Partially-Sampled Random Numbers for Accurate Sampling of Continuous Distributions

This version of the document is dated 2023-03-18.

Peter Occil

2020 Mathematics Subject Classification: 68W20, 60-08, 60-04.

1 Introduction

This page introduces a implementation of *partially-sampled random numbers* (PSRNs) in the Python programming language. 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 many 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 2016)[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 variates using random bits (Flajolet and Saheb 1982)[2], (Karney 2016)[3], (Devroye and Gravel 2020)[4], (Thomas and Luk 2008)[5], but most if not all of them don't deal with generating exponential PSRNs using an arbitrary rate, not just 1. (Habibizad Navin et al., 2010)[6] 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)[7] (Flajolet et al., 2010)[8], which can simulate a new probability given a

coin that shows heads with an unknown probability. 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 whose value no computer can compute exactly (such as `sqrt(p)` 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.

My audience for this article is **computer programmers with mathematics knowledge, but little or no familiarity with calculus**.

I encourage readers to implement any of the algorithms given in this page, and report their implementation experiences. In particular, **I seek comments on the following aspects**:

- Are the algorithms in this article easy to implement? Is each algorithm written so that someone could write code for that algorithm after reading the article?
- Does this article have errors that should be corrected?
- Are there ways to make this article more useful to the target audience?

Comments on other aspects of this document are welcome.

2 Contents

- **Introduction**
  - **About This Document**
- **Contents**
- **About the Beta Distribution**
- **About the Exponential Distribution**
- **About Partially-Sampled Random Numbers**

    - **Uniform Partially-Sampled Random Numbers**
    - **Exponential Partially-Sampled Random Numbers**
    - **Other Distributions**
    - **Properties**
    - **Limitations**
    - **Relation to Constructive Reals**
- **Sampling Uniform and Exponential PSRNs**
    - **Sampling Uniform PSRNs**
    - **Sampling E-rands**
- **Arithmetic and Comparisons with PSRNs**
    - **Addition and Subtraction**
    - **Multiplication**
    - **Reciprocal and Division**
    - **Using the Arithmetic Algorithms**
    - **Comparisons**
    - **Discussion**
- **Building Blocks**
    - **SampleGeometricBag**
    - **FillGeometricBag**
    - **kthsmallest**
    - **Power-of-Uniform Sub-Algorithm**
- **Algorithms for the Beta and Exponential Distributions**
    - **Beta Distribution**
    - **Exponential Distribution**
- **Sampler Code**
- **Correctness Testing**
    - **Beta Sampler**
    - **ExpRandFill**
    - **ExpRandLess**
- **Accurate Simulation of Continuous Distributions**
    - **General Arbitrary-Precision Samplers**
        - **Uniform Distribution Inside N-Dimensional Shapes**
        - **Building an Arbitrary-Precision Sampler**
        - **Continuous Distributions Supported on 0 to 1**
    - **Specific Arbitrary-Precision Samplers**
        - **Rayleigh Distribution**
        - **Hyperbolic Secant Distribution**
        - **Sum of Uniform Random Variates**
        - **Ratio of Two Uniform Random Variates**
        - **Reciprocal of Uniform Random Variate**

        - **Reciprocal of Power of Uniform Random Variate**
        - **Distribution of $U/(1-U)$**
        - **Arc-Cosine Distribution**
        - **Logistic Distribution**
        - **Cauchy Distribution**
        - **Exponential Distribution with Unknown Small Rate**
        - **Exponential Distribution with Rate ln(x)**
        - **Lindley Distribution and Lindley-Like Mixtures**
        - **Gamma Distribution**
        - **One-Dimensional Epanechnikov Kernel**
        - **Uniform Distribution Inside Rectellipse**
        - **Tulap distribution**
        - **Continuous Bernoulli Distribution**
- **Complexity**
    - **General Principles**
    - **Complexity of Specific Algorithms**
- **Application to Weighted Reservoir Sampling**
- **Acknowledgments**
- **Other Documents**
- **Notes**
- **Appendix**
    - **Equivalence of SampleGeometricBag Algorithms**
    - **UniformMultiply Algorithm**
    - **Uniform of Uniforms Produces a Product of Uniforms**
    - **Oberhoff's "Exact Rejection Sampling" Method**
    - **Probability Transformations**
    - **Ratio of Uniforms**
    - **Setting Digits by Digit Probabilities**
- **License**

3 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.

4 About the Exponential Distribution

The *exponential distribution* takes a parameter λ. Informally speaking, a random variate 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 variate is commonly generated as follows: `-ln(1 - X) / lamda`, where `X` is a uniformly-distributed random real number in the interval (0, 1). (This particular algorithm, however, is not robust in practice, for reasons that are outside the scope of this document, but see (Pedersen 2018)[9].) This page presents an alternative way to sample exponential random variates.

5 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 variate that "exactly" follows a probability 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 (for example, by using the `None`

keyword in the Python programming language, 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* (for example, *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 variate 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 the integer part is not stored, it's assumed to be 0. If the sign is not stored, it's assumed to be positive. For example, an implementation can care only about PSRNs in the interval [0, 1] by storing only a fractional part.

PSRNs ultimately represent a random variate between two numbers; one of the variate'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 variate 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.

5.1 Uniform Partially-Sampled Random Numbers

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

In this document, a **uniform PSRN** is a PSRN that represents a uniform random variate in a given interval. For example, if the PSRN is 3.356..., then it represents a uniformly distributed random variate in the interval [3.356, 3.357]. A uniform PSRN has the property that each additional digit of its fractional part (in this example, `.356...`) is sampled simply by setting it to an independent uniform random digit, an observation that dates from von Neumann (1951)[10] in the binary case.[11]

- Flajolet et al. (2010)[12] 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 (2016)[13] 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.

A uniform PSRN remains a uniform PSRN even if it was generated using a non-uniform random sampling algorithm (such as Karney's algorithm for the normal distribution).

5.2 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 variates (Karney 2016)[14]) samples each bit that, when combined with the existing bits, results in an exponentially-distributed random variate of the given rate. Also, because `-ln(1 - X)`, where `X` is a uniform random variate between 0 and 1, is exponentially distributed, e-rands can also represent the natural logarithm of a partially-sampled uniform random variate 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 variate in a given interval.)

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

5.3 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 2016)[15] 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)[16].
- Sampling uniform distributions in [0, *n*) (not just [0, 1]), is described later in "**Sampling Uniform PSRNs**".)

For all these distributions, the PSRN's unsampled trailing digits converge to the uniform distribution, and this also applies to any continuous distribution with a continuous probability density function (or, more generally, to so-called "absolutely continuous"[17] distributions) (Oberhoff 2018)[18], (Hill and Schürger 2005, Corollary 4.4)[19].

PSRNs could also be implemented via rejection from the exponential distribution.

5.4 Properties

An algorithm that samples from a non-discrete distribution[20] using PSRNs has the following properties:

1. The algorithm relies only on a source of independent and unbiased random bits for randomness.
2. The algorithm does not rely on floating-point arithmetic or fixed-precision approximations of irrational numbers or transcendental functions. (The algorithm may calculate approximations that converge to an irrational number, as long as those approximations are rational numbers of arbitrary precision. However, the more implicitly the algorithm works with irrational numbers or transcendental functions, the better.)

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 algorithm's ideal distribution by a distance of not more than $1/(b^m)$, 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 2020)[21] suggests Wasserstein distance, or "earth-mover distance", as the distance to use for this purpose.)
5. After the algorithm outputs a PSRN, if the algorithm's caller fills the PSRN's unsampled fractional digits at random (for example, uniformly at random in the case of a uniform PSRN), so that the PSRN's fractional part's first m digits are sampled, the PSRN's distribution must remain close to the algorithm's ideal distribution by a distance of not more than $1/(b^m)$.

> **Example**: Suppose an algorithm samples from a normal distribution using base-2 uniform PSRNs. If it outputs a PSRN whose fractional part has three sampled bits (and no unsampled bits before the rightmost sampled bit), the PSRN's distribution must be within a distance of—
>
> - $1/(2^3)$ to the ideal normal distribution at the time the PSRN is output, and
> - $1/(2^6)$ to the ideal normal distribution if the caller later inserts three more uniform random bits to the end of the PSRN's fractional part.
>
> **Notes:**
>
> 1. It is not easy to turn a sampler for a non-discrete distribution into an algorithm that meets these properties. Some reasons for this are given in the section "**Discussion**" later in this document.
> 2. The *exact rejection sampling* algorithm described by Oberhoff (2018)[22] produces samples that act like PSRNs. However, in general, the algorithm doesn't have the properties described in this section because some of its operations can introduce numerical error unless care is taken, and these operations include calculating minimums and maximums of probabilities. 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)[23]). For completeness, Oberhoff's method appears in the appendix.

5.5 Limitations

Because a PSRN stores a random variate in a certain interval, PSRNs are not well suited for representing numbers in zero-volume sets. Such sets include:

- Sets of integers or rational numbers.
- Sets of individual points.
- Curves on two- or higher-dimensional real number space.
- Surfaces on three- or higher-dimensional real number space.

In the case of curves and surfaces, a PSRN can't directly store the coordinates, in space, of a random point on that curve or surface (because the exact value of those coordinates may be an irrational number that no computer can store, and no interval can bound those exact coordinates "tightly" enough), but the PSRN *can* store upper and lower bounds that indirectly give that point's position on that curve or surface.

> **Examples:**
>
> 1. To represent a point on the edge of a circle, a PSRN can store a random variate in the interval [0, 2*π), via the **RandUniformFromReal** method, given later, for 2*π (for example, it can store an integer part of 2 and a fractional part of [1, 3, 5] and thus represent a number in the interval [2.135, 2.136]), and the number stored this way indicates the distance on the circular arc relative to its starting position. A program that cares about the point's X and Y coordinates can then generate enough digits of the PSRN to compute an approximation of cos(P) and sin(P), respectively, to the desired accuracy, where P is the number stored by the PSRN. (However, the direct use of mathematical functions such as `cos` and `sin` is outside the scope of this document.)
> 2. Example 1 is trivial, because each point on the interval maps evenly to a point on the circle. But this is not true in general:

> an interval's or box's points don't map evenly to points on a curve or surface in general. For example, take two PSRNs describing the U and V coordinates of a 3 dimensional cone's surface: [1.135, 1.136] for U and [0.288, 0.289] for V, and the cone's coordinates are X = U*cos(V), Y = U*sin(V), Z = U. In this example, the PSRNs form a box that's mapped to a small part of the cone surface. However, the points in the box don't map to the cone evenly this way, so generating enough digits to calculate X, Y, and Z to the desired accuracy will not sample uniformly from that part of the cone without more work (see Williamson (1987)[24] for one solution).

5.6 Relation to Constructive Reals

Partially-sampled random numbers are related to a body of work dealing with so-called "constructive reals" or "recursive reals", or operations on real numbers that compute an approximation of the exact result to a user-specified number of digit places. For example, in Hans-J. Boehm's implementation (Boehm 2020)[25], (Boehm 1987)[26], each operation on "constructive reals" (such as addition, multiplication, `exp`, `ln`, and so on) is associated with a function `f(n)` (where `n` is usually 0 or greater) that returns an integer `m` such that `abs(m/pow(2, n) - x) < 1/pow(2, n)`, where `x` is the exact result of the operation. In addition, comparisons such as "less than" or "greater than" can operate on "constructive reals". As suggested in Goubault-Larrecq et al. (2021)[27], there can also be an operation that samples the digits of a uniform random variate between 0 and 1 and gives access to approximations of that variate, sampling random digits as necessary. Similarly, operations of this kind can be defined to access approximations of the value stored in a PSRN (including a uniform or exponential PSRN), sampling digits for the PSRN as necessary.

Details on "constructive real" operations and comparisons are outside the scope of this document.

6 Sampling Uniform and Exponential PSRNs

6.1 Sampling Uniform PSRNs

There are several 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**) with probability 1. 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 (for example, 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. (We now set **a**'s integer part and sign.) 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 (note that bi is included). 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 (such as an unbiased random bit if β is 2). (Positions start at 0 where 0 is the most significant digit after the point, 1 is the next, etc.)
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 an unbiased random bit.)
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)[28] 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 variate (**b**) and a uniform random variate between 0 and 1.
> 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 random variates between 0 and 1.

The **RandUniformInRangePositive** algorithm generates a uniformly distributed PSRN (**a**) that is greater than one nonnegative real number **bmin** and less than another positive real number **bmax** with probability 1. This algorithm works whether **bmin** or **bmax** is known to be a rational number or not (for example, either number can be the result of an expression such as `exp(-2)` or `ln(20)`), but the algorithm notes how it can be more efficiently implemented if **bmin** or **bmax** is known to be a rational number.

1. If **bmin** is greater than or equal to **bmax**, if **bmin** is less than 0, or if **bmax** is 0 or less, return an error.
2. Create an empty uniform PSRN **a**.
3. Special case: If **bmax** is 1 and **bmin** is 0, set **a**'s sign to positive, set **a**'s integer part to 0, and return **a**.
4. Special case: If **bmax** and **bmin** are rational numbers and each of their denominators is a power of β, including 1 (where β is the desired digit base, or radix, of the uniform PSRN, such as 10 for decimal or 2 for binary), then do the following:
    1. Let *denom* be **bmax**'s or **bmin**'s denominator, whichever is greater.

2. Set *c1* to floor(**bmax*****denom*) and *c2* to floor((**bmax**−**bmin**)**denom*).
3. If *c2* is greater than 1, add to *c1* an integer chosen uniformly at random in [0, *c2*) (note that *c2* is excluded).
4. Let *d* be the base-*β* logarithm of *denom* (this is equivalent to finding the minimum number of base-*β* digits needed to store *denom* and subtracting 1). Transfer *c1*'s least significant digits to **a**'s fractional part; the variable *d* tells how many digits to transfer to each PSRN this way. Then set **a**'s sign to positive and **a**'s integer part to floor($c1/\beta^d$). (For example, if *β* is 10, *d* is 3, and *c1* is 7342, set **a**'s fractional part to [3, 4, 2] and **a**'s integer part to 7.) Finally, return **a**.

5. Calculate floor(**bmax**), and set *bmaxi* to the result. Likewise, calculate floor(**bmin**) and set *bmini* to the result.
6. If *bmini* is equal to **bmin** and *bmaxi* is equal to **bmax**, set **a**'s sign to positive and **a**'s integer part to an integer chosen uniformly at random in [*bmini*, *bmaxi*) (note that *bmaxi* is excluded), then return **a**. (It should be noted that determining whether a real number is equal to another is undecidable in general.)
7. (We now set **a**'s integer part and sign.) Set **a**'s sign to positive and **a**'s integer part to an integer chosen uniformly at random in the interval [*bmini*, *bmaxi*] (note that *bmaxi* is included). If *bmaxi* is equal to **bmax**, the integer is chosen from the interval [*bmini*, *bmaxi*−1] instead. Return **a** if—
   - **a**'s integer part is greater than *bmini* and less than *bmaxi*, or
   - *bmini* is equal to **bmin**, and **a**'s integer part is equal to *bmini* and less than *bmaxi*.
8. (We now sample **a**'s fractional part.) Set *i* to 0 and *istart* to 0. (Then, *if **bmax** is known rational:* set *bmaxf* to **bmax** minus *bmaxi*, and *if **bmin** is known rational*, set *bminf* to **bmin** minus *bmini*.)
9. (This step is not crucial for correctness, but helps improve its efficiency. It sets **a**'s fractional part to the initial digits shared by **bmin** and **bmax**.) If **a**'s integer part is equal to *bmini* and *bmaxi*, then do the following in a loop: 1. Calculate the base-*β* digit at position *i* of **bmax**'s and **bmin**'s fractional parts, and set *dmax* and *dmin* to those digits, respectively. (*If **bmax** is known rational:* Do this step by setting *dmax* to floor(*bmaxf***β*) and *dmin* to floor(*bminf***β*).) 2. If *dmin* equals *dmax*, append *dmin* to **a**'s fractional part, then add 1 to *i* (and, if **bmax** and/or **bmin** is known to be rational, set *bmaxf* to *bmaxf***β*−*d* and set *bminf* to *bminf***β*−*d*). Otherwise, break from this loop and set *istart* to *i*.

