithm compares a PSRN **a** with a real number **b** and returns 1 if **a** turns out to be less than **b** almost surely, or 0 otherwise. This algorithm samples digits of **a**'s fractional part as necessary. This algorithm works whether **b** is known to be a rational number or not (for example, **b** can be the result of an expression such as `exp(-2)` or `log(20)`), but the algorithm notes how it can be more efficiently implemented if **b** is known to be a rational number.

1. If **a**'s integer part or sign is unsampled, return an error.
2. Set *bs* to −1 if **b** is less than 0, or 1 otherwise. Calculate floor(abs(**b**)), and set *bi* to the result. (*If **b** is known rational:* Then set *bf* to abs(**b**) minus *bi*.)
3. If **a**'s sign is different from *bs*'s sign, return 1 if **a** is negative and 0 if non-negative. If **a** is non-negative, return 1 if **a**'s integer part is less than *bi*, or 0 if greater. If **a** is negative, return 0 if **a**'s integer part is less than *bi*, or 1 if greater.
4. Set *i* to 0.
5. If the digit at position *i* of **a**'s fractional part is unsampled, set the digit at that position according to the kind of PSRN **a** is. (Positions start at 0 where 0 is the most

significant digit after the point, 1 is the next, etc.)
6. Calculate the base-β digit at position *i* of **b**'s fractional part, and set *d* to that digit. (*If **b** is known rational:* Do this step by multiplying *bf* by β, then setting *d* to floor(*bf*), then subtracting *d* from *bf*.)
7. Let *ad* be the digit at position *i* of **a**'s fractional part.
8. Return 1 if—
    - *ad* is less than *d* and **a** is non-negative,
    - *ad* is greater than *d* and **a** is negative, or
    - *ad* is equal to *d*, **a** is negative, and—
        - ***b** is not known to be rational* and all the digits after the digit at position *i* of **b**'s fractional part are zeros (indicating **a** is less than **b** almost surely), or
        - ***b** is known to be rational* and *bf* is 0 (indicating **a** is less than **b** almost surely).
9. Return 0 if—
    - *ad* is less than *d* and **a** is negative,
    - *ad* is greater than *d* and **a** is non-negative, or
    - *ad* is equal to *d*, **a** is non-negative, and—
        - ***b** is not known to be rational* and all the digits after the digit at position *i* of **b**'s fractional part are zeros (indicating **a** is greater than **b** almost surely), or
        - ***b** is known to be rational* and *bf* is 0 (indicating **a** is greater than **b** almost surely).
10. Add 1 to *i* and go to step 5.

An alternative version of steps 6 through 9 in the algorithm above are as follows (see also (Brassard et al. 2019)**(14)**):

- (6.) Calculate *bp*, which is an approximation to **b** such that abs(**b** − *bp*) <= β^{-i-1}. Let *bk* be *bp*'s digit expansion up to the *i* + 1 digits after the point. For example, if **b** is π, β is 10, and *i* is 4, one possibility is *bp* = 3.14159 and *bk* = 314159.
- (7.) Let *ak* be **a**'s digit expansion up to the *i* + 1 digits after the point.
- (8.) Return 1 if *ak* <= *bk* − 2.
- (9.) Return 0 if *ak* >= *bk* + 1.

URandLessThanReal is a version of **RandLessThanReal** in which **a** is a uniform PSRN. The algorithm for **URandLessThanReal** samples digit *i* in step 4 by setting the digit at position *i* to a digit chosen uniformly at random.

The following is a simpler way to implement **URandLessThanReal** when **a** is non-negative and its integer part is 0, and when **b** is known to be a rational number.

1. If **a**'s integer part or sign is unsampled, or if **a** is negative or its integer part is other than 0, return an error. If **b** is 0 or less, return 0. If **b** is 1 or greater, return 1. (The case of 1 is a degenerate case since **a** could, at least in theory, represent an infinite sequence of ones, making it equal to 1.)
2. Set *pt* to 1/*base*, and set *i* to 0. (*base* is the base, or radix, of **a**'s digits, such as 2 for binary or 10 for decimal.)
3. Set *d1* to the digit at the i^{th} position (starting from 0) of **a**'s fractional part. If the digit at that position is unsampled, put a digit chosen uniformly at random at that position and set *d1* to that digit.
4. Set *d2* to floor(**b** / *pt*). (For example, in base 2, set *d2* to 0 if **b** is less than *pt*, or 1 otherwise.)
5. If *d1* is less than *d2*, return 1. If *d1* is greater than *d2*, return 0.

6. If **b** >= *pt*, subtract *pt* from **b**.
7. If **b** is 0, return 0 (indicating that **a** is greater than the original fraction almost surely).
8. Divide *pt* by *base*, add 1 to *i*, and go to step 3.

The following is a simpler way to implement **URandLessThanReal** when **a** is non-negative and its integer part is 0, and when **b** is a fraction known by its numerator and denominator, *num*/*den*.

1. If **a**'s integer part or sign is unsampled, or if *den* is 0, return an error. If *num* and *den* are both less than 0, set them to their absolute values. If **a** is negative or its integer part is other than 0, return an error. If *num* is 0, or if *num* < 0 or *den* < 0, return 0. If *num* >= *den*, return 1.
2. Set *pt* to *base*, and set *i* to 0. (*base* is the base, or radix, of **a**'s digits, such as 2 for binary or 10 for decimal.)
3. Set *d1* to the digit at the i^{th} position (starting from 0) of **a**'s fractional part. If the digit at that position is unsampled, put a digit chosen uniformly at random at that position and set *d1* to that digit.
4. Set *c* to 1 if *num* * *pt* >= *den*, and 0 otherwise.
5. Set *d2* to floor(*num* * *pt* / *den*). (In base 2, this is equivalent to setting *d2* to *c*.)
6. If *d1* is less than *d2*, return 1. If *d1* is greater than *d2*, return 0.
7. If *c* is 1, set *num* to *num* * *pt* − *den*, then multiply *den* by *pt*.
8. If *num* is 0, return 0.
9. Multiply *pt* by *base*, add 1 to *i*, and go to step 3.

7 Sampling Uniform and Exponential PSRNs

7.1 Sampling Uniform PSRNs

There are two algorithms for sampling uniform partially-sampled random numbers given another number.

The **RandUniform** algorithm generates a uniformly distributed PSRN (**a**) that is greater than 0 and less than another PSRN (**b**) almost surely. This algorithm samples digits of **b**'s fractional part as necessary. This algorithm should not be used if **b** is known to be a real number rather than a partially-sampled random number, since this algorithm could overshoot the value **b** had (or appeared to have) at the beginning of the algorithm; instead, the **RandUniformFromReal** algorithm, given later, should be used. (For example, if **b** is 3.425..., one possible result of this algorithm is **a** = 3.42574... and **b** = 3.42575... Note that in this example, 3.425... is not considered an exact number.)

1. Create an empty uniform PSRN **a**. Let β be the base (or radix) of digits stored in **b**'s fractional part (e.g., 2 for binary or 10 for decimal). If **b**'s integer part or sign is unsampled, or if **b**'s sign is negative, return an error.
2. Set **a**'s sign to positive and **a**'s integer part to an integer chosen uniformly at random in [0, *bi*], where *bi* is **b**'s integer part (e.g., `RNDINT(0, bi)`). If **a**'s integer part is less than *bi*, return **a**.
3. (We now sample **a**'s fractional part.) Set *i* to 0.
4. If **b**'s integer part is 0 and **b**'s fractional part begins with a sampled 0-digit, set *i* to the number of sampled zeros at the beginning of **b**'s fractional part. A nonzero digit or an unsampled digit ends this sequence. Then append *i* zeros to **a**'s fractional part.

(For example, if **b** is 5.000302 or 4.000 or 0.0008, there are three sampled zeros that begin **b**'s fractional part, so *i* is set to 3 and three zeros are appended to **a**'s fractional part.)

5. If the digit at position *i* of **a**'s fractional part is unsampled, set the digit at that position to a base-β digit chosen uniformly at random. (Positions start at 0 where 0 is the most significant digit after the point, 1 is the next, etc. An example if β is 2, or binary, is `RNDINTEXC(2)`.)
6. If the digit at position *i* of **b**'s fractional part is unsampled, sample the digit at that position according to the kind of PSRN **b** is. (For example, if **b** is a uniform PSRN and β is 2, this can be done by setting the digit at that position to `RNDINTEXC(2)`.)
7. If the digit at position *i* of **a**'s fractional part is less than the corresponding digit for **b**, return **a**.
8. If that digit is greater, then discard **a**, then create a new empty uniform PSRN **a**, then go to step 2.
9. Add 1 to *i* and go to step 5.