10. (Ensure the fractional part is greater than **bmin**'s.) Set *i* to *istart*, then if **a**'s integer part is equal to *bmini*:
    1. Calculate the base-β digit at position *i* of **bmin**'s fractional part, and set *dmin* to that digit.
    2. 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 (such as an unbiased random bit if β is 2, or binary). (Positions start at 0 where 0 is the most significant digit after the point, 1 is the next, etc.)
    3. Let *ad* be the digit at position *i* of **a**'s fractional part. If *ad* is greater than *dmin*, abort these substeps and go to step 11.
    4. Discard **a**, create a new empty uniform PSRN **a**, and abort these substeps and go to step 7 if *ad* is less than *dmin*.
    5. Add 1 to *i* and go to the first substep.
11. (Ensure the fractional part is less than **bmax**'s.) Set *i* to *istart*, then if **a**'s integer part is equal to *bmaxi*:
    1. If *bmaxi* is 0 and not equal to **bmax**, and if **a** has no digits in its fractional part, then do the following in a loop:
        1. Calculate the base-β digit at position *i* of **bmax**'s fractional part, and set *d* to that digit. (*If **bmax** is known rational:* Do this step by setting *d* to floor(*bmaxf**β).)
        2. If *d* is 0, append a 0-digit to **a**'s fractional part, then add 1 to *i* (and, if **bmax** is known to be rational, set *bmaxf* to *bmaxf**$\beta-d$). Otherwise, break from this loop.
    2. Calculate the base-β digit at position *i* of **bmax**'s fractional part, and set *dmax* to that digit. (*If **bmax** is known rational:* Do this step by multiplying *bmaxf* by β, then setting *dmax* to floor(*bmaxf*), then subtracting *dmax* from *bmaxf*.)
    3. 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.
    4. Let *ad* be the digit at position *i* of **a**'s fractional part. Return **a** if *ad* is less than *dmax*.
    5. Discard **a**, create a new empty uniform PSRN **a**, and abort these substeps and go to step 7 if—
        - ***bmax** is not known to be rational*, and either *ad* is greater than *dmax* or all the digits after the digit at position *i* of **bmax**'s fractional part are zeros, or
        - ***bmax** is known to be rational*, and either *ad* is greater than *dmax* or *bmaxf* is 0
    6. Add 1 to *i* and go to the second substep.

12. Return **a**.

The **RandUniformInRange** algorithm generates a uniformly distributed PSRN (**a**) that is greater than one real number **bmin** and less than another real number **bmax** with probability 1. It works for both positive and negative real numbers, but it's specified separately from **RandUniformInRangePositive** to reduce clutter.

1. If **bmin** is greater than or equal to **bmax**, return an error. If **bmin** and **bmax** are each 0 or greater, return the result of **RandUniformInRangePositive**.
2. If **bmin** and **bmax** are each 0 or less, call **RandUniformInRangePositive** with **bmin** = abs(**bmax**) and **bmax** = abs(**bmin**), set the result's fractional part to negative, and return the result.
3. (At this point, **bmin** is less than 0 and **bmax** is greater than 0.) Set *bmaxi* to either floor(**bmax**) if **bmax** is 0 or greater, or −ceil(abs(**bmax**)) otherwise, and set *bmini* to either floor(**bmin**) if **bmin** is 0 or greater, or −ceil(abs(**bmin**)) otherwise. (Described this way to keep implementers from confusing floor with the integer part.)
4. Set *ipart* to an integer chosen uniformly at random in the interval [*bmini*, *bmaxi*] (note that *bmaxi* is included). If *bmaxi* is equal to **bmax**, the integer is chosen from the interval [*bmini*, *bmaxi*−1] instead.
5. If *ipart* is neither *bmini* nor *bmaxi*, create a uniform PSRN **a** with an empty fractional part; then set **a**'s sign to either positive if *ipart* is 0 or greater, or negative otherwise; then set **a**'s integer part to abs(*ipart*+1) if *ipart* is less than 0, or *ipart* otherwise; then return **a**.
6. If *ipart* is *bmini*, then create a uniform PSRN **a** with a positive sign, an integer part of abs(*ipart*+1), and an empty fractional part; then run **URandLessThanReal** with **a** = **a** and **b** = abs(**bmin**). If the result is 1, set **a**'s sign to negative and return **a**. Otherwise, go to step 3.
7. If *ipart* is *bmaxi*, then create a uniform PSRN **a** with a positive sign, an integer part of *ipart*, and an empty fractional part; then run **URandLessThanReal** with **a** = **a** and **b** = **bmax**. If the result is 1, return **a**. Otherwise, go to step 3.

The **RandUniformFromReal** algorithm generates a uniformly distributed PSRN (**a**) that is greater than 0 and less than a real number **b** with probability 1. It is equivalent to the **RandUniformInRangePositive** algorithm with **a** = **a**, **bmin** = 0, and **bmax** = **b**.

The **UniformComplement** algorithm generates 1 minus the value of a uniform PSRN (**a**) as follows:

1. If **a**'s sign is negative or its integer part is other than 0, return an error.
2. For each sampled digit in **a**'s fractional part, set it to $base-1-digit$, where *digit* is the digit and *base* is the base of digits stored by the PSRN, such as 2 for binary.
3. Return **a**.

6.2 Sampling E-rands

Sampling an e-rand (a exponential PSRN) makes use of two observations, based on the parameter λ of the exponential distribution (with λ greater than 0):

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

Devroye and Gravel (2020, sec. 3.8)[29] already made these observations, but only for $\lambda = 1$.

To implement these probabilities using just random bits, the sampler uses the **ExpMinus** and **LogisticExp** algorithms from "**Bernoulli Factory Algorithms**".

> **Note:** An exponential PSRN is an exponential random variate built up digit by digit, but an exponential random variate can also be stored in a *uniform PSRN* and generated by other algorithms.

7 Arithmetic and Comparisons with

PSRNs

This section describes addition, subtraction, multiplication, reciprocal, and division involving uniform PSRNs, and discusses other aspects of arithmetic involving PSRNs.

7.1 Addition and Subtraction

The following algorithm (**UniformAdd**) 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 have a positive or negative sign, 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**'s sign is negative, or 1 otherwise. Let *bsign* be −1 if **b**'s sign is negative, or 1 otherwise. Let *afp* be **a**'s integer and fractional parts packed into an integer, as explained in the example, and let *bfp* be **b**'s integer and fractional parts packed the same way. (For example, if **a** represents the number 83.12344..., *afp* is 8312344.) 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 variates.

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 (**UniformAddRational**) shows how to add a uniform PSRN (**a**) and a rational number **b**. The input PSRN may have a positive or negative sign, and it is assumed that its integer part and sign were sampled. Similarly, the rational number may be positive, negative, or zero. *Python code implementing this algorithm is given later in this document.*

1. Let *ai* be **a**'s integer part. Special cases:
    - If **a**'s sign is positive and has no sampled digits in its fractional part, and if **b** is an integer 0 or greater, return a uniform PSRN with a positive sign, an integer part equal to *ai* + **b**, and an empty fractional part.
    - If **a**'s sign 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**'s sign is positive, 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 empty fractional part. If **b** is less than 0, the PSRN's sign is negative and its integer part is abs(**b**)−1. If **b** is 0 or greater, the PSRN's sign is positive and its integer part is abs(**b**).
    - 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**'s sign 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 **a**'s integer and fractional parts packed into an integer, as explained in the example. (For example, if **a** represents the number 83.12344..., *afp* is 8312344.) 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.

7.2 Multiplication

The following algorithm (**UniformMultiplyRational**) shows how to multiply a uniform PSRN (**a**) by a nonzero rational number **b**. The input PSRN may have a positive or negative sign, 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. Create a uniform PSRN, call it *ret*. Set *ret*'s sign to be −1 if **a**'s

sign is positive and **b** is less than 0 or if **a**'s sign 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*).

3. Let *afp* be **a**'s integer and fractional parts packed into an integer, as explained in the example. (For example, if **a** represents the number 83.12344..., *afp* is 8312344.)
4. 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*+1)/*ddc*)**absfrac* (again using rational arithmetic).
5. Set *rv* to a uniform random integer in the interval [floor(*lower***ddc*), floor(*upper***ddc*)).
6. Set *rvlower* to *rv*/*ddc* (as a rational number), then set *rvupper* to (*rv*+1)/*ddc* (as a rational number).
7. 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.)
8. If *rvlower* is greater than *upper* or if *rvupper* is less than *lower*, go to step 4.
9. 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 6.

Another algorithm (**UniformMultiply**) shows how to multiply two uniform PSRNs (**a** and **b**) is given in the appendix — the algorithm is complicated and it may be simpler to instead connect PSRNs with "constructive reals" (described earlier) that implement multiplication to arbitrary precision.

> **Note:** Let $b>0$, $c\geq 0$, and $d>0$ be rational numbers where $d>c$. To generate the product of two uniform variates, one in $[0, b]$ and the other in $[c, d]$, the following algorithm can be used.
> (1) Generate a uniform PSRN using **RandUniformFromReal** with parameter $b*(d-c)$, call it **K**;
> (2) Get the result of **UniformAddRational** with parameters **K** and $b*c$, call it **M**;
> (3) Generate a uniform PSRN using **RandUniform** with

> parameter **M**; return the PSRN.
> Broadly speaking: "generate a uniform(0, b*(d−c)) random variate X, then return a uniform(0, X+b*c) random variate". See the **appendix** for evidence that this algorithm works, at least when c = 0.

7.3 Reciprocal and Division

The following algorithm (**UniformReciprocal**) generates 1/**a**, where **a** is a uniform PSRN, and generates a new uniform PSRN with that reciprocal. The input PSRN may have a positive or negative sign, and it is assumed that its integer part and sign were sampled. All divisions mentioned here should be done using rational arithmetic. *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. Create a uniform PSRN, call it *ret*. Set *ret*'s sign to **a**'s sign. Let *base* be the base of digits stored in **a**'s fractional part (such as 2 for binary or 10 for decimal).
3. If **a** has no non-zero digit in its fractional part, and has an integer part of 0, then append a digit chosen uniformly at random to **a**'s fractional part. If that digit is 0, repeat this step. (This step is crucial for correctness when both PSRNs' intervals cover the number 0, since the distribution of their product is different from the usual case.)
4. Let *afp* be **a**'s integer and fractional parts packed into an integer, as explained in the example. (For example, if **a** represents the number 83.12344..., *afp* is 8312344.)
5. (Calculate lower and upper bounds of 1/**a**, disregarding **a**'s sign.) Set *dcount* to *digitcount*, then set *ddc* to $base^{dcount}$, then set *lower* to (*ddc*/(*afp*+1)), then set *upper* to (*ddc*/*afp*).
6. Set *lowerdc* to floor(*lower***ddc*). If *lowerdc* is 0, add 1 to *dcount*, multiply *ddc* by *base*, then repeat this step. (This step too is important for correctness.)
7. (*rv* represents a tight interval between the lower and upper bounds, and *rv2* represents a uniform random variate between 0 and 1 to compare with the density function for the reciprocal.) Set *rv* to a uniform random integer in the interval [*lowerdc*,

floor(*upper***ddc*)). Set *rv2* to a uniform random integer in the interval [0, *lowerdc*).

8. (Get the bounds of the tight interval *rv*.) Set *rvlower* to *rv*/*ddc*, then set *rvupper* to (*rv*+1)/*ddc*.
9. If *rvlower* is greater than or equal to *lower* and *rvupper* is less than *upper*:
    1. Set *rvd* to *lowerdc*/*ddc*, then set *rvlower2* to *rv2*/*lowerdc*, then set *rvupper2* to (*rv2*+1)/*lowerdc*. (*rvlower2* and *rvupper2* are bounds of the uniform(0, 1) variate.)
    2. (Compare with upper bounded density: y^2/x^2, where y is the lower bound of 1/**a** and x is between the lower and upper bounds.) If *rvupper2* is less than (*rvd***rvd*)/(*rvupper***rvupper*), 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.)
    3. (Compare with lower bounded density.) If *rvlower2* is greater than (*rvd***rvd*)/(*rvlower***rvlower*), then abort these substeps and go to step 5. (This is a rejection event.)
10. If *rvlower* is greater than *upper* or if *rvupper* is less than *lower*, go to step 5. (This is a rejection event.)
11. Multiply *rv*, *rv2*, *lowerdc*, and *ddc* each by *base*, then add 1 to *dcount*, then add a digit chosen uniformly at random to *rv*, then add a digit chosen uniformly at random to *rv2*, then go to step 8.

With this algorithm it's now trivial to describe an algorithm for dividing one uniform PSRN **a** by another uniform PSRN **b**, here called **UniformDivide**:

1. Run the **UniformReciprocal** algorithm on **b** to create a new uniform PSRN **c**.
2. If **c**'s fractional part has no digits or all of them are zeros, append uniform random digits to **c** until a nonzero digit is appended this way.
3. Run the **UniformMultiply** algorithm (given in the appendix) on **a** and **b**, in that order, and return the result of that algorithm.

It's likewise trivial to describe an algorithm for multiplying a uniform PSRN **a** by a nonzero rational number **b**, here called **UniformDivideRational**:

1. If **b** is 0, return an error.
2. Run the **UniformMultiplyRational** algorithm on **a** and 1/**b**, in that order, and return the result of that algorithm.

7.4 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 variate 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 2016)[30])). This procedure won't work for exponential PSRNs (e-rands).
2. Run the **UniformMultiplyRational** algorithm to multiply the uniform PSRN by the rational parameter *scale* to get a new uniform PSRN.
3. Run the **UniformAddRational** algorithm to add the new uniform PSRN and the rational parameter *location* to get a third uniform PSRN. Return this third PSRN.

See also the section "Discussion" later in this article.

7.5 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** with probability 1, or 0 otherwise (see also (Karney

2016)[31])).

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**'s sign is negative and 0 if it's positive. If **a**'s sign is positive, return 1 if **a**'s integer part is less than **b**'s, or 0 if greater. If **a**'s sign 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**'s sign is positive, return 1 if *da* is less than *db*, or 0 if *da* is greater than *db*.
7. If **a**'s sign 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 an unbiased random bit.)

> **Note**: To sample the **maximum** of two uniform random variates between 0 and 1, or the **square root** of a uniform random variate between 0 and 1: (1) Generate two uniform PSRNs **a** and **b** each with a positive sign, an integer part of 0, and an empty fractional part. (2) Run **RandLess** on **a** and **b** in that order. If the call returns 0, return **a**; otherwise, return **b**.

The **RandLessThanReal** algorithm compares a PSRN **a** with a real number **b** and returns 1 if **a** turns out to be less than **b** with probability 1, 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 `ln(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**'s sign is negative and 0 if it's positive. If **a**'s sign is positive, return 1 if **a**'s integer part is less than *bi*, or 0 if greater. (Continue if both are equal.) If **a**'s sign is negative, return 0 if **a**'s integer part is less than *bi*, or 1 if greater. (Continue if both are equal.)
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**'s sign is positive,
    - *ad* is greater than *d* and **a**'s sign is negative, or
    - *ad* is equal to *d*, **a**'s sign 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** with probability 1), or
        - ***b** is known to be rational* and *bf* is 0 (indicating **a** is less than **b** with probability 1).
9. Return 0 if—
    - *ad* is less than *d* and **a**'s sign is negative,
    - *ad* is greater than *d* and **a**'s sign is positive, or
    - *ad* is equal to *d*, **a**'s sign is positive, 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** with probability 1), or
        - ***b** is known to be rational* and *bf* is 0 (indicating **a** is greater than **b** with probability 1).

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)[32]):

- (6.) Calculate bp, which is an approximation to **b** such that abs(**b** − bp) <= β^{-i-1}, and such that bp has the same sign as **b**. Let bk be bp's digit expansion up to the i + 1 digits after the point (ignoring its sign). For example, if **b** is π or −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 (ignoring its sign).
- (8.) If ak <= bk − 2, return either 1 if **a**'s sign is positive or 0 otherwise.
- (9.) If ak >= bk + 1, return either 1 if **a**'s sign is negative or 0 otherwise.[33]

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 shows how to implement **URandLessThanReal** 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. Then, if num and den are both less than 0, set them to their absolute values. Then if **a**'s sign is positive, its integer part is 0, and num is 0, return 0. Then if **a**'s sign is positive, its integer part is 0, and num's sign is different from den's sign, return 0.
2. Set bs to −1 if num or den, but not both, is less than 0, or 1 otherwise, then set den to abs(den), then set bi to floor(abs(num)/den), then set num to rem(abs(num), den).
3. If **a**'s sign is different from bs's sign, return 1 if **a**'s sign is negative and 0 if it's positive. If **a**'s sign is positive, return 1 if **a**'s integer part is less than bi, or 0 if greater. (Continue if both are equal.) If **a**'s sign is negative, return 0 if **a**'s integer part is less than bi, or 1 if greater. (Continue if both are equal.) If num is 0 (indicating the fraction is an integer), return 0 if **a**'s sign is positive and 1 otherwise.
4. 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.)