Notes:

1. Karney (2014, end of sec. 4)**(1)** discusses how even the integer part can be partially sampled rather than generating the whole integer as in step 2 of the algorithm. However, incorporating this suggestion will add a non-trivial amount of complexity to the algorithm given above.
2. The **RandUniform** algorithm is equivalent to generating the product of a random number (**b**) and a uniform(0, 1) random number.
3. If **b** is a uniform PSRN with a positive sign, an integer part of 0, and an empty fractional part, the **RandUniform** algorithm is equivalent to generating the product of two uniform(0, 1) random numbers.

The **RandUniformFromReal** algorithm generates a uniformly distributed PSRN (**a**) that is greater than 0 and less than a real number **b** almost surely. This algorithm works whether **b** is known to be a rational number or not (for example, **b** can be the result of an expression such as `exp(-2)` or `log(20)`), but the algorithm notes how it can be more efficiently implemented if **b** is known to be a rational number.

1. If **b** is 0 or less, return an error.
2. Create an empty uniform PSRN **a**.
3. Calculate floor(**b**), and set *bi* to the result. (*If **b** is known rational:* Then set *bf* to **b** minus *bi*.)
4. If *bi* is equal to **b**, set **a**'s sign to positive and **a**'s integer part to an integer chosen uniformly at random in [0, *bi*) (e.g., `RNDINTEXC(0, bi)`), then return **a**. (It should be noted that determining whether a real number is equal to another is undecidable in general.)
5. Set **a**'s sign to positive and **a**'s integer part to an integer chosen uniformly at random in [0, *bi*] (e.g., `RNDINT(0, bi)`), then if **a**'s integer part is less than *bi*, return **a**.
6. (We now sample **a**'s fractional part.) Set *i* to 0.
7. If *bi* is 0 and not equal to **b**, then do the following in a loop:
    1. Calculate the base-β digit at position *i* of **b**'s fractional part, and set *d* to that digit. (β is the desired digit base, or radix, of the uniform PSRN, such as 10 for decimal or 2 for binary). (*If **b** is known rational:* Do this step by multiplying *bf* by β, then setting *d* to floor(*bf*), then subtracting *d* from *bf*.)
    2. If *d* is 0, append a 0-digit to **a**'s fractional part, then add 1 to *i*. Otherwise, break from this loop.
8. If the digit at position *i* of **a**'s fractional part is unsampled, set the digit at that position to a base-β digit chosen uniformly at random. (Positions start at 0 where 0 is the most significant digit after the point, 1 is the next, etc. An example if β is 2, or

binary, is `RNDINTEXC(2)`.)

9. Calculate the base-β digit at position *i* of **b**'s fractional part, and set *d* to that digit. (*If **b** is known rational:* Do this step by multiplying *bf* by β, then setting *d* to floor(*bf*), then subtracting *d* from *bf*.)
10. Let *ad* be the digit at position *i* of **a**'s fractional part. If *ad* is less than *d*, return **a**.
11. *If **b** is not known to be rational:* If *ad* is greater than *d*, or if all the digits after the digit at position *i* of **b**'s fractional part are zeros, then discard **a**, then create a new empty uniform PSRN **a**, then go to step 5.
12. *If **b** is known rational:* If *ad* is greater than *d*, or if *bf* is 0, then discard **a**, then create a new empty uniform PSRN **a**, then set *bf* to **b** minus *bi*, then go to step 5.
13. Add 1 to *i* and go to step 8.

7.2 Sampling E-rands

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, which both enable e-rands with rational valued λ parameters:

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**").

8 Arithmetic with PSRNs

8.1 In General

Arithmetic between two PSRNs is not always trivial.

- Adding, multiplying, or dividing two PSRNs *A* and *B* (see also (Brassard et al., 2019) **(14)**) is not as simple as adding their integer and fractional parts.

  An example illustrates this. Say we have two uniform PSRNs: *A* = 0.12345... and *B* = 0.38901.... They represent random numbers in the intervals *AI* = [0.12345, 0.12346] and *BI* = [0.38901, 0.38902], respectively. Adding two uniform PSRNs is akin to adding their intervals (using interval arithmetic), so that in this example, the result *C* lies in *CI* = [0.12345 + 0.38901, 0.12346 + 0.38902] = [0.51246, 0.51248]. However, the resulting random number is *not* uniformly distributed in [0.51246, 0.51248], so that simply choosing a uniform random number in the interval won't work. This can be demonstrated by generating many pairs of uniform random numbers in the intervals *AI* and *BI*, summing the numbers in each pair, and building

a histogram using the sums (which will all lie in the interval *CI*). In this case, the histogram will show a triangular distribution that peaks at 0.51247.

This example can also apply to other arithmetic operations besides addition: do the interval operation (multiplication, division, etc.) on the intervals *AI* and *BI*, and build a histogram of random results (products, quotients, etc.) that lie in the resulting interval to find out what distribution forms this way.

Another reason most operations are nontrivial is that the result of the operation may be an irrational number (as in `log`, `sin`, etc.), or can even have a non-terminating digit expansion (as in most cases of division). For these operations, although interval arithmetic can tightly bound the possible result to a finite number of digits, the resulting interval can include numbers with a probability density of zero.

- On the other hand, some other arithmetic operations are trivial to carry out in PSRNs. They include negation, as mentioned in (Karney 2014)**(1)**, and operations affecting the PSRN's integer part only.
- Partially-sampled-number arithmetic may also be possible by relating the relative probabilities of each digit, in the result's digit expansion, to some kind of formula.
    - 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)**(15)**, (Nezhad et al., 2013)**(16)**. This previous work points to building a probability tree, where the probability of the next digit depends on the value of the previous digits. However, calculating each probability requires knowing the distribution's cumulative distribution function (CDF), and the calculations can incur rounding errors especially when the digit probabilities are not rational numbers or they have no simple mathematical form, as is often the case.
    - For some distributions (including the uniform and exponential distributions), the digit probabilities don't depend on previous digits, only on the position of the digit. In this case, however, there appear to be limits on how practical this approach is; see the **appendix** for details.
- Finally, arithmetic with PSRNs may be possible if the result of the arithmetic is distributed with a known probability density function (PDF) (e.g., one found via Rohatgi's formula (Rohatgi 1976)**(17)**), allowing for an algorithm that implements rejection from the uniform or exponential distribution. An example of this is found in my article on **arbitrary-precision samplers for the sum of uniform random numbers**. However, that PDF 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.

8.2 Addition and Subtraction

The following algorithm shows how to add two uniform PSRNs (**a** and **b**) that store digits of the same base (radix) in their fractional parts, and get a uniform PSRN as a result. The input PSRNs may be negative or non-negative, and it is assumed that their integer parts and signs were sampled. *Python code implementing this algorithm is given later in this document.*

1. If **a** has unsampled digits before the last sampled digit in its fractional part, set each of those unsampled digits to a digit chosen uniformly at random. Do the same for **b**.