5. 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.
6. Set *c* to 1 if *num* * *pt* >= *den*, and 0 otherwise.
7. Set *d2* to floor(*num* * *pt* / *den*). (In base 2, this is equivalent to setting *d2* to *c*.)
8. If *d1* is less than *d2*, return either 1 if **a**'s sign is positive, or 0 otherwise. If *d1* is greater than *d2*, return either 0 if **a**'s sign is positive, or 1 otherwise.
9. If *c* is 1, set *num* to *num* * *pt* − *den* * *d2*, then multiply *den* by *pt*.
10. If *num* is 0, return either 0 if **a**'s sign is positive, or 1 otherwise.
11. Multiply *pt* by *base*, add 1 to *i*, and go to step 5.

7.6 Discussion

This section discusses issues involving arithmetic with PSRNs.

Uniform PSRN arithmetic produces non-uniform distributions in general. As can be seen in the arithmetic algorithms earlier in this section (such as **UniformAdd** and **UniformMultiplyRational**), addition, multiplication, and other arithmetic operations with PSRNs (see also (Brassard et al., 2019)[34]) are not as trivial as adding, multiplying, etc. their integer and fractional parts. A uniform PSRN is ultimately a uniform random variate inside an interval (this is its nature), yet arithmetic on random variates does not produce a uniform distribution in general.

An example illustrates this. Say we have two uniform PSRNs: A = 0.12345... and B = 0.38901.... They represent random variates 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 variate is *not* uniformly distributed in [0.51246, 0.51248], so that simply choosing a uniform random variate in the interval won't work. (This is true in general for other arithmetic operations besides addition.) This can be demonstrated by generating many pairs of uniform random variates 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.

The example applies in general to most other math operations besides addition (including multiplication, division, `log`, `sin`, and so on): do the math operation 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.

Implementing other operations. In contrast to addition, multiplication, and division, certain other math operations are trivial to carry out in PSRNs. They include negation, as mentioned in (Karney 2016)[35], and operations affecting the PSRN's integer part only.

A promising way to connect PSRNs with other math operations (such as multiplication, `ln`, and `exp`) is to use "constructive reals" or "recursive reals". See the section "Relation to Constructive Reals", earlier.

A sampler can be created that uses the probabilities of getting each digit under the target distribution. But if the distribution is non-discrete:

- These probabilities will depend on previous digits except for a very limited class of distributions (including uniform and exponential); see the **appendix** for details.
- For distributions outside that limited class, the sampler will be *limited-precision* (not *arbitrary-precision*) in practice, since it can hold only so many digit probabilities. For example, the works (Habibizad Navin et al., 2007)[36], (Nezhad et al., 2013)[37] point to building a "tree" of such digit probabilities. [38]

Finally, arithmetic with PSRNs may be possible if the result of the arithmetic is distributed with a known probability density function (PDF), allowing for an algorithm that implements rejection from the uniform or exponential distribution. An example of this is found in the **UniformReciprocal** algorithm above. However, that PDF may have an unbounded peak, thus ruling out rejection sampling in practice. For example, if X is a uniform PSRN in the interval [0, 1], then the distribution of X^3 has the PDF `(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.

Reusing PSRNs. The arithmetic algorithms in this section may give incorrect results if the *same PSRN* is used more than once in different runs of these algorithms.

This issue happens in general when the original sampler uses the same random variate for different purposes in the algorithm (an example is "W*Y, (1−W)*Y", where W and Y are independent random variates (Devroye 1986, p. 394)[39]). In this case, if one PSRN spawns additional PSRNs (so that they become *dependent* on the first), those additional PSRNs may become inaccurate once additional digits of the first PSRN are sampled uniformly at random. (This is not always the case, but it's hard to characterize when the additional PSRNs become inaccurate this way and when not.)

This issue is easy to see for the **UniformAddRational** or **UniformMultiplyRational** algorithm when it's called more than once with the same PSRN and the same rational number: although the same random variate ought to be returned each time, in reality different variates will be generated this way with probability 1, especially when additional digits are sampled from them afterwards.

It might be believed that the issue just described could be solved by the algorithm below:

Assume we want to multiply the same PSRN by different numbers. Let vec be a vector of rational numbers to multiply the same PSRN by, and let vec[i] be the rational number at position i of the vector (positions start at 0).

1. *Set i to 0, set **a** to the input PSRN, set num to vec[i], and set 'output' to an empty list.*
2. *Set ret to the result of **UniformMultiplyRational** with the PSRN **a** and the rational number num.*
3. *Add a pointer to ret to the list 'output'. If vec[i] was the last number in the vector, stop this algorithm.*
4. *Set **a** to point to ret, then add 1 to i, then set num to vec[i]/vec[i−1], then go to step 2.*

However, even this algorithm doesn't ensure that the output PSRNs will be exactly proportional to the same random variate. An example: Let **a** be the PSRN 0.... (or the interval [0.0, 1.0]), then let **b** be the result of **UniformMultiplyRational**(**a**, 1/2), then let **c** be the result of **UniformMultiplyRational**(**b**, 1/3). One possible result for **b** is 0.41... and for **c** is 0.138.... Now we fill **a**, **b**, and **c** with uniform random bits. Thus, as one possible result, **a** is now 0.13328133..., **b** is now 0.41792367..., and **c** is now 0.13860371.... Here, however, **c** divided by **b** is not exactly 1/3, although it's close, and **b** divided by **a** is far from 1/2 (especially since **a** was very coarse to begin with). Although this example shows PSRNs with decimal digits, the situation is worse with binary digits.

8 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)[40], which generates heads or tails with a probability that is built up digit by digit.

8.1 SampleGeometricBag

The algorithm **SampleGeometricBag** returns 1 with a probability built up by a uniform PSRN's fractional part. (Flajolet et al., 2010)[41] 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. For convenience, this algorithm ignores the PSRN's integer part and sign.

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. It likewise ignores the input PSRN's integer part and sign.

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 what the Flajolet paper calls a *geometric* random variate (with parameter 1/2), hence the name "geometric bag", but the terminology "geometric random variate" is avoided in this article since it has several conflicting meanings in academic works.)
3. If the item at position N in the uniform PSRN's fractional part (positions start at 0) is not set to a digit (for example, 0 or 1 for base 2), set the item at that position to a digit chosen uniformly at random (for example, 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 $1 - U$.

8.2 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 a digit chosen uniformly at random (for example, 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)[42].)

2. Let `sign` be -1 if the PSRN's sign 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` + bag[0] * b^{-0-1} + bag[1] * b^{-1-1} + ... + bag[`p`−1] * $b^{-(p-1)-1}$), 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,
- round the number represented by the PSRN to a number whose fractional part has `p` digits, with a rounding mode of choice (and without further modifying the PSRN), and
- return the rounded number.

For example, if `p` is 4, b is 10, and the PSRN is 0.3437500... or 0.3438500..., the implementation could use a round-to-nearest mode to round the number that the PSRN represents to the number 0.3438 or 0.3439, respectively, and return the rounded number; because this is a PSRN with an "infinite" but unsampled digit expansion, there is no tie-breaking such as "ties to even" applied here.

8.3 kthsmallest

The **kthsmallest** method generates the 'k'th smallest 'bitcount'-digit uniform random variate in the interval [0, 1] 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 variate in this case is the `a`th smallest uniform random variate between 0 and 1 out of `a + b - 1` of them (Devroye 1986, p. 431)[43].

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 random variate between 0 and 1 is either less than half or greater than half with equal probability; 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`, a binomial random variate with parameters `n` and 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` + **v**[0] + **v**[1] + ... + **v**[$i - 1$] 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), then optionally fill it with uniform random digits as necessary to give its fractional part `bitcount` many digits (similarly to **FillGeometricBag** above), then return that number. (Note that the beta sampler described later chooses to fill the PSRN this way via this algorithm.)

8.4 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 variate 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)[44]. This is needed because when px/py is greater than 1, the distribution of $U^{px/py}$ has the PDF `(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 $(a/b)^z$** described in "**Bernoulli Factory Algorithms**", with parameters `a = 1, b = 2, z = py/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 variate 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 variate 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. Optionally, fill the PSRN with uniform random digits as necessary to give its fractional part n digits (similarly to **FillGeometricBag** above), where n is a precision parameter. Then, return the PSRN.
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.

9 Algorithms for the Beta and Exponential Distributions

9.1 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)[45]):
    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. Do a separate (recursive) run of this algorithm, 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 random variate between 0 and 1, U, for the following steps, which will accept U with probability $U^{\mathrm{a}-1} * (1-U)^{\mathrm{b}-1}$ (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 $U^{\mathrm{a}-1}$). 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 $(1-U)^{\mathrm{b}-1}$). 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, optionally 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 random variate with parameters $1/x$ and 1 is the same as a uniform random variate in [0, 1] raised to the power of x.
> - For the beta distribution, the bigger `alpha` or `beta` is, the smaller the area of acceptance becomes (and the greater the probability that random variates 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).

9.2 Exponential Distribution

We also have the necessary building blocks to describe how to sample e-rands. An e-rand consists of four numbers: the first is a multiple of $1/(2^k)$, the second is k, the third is the integer part (initially -1 to indicate the integer part wasn't sampled yet), and the fourth, λ, is the rate parameter of the exponential distribution ($\lambda>0$). (Because exponential random variates are always 0 or greater, the e-rand's sign is implicitly positive.) In the Python code, e-rands are as described, except λ must be a rational number and its numerator and denominator take up a parameter each.

To sample bit k after the binary point of an exponential random variate with rate λ (where $k = 1$ means the first digit after the point, $k = 2$ means the second, etc.), call the **LogisticExp** algorithm (see "**Bernoulli Factory Algorithms** with $z=\lambda$ 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 **ExpMinus** algorithm (see "**Bernoulli Factory Algorithms** with $z=\lambda$, 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 (see "**Bernoulli Factory Algorithms** with $z=\lambda$ 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 and generates a number whose fractional part has `p` digits as follows:

1. For each position i in [0, `p`), if the item at that position in the e-rand's fractional part is unsampled, call the **LogisticExp** algorithm (see "**Bernoulli Factory Algorithms** with $z=\lambda$ and $prec = i + 1$, and set the item at position i to the result (which will be either 0 or 1), increasing the fractional part's capacity as necessary. (Bit positions start at 0 where 0 is the most significant bit after the point, 1 is the next, etc. See also (Oberhoff 2018, sec. 8)[46].)
2. Let `sign` be -1 if the e-rand's sign is negative, or 1 otherwise; let `ipart` be the e-rand's integer part; and let `bag` be the PSRN's fractional part. Take the first `p` digits of `bag` and return `sign` * (`ipart` + bag[0] * 2^{-0-1} + bag[1] * 2^{-1-1} + ... + bag[`p`−1] * $2^{-(p-1)-1}$).

See the discussion in **FillGeometricBag** for advice on how to handle the case when it somehow happens that bits beyond `p` in the PSRN's fractional part were already sampled (that is, they were already set to

a digit) after step 2 of this algorithm.

Here is a third algorithm (called **ExpRand**) that generates a *uniform PSRN*, rather than an e-rand, that follows the exponential distribution. In the algorithm, the rate λ is given as a rational number greater than 0. The method is based on von Neumann's algorithm (von Neumann 1951)[47].

1. Set *recip* to $1/\lambda$, and set *highpart* to 0.
2. Set *u* to the result of **RandUniformFromReal** with the parameter *recip*.
3. Set *val* to point to the same value as *u*, and set *accept* to 1.
4. Set *v* to the result of **RandUniformFromReal** with the parameter *recip*.
5. Run the **URandLess** algorithm on *u* and *v*, in that order. If the call returns 0, set *u* to *v*, then set *accept* to 1 minus *accept*, then go to step 4.
6. If *accept* is 1, add *highpart* to *val* via the **UniformAddRational** algorithm given earlier, then return *val*.
7. Add *recip* to *highpart* and go to step 2.

The following alternative version of the previous algorithm (called **ExpRand2**) includes Karney's improvement to the von Neumann algorithm (Karney 2016)[48], namely a so-called "early rejection step". The algorithm here allows an arbitrary rate parameter (λ), given as a rational number greater than 0, unlike with the von Neumann and Karney algorithms, where λ is 1.

1. Set *recip* to $1/\lambda$, and set *highpart* to 0.
2. Set *u* to the result of **RandUniformFromReal** with the parameter *recip*.
3. Run the **URandLessThanReal** algorithm on *u* with the parameter *recip*/2. If the call returns 0, add *recip*/2 to *highpart* and go to step 2. (This is Karney's "early rejection step", where the parameter is 1/2 when λ is 1. However, Fan et al. (2019)[49] point out that the parameter 1/2 in Karney's "early rejection step" is not optimal.)
4. Set *val* to point to the same value as *u*, and set *accept* to 1.
5. Set *v* to the result of **RandUniformFromReal** with the parameter *recip*.
6. Run the **URandLess** algorithm on *u* and *v*, in that order. If the call returns 0, set *u* to *v*, then set *accept* to 1 minus *accept*, then go to step 5.

7. If *accept* is 1, add *highpart* to *val* via the **UniformAddRational** algorithm given earlier, then return *val*.
8. Add ***recip*/2** to *highpart* and go to step 2.

> **Note:** A Laplace (double exponential) random variate is then implemented by giving the PSRN returned by **ExpRand** or **ExpRand2** a random sign (with equal probability, the PSRN's sign is either positive or negative).

10 Sampler Code

The following Python code implements the beta sampler described in this document. 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 variate 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.

The Python code below supports only rational-valued λ parameters in the exponential sampler.

```
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 variate
   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 variate, 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 variate 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_psrn(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 variate (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.rndint(1)
            else:
                ret = (ret << 1) | bag[i]
        if len(bag) < precision:
            diff = precision - len(bag)
            ret = (ret << diff) | b.rndint((1 << diff) - 1)
        # Now we have a number that is a multiple of
        # 2^-precision.
        return ret / (1 << precision)

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, `add_psrns` is a method to generate the result of multiplying or adding two uniform PSRNs, respectively.

```
def psrn_reciprocal(psrn1, digits=2):
    """ Generates the reciprocal of a partially-sampled random
number.
        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.
        digits: Digit base of PSRNs' digits.  Default is 2, or
binary. """
    if psrn1[0] == None or psrn1[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]
        )
    digitcount = len(psrn1[2])
    # Perform multiplication
    frac1 = psrn1[1]
    for i in range(digitcount):
        frac1 = frac1 * digits + psrn1[2][i]
    while frac1 == 0:
        # Avoid degenerate cases
        d1 = random.randint(0, digits - 1)
        psrn1[2].append(d1)
        frac1 = frac1 * digits + d1
        digitcount += 1
    while True:
        dcount = digitcount
        ddc = digits ** dcount
        small = Fraction(ddc, frac1 + 1)
        large = Fraction(ddc, frac1)
        if small>large: raise ValueError
        if small==0: raise ValueError
        while True:
           dc = int(small * ddc)
           if dc!=0: break
           dcount+=1
           ddc*=digits
        if dc == 0:

print(["dc",dc,"dc/ddc",float(Fraction(dc,ddc)),"small",float(small),

        dc2 = int(large * ddc) + 1
        rv = random.randint(dc, dc2 - 1)
        rvx = random.randint(0, dc - 1)
        # print([count,float(small), float(large),dcount, dc/ddc,
dc2/ddc])
        while True:
            rvsmall = Fraction(rv, ddc)
            rvlarge = Fraction(rv + 1, ddc)
            if rvsmall >= small and rvlarge < large:
                rvd = Fraction(dc, ddc)
                rvxf = Fraction(rvx, dc)
```

```
                rvxf2 = Fraction(rvx + 1, dc)
                #
print(["dcs",rvx,"rvsmall",float(rvsmall),"rvlarge",float(rvlarge),"s

                #
"rvxf",float(rvxf),float(rvxf2),"rvd",float(rvd),
                #
"sl",float((rvd*rvd)/(rvlarge*rvlarge)),float((rvd*rvd)/(rvsmall*rvsr

                if rvxf2 < (rvd * rvd) / (rvlarge * rvlarge):
                    cpsrn = [1, 0, [0 for i in range(dcount)]]
                    cpsrn[0] = psrn1[0]
                    sret = rv
                    for i in range(dcount):
                        cpsrn[2][dcount - 1 - i] = sret % digits
                        sret //= digits
                    cpsrn[1] = sret
                    return cpsrn
                elif rvxf > (rvd * rvd) / (rvsmall * rvsmall):
                    break
            elif rvsmall > large or rvlarge < small:
                break
            rv = rv * digits + random.randint(0, digits - 1)
            rvx = rvx * digits + random.randint(0, digits - 1)
            dcount += 1
            ddc *= digits
            dc *= digits

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])
    # 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]
    while True:
        dcount = digitcount
        ddc = digits ** dcount
        small = Fraction(frac1, ddc) * absfrac
        large = Fraction(frac1 + 1, ddc) * absfrac
        dc = int(small * ddc)
        dc2 = int(large * ddc) + 1
        rv = random.randint(dc, dc2 - 1)
        while True:
            rvsmall = Fraction(rv, ddc)
            rvlarge = Fraction(rv + 1, ddc)
            if rvsmall >= small and rvlarge < large:
                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 > large or rvlarge < small:
                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.denominator == 1
    # 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
            and fracsign < 0
        ):
            return [fracsign, int(absfrac) - 1, []]
        if (
            isinteger
            and psrn[0] == 1
            and psrn[1] == 0
            and len(psrn[2]) == 0
            and fracsign > 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 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 Correctness Testing

11.1 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 variates generated using the sampler above. This test can be used because the beta distribution has a probability density function; independently sampled variates from the distribution are tested; and the distribution's parameters are known. 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. If p-values tend to be close to 0 (or close to 1, since this test is two-sided), then this is evidence that the samples do not come from the corresponding beta distribution.