2. If **a** has fewer digits in its fractional part than **b** (or vice versa), sample enough digits (by setting them to uniform random digits, such as unbiased random bits if **a** and **b** store binary, or base-2, digits) so that both PSRNs' fractional parts have the same number of digits. Now, let *digitcount* be the number of digits in **a**'s fractional part.
3. Let *asign* be −1 if **a** is negative, or 1 otherwise. Let *bsign* be −1 if **b** is negative, or 1 otherwise. Let *afp* be the digits of **a**'s *fractional part*, and let *bfp* be the digits of **b**'s *fractional part*. (For example, if **a** represents the number 83.12344..., *afp* is 12344.) Let *base* be the base of digits stored by **a** and **b**, such as 2 for binary or 10 for decimal.
4. Calculate the following four numbers:
    - *afp***asign* + *bfp***bsign*.
    - *afp***asign* + (*bfp*+1)**bsign*.
    - (*afp*+1)**asign* + *bfp***bsign*.
    - (*afp*+1)**asign* + (*bfp*+1)**bsign*.
5. Set *minv* to the minimum and *maxv* to the maximum of the four numbers just calculated. These are lower and upper bounds to the result of applying interval addition to the PSRNs **a** and **b**. (For example, if **a** is 0.12344... and **b** is 0.38925..., their fractional parts are added to form **c** = 0.51269...., or the interval [0.51269, 0.51271].) However, the resulting PSRN is not uniformly distributed in its interval, and this is what the rest of this algorithm will solve, since in fact, the distribution of numbers in the interval resembles the distribution of the sum of two uniform random numbers.
6. Once the four numbers are sorted from lowest to highest, let *midmin* be the second number in the sorted order, and let *midmax* be the third number in that order.
7. Set *x* to a uniform random integer in the interval [0, *maxv*−*minv*). If *x* < *midmin*−*minv*, set *dir* to 0 (the left side of the sum density). Otherwise, if *x* > *midmax*−*minv*, set *dir* to 1 (the right side of the sum density). Otherwise, do the following:
    1. Set *s* to *minv* + *x*.
    2. Create a new uniform PSRN, *ret*. If *s* is less than 0, set *s* to abs(1 + *s*) and set *ret*'s sign to negative. Otherwise, set *ret*'s sign to positive.
    3. Transfer the *digitcount* least significant digits of *s* to *ret*'s fractional part. (Note that *ret*'s fractional part stores digits from most to least significant.) Then set *ret*'s integer part to floor(*s*/*base*^{digitcount}), then return *ret*. (For example, if *base* is 10, *digitcount* is 3, and *s* is 34297, then *ret*'s fractional part is set to [2, 9, 7], and *ret*'s integer part is set to 34.)
8. If *dir* is 0 (the left side), set *pw* to *x* and *b* to *midmin*−*minv*. Otherwise (the right side), set *pw* to *x*−(*midmax*−*minv*) and *b* to *maxv*−*midmax*.
9. Set *newdigits* to 0, then set *y* to a uniform random integer in the interval [0, *b*).
10. If *dir* is 0, set *lower* to *pw*. Otherwise, set *lower* to *b*−1−*pw*.
11. If *y* is less than *lower*, do the following:
    1. Set *s* to *minv* if *dir* is 0, or *midmax* otherwise, then set *s* to *s**(*base*^{newdigits}) + *pw*.
    2. Create a new uniform PSRN, *ret*. If *s* is less than 0, set *s* to abs(1 + *s*) and set *ret*'s sign to negative. Otherwise, set *ret*'s sign to positive.
    3. Transfer the *digitcount* + *newdigits* least significant digits of *s* to *ret*'s fractional part, then set *ret*'s integer part to floor(*s*/*base*$^{digitcount\ +\ newdigits}$), then return *ret*.
12. If *y* is greater than *lower* + 1, go to step 7. (This is a rejection event.)
13. Multiply *pw*, *y*, and *b* each by *base*, then add a digit chosen uniformly at random to *pw*, then add a digit chosen uniformly at random to *y*, then add 1 to *newdigits*, then go to step 10.

The following algorithm shows how to add a uniform PSRN (**a**) and a rational number **b**. The input PSRN may be negative or non-negative, and it is assumed that its integer part

and sign were sampled. Likewise, the rational number may be negative or non-negative. *Python code implementing this algorithm is given later in this document.*

1. Let *ai* be **a**'s integer part. Special cases:
    - If **a** is non-negative and has no sampled digits in its fractional part, and if **b** is an integer 0 or greater, return a uniform PSRN with a non-negative sign, an integer part equal to *ai* + **b**, and an empty fractional part.
    - If **a** is negative and has no sampled digits in its fractional part, and if **b** is an integer less than 0, return a uniform PSRN with a negative sign, an integer part equal to *ai* + abs(**b**), and an empty fractional part.
    - If **a** is non-negative, has an integer part of 0, and has no sampled digits in its fractional part, and if **b** is an integer, return a uniform PSRN with an integer part equal to abs(**b**), and an empty fractional part. The PSRN's sign is negative if **b** is less than 0, and non-negative otherwise.
    - If **b** is 0, return a copy of **a**.
2. If **a** has unsampled digits before the last sampled digit in its fractional part, set each of those unsampled digits to a digit chosen uniformly at random. Now, let *digitcount* be the number of digits in **a**'s fractional part.
3. Let *asign* be −1 if **a** is negative or 1 otherwise. Let *base* be the base of digits stored in **a**'s fractional part (such as 2 for binary or 10 for decimal). Set *absfrac* to abs(**b**), then set *fraction* to *absfrac* − floor(*absfrac*).
4. Let *afp* be the digits of **a**'s *fractional part*. (For example, if **a** represents the number 83.12344..., *afp* is 12344.) Let *asign* be −1 if
5. Set *ddc* to $base^{dcount}$, then set *lower* to ((*afp***asign*)/*ddc*)+**b** (using rational arithmetic), then set *upper* to (((*afp*+1)**asign*)/*ddc*)+**b** (again using rational arithmetic). Set *minv* to min(*lower*, *upper*), and set *maxv* to min(*lower*, *upper*).
6. Set *newdigits* to 0, then set *b* to 1, then set *ddc* to $base^{dcount}$, then set *mind* to floor(abs(*minv***ddc*)), then set *maxd* to floor(abs(*maxv***ddc*)). (Outer bounds): Then set *rvstart* to *mind*−1 if *minv* is less than 0, or *mind* otherwise, then set *rvend* to *maxd* if *maxv* is less than 0, or *maxd*+1 otherwise.
7. Set *rv* to a uniform random integer in the interval [0, *rvend*−*rvstart*), then set *rvs* to *rv* + *rvstart*.
8. (Inner bounds.) Set *innerstart* to *mind* if *minv* is less than 0, or *mind*+1 otherwise, then set *innerend* to *maxd*−1 if *maxv* is less than 0, or *maxd* otherwise.
9. If *rvs* is greater than *innerstart* and less than *innerend*, then the algorithm is almost done, so do the following:
    1. Create an empty uniform PSRN, call it *ret*. If *rvs* is less than 0, set *rvs* to abs(1 + *rvs*) and set *ret*'s sign to negative. Otherwise, set *ret*'s sign to positive.
    2. Transfer the *digitcount* + *newdigits* least significant digits of *rvs* to *ret*'s fractional part, then set *ret*'s integer part to floor($rvs/base^{digitcount + newdigits}$), then return *ret*.
10. If *rvs* is equal to or less than *innerstart* and (*rvs*+1)/*ddc* (calculated using rational arithmetic) is less than or equal to *minv*, go to step 6. (This is a rejection event.)
11. If *rvs*/*ddc* (calculated using rational arithmetic) is greater than or equal to *maxv*, go to step 6. (This is a rejection event.)
12. Add 1 to *newdigits*, then multiply *ddc*, *rvstart*, *rv*, and *rvend* each by *base*, then set *mind* to floor(abs(*minv***ddc*)), then set *maxd* to floor(abs(*maxv***ddc*)), then add a digit chosen uniformly at random to *rv*, then set *rvs* to *rv*+*rvstart*, then go to step 8.

8.3 Multiplication

The following algorithm shows how to multiply two uniform PSRNs (**a** and **b**) that store digits of the same base (radix) in their fractional parts, and get a uniform PSRN as a result. The input PSRNs may be negative or non-negative, and it is assumed that their

integer parts and signs were sampled. *Python code implementing this algorithm is given later in this document.*