11.2 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 variates generated using the `exprand` method above. This test can be used because the exponential distribution has a probability density function; independently sampled variates from the distribution are tested; and the distribution's parameters are known. 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. If p-values tend to be close to 0 (or close to 1, since this test is two-sided), then this is evidence that the samples do not come from the corresponding beta 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

11.3 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 variate compares as less than another; for the same λ this event should have the same probability as the event that it compares as greater. (In fact, this should be the case for *any* pair of independent random variates of the same non-degenerate distribution; see proposition 2 in my note on **randomness extraction**.) 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. If p-values tend to be close to 0, then this is evidence that the exponential random variates 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))
```

12 Accurate Simulation of Continuous Distributions

The following shows arbitrary-precision samplers (see "**Properties**") for various continuous distributions using PSRNs.

12.1 General Arbitrary-Precision Samplers

12.1.1 Uniform Distribution Inside N-Dimensional Shapes

The following is a general way to describe an arbitrary-precision sampler for generating a point uniformly at random inside a geometric shape located entirely in the hypercube [0, *d1*]×[0, *d2*]×...×[0,*dN*] in *N*-dimensional space, where *d1*, ..., *dN* are integers greater than 0. The algorithm will generally work if the shape is reasonably defined;

the technical requirements are that the shape must have a zero-volume (Lebesgue measure zero) boundary and a nonzero finite volume, and must assign zero probability to every zero-volume subset of it (such as a set of individual points).

The sampler's description has the following skeleton.

1. Generate *N* empty uniform partially-sampled random numbers (PSRNs), with a positive sign, an integer part of 0, and an empty fractional part. Call the PSRNs *p1*, *p2*, ..., *pN*.
2. Set *S* to *base*, where *base* is the base of digits to be stored by the PSRNs (such as 2 for binary or 10 for decimal). Then set *N* coordinates to 0, call the coordinates *c1*, *c2*, ..., *cN*. Then set *d* to 1. Then, for each coordinate (*c1*, ..., *cN*), set that coordinate to an integer in [0, *dX*), chosen uniformly at random, where *dX* is the corresponding dimension's size.
3. For each coordinate (*c1*, ..., *cN*), multiply that coordinate by *base* and add a digit chosen uniformly at random to that coordinate.
4. This step uses a function known as **InShape**, which takes the coordinates of a box and returns one of three values: *YES* if the box is entirely inside the shape; *NO* if the box is entirely outside the shape; and *MAYBE* if the box is partly inside and partly outside the shape, or if the function is unsure. **InShape**, as well as the divisions of the coordinates by *S*, should be implemented using rational arithmetic. Instead of dividing those coordinates this way, an implementation can pass *S* as a separate parameter to **InShape**. See the Implementation Notes for Box Shape Intersection, later in this section, for further implementation notes. In this step, run **InShape** using the current box, whose coordinates in this case are ((*c1*/*S*, *c2*/*S*, ..., *cN*/*S*), ((*c1*+1)/*S*, (*c2*+1)/*S*, ..., (*cN*+1)/*S*)).
5. If the result of **InShape** is *YES*, then the current box was accepted. If the box is accepted this way, then at this point, *c1*, *c2*, etc., will each store the *d* digits of a coordinate in the shape, expressed as a number in the interval [0, 1], or more precisely, a range of numbers. (For example, if *base* is 10, *d* is 3, and *c1* is 342, then the first coordinate is 0.342..., or more precisely, a number in the interval [0.342, 0.343].) In this case, do the following:
    1. For each coordinate (*c1*, ..., *cN*), transfer that coordinate's least significant digits to the corresponding PSRN's fractional part. The variable *d* tells how many digits to transfer to each PSRN this way. Then, for each coordinate (*c1*, ..., *cN*), set the

corresponding PSRN's integer part to floor(cX/$base^d$), where cX is that coordinate. (For example, if *base* is 10, d is 3, and $c1$ is 7342, set $p1$'s fractional part to [3, 4, 2] and $p1$'s integer part to 7.)

2. For each PSRN ($p1$, ..., pN), optionally fill that PSRN with uniform random digits as necessary to give its fractional part the desired number of digits (similarly to **FillGeometricBag**).
3. For each PSRN, optionally do the following: Generate an unbiased random bit. If that bit is 1 (which happens with probability 1/2), set that PSRN's sign to negative. (This will result in a symmetric shape in the corresponding dimension. This step can be done for some PSRNs and not others.)
4. Return the PSRNs $p1$, ..., pN, in that order.

6. If the result of **InShape** is *NO*, then the current box lies outside the shape and is rejected. In this case, go to step 2.
7. If the result of **InShape** is *MAYBE*, it is not known whether the current box lies fully inside or fully outside the shape, so multiply S by *base*, then add 1 to d, then go to step 3.

> **Notes:**
>
> 1. See Li and El Gamal (2016)[50] and Oberhoff (2018)[51] for related work on encoding random points uniformly distributed in a shape.
> 2. Rejection sampling on a shape is subject to the "curse of dimensionality", since typical shapes of high dimension will tend to cover much less volume than their bounding boxes, so that it would take a lot of time on average to accept a high-dimensional box. Moreover, the more volume the shape takes up in the bounding box, the higher the acceptance rate.
> 3. Devroye (1986, chapter 8, section 3)[52] describes grid-based methods to optimize random point generation. In this case, the space is divided into a grid of boxes each with size 1/$base^k$ in all dimensions; the result of **InShape** is calculated for each such box and that box labeled with the result; all boxes labeled *NO* are discarded; and the algorithm is modified by adding the following after step 2: "2a. Choose a precalculated box uniformly at random, then set $c1$, ..., cN to that box's coordinates, then set d to k and set S to $base^k$. If a box labeled *YES* was chosen, follow the substeps in step 5. If a box labeled *MAYBE* was chosen, multiply S by *base* and add 1 to d." (For example, if *base* is 10, k is 1, N is 2, and $d1 = d2$

= 1, the space could be divided into a 10×10 grid, made up of 100 boxes each of size (1/10)×(1/10). Then, **InShape** is precalculated for the box with coordinates ((0, 0), (1, 1)), the box ((0, 1), (1, 2)), and so on [the boxes' coordinates are stored as just given, but **InShape** instead uses those coordinates divided by $base^k$, or 10^1 in this case], each such box is labeled with the result, and boxes labeled *NO* are discarded. Finally the algorithm above is modified as just given.)

4. Besides a grid, another useful data structure is a *mapped regular paving* (Harlow et al. 2012)[53], which can be described as a binary tree with nodes each consisting of zero or two child nodes and a marking value. Start with a box that entirely covers the desired shape. Calculate **InShape** for the box. If it returns *YES* or *NO* then mark the box with *YES* or *NO*, respectively; otherwise it returns *MAYBE*, so divide the box along its first widest coordinate into two sub-boxes, set the parent box's children to those sub-boxes, then repeat this process for each sub-box (or if the nesting level is too deep, instead mark each sub-box with *MAYBE*). Then, to generate a random point (with a base-2 fractional part), start from the root, then: (1) If the box is marked *YES*, return a uniform random point between the given coordinates using the **RandUniformInRange** algorithm; or (2) if the box is marked *NO*, start over from the root; or (3) if the box is marked *MAYBE*, get the two child boxes bisected from the box, choose one of them with equal probability (for example, choose the left child if an unbiased random bit is 0, or the right child otherwise), mark the chosen child with the result of **InShape** for that child, and repeat this process with that child; or (4) the box has two child boxes, so choose one of them with equal probability and repeat this process with that child.
5. The algorithm can be adapted to return 1 with probability equal to its acceptance rate (which equals the shape's volume divided by the hyperrectangle's volume), and return 0 with the opposite probability. In this case, replace steps 5 and 6 with the following: "5. If the result of **InShape** is *YES*, return 1.; 6. If the result of **InShape** is *NO*, return 0." (I thank BruceET of the Cross Validated community for leading me to this insight.)

Implementation Notes for Box/Shape Intersection: The algorithm in this section uses a function called **InShape** to determine whether an axis-aligned box is either outside a shape, fully inside the shape, or partially inside the shape. The following are notes that will aid in developing a robust implementation of **InShape** for a particular shape, especially because the boxes being tested can be arbitrarily small.

1. **InShape**, as well as the divisions of the coordinates by S, should be implemented using rational arithmetic. Instead of dividing those coordinates this way, an implementation can pass S as a separate parameter to **InShape**.

2. If the shape is convex, and the point (0, 0, ..., 0) is on or inside that shape, **InShape** can return—

   - *YES* if all the box's corners are in the shape;
   - *NO* if none of the box's corners are in the shape and if the shape's boundary does not intersect with the box's boundary; and
   - *MAYBE* in any other case, or if the function is unsure.

   In the case of two-dimensional shapes, the shape's corners are ($c1/S$, $c2/S$), (($c1+1$)/S, $c2/S$), ($c1$,($c2+1$)/S), and (($c1+1$)/S, ($c2+1$)/S). However, checking for box/shape intersections this way is non-trivial to implement robustly, especially if interval arithmetic is not used.

3. If the shape is given as an inequality of the form $f(t1, ..., tN) \leq 0$, where f is a continuous function of one or more variables, **InShape**'s calculations should involve *rational intervals*, or objects that describe a number that's bounded by two rational numbers with numerators and denominators of arbitrary size. (See also (Duff 1992)[54]; Daumas et al. (2007)[55] give one implementation of rational intervals.) Then, **InShape** should build one interval for each dimension of the box and evaluate f using those intervals[56] with an accuracy that increases as S increases. Then, **InShape** can return—

   - *YES* if the interval result of f has an upper bound less than or equal to 0;
   - *NO* if the interval result of f has a lower bound greater than 0; and

- *MAYBE* in any other case.

For example, if f is $(t1^2+t2^2-1)$, which describes a quarter disk, **InShape** should build two intervals, namely $t1 = [c1/S, (c1+1)/S]$ and $t2 = [c2/S, (c2+1)/S]$, and evaluate $f(t1, t2)$ using interval arithmetic.

One thing to point out, though: If f calls the $\exp(x)$ function where x can potentially have a high absolute value, say 10000 or higher, the exp function can run a very long time in order to calculate proper bounds for the result, since the number of digits in $\exp(x)$ grows linearly with x. In this case, it may help to transform the inequality to its logarithmic version. For example, by applying ln(.) to each side of the inequality $y^2 \le \exp(-(x/y)^2/2)$, the inequality becomes $2*\ln(y) \le -(x/y)^2/2$ and thus becomes $2*\ln(y) + (x/y)^2/2 \le 0$ and thus becomes $4*\ln(y) + (x/y)^2 \le 0$.

4. If the shape is such that every axis-aligned line segment that begins in one face of the hypercube and ends in another face crosses the shape at most once, ignoring the segment's endpoints (an example is an axis-aligned quarter of a circular disk where the disk's center is (0, 0)), then **InShape** can return—

    - *YES* if all the box's corners are in the shape;
    - *NO* if none of the box's corners are in the shape; and
    - *MAYBE* in any other case, or if the function is unsure.

    If **InShape** uses rational interval arithmetic, it can build an interval per dimension *per corner*, evaluate the shape for each corner individually and with an accuracy that increases as S increases, and treat a corner as inside or outside the shape only if the result of the evaluation clearly indicates that. Using the example of a quarter disk, **InShape** can build eight intervals, namely an x- and y-interval for each of the four corners; evaluate (x^2+y^2-1) for each corner; and return *YES* only if all four results have upper bounds less than or equal to 0, *NO* only if all four results have lower bounds greater than 0, and *MAYBE* in any other case.

5. If **InShape** expresses a shape in the form of a ***signed distance function***, namely a function that describes the closest distance from any point in space to the shape's boundary, it can return—

    - *YES* if the signed distance (or an upper bound of such distance) at each of the box's corners, after dividing their coordinates by S, is less than or equal to $-\sigma$ (where σ is an upper bound for sqrt(N)/(S*2), such as 1/S);
    - *NO* if the signed distance (or a lower bound of such distance) at each of the box's corners is greater than σ; and
    - *MAYBE* in any other case, or if the function is unsure.

6. **InShape** implementations can also involve a shape's *implicit curve* or *algebraic curve* equation (for closed curves), its *implicit surface* equation (for closed surfaces), or its *signed distance field* (a quantized version of a signed distance function).

7. An **InShape** function can implement a set operation (such as a union, intersection, or difference) of several simpler shapes, each with its own **InShape** function. The final result depends on the set operation (such as union or intersection) as well as the result returned by each component for a given box. The following are examples of set operations:

    - For unions, the final result is *YES* if any component returns *YES*; *NO* if all components return *NO*; and *MAYBE* otherwise.
    - For intersections, the final result is *YES* if all components return *YES*; *NO* if any component returns *NO*; and *MAYBE* otherwise.
    - For differences between two shapes, the final result is *YES* if the first shape returns *YES* and the second returns *NO*; *NO* if the first shape returns *NO* or if both shapes return *YES*; and *MAYBE* otherwise.
    - For the exclusive OR of two shapes, the final result is *YES* if one shape returns *YES* and the other returns *NO*; *NO* if both shapes return *NO* or both return *YES*; and *MAYBE* otherwise.

Examples:

- The following example generates a point inside a quarter diamond (centered at (0, ..., 0), "radius" k where k is an integer greater than 0): Let $d1$, ..., dN be k. Let **InShape** return *YES* if $((c1+1) + ... + (cN+1)) < S*k$; *NO* if $(c1 + ... + cN) > S*k$; and *MAYBE* otherwise. For $N=2$, the acceptance rate (see note 5) is 1/2. For a full diamond, step 5.3 in the algorithm is done for each of the N dimensions.
- The following example generates a point inside a quarter hypersphere (centered at (0, ..., 0), radius k where k is an integer greater than 0): Let $d1$, ..., dN be k. Let **InShape** return *YES* if $((c1+1)^2 + ... + (cN+1)^2) < (S*k)^2$; *NO* if $(c1^2 + ... + cN^2) > (S*k)^2$; and *MAYBE* otherwise. For $N=2$, the acceptance rate (see note 5) is $\pi/4$. For a full hypersphere with radius 1, step 5.3 in the algorithm is done for each of the N dimensions. In the case of a 2-dimensional disk, this algorithm thus adapts the well-known rejection technique of generating X and Y coordinates until $X^2+Y^2 < 1$ (for example, Devroye (1986, p. 230 ff.)[57]).
- The following example generates a point inside a quarter *astroid* (centered at (0, ..., 0), radius k where k is an integer greater than 0): Let $d1$, ..., dN be k. Let **InShape** return *YES* if $((sk-c1-1)^2 + ... + (sk-cN-1)^2) > sk^2$; *NO* if $((sk-c1)^2 + ... + (sk-cN)^2) < sk^2$; and *MAYBE* otherwise, where $sk = S*k$. For $N=2$, the acceptance rate (see note 5) is $1 - \pi/4$. For a full astroid, step 5.3 in the algorithm is done for each of the N dimensions.

12.1.2 Building an Arbitrary-Precision Sampler

Suppose a probability distribution—

- takes on only values 0 or greater, and
- is described by a cumulative distribution function (CDF, or $CDF(x)$, or the probability of getting x or less under the distribution) and by a probability distribution (PDF) or a function proportional to the PDF.

Also, suppose that—

- the CDF and PDF or proportional function have known symbolic forms, and
- the PDF that has an infinite tail to the right, is less than or equal to

a finite number (so that *PDF(0)* is other than infinity), and is strictly decreasing.

Then it may be possible to describe an arbitrary-precision sampler (see "**Properties**") for that distribution. Such a description has the following skeleton.

1. With probability A, set *intval* to 0, then set *size* to 1, then go to step 4.
    - A is calculated as ($CDF(1) - CDF(0)$) / ($1-CDF(0)$), where CDF is the distribution's CDF. This should be found analytically using a computer algebra system such as SymPy.
    - The symbolic form of A will help determine which Bernoulli factory algorithm, if any, will simulate the probability; if a Bernoulli factory exists, it should be used.
2. Set *intval* to 1 and set *size* to 1.
3. With probability $B(size, intval)$, go to step 4. Otherwise, add *size* to *intval*, then multiply *size* by 2, then repeat this step.
    - This step chooses an interval beyond 1, and grows this interval by geometric steps, so that an appropriate interval is chosen with the correct probability.
    - The probability $B(size, intval)$ is the probability that the interval is chosen given that the previous intervals weren't chosen, and is calculated as ($CDF(size + intval) - CDF(intval)$) / ($1-CDF(intval)$). This should be found analytically using a computer algebra system such as SymPy.
    - The symbolic form of B will help determine which Bernoulli factory algorithm, if any, will simulate the probability; if a Bernoulli factory exists, it should be used.
4. Generate an integer in the interval $intval, intval + size$) uniformly at random, call it i.
5. Create *ret*, a uniform PSRN with a positive sign and an integer part of 0.
6. Create an input coin that calls **SampleGeometricBag** on the PSRN *ret*. Run a Bernoulli factory algorithm that returns 1 with probability equal to $C(i, \lambda)$, using the input coin (here, λ, the heads probability of the input coin, is the probability built up in *ret* via **SampleGeometricBag**, and satisfies $0 \le \lambda \le 1$). If the call returns 0, go to step 4.
    - The probability $C(i, \lambda)$ is calculated as $PDF(i + \lambda) / M$, where PDF is the distribution's PDF or a function proportional to the PDF, and should be found analytically using a computer algebra

system such as SymPy.
    - In this formula, M is any convenient number that satisfies $PDF(intval) \le M \le \max(1, PDF(intval))$, and should be as low as feasible. M serves to ensure that C is as close as feasible to 1 (to improve acceptance rates), but no higher than 1. The choice of M can vary for each interval (each value of $intval$, which can only be 0, 1, or a power of 2). Any such choice for M preserves the algorithm's correctness because the PDF has to be strictly decreasing and a new interval isn't chosen when λ is rejected.
    - The symbolic form of C will help determine which Bernoulli factory algorithm, if any, will simulate the probability; if a Bernoulli factory exists, it should be used.
    - This step could be implemented as follows instead: "Generate z, a uniform PSRN with a positive sign and an integer part of 0. Determine whether z is greater than $C(i, ret)$, using operations for 'constructive reals' and treating z and ret as 'constructive reals' whose fractional digits are sampled using **SampleGeometricBag** as necessary. If z is greater, go to step 4." See "[**Relation to Constructive Reals**", earlier. However, although this change is also correct, the use of "constructive reals" (or "recursive reals") here is not preferred when a Bernoulli factory is available to implement this step.
7. The PSRN ret was accepted, so set ret's integer part to i, then optionally fill ret with uniform random digits as necessary to give its fractional part the desired number of digits (similarly to **FillGeometricBag**), then return ret.

Examples of algorithms that use this skeleton are the algorithm for the **ratio of two uniform random variates**, as well as the algorithms for the Rayleigh distribution and for the reciprocal of power of uniform, both given later.

Perhaps the most difficult part of describing an arbitrary-precision sampler with this skeleton is finding the appropriate Bernoulli factory algorithm for the probabilities A, B, and C, especially when these probabilities have a non-trivial symbolic form.

> **Note:** The algorithm skeleton uses ideas similar to the inversion-rejection method described in Devroye (1986, ch. 7, sec. 4.6)[58]; an exception is that instead of generating a uniform random variate and comparing it to calculations of a CDF, this algorithm uses conditional probabilities of choosing a given piece,

> probabilities labeled A and B. This approach was taken so that the CDF of the distribution in question is never directly calculated in the course of the algorithm, which furthers the goal of sampling with arbitrary precision and without using floating-point arithmetic.

12.1.3 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 thanks to Bernoulli factories, as long as the distributions have probability density functions (PDFs) and take on only values 0 or greater and 1 or less. 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). (See also Devroye (1986, ch. 2, sec. 3.8, exercise 14)[59])

1. Create a 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 variate, 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 density function (PDF) modeled by this Bernoulli factory, where U is the uniform random variate built up by the PSRN. f has a domain equal to the open interval (0, 1) or a subset of that interval, and returns a value of [0, 1] everywhere in its domain. f is the PDF for the underlying continuous distribution, or the PDF times a (possibly unknown) constant factor. As shown by Keane and O'Brien (1994)[60], however, this step works if and only if—

    - $f(\lambda)$ is constant on its domain, or
    - $f(\lambda)$ is continuous and polynomially bounded on its domain (polynomially bounded means that both $f(\lambda)$ and $1-f(\lambda)$ are greater than or equal to $\min(\lambda^n, (1-\lambda)^n)$ for some integer n),

    and they show that $2 * \lambda$, where $0 \le \lambda < 1/2$, is one function that does not admit a Bernoulli factory. $f(\lambda)$ can be a constant, including an irrational number; see "**Algorithms for Specific Constants**"

for ways to simulate constant probabilities.

3. If the PSRN is accepted, optionally fill the PSRN with uniform random digits as necessary to give its fractional part n digits (similarly to **FillGeometricBag** above), where n is a precision parameter, then return the PSRN.

However, the speed of this algorithm depends crucially on the mode (highest point) of $f(\lambda)$ with $0 \le \lambda \le 1$.[61] As the 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 the 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 PDF 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 variates in the interval $[m, m + y]$ (where m and y are rational numbers and y is greater than 0), rather than [0, 1], as follows:

1. Apply the algorithm above, except that a modified function $f'(x) = f(x * y + m)$ is used rather than f, where x is the number in [0, 1] that is built up by the PSRN, and that the choice is not made to fill the PSRN as given in step 3 of that algorithm.
2. Multiply the resulting random PSRN by y via the second algorithm in "**Multiplication**". (Note that if y has the form b^i, this step is relatively trivial.)
3. Add m to the resulting random PSRN via the second algorithm in "**Addition and Subtraction**".

Note that here, the function f' must meet the requirements of Keane and O'Brien. (For example, take the 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.)

12.2 Specific Arbitrary-Precision Samplers

12.2.1 Rayleigh Distribution

The following is an arbitrary-precision sampler for the Rayleigh distribution with parameter s, which is a rational number greater than 0.

1. Set k to 0, and set y to 2 * s * s.
2. Run the **ExpMinus** algorithm with parameter (k * 2 + 1)/y. If it returns 1, go to step 3. Otherwise, add 1 to k and repeat this step.
3. (Now, the piece located at [k, k + 1) is sampled.) Create a uniform PSRN with a positive sign and an integer part of 0.
4. Set ky to k * k / y.
5. (Now simulating $\exp(-U^2/y)$, $\exp(-k^2/y)$, $\exp(-U*k*2/y)$, as well as a scaled-down version of U + k, where U is the number built up by the uniform PSRN.) Call the **ExpMinus** algorithm with parameter k*y, then call the **exp(−(λ*z))** with z = 1/y and λ representing a coin that does: "Run **SampleGeometricBag** on the uniform PSRN twice, and return 1 if both flips return 1, or 0 otherwise", then run the **algorithm for exp(−(λ* z))** with z= floor(k * 2 / y), and λ being a coin that does: "Run **SampleGeometricBag** on the uniform PSRN once, then return the result", then call the **sub-algorithm** given later with the uniform PSRN and k = k. If all of these calls return 1, the uniform PSRN was accepted. Otherwise, remove all digits from the uniform PSRN's fractional part and go to step 4.
6. If the uniform PSRN, call it *ret*, was accepted by step 5, set *ret*'s integer part to k, then optionally fill *ret* with uniform random digits as necessary to give its fractional part the desired number of digits (similarly to **FillGeometricBag**), and return *ret*.

The sub-algorithm below simulates a probability equal to (U+k)/$base^z$, where U is the number built by the uniform PSRN, *base* is the base (radix) of digits stored by that PSRN, k is an integer 0 or greater, and z is the number of significant digits in k (for this purpose, z is 0 if k is 0).

For base 2:

1. Set N to 0.
2. Generate an unbiased random bit. If that bit is 1 (which happens with probability 1/2), go to the next step. Otherwise, add 1 to N and repeat this step.

3. If N is less than z, return rem(k / 2^{z-1-N}, 2). (Alternatively, shift k to the right, by $z - 1 - N$ bits, then return k *AND* 1, where "*AND*" is a bitwise AND-operation.)
4. Subtract z from N. Then, if the item at position N in the uniform PSRN's fractional part (positions start at 0) is not set to a digit (for example, 0 or 1 for base 2), set the item at that position to a digit chosen uniformly at random (for example, either 0 or 1 for base 2), increasing the capacity of the uniform PSRN's fractional part as necessary.
5. Return the item at position N.

For bases other than 2, such as 10 for decimal, this can be implemented as follows (based on **URandLess**):

1. Set i to 0.
2. If i is less than z:
    1. Set da to rem(k / 2^{z-1-i}, $base$), and set db to a digit chosen uniformly at random (that is, an integer in the interval [0, $base$)).
    2. Return 1 if da is less than db, or 0 if da is greater than db.
3. If i is z or greater:
    1. If the digit at position ($i - z$) in the uniform PSRN's fractional part is not set, set the item at that position to a digit chosen uniformly at random (positions start at 0 where 0 is the most significant digit after the point, 1 is the second most significant, etc.).
    2. Set da to the item at that position, and set db to a digit chosen uniformly at random (so that $0 \le db < base$).
    3. Return 1 if da is less than db, or 0 if da is greater than db.
4. Add 1 to i and go to step 3.

12.2.2 Hyperbolic Secant Distribution

The following algorithm adapts the rejection algorithm from p. 472 in Devroye (1986)[62] for arbitrary-precision sampling.

1. Generate *ret*, an exponential random variate with a rate of 1 via the **ExpRand** or **ExpRand2** algorithm. This number will be a uniform PSRN.
2. Set *ip* to 1 plus *ret*'s integer part.
3. (The rest of the algorithm accepts *ret* with probability 1/(1+*ret*).) With probability *ip*/(1+*ip*), generate a number that is 1 with

probability 1/*ip* and 0 otherwise. If that number is 1, *ret* was accepted, in which case optionally fill it with uniform random digits as necessary to give its fractional part the desired number of digits (similarly to **FillGeometricBag**), then set *ret*'s sign to positive or negative with equal probability, then return *ret*.

4. Call **SampleGeometricBag** on *ret*'s fractional part (ignore *ret*'s integer part and sign). If the call returns 1, go to step 1. Otherwise, go to step 3.

12.2.3 Sum of Uniform Random Variates

The sum of n uniform random variates between 0 and 1 has the following probability density function (PDF) (see **MathWorld**):

$$f(x)=\left(p(x,n,0) + p(x,n,1) + ... + p(x,n,n)\right)/(2(n-1)!),$$

where $p(x,n,k) = {n\choose k} (x-k)^{n-1} \text{sign}(x-k)$, ${n\choose k}$ is a *binomial coefficient*, or the number of ways to choose k out of n labeled items, and sign(x) is 1 if x is greater than 0, or 0 if x is 0, or −1 is less than 0.[63] [64]

This is a polynomial of degree $n - 1$. For n uniform numbers, the distribution can take on values that are 0 or greater and n or less.

This new algorithm samples the sum of uniform random variates between 0 and 1 and the ratio of two uniform random variates between 0 and 1, with the help of PSRNs. It logically works as follows:

1. The distribution is divided into pieces that are each 1 unit long (thus, for example, if n is 4, there will be four pieces).

2. An integer in [0, n) is chosen uniformly at random, call it i, then the piece identified by i is chosen. There are **many algorithms to choose an integer** this way, but an algorithm that is "optimal" in terms of the number of bits it uses, as well as unbiased, should be chosen.

3. The PDF at [i, i + 1] is shifted so the desired piece of the PDF is at the closed interval [0, 1] rather than its usual place. More specifically, the PDF is now as follows: $$\mathtt{ShiftedF}(x)=\left(p(x+i,n,0) + p(x+i,n,1) + ... + p(x+i,n,n)\right)/(2(n-1)!),$$ where $0 \le x \le 1$. Since `ShiftedF` is a polynomial, it can be rewritten in Bernstein form, so that it has *Bernstein coefficients*, which are equivalent to control points describing the shape of the

curve drawn out by `ShiftedF`. (The Bernstein coefficients are the backbone of the well-known Bézier curve.) A polynomial can be written in *Bernstein form* as— $${n\choose 0}\lambda^0 (1-\lambda)^{n-0} a[0] + {n\choose 1}\lambda^1 (1-\lambda)^{n-1} a[1] + ... + {n\choose n}\lambda^n (1-\lambda)^{n-n} a[n],$$ where n is the polynomial's *degree* and $a[0]$, $a[1]$, ..., $a[n]$ are its n plus one *coefficients*. The coefficients serve as control points that together trace out a 1-dimensional Bézier curve. For example, given coefficients (control points) 0.2, 0.3, and 0.6, the curve is at 0.2 when $x = 0$, and 0.6 when $x = 1$. (Note that the curve is not at 0.3 when $x = 1/2$; in general, Bézier curves do not cross their control points other than the first and the last.)

Moreover, this polynomial can be simulated because its Bernstein coefficients all lie in [0, 1] (Goyal and Sigman 2012)[65].

4. The sampler creates a "coin" made up of a uniform partially-sampled random number (PSRN) whose contents are built up on demand using an algorithm called **SampleGeometricBag**. It flips this "coin" $n - 1$ times and counts the number of times the coin returned 1 this way, call it j. (The "coin" will return 1 with probability equal to the to-be-determined uniform random variate.)

5. Based on j, the sampler accepts the PSRN with probability equal to the coefficient $a[j]$. (See (Goyal and Sigman 2012)[66].)

6. If the PSRN is accepted, the sampler optionally fills it up with uniform random digits, then sets the PSRN's integer part to i, then the sampler returns the finished PSRN. If the PSRN is not accepted, the sampler starts over from step 2.

Finding Parameters:

Using the uniform sum sampler for an arbitrary n requires finding the Bernstein coefficients for each of the n pieces of the uniform sum PDF. This can be found, for example, with the Python code below, which uses the SymPy computer algebra library. In the code:

- `unifsum(x,n,v)` calculates the PDF of the sum of `n` uniform random variates when the variable `x` is shifted by `v` units.
- `find_control_points` returns the coefficients for each piece of the PDF for the sum of `n` uniform random variates, starting with piece 0.
- `find_areas` returns the relative areas for each piece of that PDF.

This can be useful to implement a variant of the sampler above, as detailed later in this section.