1. If **a** has unsampled digits before the last sampled digit in its fractional part, set each of those unsampled digits to a digit chosen uniformly at random. Do the same for **b**.
2. If **a** has fewer digits in its fractional part than **b** (or vice versa), sample enough digits (by setting them to uniform random digits, such as unbiased random bits if **a** and **b** store binary, or base-2, digits) so that both PSRNs' fractional parts have the same number of digits.
3. To simplify matters: If **a** has no digits in its fractional part, append a digit chosen uniformly at random to that fractional part. Do the same for **b**.
4. Let *afp* be the digits of **a**'s *fractional part*, and let *bfp* be the digits of **b**'s *fractional part*. (For example, if **a** represents the number 83.12344..., *afp* is 12344.) Let *digitcount* be the number of digits in **a**'s fractional part.
5. Calculate *n1* = *afp***bfp*, *n2* = *afp**(*bfp*+1), *n3* = (*afp*+1)**bfp*, and *n4* = (*afp*+1)*(*bfp*+1).
6. Set *minv* to the minimum and *maxv* to the maximum of the four numbers just calculated. Set *midmin* to min(*n2*, *n3*) and *midmax* to max(*n2*, *n3*). The numbers *minv* and *maxv* are lower and upper bounds to the result of applying interval multiplication to the PSRNs **a** and **b**. (For example, if **a** is 0.12344... and **b** is 0.38925..., their fractional parts are added to form **c** = 0.51269...., or the interval [0.51269, 0.51271].) However, the resulting PSRN is not uniformly distributed in its interval; in the case of multiplication the distribution resembles a trapezoid whose domain is the interval [*minv*, *maxv*] and whose top is delimited by *midmin* and *midmax*.
7. Create a new uniform PSRN, *ret*. If **a** is negative and **b** is negative, or vice versa, set *ret*'s sign to negative. Otherwise, set *ret*'s sign to non-negative.
8. Set *z* to a uniform random integer in the interval [0, *maxv*−*minv*).
9. If *z* is less than *midmin*−*minv*, we will sample from the left side of the trapezoid. In this case, do the following:
    1. Set *x* to *z*, then set *newdigits* to 0, then set *b* to *midmin*−*minv*, then set *y* to a uniform random integer in the interval [0, *b*).
    2. If *y* is less than *x*, the algorithm succeeds, so do the following:
        1. Set *s* to *minv***base*^{newdigits} + *x* (where *base* is the base of digits stored by **a** and **b**, such as 2 for binary or 10 for decimal).
        2. Transfer the (*n**2 + *newdigits*) least significant digits of *s* to *ret*'s fractional part, where *n* is the number of digits in **a**'s fractional part. (Note that *ret*'s fractional part stores digits from most to least significant.) Then set *ret*'s integer part to floor(*s*/*base*$^{n*2 + newdigits}$). (For example, if *base* is 10, (*n**2 + *newdigits*) is 4, and *s* is 342978, then *ret*'s fractional part is set to [2, 9, 7, 8], and *ret*'s integer part is set to 34.) Finally, return *ret*.
    3. If *y* is greater than *x* + 1, abort these substeps and go to step 8. (This is a rejection event.)
    4. Multiply *x*, *y*, and *b* each by *base*, then add a digit chosen uniformly at random to *x*, then add a digit chosen uniformly at random to *y*, then add 1 to *newdigits*, then go to the second substep.
10. If *z* is greater than or equal to *midmax*−*minv*, we will sample from the right side of the trapezoid. In this case, do the following:
    1. Set *x* to *z*−(*midmax*−*minv*), then set *newdigits* to 0, then set *b* to *maxv*−*midmax*, then set *y* to a uniform random integer in the interval [0, *b*).
    2. If *y* is less than *b*−1−*x*, the algorithm succeeds, so do the following: Set *s* to *midmax***base*^{newdigits} + *x*, then transfer the (*n**2+ *newdigits*) least significant digits of *s* to *ret*'s fractional part, then set *ret*'s integer part to floor(*s*/*base*$^{n*2 + newdigits}$), then return *ret*.

    3. If y is greater than $(b-1-x)+1$, abort these substeps and go to step 8. (This is a rejection event.)
    4. Multiply x, y, and b each by *base*, then add a digit chosen uniformly at random to x, then add a digit chosen uniformly at random to y, then add 1 to *newdigits*, then go to the second substep.
11. If we reach here, we have reached the middle part of the trapezoid, which is flat and uniform, so no rejection is necessary. Set s to $minv + z$, then transfer the (n*2) least significant digits of s to *ret*'s fractional part, then set *ret*'s integer part to floor($s/base^{n*2}$), then return *ret*.

The following algorithm shows how to multiply a uniform PSRN (**a**) by a rational number **b**. The input PSRN may be negative or non-negative, and it is assumed that its integer part and sign were sampled. *Python code implementing this algorithm is given later in this document.*

1. If **a** has unsampled digits before the last sampled digit in its fractional part, set each of those unsampled digits to a digit chosen uniformly at random. Now, let *digitcount* be the number of digits in **a**'s fractional part.
2. To simplify matters: If **a** has no digits in its fractional part, append a digit chosen uniformly at random to that fractional part.
3. Create a uniform PSRN, call it *ret*. Set *ret*'s sign to be −1 if **a** is non-negative and **b** is less than 0 or if **a** is negative and **b** is 0 or greater, or 1 otherwise, then set *ret*'s integer part to 0. Let *base* be the base of digits stored in **a**'s fractional part (such as 2 for binary or 10 for decimal). Set *absfrac* to abs(**b**), then set *fraction* to *absfrac* − floor(*absfrac*).
4. Let *afp* be the digits of **a**'s *fractional part*. (For example, if **a** represents the number 83.12344..., *afp* is 12344.)
5. Set *dcount* to *digitcount*, then set *ddc* to $base^{dcount}$, then set *lower* to (*afp*/*ddc*)**absfrac* (using rational arithmetic), then set *upper* to (*afp*/*ddc*)**absfrac* (again using rational arithmetic).
6. Set *rv* to a uniform random integer in the interval [floor(*lower***ddc*), floor(*lower***ddc*)).
7. Set *rvlower* to *rv*/*ddc* (as a rational number), then set *rvupper* to (*rv*+1)/*ddc* (as a rational number).
8. If *rvlower* is greater than or equal to *lower* and *rvupper* is less than *upper*, then the algorithm is almost done, so do the following: Transfer the *dcount* least significant digits of *rv* to *ret*'s fractional part (note that *ret*'s fractional part stores digits from most to least significant), then set *ret*'s integer part to floor($rv/base^{dcount}$), then return *ret*. (For example, if *base* is 10, (*dcount*) is 4, and *rv* is 342978, then *ret*'s fractional part is set to [2, 9, 7, 8], and *ret*'s integer part is set to 34.)
9. If *rvlower* is greater than *upper* or if *rvupper* is less than *lower*, go to step 5.
10. Multiply *rv* and *ddc* each by *base*, then add 1 to *dcount*, then add a digit chosen uniformly at random to *rv*, then go to step 8.

8.4 Using the Arithmetic Algorithms

The algorithms given above for addition and multiplication are useful for scaling and shifting PSRNs. For example, they can transform a normally-distributed PSRN into one with an arbitrary mean and standard deviation (by first multiplying the PSRN by the standard deviation, then adding the mean). Here is a sketch of a procedure that achieves this, given two parameters, *location* and *scale*, that are both rational numbers.

1. Generate a uniform PSRN, then transform it into a random number of the desired distribution via an algorithm that employs rejection from the uniform distribution

(such as Karney's algorithm for the standard normal distribution (Karney 2014)[1])). This procedure won't work for exponential PSRNs (e-rands).
2. Run the algorithm to multiply the uniform PSRN by the rational parameter *scale* to get a new uniform PSRN.
3. Run the algorithm to add the new uniform PSRN and the rational parameter *location* to get a third uniform PSRN. Return this third PSRN.

9 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.

9.1 SampleGeometricBag

The algorithm **SampleGeometricBag** returns 1 with a probability built up by a uniform PSRN. (Flajolet et al., 2010)[7] described an algorithm for the base-2 (binary) case, but that algorithm is difficult to apply to other digit bases. Thus the following is a general version of the algorithm for any digit base.

1. Set i to 0, and set **b** to a uniform PSRN with a positive sign and an integer part of 0.
2. If the item at position i of the input PSRN's fractional part is unsampled (that is, not set to a digit), set the item at that position to a digit chosen uniformly at random, increasing the fractional part's capacity as necessary (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**.
3. Let da be the digit at position i of the input PSRN's fractional part, and let db be the corresponding digit for **b**. Return 0 if da is less than db, or 1 if da is greater than db.
4. Add 1 to i and go to step 2.

For base 2, the following **SampleGeometricBag** algorithm can be used, which is closer to the one given in the Flajolet paper:

1. Set N to 0.
2. With probability 1/2, go to the next step. Otherwise, add 1 to N and repeat this step. (When the algorithm moves to the next step, N is a geometric(1/2) random number.)
3. If the item at position N in the uniform PSRN's fractional part (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 fractional part's capacity as necessary. (As a result of this step, there may be "gaps" in the uniform PSRN where no digit was sampled yet.)
4. Return the item at position N.

For more on why these two algorithms are equivalent, see the appendix.

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 $1 - U$.

9.2 FillGeometricBag

FillGeometricBag takes a uniform PSRN 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 in the uniform PSRN's fractional part is unsampled, set the item there to to a digit chosen uniformly at random (e.g., either 0 or 1 for binary), increasing the fractional part's capacity as necessary. (Positions start at 0 where 0 is the most significant digit after the point, 1 is the next, etc. See also (Oberhoff 2018, sec. 8)[**(12)**.)
2. Let `sign` be -1 if the PSRN is negative, or 1 otherwise; let `ipart` be the PSRN's integer part; and let `bag` be the PSRN's fractional part. Take the first `p` digits of `bag` and return `sign` * (`ipart` + $\Sigma_{i=0,\ ...,\ p-1}$ bag[i] * b^{-i-1}), where b is the base, or radix.

After step 2, if it somehow happens that digits beyond `p` in the PSRN's fractional part were already sampled (that is, they were already set to a digit), 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. (For example, if `p` is 4, b is 10, and the PSRN is 0.3437500... or 0.3438500..., it could use a round-to-nearest mode to round the PSRN to the number 0.3438 or 0.3439, respectively; because this a PSRN with an "infinite" but unsampled digit expansion, there is no tie-breaking such as "ties to even" applied here.)

9.3 kthsmallest

The **kthsmallest** method generates the 'k'th smallest 'bitcount'-digit uniform random number out of 'n' of them (also known as the 'n'th *order statistic*), 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)**(18)**.

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` uniform PSRNs with positive sign and an integer part of 0.
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] PSRNs, a 1 digit to the next **v**[1] PSRNs, and so on to appending a $b - 1$ digit to the last **v**[$b - 1$] PSRNs (for the binary case, this means appending a 0 bit to the first `LC` PSRNs and a 1 bit to the next `n - LC` PSRNs).
    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.)

9.4 Power-of-Uniform Sub-Algorithm

The power-of-uniform sub-algorithm is used for certain cases of the beta sampler below. 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 PDFs**, which in turn is similar to the inversion-rejection algorithm in (Devroye 1986, ch. 7, sec. 4.4)**(18)**. 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 currently only supports generating a PSRN with base-2 (binary) digits in its fractional part.

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 1 or 0 otherwise (or return a uniform PSRN with a positive sign, an integer part of 0, and a fractional part 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 a positive-sign zero-integer-part uniform PSRN, then create a *geobag* input coin that returns the result of **SampleGeometricBag** on that PSRN.
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 PSRN's fractional part. 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 $\epsilon = py/(px * 2^i)$ (which

represents a), thus returning 1 with probability a/b. If the call returns 1, the PSRN was accepted, so do the following:
    1. If the *complement* flag is 1, make each zero-digit in the PSRN's fractional part a one-digit and vice versa.
    2. Either return the PSRN as is or fill the unsampled digits of the PSRN's fractional part 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 PSRN's fractional part, then go to step 7.

10 Algorithms for the Beta and Exponential Distributions

10.1 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 positive-sign zero-integer-part uniform PSRN.
    - If a and b are both integers, return the result of **kthsmallest** with `n = a - b + 1` and `k = a`
    - In the binary case, if a is 1 and b is less than 1, call the **power-of-uniform sub-algorithm** described below, with $px/py = 1/b$, and the *complement* flag set to 1, and return the result of that algorithm as is (without filling it as described in substep 7.2 of that algorithm).
    - In the binary case, if b is 1 and a is less than 1, call the **power-of-uniform sub-algorithm** described below, with $px/py = 1/a$, and the *complement* flag set to 0, and return the result of that algorithm as is (without filling it as described in substep 7.2 of that algorithm).
2. If $a > 2$ and $b > 2$, do the following steps, which split a and b into two parts that are faster to simulate (and implement the generalized rejection strategy in (Devroye 1986, top of page 47)[18]):
    1. Set *aintpart* to floor(a) − 1, set *bintpart* to floor(b) − 1, set *arest* to a − *aintpart*, and set *brest* to b − *bintpart*.
    2. Run this algorithm recursively, but with a = *aintpart* and b = *bintpart*. Set *bag* to the PSRN created by the run.
    3. Create an input coin *geobag* that returns the result of **SampleGeometricBag** using the given PSRN. Create another input coin *geobagcomp* that returns the result of **SampleGeometricBagComplement** using the given PSRN.
    4. Call the **algorithm for $\lambda^{x/y}$**, described in "**Bernoulli Factory Algorithms**", using the *geobag* input coin and x/y = *arest*/1, then call the same algorithm using the *geobagcomp* input coin and x/y = *brest*/1. If both calls return 1, return *bag*. Otherwise, go to substep 2.
3. Create an positive-sign zero-integer-part uniform PSRN. Create an input coin *geobag* that returns the result of **SampleGeometricBag** using the given PSRN. Create another input coin *geobagcomp* that returns the result of

SampleGeometricBagComplement using the given PSRN.

4. Remove all digits from the PSRN's fractional part. This will result in an "empty" uniform(0, 1) 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.
5. 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 4.
6. 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 4. (Note that this step and the previous step 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.)
7. U was accepted, so return the result of **FillGeometricBag**.

Once a PSRN is accepted by the steps above, either return the PSRN as is or fill the unsampled digits of the PSRN's fractional part with uniform random digits as necessary to give the number a p-digit fractional part (similarly to **FillGeometricBag**), then return the resulting number.

Notes:

- A beta(1/x, 1) random number is the same as a uniform random number raised to the power of x.
- For the beta distribution bigger `alpha` or `beta` is, the smaller the area of acceptance becomes (and the more likely random numbers get rejected by steps 5 and 6, raising its run-time). This is because `max(u^(alpha-1)*(1-u)^(beta-1))`, the peak of the PDF, approaches 0 as the parameters get bigger. To deal with this, step 2 was included, which under certain circumstances breaks the PDF into two parts that are relatively trivial to sample (in terms of bit complexity).

10.2 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. (Because exponential random numbers are always non-negative, the e-rand's sign is implicitly positive).

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 1 if **a** turns out to be less than **b**, or 0 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 1 if **a**'s integer part is less than **b**'s, or 0 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. (For example, if the implementation stores the binary expansion as a packed integer and a size, the implementation can shift the packed integer by 1, add the result of the algorithm to that integer, then add 1 to the size.) Do the same for **b**.
5. Return 1 if **a**'s fractional part is less than **b**'s, or 0 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**.

11 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 _urand_to_geobag(bag):
  return [(bag[0]>>(bag[1]-1-i))&1 for i in range(bag[1])]

def _power_of_uniform_greaterthan1(bern, power, complement=False, precision=53):
```

```
    return bern.fill_geometric_bag(
        _power_of_uniform_greaterthan1_geobag(bern, power, complement), precision
    )

def _power_of_uniform_greaterthan1_geobag(bern, power, complement=False, precision=53):
   if power<1:
     raise ValueError("Not supported")
   if power==1:
       return []  # Empty uniform random number
   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:
      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
          return bag

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)

    return b.fill_geometric_bag(
        betadist_geobag(b, ax, ay, bx, by), precision
    )

def betadist_geobag(b, ax=1, ay=1, bx=1, by=1):
    """ Generates a beta-distributed random number with arbitrary
          (user-defined) precision.  Currently, this sampler only works if (ax/ay) and
          (bx/by) are both 1 or greater, or if one of these parameters is
         1 and the other is less than 1.
         - b: Bernoulli object (from the "bernoulli" module).
         - ax, ay: Numerator and denominator of first shape parameter.
         - bx, by: Numerator and denominator of second shape parameter.
         - precision: Number of bits after the point that the result will contain.
        """
    # Beta distribution for alpha>=1 and beta>=1
    bag = []
    afrac=(Fraction(ax) if ay==1 else Fraction(ax, ay))
    bfrac=(Fraction(bx) if by==1 else Fraction(bx, by))
```

```
    bpower = bfrac - 1
    apower = afrac - 1
    # Special case for a=b=1
    if bpower == 0 and apower == 0:
        return bag
    # Special case if a=1
    if apower == 0 and bpower < 0:
        return _power_of_uniform_greaterthan1_geobag(b, Fraction(by, bx), True)
    # Special case if b=1
    if bpower == 0 and apower < 0:
        return _power_of_uniform_greaterthan1_geobag(b, Fraction(ay, ax), False)
    if apower <= -1 or bpower <= -1:
        raise ValueError
    # Special case if a and b are integers
    if int(bpower) == bpower and int(apower) == apower:
        a = int(afrac)
        b = int(bfrac)
        return _urand_to_geobag(randomgen.RandomGen().kthsmallest_urand(a + b - 1, a))
    # Split a and b into two parts which are relatively trivial to simulate
    if bfrac > 2 and afrac > 2:
        bintpart = int(bfrac) - 1
        aintpart = int(afrac) - 1
        brest = bfrac - bintpart
        arest = afrac - aintpart
        # Generalized rejection method, p. 47
        while True:
           bag = betadist_geobag(b, aintpart, 1, bintpart, 1)
           gb = lambda: b.geometric_bag(bag)
           gbcomp = lambda: b.geometric_bag(bag) ^ 1
           if (b.power(gbcomp, brest)==1 and \
              b.power(gb, arest)==1):
              return bag
    # 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
    bp1=lambda: (1 if b.power(gbcomp, bpower)==1 and \
           b.power(gb, apower)==1 else 0)
    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()
        if bp1() == 1:
            # Accepted
            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