```
def unifsum(x,n,v):
    # Builds up the PDF at x (with offset v)
    # of the sum of n uniform random variates
    ret=0
    x=x+v # v is an offset
    for k in range(n+1):
           s=(-1)**k*binomial(n,k)*(x-k)**(n-1)
           # Equivalent to k>x+v since x is limited
           # to [0, 1]
           if k>v: ret-=s
           else: ret+=s
    return ret/(2*factorial(n-1))

def find_areas(n):
   x=symbols('x', real=True)
   areas=[integrate(unifsum(x,n,i),(x,0,1)) for i in range(n)]
   g=prod([v.q for v in areas])
   areas=[int(v*g) for v in areas]
   g=gcd(areas)
   areas=[v/int(g) for v in areas]
   return areas

def find_control_points(n, scale_pieces=False):
 x=symbols('x', real=True)
 controls=[]
 for i in range(n):
  # Find the "usual" coefficients of the uniform
  # sum polynomial at offset i.
  poly=Poly(unifsum(x, n, i))
  coeffs=[poly.coeff_monomial(x**i) for i in range(n)]
  # Build coefficient vector
  coeffs=Matrix(coeffs)
  # Build power-to-Bernstein basis matrix
  mat=[[0 for _ in range(n)] for _ in range(n)]
  for j in range(n):
    for k in range(n):
       if k==0 or j==n-1:
         mat[j][k]=1
       elif k<=j:
```

```
        mat[j][k]=binomial(j, j-k) / binomial(n-1, k)
      else:
        mat[j][k]=0
  mat=Matrix(mat)
  # Get the Bernstein coefficients
  mv = mat*coeffs
  mvc = [Rational(mv[i]) for i in range(n)]
  maxcoeff = max(mvc)
  # If requested, scale up coefficients to raise acceptance rate
  if scale_pieces:
     mvc = [v/maxcoeff for v in mvc]
  mv = [[v.p, v.q] for v in mvc]
  controls.append(mv)
 return controls
```

The basis matrix is found, for example, as Equation 42 of (Ray and Nataraj 2012)[67].

For example, if $n = 4$ (so a sum of four uniform random variates is desired), the following Bernstein coefficients are used for each piece of the PDF:

Piece	Coefficients
0	0, 0, 0, 1/6
1	1/6, 1/3, 2/3, 2/3
2	2/3, 2/3, 1/3, 1/6
3	1/6, 0, 0, 0

For more efficient results, all these coefficients could be scaled so that the highest coefficient is equal to 1. This doesn't affect the algorithm's correctness because scaling a Bézier curve's control points scales the curve accordingly, as is well known. In the example above, after multiplying by 3/2 (the reciprocal of the highest coefficient, which is 2/3), the table would now look like this:

Piece	Coefficients
0	0, 0, 0, 1/4
1	1/4, 1/2, 1, 1
2	1, 1, 1/2, 1/4
3	1/4, 0, 0, 0

Notice the following:

- All these Bernstein coefficients are rational numbers, and the sampler may have to determine whether an event is true with probability equal to a Bernstein coefficient. For rational numbers like these, it is possible to determine this exactly (using only random bits), using the **ZeroOrOne** method given in my **article on randomization and sampling methods**.
- The first and last piece of the PDF have predictable Bernstein coefficients. Namely the coefficients are as follows:
  - Piece 0: 0, 0, ..., 0, 1/((*n* − 1)!), where (*n* − 1)! = 1*2*3*...*(*n*−1).
  - Piece *n* − 1: 1/((*n* − 1)!), 0, 0, ..., 0.

If the areas of the PDF's pieces are known in advance (and SymPy makes them easy to find as the `find_areas` method shows), then the sampler could be modified as follows, since each piece is now chosen with probability proportional to the chance that a random variate there will be sampled:

- Step 2 is changed to read: "An integer in [0, *n*) is chosen with probability proportional to the corresponding piece's area, call the integer *i*, then the piece identified by *i* is chosen. There are many **algorithms to choose an integer** this way."

- The last sentence in step 6 is changed to read: "If the PSRN is not accepted, the sampler starts over from step 3." With this, the same piece is sampled again.

- The following are additional modifications that should be done to the sampler. However, not applying them does not affect the sampler's correctness.

  - The Bernstein coefficients should be scaled so that the highest coefficient of *each* piece is equal to 1. See the table below for an example.
  - If piece 0 is being sampled and the PSRN's digits are binary (base 2), the "coin" described in step 4 uses a modified version of **SampleGeometricBag** in which a 1 (rather than any other digit) is sampled from the PSRN when it reads from or writes to that PSRN. Moreover, the PSRN is always accepted regardless of the result of the "coin" flip.
  - If piece *n* − 1 is being sampled and the PSRN's digits are binary (base 2), the "coin" uses a modified version of

SampleGeometricBag in which a 0 (rather than any other digit) is sampled, and the PSRN is always accepted.

Piece	Coefficients
0	0, 0, 0, 1
1	1/4, 1/2, 1, 1
2	1, 1, 1/2, 1/4
3	1, 0, 0, 0

Sum of Two Uniform Random Variates:

The following algorithm samples the sum of two uniform random variates.

1. Create a uniform PSRN (partially-sampled random number) with an empty fractional part, an integer part of 0, and a positive sign, call it *ret*.
2. Generate an unbiased random bit (that is, either 0 or 1, chosen with equal probability).
3. Remove all digits from *ret*. (This algorithm works for digits of any base, including base 10 for decimal, or base 2 for binary.)
4. Call the **SampleGeometricBag** algorithm on *ret*, then generate an unbiased random bit.
5. If the bit generated in step 2 is 1 and the result of **SampleGeometricBag** is 1, optionally fill *ret* with uniform random digits as necessary to give its fractional part the desired number of digits (similarly to **FillGeometricBag**), then return *ret*.
6. If the bit generated in step 2 is 0 and the result of **SampleGeometricBag** is 0, optionally fill *ret* as in step 5, then set *ret*'s integer part to 1, then return *ret*.
7. Go to step 3.

For base 2, the following algorithm also works, using certain "tricks" described in the next section.

1. Generate an unbiased random bit (that is, either 0 or 1, chosen with equal probability), call it d.
2. Generate unbiased random bits until 0 is generated this way. Set g to the number of one-bits generated by this step.
3. Create a uniform PSRN with an empty fractional part, an integer part of 0, and a positive sign, call it *ret*. Then, set the digit at position g of the PSRN's fractional part to d (positions start at 0 in

the PSRN).

4. Optionally, fill *ret* with uniform random digits as necessary to give its fractional part the desired number of digits (similarly to **FillGeometricBag**). Then set *ret*'s integer part to $(1 - d)$, then return *ret*

Sum of Three Uniform Random Variates:

The following algorithm samples the sum of three uniform random variates.

1. Create a positive-sign zero-integer-part uniform PSRN, call it *ret*.
2. Choose an integer in [0, 6), uniformly at random. (With this, the left piece is chosen at a 1/6 chance, the right piece at 1/6, and the middle piece at 2/3, corresponding to the relative areas occupied by the three pieces.)
3. Remove all digits from *ret*.
4. If 0 was chosen by step 2, we will sample from the left piece of the function for the sum of three uniform random variates. This piece runs along the interval [0, 1) and is a polynomial with Bernstein coefficients of (0, 1, 1/2) (and is thus a Bézier curve with those control points). Due to the particular form of the coefficients, the piece can be sampled in one of the following ways:
    - Call the **SampleGeometricBag** algorithm twice on *ret*. If both of these calls return 1, then accept *ret* with probability 1/2. This is the most "naïve" approach.
    - Call the **SampleGeometricBag** algorithm twice on *ret*. If both of these calls return 1, then accept *ret*. This version of the step is still correct since it merely scales the polynomial so its upper bound is closer to 1, which is the top of the left piece, thus improving the acceptance rate of this step.
    - Base-2 only: Call a modified version of **SampleGeometricBag** twice on *ret*; in this modified algorithm, a 1 (rather than any other digit) is sampled from *ret* when that algorithm reads or writes a digit in *ret*. Then *ret* is accepted. This version will always accept *ret* on the first try, without rejection, and is still correct because *ret* would be accepted by this step only if **SampleGeometricBag** succeeds both times, which will happen only if that algorithm reads or writes out a 1 each time (because otherwise the Bernstein coefficient is 0, meaning that

ret is accepted with probability 0).

If *ret* was accepted, optionally fill *ret* with uniform random digits as necessary to give its fractional part the desired number of digits (similarly to **FillGeometricBag**), then return *ret*.

5. If 2, 3, 4, or 5 was chosen by step 2, we will sample from the middle piece of the PDF, which runs along the interval [1, 2) and is a polynomial with Bernstein coefficients (control points) of (1/2, 1, 1/2). Call the **SampleGeometricBag** algorithm twice on *ret*. If neither or both of these calls return 1, then accept *ret*. Otherwise, if one of these calls returns 1 and the other 0, then accept *ret* with probability 1/2. If *ret* was accepted, optionally fill *ret* as given in step 4, then set *ret*'s integer part to 1, then return *ret*.

6. If 1 was chosen by step 2, we will sample from the right piece of the PDF, which runs along the interval [2, 3) and is a polynomial with Bernstein coefficients (control points) of (1/2, 0, 0). Due to the particular form of the coefficients, the piece can be sampled in one of the following ways:

    - Call the **SampleGeometricBag** algorithm twice on *ret*. If both of these calls return 0, then accept *ret* with probability 1/2. This is the most "naïve" approach.
    - Call the **SampleGeometricBag** algorithm twice on *ret*. If both of these calls return 0, then accept *ret*. This version is correct for a similar reason as in step 4.
    - Base-2 only: Call a modified version of **SampleGeometricBag** twice on *ret*; in this modified algorithm, a 0 (rather than any other digit) is sampled from *ret* when that algorithm reads or writes a digit in *ret*. Then *ret* is accepted. This version is correct for a similar reason as in step 4.

    If *ret* was accepted, optionally fill *ret* as given in step 4, then set *ret*'s integer part to 2, then return *ret*.

7. Go to step 3.

12.2.4 Ratio of Two Uniform Random Variates

The ratio of two uniform random variates between 0 and 1 has the following probability density function (see **MathWorld**):

- 1/2 if $x >= 0$ and $x <= 1$,

- (1/ x^2) / 2 if $x > 1$, and
- 0 otherwise.

The following algorithm generates the ratio of two uniform random variates.

1. Generate an unbiased random bit. If that bit is 1 (which happens with probability 1/2), we have a uniform random variate between 0 and 1. Create a positive-sign zero-integer-part uniform PSRN, then optionally fill the PSRN with uniform random digits as necessary to give the number's fractional part the desired number of digits (similarly to **FillGeometricBag**), then return the PSRN.
2. At this point, the result will be 1 or greater. Set *intval* to 1 and set *size* to 1.
3. Generate an unbiased random bit. If that bit is 1 (which happens with probability 1/2), go to step 4. Otherwise, add *size* to *intval*, then multiply *size* by 2, then repeat this step. (This step chooses an interval beyond 1, taking advantage of the fact that the area under the PDF between 1 and 2 is 1/4, between 2 and 4 is 1/8, between 4 and 8 is 1/16, and so on, so that an appropriate interval is chosen with the correct probability.)
4. Generate an integer in the interval [*intval*, *intval* + *size*) uniformly at random, call it *i*.
5. Create a positive-sign zero-integer-part uniform PSRN, *ret*.
6. Call the **sub-algorithm** below with d = *intval* and $c = i$. If the call returns 0, go to step 4. (Here we simulate $intval/(i+\lambda)$ rather than $1/(i+\lambda)$ in order to increase acceptance rates in this step. This is possible without affecting the algorithm's correctness.)
7. Call the **sub-algorithm** below with $d = 1$ and $c = i$. If the call returns 0, go to step 4.
8. The PSRN *ret* was accepted, so set *ret*'s integer part to *i*, then optionally fill *ret* with uniform random digits as necessary to give its fractional part the desired number of digits (similarly to **FillGeometricBag**), then return *ret*.

The algorithm above uses a sub-algorithm that simulates the probability $d / (c + \lambda)$, where λ is the probability built up by the uniform PSRN, as follows:

1. With probability $c / (1 + c)$, return a number that is 1 with probability d/c and 0 otherwise.
2. Call **SampleGeometricBag** on *ret* (the uniform PSRN). If the call returns 1, return 0. Otherwise, go to step 1.

And the following Python code implements this algorithm.

```
def numerator_div(bern, numerator, intpart, bag):
   # Simulates numerator/(intpart+bag)
   while True:
      if bern.zero_or_one(intpart,1+intpart)==1:
         return bern.zero_or_one(numerator,intpart)
      if bern.geometric_bag(bag)==1: return 0

def ratio_of_uniform(bern):
    """ Exact simulation of the ratio of uniform random variates."""
    # First, simulate the integer part
    if bern.randbit():
       # This is 0 to 1, which follows a uniform distribution
       bag=[]
       return bern.fill_geometric_bag(bag)
    else:
       # This is 1 or greater
       intval=1
       size=1
       # Determine which range of integer parts to draw
       while True:
           if bern.randbit()==1:
                break
           intval+=size
           size*=2
       while True:
         # Draw the integer part
         intpart=bern.rndintexc(size) + intval
         bag=[]
         # Note: Density at [intval,intval+size) is multiplied
         # by intval, to increase acceptance rates
         if numerator_div(bern,intval,intpart,bag)==1 and \
            numerator_div(bern,1,intpart,bag)==1:
             return intpart + bern.fill_geometric_bag(bag)
```

12.2.5 Reciprocal of Uniform Random Variate

The reciprocal of a uniform random variate between 0 and 1 has the probability density function—

- $1 / x^2$ if $x > 1$, and

- 0 otherwise.

The algorithm to generate the reciprocal of a uniform random variate is the same as the algorithm for the ratio of two uniform random variates, except step 1 is omitted.

12.2.6 Reciprocal of Power of Uniform Random Variate

The following algorithm generates a PSRN of the form $1/U^{1/x}$, where U is a uniform random variate greater than 0 and less than 1 and x is an integer greater than 0.

1. Set *intval* to 1 and set *size* to 1.
2. With probability $(4^x-2^x)/4^x$, go to step 3. Otherwise, add *size* to *intval*, then multiply *size* by 2, then repeat this step.
3. Generate an integer in the interval [*intval*, *intval* + *size*) uniformly at random, call it i.
4. Create *ret*, a uniform PSRN with a positive sign and an integer part of 0.
5. Create an input coin that calls **SampleGeometricBag** on the PSRN *ret*. Call the **algorithm for d^k / $(c + \lambda)^k$** in "**Bernoulli Factory Algorithms**", using the input coin, where d = *intval*, c = i, and k = x + 1 (here, λ is the probability built up in *ret* via **SampleGeometricBag**, and satisfies $0 \le \lambda \le 1$). If the call returns 0, go to step 3.
6. The PSRN *ret* was accepted, so set *ret*'s integer part to i, then optionally fill *ret* with uniform random digits as necessary to give its fractional part the desired number of digits (similarly to **FillGeometricBag**), then return *ret*.

This algorithm uses the skeleton described earlier in "Building an Arbitrary-Precision Sampler". Here, the probabilities A, B, and C are as follows:

- A = 0, since the random variate can't be zero or come between 0 and 1.
- $B = (4^x-2^x)/4^x$.
- $C = (x/(i + \lambda)^{x+1})$ / M. Ideally, M is either x if *intval* is 1, or $x/intval^{x+1}$ otherwise. Thus, the ideal form for C is $intval^{x+1}/(i+\lambda)^{x+1}$.

12.2.7 Distribution of $U/(1-U)$

The following algorithm generates a PSRN distributed as $U/(1-U)$, where U is a uniform random variate greater than 0 and less than 1.

1. Generate an unbiased random bit. If that bit is 1 (which happens with probability 1/2), set *intval* to 0, then set *size* to 1, then go to step 4.
2. Set *intval* to 1 and set *size* to 1.
3. With probability *size*/(*size* + *intval* + 1), go to step 4. Otherwise, add *size* to *intval*, then multiply *size* by 2, then repeat this step.
4. Generate an integer in the interval [*intval*, *intval* + *size*) uniformly at random, call it *i*.
5. Create *ret*, a uniform PSRN with a positive sign and an integer part of 0.
6. Create an input coin that calls **SampleGeometricBag** on the PSRN *ret*. Call the **algorithm for d^k / $(c + \lambda)^k$** in "**Bernoulli Factory Algorithms**", using the input coin, where $d = intval + 1$, $c = i + 1$, and $k = 2$ (here, λ is the probability built up in *ret* via **SampleGeometricBag**, and satisfies $0 \le \lambda \le 1$). If the call returns 0, go to step 4.
7. The PSRN *ret* was accepted, so set *ret*'s integer part to *i*, then optionally fill *ret* with uniform random digits as necessary to give its fractional part the desired number of digits (similarly to **FillGeometricBag**), then return *ret*.

This algorithm uses the skeleton described earlier in "Building an Arbitrary-Precision Sampler". Here, the probabilities A, B, and C are as follows:

- $A = 1/2$.
- $B = size/(size + intval + 1)$.
- $C = (1/(i+\lambda+1)^2)$ / M. Ideally, M is $1/(intval+1)^2$. Thus, the ideal form for C is $(intval+1)^2/(i+\lambda+1)^2$.

12.2.8 Arc-Cosine Distribution

The following is a reimplementation of an example from Devroye's book *Non-Uniform Random Variate Generation* (Devroye 1986, pp. 128–129)[68]. The following arbitrary-precision sampler generates a random variate from a distribution with the following cumulative distribution function (CDF): `1 - cos(pi*x/2)`. The random variate will

be 0, 1, or a real number in between. This algorithm's result is the same as applying acos(*U*)*2/π, where *U* is a uniform random variate between 0 and 1, as pointed out by Devroye. (acos(*x*) is the inverse cosine function.) The algorithm follows.

1. Call the **kthsmallest** algorithm with `n = 2` and `k = 2`, but without filling it with digits at the last step. Let *ret* be the result.
2. Set *m* to 1.
3. Call the **kthsmallest** algorithm with `n = 2` and `k = 2`, but without filling it with digits at the last step. Let *u* be the result.
4. With probability 4/(4**m***m* + 2**m*), call the **URandLess** algorithm with parameters *u* and *ret* in that order, and if that call returns 1, call the **algorithm for π / 4**, described in "**Bernoulli Factory Algorithms**", twice, and if both of these calls return 1, add 1 to *m* and go to step 3. (Here, an erratum in the algorithm on page 129 of the book is incorporated.)
5. If *m* is odd[69], optionally fill *ret* with uniform random digits as necessary to give its fractional part the desired number of digits (similarly to **FillGeometricBag**), and return *ret*.
6. If *m* is even[70], go to step 1.

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