<h1>Example of use</h1>

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

In the following Python code, `multiply_psrns` and `add_psrns` are methods to generate the result of multiplying or adding two uniform PSRNs, respectively.

```
def multiply_psrns(psrn1, psrn2, digits=2):
    """ Multiplies two uniform partially-sampled random numbers.
        psrn1: List containing the sign, integer part, and fractional part
            of the first PSRN.  Fractional part is a list of digits
            after the point, starting with the first.
        psrn2: List containing the sign, integer part, and fractional part
            of the second PSRN.
        digits: Digit base of PSRNs' digits.  Default is 2, or binary. """
    if psrn1[0] == None or psrn1[1] == None or psrn2[0] == None or psrn2[1] == None:
        raise ValueError
    for i in range(len(psrn1[2])):
        psrn1[2][i] = (
            random.randint(0, digits - 1) if psrn1[2][i] == None else psrn1[2][i]
        )
    for i in range(len(psrn2[2])):
        psrn2[2][i] = (
            random.randint(0, digits - 1) if psrn2[2][i] == None else psrn2[2][i]
        )
    while len(psrn1[2]) < len(psrn2[2]):
        psrn1[2].append(random.randint(0, digits - 1))
    while len(psrn1[2]) > len(psrn2[2]):
        psrn2[2].append(random.randint(0, digits - 1))
    digitcount = len(psrn1[2])
    if digitcount == 0:
        # Make sure fractional part has at least one digit, to simplify matters
        psrn1[2].append(random.randint(0, digits - 1))
        psrn2[2].append(random.randint(0, digits - 1))
        digitcount += 1
    if len(psrn2[2]) != digitcount:
        raise ValueError
    # Perform multiplication
    frac1 = psrn1[1]
    frac2 = psrn2[1]
    for i in range(digitcount):
        frac1 = frac1 * digits + psrn1[2][i]
    for i in range(digitcount):
        frac2 = frac2 * digits + psrn2[2][i]
    small = frac1 * frac2
    mid1 = frac1 * (frac2 + 1)
    mid2 = (frac1 + 1) * frac2
    large = (frac1 + 1) * (frac2 + 1)
    midmin = min(mid1, mid2)
```

```
midmax = max(mid1, mid2)
cpsrn = [1, 0, [0 for i in range(digitcount * 2)]]
cpsrn[0] = psrn1[0] * psrn2[0]
while True:
    rv = random.randint(0, large - small - 1)
    if rv < midmin - small:
        # Left side of product density; rising triangular
        pw = rv
        newdigits = 0
        b = midmin - small
        y = random.randint(0, b - 1)
        while True:
            if y < pw:
                # Success
                sret = small * (digits ** newdigits) + pw
                for i in range(digitcount * 2 + newdigits):
                    idx = (digitcount * 2 + newdigits) - 1 - i
                    while idx >= len(cpsrn[2]):
                        cpsrn[2].append(None)
                    cpsrn[2][idx] = sret % digits
                    sret //= digits
                cpsrn[1] = sret
                return cpsrn
            elif y > pw + 1:  # Greater than upper bound
                # Rejected
                break
            pw = pw * digits + random.randint(0, digits - 1)
            y = y * digits + random.randint(0, digits - 1)
            b *= digits
            newdigits += 1
    elif rv >= midmax - small:
        # Right side of product density; falling triangular
        pw = rv - (midmax - small)
        newdigits = 0
        b = large - midmax
        y = random.randint(0, b - 1)
        while True:
            lowerbound = b - 1 - pw
            if y < lowerbound:
                # Success
                sret = midmax * (digits ** newdigits) + pw
                for i in range(digitcount * 2 + newdigits):
                    idx = (digitcount * 2 + newdigits) - 1 - i
                    while idx >= len(cpsrn[2]):
                        cpsrn[2].append(None)
                    cpsrn[2][idx] = sret % digits
                    sret //= digits
                cpsrn[1] = sret
                return cpsrn
            elif y > lowerbound + 1:  # Greater than upper bound
                # Rejected
                break
            pw = pw * digits + random.randint(0, digits - 1)
            y = y * digits + random.randint(0, digits - 1)
            b *= digits
            newdigits += 1
    else:
        # Middle, or uniform, part of product density
        sret = small + rv
        for i in range(digitcount * 2):
            cpsrn[2][digitcount * 2 - 1 - i] = sret % digits
            sret //= digits
```

```
            cpsrn[1] = sret
            return cpsrn

def multiply_psrn_by_fraction(psrn1, fraction, digits=2):
    """ Multiplies a partially-sampled random number by a fraction.
        psrn1: List containing the sign, integer part, and fractional part
            of the first PSRN.  Fractional part is a list of digits
            after the point, starting with the first.
        fraction: Fraction to multiply by.
        digits: Digit base of PSRNs' digits.  Default is 2, or binary. """
    if psrn1[0] == None or psrn1[1] == None:
        raise ValueError
    fraction = Fraction(fraction)
    for i in range(len(psrn1[2])):
        psrn1[2][i] = (
            random.randint(0, digits - 1) if psrn1[2][i] == None else psrn1[2][i]
        )
    digitcount = len(psrn1[2])
    if digitcount == 0:
        # Make sure fractional part has at least one digit, to simplify matters
        psrn1[2].append(random.randint(0, digits - 1))
        digitcount += 1
    # Perform multiplication
    frac1 = psrn1[1]
    fracsign = -1 if fraction < 0 else 1
    absfrac = abs(fraction)
    for i in range(digitcount):
        frac1 = frac1 * digits + psrn1[2][i]
    # Result is "inexact", and "small" expresses a lower bound
    while True:
        dcount = digitcount
        ddc = digits ** dcount
        small1 = Fraction(frac1, ddc) * absfrac
        large1 = Fraction(frac1 + 1, ddc) * absfrac
        dc = int(small1 * ddc)
        dc2 = int(large1 * ddc) + 1
        rv = random.randint(dc, dc2 - 1)
        while True:
            rvsmall = Fraction(rv, ddc)
            rvlarge = Fraction(rv + 1, ddc)
            if rvsmall >= small1 and rvlarge < large1:
                cpsrn = [1, 0, [0 for i in range(dcount)]]
                cpsrn[0] = psrn1[0] * fracsign
                sret = rv
                for i in range(dcount):
                    cpsrn[2][dcount - 1 - i] = sret % digits
                    sret //= digits
                cpsrn[1] = sret
                return cpsrn
            elif rvsmall > large1 or rvlarge < small1:
                break
            else:
                rv = rv * digits + random.randint(0, digits - 1)
                dcount += 1
                ddc *= digits

def add_psrns(psrn1, psrn2, digits=2):
    """ Adds two uniform partially-sampled random numbers.
        psrn1: List containing the sign, integer part, and fractional part
            of the first PSRN.  Fractional part is a list of digits
            after the point, starting with the first.
        psrn2: List containing the sign, integer part, and fractional part
```

```
        of the second PSRN.
    digits: Digit base of PSRNs' digits.  Default is 2, or binary. """
if psrn1[0] == None or psrn1[1] == None or psrn2[0] == None or psrn2[1] == None:
    raise ValueError
for i in range(len(psrn1[2])):
    psrn1[2][i] = (
        random.randint(0, digits - 1) if psrn1[2][i] == None else psrn1[2][i]
    )
for i in range(len(psrn2[2])):
    psrn2[2][i] = (
        random.randint(0, digits - 1) if psrn2[2][i] == None else psrn2[2][i]
    )
while len(psrn1[2]) < len(psrn2[2]):
    psrn1[2].append(random.randint(0, digits - 1))
while len(psrn1[2]) > len(psrn2[2]):
    psrn2[2].append(random.randint(0, digits - 1))
digitcount = len(psrn1[2])
if len(psrn2[2]) != digitcount:
    raise ValueError
# Perform addition
frac1 = psrn1[1]
frac2 = psrn2[1]
for i in range(digitcount):
    frac1 = frac1 * digits + psrn1[2][i]
for i in range(digitcount):
    frac2 = frac2 * digits + psrn2[2][i]
small = frac1 * psrn1[0] + frac2 * psrn2[0]
mid1 = frac1 * psrn1[0] + (frac2 + 1) * psrn2[0]
mid2 = (frac1 + 1) * psrn1[0] + frac2 * psrn2[0]
large = (frac1 + 1) * psrn1[0] + (frac2 + 1) * psrn2[0]
minv = min(small, mid1, mid2, large)
maxv = max(small, mid1, mid2, large)
# Difference is expected to be a multiple of two
if abs(maxv - minv) % 2 != 0:
    raise ValueError
vs = [small, mid1, mid2, large]
vs.sort()
midmin = vs[1]
midmax = vs[2]
while True:
    rv = random.randint(0, maxv - minv - 1)
    if rv < 0:
        raise ValueError
    side = 0
    start = minv
    if rv < midmin - minv:
        # Left side of sum density; rising triangular
        side = 0
        start = minv
    elif rv >= midmax - minv:
        # Right side of sum density; falling triangular
        side = 1
        start = midmax
    else:
        # Middle, or uniform, part of sum density
        sret = minv + rv
        cpsrn = [1, 0, [0 for i in range(digitcount)]]
        if sret < 0:
            sret += 1
            cpsrn[0] = -1
        sret = abs(sret)
        for i in range(digitcount):
```

```
                cpsrn[2][digitcount - 1 - i] = sret % digits
                sret //= digits
            cpsrn[1] = sret
            return cpsrn
        if side == 0:  # Left side
            pw = rv
            b = midmin - minv
        else:
            pw = rv - (midmax - minv)
            b = maxv - midmax
        newdigits = 0
        y = random.randint(0, b - 1)
        while True:
            lowerbound = pw if side == 0 else b - 1 - pw
            if y < lowerbound:
                # Success
                sret = start * (digits ** newdigits) + pw
                cpsrn = [1, 0, [0 for i in range(digitcount + newdigits)]]
                if sret < 0:
                    sret += 1
                    cpsrn[0] = -1
                sret = abs(sret)
                for i in range(digitcount + newdigits):
                    idx = (digitcount + newdigits) - 1 - i
                    while idx >= len(cpsrn[2]):
                        cpsrn[2].append(None)
                    cpsrn[2][idx] = sret % digits
                    sret //= digits
                cpsrn[1] = sret
                return cpsrn
            elif y > lowerbound + 1:  # Greater than upper bound
                # Rejected
                break
            pw = pw * digits + random.randint(0, digits - 1)
            y = y * digits + random.randint(0, digits - 1)
            b *= digits
            newdigits += 1

def add_psrn_and_fraction(psrn, fraction, digits=2):
    if psrn[0] == None or psrn[1] == None:
        raise ValueError
    fraction = Fraction(fraction)
    fracsign = -1 if fraction < 0 else 1
    absfrac = abs(fraction)
    origfrac = fraction
    isinteger = absfrac.numerator == absfrac.denominator
    # Special cases
    # positive+pos. integer or negative+neg. integer
    if ((fracsign < 0) == (psrn[0] < 0)) and isinteger and len(psrn[2]) == 0:
        return [fracsign, psrn[1] + int(absfrac), []]
    # PSRN has no fractional part, fraction is integer
    if isinteger and psrn[0] == 1 and psrn[1]==0 and len(psrn[2]) == 0:
        return [fracsign, int(absfrac), []]
    if fraction == 0:  # Special case of 0
        return [psrn[0], psrn[1], [x for x in psrn[2]]]
    # End special cases
    for i in range(len(psrn[2])):
        psrn[2][i] = random.randint(0, digits - 1) if psrn[2][i] == None else psrn[2][i]
    digitcount = len(psrn[2])
    # Perform addition
    frac1 = psrn[1]
    frac2 = int(absfrac)
```

```
fraction = absfrac - frac2
for i in range(digitcount):
    frac1 = frac1 * digits + psrn[2][i]
for i in range(digitcount):
    digit = int(fraction * digits)
    fraction = (fraction * digits) - digit
    frac2 = frac2 * digits + digit
ddc = digits ** digitcount
small = Fraction(frac1 * psrn[0], ddc) + origfrac
large = Fraction((frac1 + 1) * psrn[0], ddc) + origfrac
minv = min(small, large)
maxv = max(small, large)
while True:
    newdigits = 0
    b = 1
    ddc = digits ** digitcount
    mind = int(minv * ddc)
    maxd = int(maxv * ddc)
    rvstart = mind - 1 if minv < 0 else mind
    rvend = maxd if maxv < 0 else maxd + 1
    rv = random.randint(0, rvend - rvstart - 1)
    rvs = rv + rvstart
    if rvs >= rvend:
        raise ValueError
    while True:
        rvstartbound = mind if minv < 0 else mind + 1
        rvendbound = maxd - 1 if maxv < 0 else maxd
        if newdigits > 50:
            return None
        if rvs > rvstartbound and rvs < rvendbound:
            sret = rvs
            cpsrn = [1, 0, [0 for i in range(digitcount + newdigits)]]
            if sret < 0:
                sret += 1
                cpsrn[0] = -1
            sret = abs(sret)
            for i in range(digitcount + newdigits):
                idx = (digitcount + newdigits) - 1 - i
                cpsrn[2][idx] = sret % digits
                sret //= digits
            cpsrn[1] = sret
            return cpsrn
        elif rvs <= rvstartbound:
            rvd = Fraction(rvs + 1, ddc)
            if rvd <= minv:
                # Rejected
                break
            else:
                # print(["rvd",rv+rvstart,float(rvd),float(minv)])
                newdigits += 1
                ddc *= digits
                rvstart *= digits
                rvend *= digits
                mind = int(minv * ddc)
                maxd = int(maxv * ddc)
                rv = rv * digits + random.randint(0, digits - 1)
                rvs = rv + rvstart
        else:
            rvd = Fraction(rvs, ddc)
            if rvd >= maxv:
                # Rejected
                break
```

```
    else:
        newdigits += 1
        ddc *= digits
        rvstart *= digits
        rvend *= digits
        mind = int(minv * ddc)
        maxd = int(maxv * ddc)
        rv = rv * digits + random.randint(0, digits - 1)
        rvs = rv + rvstart
```

11.1 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.[(19)].

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.

12 Correctness Testing

12.1 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.

12.2 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

12.3 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))
```

13 Accurate Simulation of Continuous Distributions Supported on 0 to 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 uniform PSRN with a positive sign, an integer part of 0, and an empty fractional part. Create a **SampleGeometricBag** Bernoulli factory that uses that PSRN.

2. As the PSRN builds up a uniform random number, accept the PSRN with a probability that can be represented by a Bernoulli factory algorithm (that takes the **SampleGeometricBag** factory from step 1 as part of its input), or reject it

otherwise. (A number of these algorithms can be found in "**Bernoulli Factory Algorithms**".) Let $f(U)$ be the probability function modeled by this Bernoulli factory, where U is the uniform random number built up by the PSRN. f is a multiple of the PDF for the underlying continuous distribution (as a result, this algorithm can be used even if the distribution's PDF 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 set in [0, 1]—

- is continuous everywhere,
- does not go to 0 or 1 exponentially fast in value, 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 * λ as a probability function that cannot be represented by a Bernoulli factory. Notice that the probability can be a constant, including an irrational number; see "**Algorithms for Irrational Constants**" for ways to simulate constant probabilities.

3. If the PSRN is accepted, either return the PSRN as is or fill the unsampled digits of the PSRN's fractional part 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, where x is the number in [0, 1] that is built up by the PSRN.
2. Multiply the resulting random number or PSRN by b^i, then add m (this step is relatively trivial given that the PSRN's fractional part stores base-b digits).
3. If the random number (rather than the PSRN that holds it) 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).

13.1 An Example: The Continuous Bernoulli

Distribution

The continuous Bernoulli distribution (Loaiza-Ganem and Cunningham 2019)[20] 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 a positive-sign zero-integer-part uniform PSRN.
2. Create a **complementary lambda Bernoulli factory** that returns 1 minus the result of the input coin.
3. Remove all digits from the uniform PSRN's fractional part. This will result in an "empty" uniform(0,1) 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
```

14 Complexity

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

14.1 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) [(21)]. (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 PDF.

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.

14.2 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)[22]. 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 partially-sampled random number's fractional part with random bits. The complexity here depends on the number of times **SampleGeometricBag** is called for the same PSRN, 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)[23], but not in terms of its bit complexity. The special case of $\gamma = x/y = 0$ requires no bits. If γ is an integer greater than 1, then the bit complexity is the same as that of sampling a random number G, where G is γ or the number of successes before the first failure, whichever is less, and where a success has probility $\exp(-1)$.

- 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 $\gamma = 1$), which is the expected value of G as 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 $\exp(\gamma)$ in (Canonne et al. 2020)[23], times
  - the bit complexity for each such call (which is generally 2, but is lower in the case of $\gamma = 1$, 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.

15 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 $\lambda = w$, call it a key, and
- choosing the item with the smallest key

(see also (Efraimidis 2015)[24]). 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.

16 Open Questions

There are some open questions on PSRNs:

1. Are there constructions for PSRNs other than for cases given earlier in this document?
2. Doing an arithmetic operation between two PSRNs is akin to doing an interval operation between those PSRNs, since a PSRN is ultimately a random number that lies in an interval. However, as explained in "**Arithmetic with PSRNs**", the result of the operation is an interval that bounds a random number that is *not* always uniformly distributed in that interval. For example, in the case of addition this distribution is triangular with a peak in the middle, and in the case of multiplication this distribution resembles a trapezoid. What are the exact distributions of this kind

for other interval arithmetic operations, such as division?