```
def example_4_2_1(rg, bern, precision=53):
    while True:
       ret=rg.kthsmallest_psrn(2,2)
       k=1
       while True:
          u=rg.kthsmallest_psrn(2,2)
          kden=4*k*k+2*k # erratum incorporated
          if randomgen.urandless(rg,u, ret) and \
             rg.zero_or_one(4, kden)==1 and \
             bern.zero_or_one_pi_div_4()==1 and \
             bern.zero_or_one_pi_div_4()==1:
             k+=1
          elif (k&1)==1:
             return
randomgen.urandfill(rg,ret,precision)/(1<<precision)
          else: break
```

12.2.9 Logistic Distribution

The following new algorithm generates a partially-sampled random number that follows the logistic distribution.

1. Set k to 0.
2. (Choose a 1-unit-wide piece of the logistic density.) Run the **algorithm for (1 + exp(z − w)) / (1 + exp(z))** described in "Bernoulli Factory Algorithms" with $z = k + 1$ and $w = 1$. If the call returns 0, add 1 to k and repeat this step. Otherwise, go to step 3.
3. (The rest of the algorithm samples from the chosen piece.) Create a uniform PSRN with integer part 0 and a positive sign, call it f.
4. (Steps 4 through 7 succeed with probability $\exp(-(f+k))/(1+\exp(-(f+k)))^2$.) Generate an unbiased random bit. If that bit is 1 (which happens with probability 1/2), go to step 3.
5. Call the **ExpMinus** algorithm with parameter k, then call the **ExpMinus** algorithm with parameter f. If any of these calls returns 0, go to step 4.
6. Generate an unbiased random bit. If that bit is 1 (which happens with probability 1/2), accept f. If f is accepted this way, set f's integer part to k, then optionally fill f with uniform random digits as necessary to give its fractional part the desired number of digits (similarly to **FillGeometricBag**), then set f's sign to positive or negative with equal probability, then return f.
7. Call the **ExpMinus** algorithm with parameter k, then call the **ExpMinus** algorithm with parameter f. If both calls return 1, go to step 3. Otherwise, go to step 6.

12.2.10 Cauchy Distribution

Uses the skeleton for the uniform distribution inside N-dimensional shapes.

1. Generate two empty PSRNs, with a positive sign, an integer part of 0, and an empty fractional part. Call the PSRNs *p1* and *p2*.
2. Set S to *base*, where *base* is the base of digits to be stored by the PSRNs (such as 2 for binary or 10 for decimal). Then set *c1* and *c2* each to 0. Then set d to 1.
3. Multiply *c1* and *c2* each by *base* and add a digit chosen uniformly at random to that coordinate.
4. If $((c1+1)^2 + (c2+1)^2) < S^2$, then do the following:
    1. Transfer *c1*'s least significant digits to *p1*'s fractional part, and

transfer *c2*'s least significant digits to *p2*'s fractional part. The variable *d* tells how many digits to transfer to each PSRN this way. (For example, if *base* is 10, *d* is 3, and *c1* is 342, set *p1*'s fractional part to [3, 4, 2].)

2. Run the **UniformDivide** algorithm on *p1* and *p2*, in that order, then set the resulting PSRN's sign to positive or negative with equal probability, then return that PSRN.

5. If $(c1^2 + c2^2) > S^2$, then go to step 2.
6. Multiply *S* by *base*, then add 1 to *d*, then go to step 3.

12.2.11 Exponential Distribution with Unknown Small Rate

Exponential random variates can be generated using an input coin of unknown probability of heads of λ (which can either be 1 or come between 0 and 1), by generating arrival times in a *Poisson process* of rate 1, then *thinning* the process using the coin. The arrival times that result will be exponentially distributed with rate λ. I found the basic idea in the answer to a **Mathematics Stack Exchange question**, and thinning of Poisson processes is discussed, for example, in Devroye (1986, chapter six)[71]. The algorithm follows:

1. Generate an exponential(1) random variate using the **ExpRand** or **ExpRand2** algorithm (with $\lambda = 1$), call it *ex*.
2. (Thinning step.) Flip the input coin. If it returns 1, return *ex*.
3. Generate another exponential(1) random variate using the **ExpRand** or **ExpRand2** algorithm (with $\lambda = 1$), call it *ex2*. Then run **UniformAdd** on *ex* and *ex2* and set *ex* to the result. Then go to step 2.

Notice that the algorithm's average running time increases as λ decreases.

12.2.12 Exponential Distribution with Rate ln(*x*)

The following new algorithm generates a partially-sampled random number that follows the exponential distribution with rate ln(*x*). This is useful for generating a base-*x* logarithm of a uniform(0,1) random variate. This algorithm has two supported cases:

- *x* is a rational number that's greater than 1. In that case, let *b* be floor(ln(*x*)/ln(2)).

- x is a uniform PSRN with a positive sign and an integer part of 1 or greater. In that case, let b be floor(ln(i)/ln(2)), where i is x's integer part.

The algorithm follows.

1. (Samples the integer part of the random variate.) Generate a number that is 1 with probability 1/x and 0 otherwise, repeatedly until a zero is generated this way. Set k to the number of ones generated this way. (This is also known as a "geometric random variate", but this terminology is avoided because it has conflicting meanings in academic works.)
    - If x is a rational number and a power of 2, this step can be implemented by generating blocks of b unbiased random bits until a **non-zero** block of bits is generated this way, then setting k to the number of **all-zero** blocks of bits generated this way.
    - If x is a uniform PSRN, this step is implemented as follows: Run the first subalgorithm (later in this section) repeatedly until a run returns 0. Set k to the number of runs that returned 1 this way.
2. (The rest of the algorithm samples the fractional part.) Create f, a uniform PSRN with a positive sign, an empty fractional part, and an integer part of 0.
3. Create a μ input coin that does the following: "Call **SampleGeometricBag** on f, then run the **algorithm for ln(1+y/z)** (given in "Bernoulli Factory Algorithms") with y/z = 1/1. If both calls return 1, return 1. Otherwise, return 0." (This simulates the probability $\lambda = f$*ln(2).) Then:
    - If x is a rational number, but not a power of 2, also create a ν input coin that does the following: "Call **SampleGeometricBag** on f, then run the **algorithm for ln(1 + y/z)** with $y/z = (x-2^b)/2^b$. If both calls return 1, return 1. Otherwise, return 0."
    - If x is a uniform PSRN, also create a ρ input coin that does the following: "Return the result of the second subalgorithm (later in this section), given x and b", and a ν input coin that does the following: "Call **SampleGeometricBag** on f, then run the **algorithm for ln(1 + λ)**, using the ρ input coin. If both calls return 1, return 1. Otherwise, return 0."
4. Call the **ExpMinus** algorithm with parameter μ (using the μ input coin), b times. If a ν input coin was created in step 3, run the same

algorithm once, using the ν input coin. If all these calls return 1, accept f. If f is accepted this way, set f's integer part to k, then optionally fill f with uniform random digits as necessary to give its fractional part the desired number of digits (similarly to **FillGeometricBag**), then return f.

5. If f was not accepted by the previous step, go to step 2.

> **Note**: A *bounded exponential* random variate with rate $\ln(x)$ and bounded by m has a similar algorithm to this one. Step 1 is changed to read as follows: "Do the following m times or until a zero is generated, whichever happens first: 'Generate a number that is 1 with probability $1/x$ and 0 otherwise'. Then set k to the number of ones generated this way. (k is a so-called bounded-geometric($1-1/x$, m) random variate, which an algorithm of Bringmann and Friedrich (2013)[72] can generate as well. If x is a power of 2, this can be implemented by generating blocks of b unbiased random bits until a **non-zero** block of bits or m blocks of bits are generated this way, whichever comes first, then setting k to the number of **all-zero** blocks of bits generated this way.) If k is m, return m (this m is a constant, not a uniform PSRN; if the algorithm would otherwise return a uniform PSRN, it can return something else in order to distinguish this constant from a uniform PSRN)." Additionally, instead of generating a uniform(0,1) random variate in step 2, a uniform(0,μ) random variate can be generated instead, such as a uniform PSRN generated via **RandUniformFromReal**, to implement an exponential distribution bounded by $m+\mu$ (where μ is a real number greater than 0 and less than 1).

The following generator for the **rate ln(2)** is a special case of the previous algorithm and is useful for generating a base-2 logarithm of a uniform(0,1) random variate. Unlike the similar algorithm of Ahrens and Dieter (1972)[73], this one doesn't require a table of probability values.

1. (Samples the integer part of the random variate. This will be geometrically distributed with parameter 1/2.) Generate unbiased random bits until a zero is generated this way. Set k to the number of ones generated this way.
2. (The rest of the algorithm samples the fractional part.) Create f, a uniform PSRN with integer part 0 and a positive sign.
3. Create an input coin that does the following: "Call

SampleGeometricBag on f, then run the **algorithm for ln(1+y/z)** (given in "Bernoulli Factory Algorithms") with $y/z = 1/1$. If both calls return 1, return 1. Otherwise, return 0." (This simulates the probability $\lambda = f*\ln(2)$.)

4. Run the **ExpMinus** algorithm with parameter λ, using the input coin from the previous step. If the call returns 1, accept f. If f is accepted this way, set f's integer part to k, then optionally fill f with uniform random digits as necessary to give its fractional part the desired number of digits (similarly to **FillGeometricBag**), then return f.
5. If f was not accepted by the previous step, go to step 2.

The first subalgorithm samples the probability $1/x$, where $x \geq 1$ is a uniform PSRN:

1. Set c to x's integer part. With probability $c / (1 + c)$, return a number that is 1 with probability $1/c$ and 0 otherwise.
2. Run **SampleGeometricBag** on x (which ignores x's integer part and sign). If the run returns 1, return 0. Otherwise, go to step 1.

The second subalgorithm samples the probability $(x-2^b)/2^b$, where $x \geq 1$ is a uniform PSRN and $b \geq 0$ is an integer:

1. Subtract 2^b from x's integer part, then create y as **RandUniformFromReal**(2^b), then run **URandLessThanReal**(x, y), then add 2^b back to x's integer part.
2. Return the result of **URandLessThanReal** from step 1.

12.2.13 Lindley Distribution and Lindley-Like Mixtures

A *mixture* involves one or more distributions, where each distribution has a separate probability of being sampled, and sampling one of them at random.

> **Example:** One example of a mixture is two beta distributions, with separate parameters. One beta distribution is chosen with probability $\exp(-3)$ (which can be simulated, for example, using the **ExpMinus** algorithm with parameter 3 can be used) and the other is chosen with the opposite probability. For the two beta distributions, an arbitrary-precision sampling algorithm exists.

The *Lindley distribution* is one example of a *mixture*, namely, there are two probability distributions, with each of the two distributions having a separate probability of being sampled (w and $1-w$ in the algorithm below). A random variate that follows the Lindley distribution (Lindley 1958)[74] with parameter θ (a real number greater than 0) can be generated as follows:

Let A be 1 and let B be 2. Then:

1. With probability $w = \theta/(1+\theta)$, generate A exponential random variates with a rate of $r = \theta$ via **ExpRand** or **ExpRand2** (described in my article on PSRNs), then generate their sum by applying the **UniformAdd** algorithm, then return that sum.
2. Otherwise, generate B exponential random variates with a rate of r via **ExpRand** or **ExpRand2**, then generate their sum by applying the **UniformAdd** algorithm, then return that sum.

Mixtures similar to the Lindley distribution are shown in the following table and use the same algorithm, but with the values given in the table.

Distribution	***w* is:**	***r* is:**	***A* is:**	***B* is:**
Garima (Shanker 2016)[75].	$(1+\theta)/(2+\theta)$.	θ.	1.	2.
i-Garima (Singh and Das 2020)[76].	$(2+\theta)/(3+\theta)$.	θ.	1.	2.
Mixture-of-weighted-exponential-and-weighted-gamma (Iqbal and Iqbal 2020)[77].	$\theta/(1+\theta)$.	θ.	2.	3.
Xgamma (Sen et al. 2016)[78].	$\theta/(1+\theta)$.	θ.	1.	3.
Mirra (Sen et al. 2021)[79].	$\theta^2/(\alpha+\theta^2)$ where $\alpha>0$.	θ.	1.	3.

> **Notes:**
>
> 1. If θ is a uniform PSRN, then the check "With probability $w = \theta/(1+\theta)$" can be implemented by running the Bernoulli factory algorithm for **$(d + \mu) / ((d + \mu) + (c + \lambda))$**, where c is 1; λ represents an input coin that always returns 0; d is θ's integer part, and μ is an input coin that runs **SampleGeometricBag** on θ's fractional part. The check succeeds if the Bernoulli factory algorithm returns 1.

2. A **Sushila** random variate is α times a Lindley random variate, where $\alpha > 0$ (Shanker et al. 2013)[80].
3. An **X-Lindley** random variate (Chouia and Zeghdoudi 2021)[81] is, with probability $\theta/(1+\theta)$, an exponential random variate with a rate of θ (see step 1), and otherwise, a Lindley random variate with parameter θ.

12.2.14 Gamma Distribution

The following arbitrary-precision sampler generates the sum of n independent exponential random variates (also known as the Erlang(n) or gamma(n) distribution), implemented via partially-sampled uniform random variates. Obviously, this algorithm is inefficient for large values of n.

1. Generate n exponential random variates with a rate of 1 via the **ExpRand** or **ExpRand2** algorithm. These numbers will be uniform PSRNs; this algorithm won't work for exponential PSRNs (e-rands) because the sum of two e-rands may follow a subtly wrong distribution. By contrast, generating exponential random variates via rejection from the uniform distribution will allow unsampled digits to be sampled uniformly at random without deviating from the exponential distribution.
2. Generate the sum of the random variates generated in step 1 by applying the **UniformAdd** algorithm given in another document.

12.2.15 One-Dimensional Epanechnikov Kernel

Adapted from Devroye and Györfi (1985, p. 236)[82].

1. Generate three uniform PSRNs a, b, and c, with a positive sign, an integer part of 0, and an empty fractional part.
2. Run **URandLess** on a and c in that order, then run **URandLess** on b and c in that order. If both runs return 1, set c to point to b.
3. Generate an unbiased random bit. If that bit is 1 (which will happen with probability 1/2), set c's sign to negative.
4. Return c.

12.2.16 Uniform Distribution Inside Rectellipse

The following example generates a point inside a quarter ***rectellipse*** centered at (0, 0) with—

- horizontal "radius" *d1*, which is an integer greater than 0,
- vertical "radius" *d2*, which is an integer greater than 0, and
- squareness parameter σ, which is a rational number in [0, 1].

Use the algorithm in "**Uniform Distribution Inside N-Dimensional Shapes**", except:

- Let **InShape** return—
  - *YES* if $(\sigma*x1*y1)^2/(d1*d2)^2 - (x2/d1)^2 - (y2/d2)^2 + 1 > 0$,
  - *NO* if $(\sigma*x2*y2)^2/(d1*d2)^2 - (x1/d1)^2 - (y1/d2)^2 + 1 < 0$, and
  - *MAYBE* otherwise,

  where $x1 = C1/S$, $x2 = (C1+1)/S$, $y1 = C2/S$, and $y2 = (C2+1)/S$ (these four values define the bounds, along the X and Y dimensions, of the box passed to **InShape**).

For a full rectellipse, step 5.3 in the algorithm is done for each of the two dimensions.

12.2.17 Tulap distribution

The algorithm below samples a variate from the Tulap(m, b, q) distribution ("truncated uniform Laplace"; Awan and Slavković (2019)[83]) and returns it as a uniform PSRN.

- The parameter b is greater than 0 and less than 1. Logically, this is implemented as a "coin" whose probability of returning 1 (showing heads) is not known by the algorithm. An example of this "coin" is a uniform PSRN with an initially empty fractional part, where "flipping" this "coin" means running **SampleGeometricBag** on that PSRN.
- The parameter m is a rational number or a uniform PSRN.
- The parameter q is not used in this algorithm, and is treated as 0.

---

1. Flip the coin for b until the result is 0. Set *g1* to the number of times the result is 1 this way.
2. Flip the coin for b until the result is 0. Set *g2* to the number of times the result is 1 this way.
3. Set n to $g2 - g1$. Create *ret*, a uniform PSRN as follows: The sign part is positive if $n \geq 0$ and negative otherwise; the integer part is abs(n); the fractional part is empty.
4. Add a uniform random variate in $(-1/2, 1/2)$ to *ret*. If *ret* stores

base-2 fractional digits, this can be done as follows. Set u to an unbiased random bit, then:
    - If $n < 0$ and u is 1, subtract 1 from the integer part and append 1 to the fractional part.
    - If $n > 0$ and u is 1, add 1 to the integer part and append 1 to the fractional part.
    - If n is 0 or u is 0, append 0 to the fractional part.
5. If m is given and is not 0, run the **UniformAdd** or **UniformAddRational** algorithm with parameters *ret* and m, respectively, and set *ret* to the result.
6. Return *ret*.

12.2.18 Continuous Bernoulli Distribution

The continuous Bernoulli distribution (Loaiza-Ganem and Cunningham 2019)[84] 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` (where $0 \le$ `lamda` ≤ 1), and takes on values in the closed 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 random variate, U, in the interval [0, 1] for the following steps, which will accept U with probability $\texttt{lamda}^{U} * (1-\texttt{lamda})^{1-U}$ 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 variate (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
```