17 Acknowledgments

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

18 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**
- **More Algorithms for Arbitrary-Precision Sampling**
- **Testing PRNGs for High-Quality Randomness**
- **Examples of High-Quality PRNGs**

19 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) As noted by von Neumann (1951), a uniform random number bounded by 0 and 1 can be produced by "juxtapos[ing] enough random binary digits". In this sense, the random number is `RNDINTEXC(2)/2 + RNDINTEXC(2)/4 + RNDINTEXC(2)/8 + ...` (where the `2` in `RNDINTEXC(2)` is the digit base 2), perhaps "forc[ing] the last [random bit] to be 1" "[t]o avoid any bias". It is not hard to see that this approach can be applied to generate any digit expansion of any base, not just 2.
- (11) Yusong Du, Baoying Fan, and Baodian Wei, "**An Improved Exact Sampling Algorithm for the Standard Normal Distribution**", arXiv:2008.03855 [cs.DS], 2020.
- (12) Oberhoff, Sebastian, "**Exact Sampling and Prefix Distributions**", *Theses and Dissertations*, University of Wisconsin Milwaukee, 2018.
- (13) S. Kakutani, "On equivalence of infinite product measures", *Annals of Mathematics* 1948.
- (14) Brassard, G., Devroye, L., Gravel, C., "Remote Sampling with Applications to General Entanglement

Simulation", *Entropy* 2019(21)(92), doi:10.3390/e21010092.

- [15] 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.
- [16] 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.
- [17] Rohatgi, V.K., 1976. An Introduction to Probability Theory Mathematical Statistics.
- [18] Devroye, L., ***Non-Uniform Random Variate Generation***, 1986.
- [19] 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.
- [20] Loaiza-Ganem, G., Cunningham, J.P., "**The continuous Bernoulli: fixing a pervasive error in variational autoencoders**", arXiv:1907.06845v5 [stat.ML], 2019.
- [21] 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.
- [22] Lumbroso, J., "**Optimal Discrete Uniform Generation from Coin Flips, and Applications**", arXiv:1304.1916 [cs.DS].
- [23] Canonne, C., Kamath, G., Steinke, T., "**The Discrete Gaussian for Differential Privacy**", arXiv:2004.00010v2 [cs.DS], 2020.
- [24] Efraimidis, P. "**Weighted Random Sampling over Data Streams**", arXiv:1012.0256v2 [cs.DS], 2015.
- [25] Devroye, L., Gravel, C., "**The expected bit complexity of the von Neumann rejection algorithm**", arXiv:1511.02273 [cs.IT], 2016.
- [26] This means that every zero-volume (measure-zero) subset of the distribution's domain (such as a set of points) has zero probability.

20 Appendix

20.1 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))
```

20.2 Equivalence of SampleGeometricBag Algorithms

For the **SampleGeometricBag**, there are two versions: one for binary (base 2) and one for other bases. Here is why these two versions are equivalent in the binary case. Step 2 of the first algorithm samples a temporary random number N. This can be implemented by generating unbiased random bits (that is, each bit is either 0 or 1, chosen with equal probability) until a zero is generated this way. There are three cases relevant here.

- The generated bit is one, which will occur at a 50% chance. This means the bit position is skipped and the algorithm moves on to the next position. In algorithm 3, this corresponds to moving to step 3 because **a**'s fractional part is equal to **b**'s, which likewise occurs at a 50% chance compared to the fractional parts being unequal (since **a** is fully built up in the course of the algorithm).
- The generated bit is zero, and the algorithm samples (or retrieves) a zero bit at position N, which will occur at a 25% chance. In algorithm 3, this corresponds to returning 0 because **a**'s fractional part is less than **b**'s, which will occur with the same probability.
- The generated bit is zero, and the algorithm samples (or retrieves) a one bit at position N, which will occur at a 25% chance. In algorithm 3, this corresponds to returning 1 because **a**'s fractional part is greater than **b**'s, which will occur with the same probability.

20.3 Oberhoff's "Exact Rejection Sampling" Method

The following describes an algorithm described by Oberhoff for sampling a continuous distribution supported on the interval [0, 1], as long as its probability function is continuous almost everywhere and bounded from above (Oberhoff 2018, section 3)[(12)], see also (Devroye and Gravel 2016)[(25)]. (Note that if the probability function's domain is wider than [0, 1], then the function needs to be divided into one-unit-long pieces, one piece chosen at random with probability proportional to its area, and that piece shifted so that it lies in [0, 1] rather than its usual place; see Oberhoff pp. 11-12.)

1. Set *pdfmax* to an upper bound of the probability function on the domain at [0, 1]. Let *base* be the base, or radix, of the digits in the return value (such as 2 for binary or 10 for decimal).
2. Set *prefix* to 0 and *prefixLength* to 0.
3. Set *y* to a uniform random number in the interval [0, *pdfmax*].
4. Let *pw* be $base^{-prefixLength}$. Set *lower* and *upper* to a lower or upper bound, respectively, of the probability function's value on the domain at [*prefix* * *pw*, *prefix* * *pw* + *pw*].
5. If *y* turns out to be greater than *upper*, the prefix was rejected, so go to step 2.
6. If *y* turns out to be less than *lower*, the prefix was accepted. Now do the following:
    1. While *prefixLength* is less than the desired precision, set *prefix* to *prefix* * *base* + *r*, where *r* is a uniform random digit, then add 1 to *prefixLength*.
    2. Return $prefix * base^{-prefixLength}$. (If *prefixLength* is somehow greater than the desired precision, then the algorithm could choose to round the return value to a number whose fractional part has the desired number of digits, with a rounding mode of choice.)
7. Set *prefix* to *prefix* * *base* + *r*, where *r* is a uniform random digit, then add 1 to *prefixLength*, then go to step 4.

Because this algorithm requires evaluating the probability function and finding its maximum and minimum values at an interval (which often requires floating-point arithmetic and is often not trivial), this algorithm appears here in the appendix rather than in the main text. Moreover, there is additional approximation error from generating *y* with a fixed number of digits, unless *y* is a uniform PSRN (see also "**Application to Weighted Reservoir Sampling**").

Oberhoff also describes *prefix distributions* that sample a box that covers the probability function, with probability proportional to the box's area, but these distributions will have to support a fixed maximum prefix length and so will only approximate the underlying continuous distribution.

20.4 Setting Digits by Digit Probabilities

In principle, a partially-sampled random number is possible by finding a sequence of digit probabilities and setting that number's digits according to those probabilities. However, there seem to be limits on how practical this approach is.

The following is part of Kakutani's theorem (Kakutani 1948)[13]: Let a_j be the j^{th} binary digit probability in a random number's binary expansion, where the random number is in [0, 1] and each digit is independently set. Then the random number's distribution is *absolutely continuous*[26] if and only if the sum of squares of ($a_j - 1/2$) converges. In other words, the random number's bits become less and less biased as they move farther and farther from the binary point.

An absolutely continuous distribution can thus be built if we can find a sequence a_j that converges to 1/2. Then a random number could be formed by setting each of its digits to 1 with probability equal to the corresponding a_j. However, experiments show that the resulting distribution will have a discontinuous *PDF*, except if the sequence has the form —

- $a_j = y^{w/\beta^j}/(1 + y^{w/\beta^j})$,

where $\beta = 2$, $y > 0$, and $w > 0$, and special cases include the uniform distribution ($y = 1$, $w = 1$), the truncated exponential(1) distribution ($y = (1/\exp(1))$, $w = 1$; (Devroye and Gravel 2015)[3]), and the more general exponential(λ) distribution ($y = (1/\exp(1))$, $w = \lambda$). Other sequences of the form $z(j)/(1 + z(j))$ will generally result in a discontinuous PDF even if $z(j)$ converges to 1.

For reference, the following calculates the relative probability for x for a given sequence, where x is in [0, 1), and plotting this probability function (which is proportional to the PDF) will often show whether the function is discontinuous:

- Let b_j be the j^{th} base-β digit after the point (e.g., `rem(floor(x*pow(beta, j)), beta)` where `beta` = β).
- Let $t(x) = \Pi_{j = 1, 2, \ldots} b_j * a_j + (1 - b_j) * (1 - a_j)$.
- The relative probability for x is $t(x)$ / ($\text{argmax}_z\, t(z)$).

It appears that the distribution's PDF will be continuous only if—

- the probabilities of the first half, interval (0, 1/2), are proportional to those of the second half, interval (1/2, 1), and
- the probabilities of each quarter, eighth, etc. are proportional to those of every other quarter, eighth, etc.

It may be that something similar applies for β other than 2 (non-base-2 or non-binary cases) as it does to β = 2 (the base-2 or binary case).

21 License