13 Complexity

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

13.1 General Principles

Existing work shows how to calculate the bit complexity for any probability distribution:

- For a 1-dimensional distribution with a probability density function (PDF), the bit complexity is greater than or equal to `DE + prec - 1` random bits, where `DE` is the differential entropy for the distribution and *prec* is the number of bits in the random variate's fractional part (Devroye and Gravel 2020)[85].
- For a discrete distribution (a distribution of random integers with separate probabilities of occurring), the bit complexity is greater than or equal to the binary entropies of all the probabilities involved, summed together (Knuth and Yao 1976)[86]. (For a given probability *p*, the binary entropy is `0 - p*log2(p)` where `log2(x) = ln(x)/ln(2)`.) 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 distribution with a PDF, `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 variates, 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.

13.2 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)[87]. 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 the number of heads from a fair coin until the first tails, as follows:
  - Optimal lower bound: Since the binary entropy of the random variate 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.)

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

(see also (Efraimidis 2015)[88]). However, using fully-sampled exponential random variates as keys (such as the naïve idiom `-ln(1-X)/w`, where `X` is a uniform random variate between 0 and 1, 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 with probability 1.) Another reason is that partially-sampled e-rands have potentially arbitrary precision.

15 Acknowledgments

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

Due to a suggestion by Michael Shoemate who suggested it was "easy to get lost" in this and related articles, some sections that related to PSRNs and were formerly in "More Algorithms for Arbitrary-Precision Sampling" were moved here.

16 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 Sampling Methods**
- **Code Generator for Discrete Distributions**
- **The Most Common Topics Involving Randomization**
- **Bernoulli Factory Algorithms**
- **Testing PRNGs for High-Quality Randomness**
- **Examples of High-Quality PRNGs**

17 Notes

18 Appendix

18.1 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 variate 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.

18.2 UniformMultiply Algorithm

The following algorithm (**UniformMultiply**) 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 have a positive or negative sign, and it is assumed that their integer parts and signs were sampled.

The algorithm currently works only if each PSRN's fractional part has at least one nonzero digit; otherwise, it can produce results that follow an incorrect distribution. This case might be handled by applying the note in the section "Multiplication", but this will further complicate the algorithm below.

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. If either **a** or **b** has an integer part of 0 and a fractional part with no non-zero digits, then do the following.
    1. Append a digit chosen uniformly at random to **a**'s fractional part. Do the same for **b**.
    2. If either **a** or **b** has an integer part of 0 and a fractional part with no non-zero digits, go to the previous substep.
4. Let *afp* be **a**'s integer and fractional parts packed into an integer, as explained in the example, and let *bfp* be **b**'s integer and fractional parts packed the same way. (For example, if **a** represents the number 83.12344..., *afp* is 8312344.) 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 *n1* and *maxv* to *n2*. 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 is almost a trapezoid whose domain is the interval [*minv*, *maxv*] and whose top is delimited by *midmin* and *midmax*. (See note 1 at the end of this section.)
7. Create a new uniform PSRN, *ret*. If **a**'s sign is negative and **b**'s sign is negative, or vice versa, set *ret*'s sign to negative. Otherwise, set *ret*'s sign to positive.
8. Set z to a uniform random integer in the interval $[0, maxv-minv)$.
9. If $z < midmin-minv$ or if $z \geq midmax - minv$, we will sample from the left side or right side of the "trapezoid", respectively. In this case, do the following:
    1. Set x to $minv + z$. Create *psrn*, a PSRN with positive sign and empty fractional part.
    2. If $z < midmin - minv$ (left side), set *psrn*'s integer part to $x - minv$, then run **sub-algorithm 1** given later, with the

parameters *minv* and *psrn*. (The sub-algorithm returns 1 with probability ln((*minv*+*psrn*)/*minv*).)

3. If $z \geq midmin - minv$ (right side), set *psrn*'s integer part to $x - midmax$, then run **sub-algorithm 2** given later, with the parameters *maxv*, *midmax* and *psrn*. (The sub-algorithm returns 1 with probability ln(*maxv*/(*midmax*+*psrn*)).)
4. If sub-algorithm 1 or 2 returns 1, the algorithm succeeds, so do the following:
    1. Set *s* to *ru*.
    2. Transfer the *n**2 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.)
    3. Append the digits in *psrn*'s fractional part to the end of *ret*'s fractional part.
    4. Set *ret*'s integer part to floor($s/base^{n*2}$). (For example, if *base* is 10, *n**2 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*.
5. If sub-algorithm 1 or 2 returns 0, abort these substeps and go to step 8.

10. (If we reach here, we have reached the middle part of the trapezoid, which is flat and uniform.) If *n2* > *n3*, run **sub-algorithm 3** given later, with the parameter *afp* (returns 1 with probability ln(1+1/*afp*)). Otherwise, run **sub-algorithm 3** with the parameter *bfp* (returns 1 with probability ln(1+1/*bfp*)). In either case, if the sub-algorithm returns 0, go to step 8.
11. (The algorithm succeeds.) 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 sub-algorithms are used by **UniformMultiply**. They all involve the same underlying function, ln(1+*x*), with an **algorithm** mentioned in the page "Bernoulli Factory Algorithms".

- The sub-algorithm **ln(1+*x*)** takes an **input algorithm** and returns 1 with probability ln(1+*x*), where *x* is the probability that the input algorithm returns 1.
    - Do the following process repeatedly, until this sub-algorithm returns a value:
        1. Generate an unbiased random bit. If that bit is 1 (which happens with probability 1/2), run the **input algorithm** and

return the result.
    2. If u wasn't created yet, create u, a uniform PSRN with positive sign, an integer part of 0, and an empty fractional part.
    3. Run the **SampleGeometricBag** algorithm on u's fractional part, then run the **input algorithm**. If the call and the run both return 1, return 0.
- **Sub-algorithm 1** takes two parameters ($minv$ and $psrn$) and returns 1 with probability $\ln((minv+psrn)/minv)$. Run the **ln(1+*x*)** sub-algorithm with an **input algorithm** as follows:
    1. Let p be $psrn$'s integer part. Generate an integer in [0, $minv$) uniformly at random, call it i.
    2. If $i < p$, return 1. If $i = p$, flip the input coin and return the result. If neither is the case, return 0.
- **Sub-algorithm 2** takes three parameters ($maxv$, $midmax$ and $psrn$) and returns 1 with probability $\ln(maxv/(midmax+psrn))$. Run the **ln(1+*x*)** sub-algorithm with an **input algorithm** as follows:
    1. Let p be $psrn$'s integer part. Set d to $maxv - p - midmax - 1$, and set c to $p + midmax$.
    2. With probability $c / (1 + c)$, do the following:
        - Generate an integer in [0, c) uniformly at random, call it i. If $i < d$, return 1. If $i = d$, run **SampleGeometricBag** on $psrn$'s fractional part and return 1 minus the result. If $i > d$, return 0.
    3. Run **SampleGeometricBag** on $psrn$'s fractional part. If the result is 1, return 0. Otherwise, go to step 2.
- **Sub-algorithm 3** takes one parameter (called n here) and returns 1 with probability $\ln(1+1/n)$. Run the **ln(1+*x*)** sub-algorithm with an **input algorithm** as follows: "Return a number that is 1 with probability $1/n$ and 0 otherwise."

> **Note:** The product distribution of two uniform PSRNs is not exactly a trapezoid, but follows a not-so-trivial distribution; when each PSRN is bounded away from 0, the distribution's left and right sides are not exactly "triangular", but are based on logarithmic functions. However, these logarithmic functions approach a triangular shape as the distribution's "width" gets smaller. See Glen et al. (2004)[89] and a **Stack Exchange question**.

18.3 Uniform of Uniforms Produces a Product of Uniforms

This section contains evidence that the algorithm given in the note in the section "Multiplication" correctly produces the product of two uniform random variates, one in [0, b] and the other in [c, d], at least when $c = 0$.

The probability density function (PDF) for a uniform(α, β) random variate is $1/(\beta-\alpha)$ if x is in [α, β], and 0 elsewhere. It will be called UPDF(x, α, β) here.

Let $K = b*(d-c)$. To show the result, we find two PDFs as described below.

- To find the PDF for the algorithm, find the expected value of UPDF(x, 0, $Z+b*c$), where Z is distributed as uniform(0, K). This is done by finding the integral (area under the graph) with respect to z of UPDF(x, 0, $z+b*c$)*UPDF(z, 0, K) in the interval [0, K] (the set of values Z can take on). The result is `PDF1(x) = ln(b**2*c**2 - b**2*c*d + (b*c - b*d)*min(b*(-c + d), max(0, -b*c + x)))/(b*c - b*d) - ln(b**2*c**2 - b**2*c*d + b*(-c + d)*(b*c - b*d))/(b*c - b*d)`.
- The second PDF is the PDF for the product of two uniform random variates, one in [0, b] and the other in [c, d]. By Rohatgi's formula (see also (Glen et al. 2004)[90]), it can be found by finding the integral with respect to z of UPDF(z, 0, b)*UPDF(x/z, c, d)/z, in the interval [0, ∞) (noting that z is never negative here). The result is `PDF2(x) = (ln(max(c,x/b)) - ln(max(c,d,x/b)))/(b*c-b*d)`.

Now it must be shown that `PDF1` and `PDF2` are equal whenever x is in the interval (0, $b*d$). Subtracting one PDF from the other and simplifying, it is seen that:

- Both PDFs are equal at least when $c = 0$ (and when b, d, and x are all greater than 0), and they are equal in all calculations so far when b, c, and d are replaced with specific values.
- The simplified difference between the PDFs has an integral equal to 0, which strongly suggests the PDFs are equal (this is not conclusive because the simplified difference can be negative).

18.4 Oberhoff's "Exact Rejection Sampling" Method

The following describes an algorithm described by Oberhoff for sampling a continuous distribution taking on values in [0, 1], as long as the distribution has a probability density function (PDF) and the PDF is continuous "almost everywhere" and less than or equal to a finite number (Oberhoff 2018, section 3)[91], see also (Devroye and Gravel 2020)[92]. (Note that if the PDF'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 PDF (or the PDF times a possibly unknown constant factor) 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 variate that satisfies $0 \le y \le pdfmax$.
4. Let *pw* be $base^{-prefixLength}$. Set *lower* and *upper* to a lower or upper bound, respectively, of the value of the PDF (or the PDF times a possibly unknown constant factor) 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.

This algorithm appears here in the appendix rather than in the main text, because:

- Certain operations it uses can introduce numerical errors unless care is taken. These operations include evaluating the PDF (or a constant times the PDF) and finding its maximum and minimum values at an interval.
- This article is focused on algorithms that don't rely on calculations of irrational numbers.

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**"). For practical purposes, the lower and upper bounds calculated in step 4 should depend on *prefixLength* (the higher *prefixLength* is, the more accurate).

Oberhoff also describes *prefix distributions* that sample a box that covers the PDF, 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 distribution.

18.5 Probability Transformations

The following algorithm takes a uniform partially-sampled random number (PSRN) as a "coin" and flips that "coin" using **SampleGeometricBag**. Given that "coin" and a function f as described below, the algorithm returns 1 with probability $f(U)$, where U is the number built up by the uniform PSRN (see also Brassard et al., (2019)[93], (Devroye 1986, p. 769)[94], (Devroye and Gravel 2020)[95]. In the algorithm:

- The uniform PSRN's sign must be positive and its integer part must be 0.
- For correctness, $f(U)$ must meet the following conditions:
    - If the algorithm will be run multiple times with the same PSRN, $f(U)$ must be the constant 0 or 1, or be continuous and polynomially bounded on the open interval (0, 1) (polynomially bounded means that both $f(U)$ and $1-f(U)$ are greater than or equal to $\min(U^n, (1-U)^n)$ for some integer n (Keane and O'Brien 1994)[96]).
    - Otherwise, $f(U)$ must map the interval [0, 1] to [0, 1] and be continuous everywhere or "almost everywhere" (the set of

discontinuous points must be "zero-volume", or have Lebesgue measure zero).

The first set of conditions is the same as those for the Bernoulli factory problem (see "**About Bernoulli Factories**) and ensure this algorithm is unbiased (see also Łatuszyński et al. (2009/2011)[97]).

The algorithm follows.

1. Set v to 0 and k to 1.
2. (v acts as a uniform random variate greater than 0 and less than 1 to compare with $f(U)$.) Set v to $b * v + d$, where b is the base (or radix) of the uniform PSRN's digits, and d is a digit chosen uniformly at random.
3. Calculate an approximation of $f(U)$ as follows:
    1. Set n to the number of items (sampled and unsampled digits) in the uniform PSRN's fractional part.
    2. Of the first n digits (sampled and unsampled) in the PSRN's fractional part, sample each of the unsampled digits uniformly at random. Then let uk be the PSRN's digit expansion up to the first n digits after the point.
    3. Calculate the lowest and highest values of f in the interval $[uk, uk + b^{-n}]$, call them $fmin$ and $fmax$. If $\text{abs}(fmin - fmax) \le 2 * b^{-k}$, calculate $(fmax + fmin) / 2$ as the approximation. Otherwise, add 1 to n and go to the previous substep.
4. Let pk be the approximation's digit expansion up to the k digits after the point. For example, if $f(U)$ is $\pi/5$, b is 10, and k is 3, pk is 628.
5. If $pk + 1 \le v$, return 0. If $pk - 2 \ge v$, return 1. If neither is the case, add 1 to k and go to step 2.

> **Notes:** This algorithm is related to the Bernoulli factory problem, where the input probability is unknown. However, the algorithm doesn't exactly solve that problem because it has access to the input probability's value to some extent. Moreover, this article is focused on algorithms that don't rely on calculations of irrational numbers. For these two reasons, this section appears in the appendix.

18.6 Ratio of Uniforms

The Cauchy sampler given earlier demonstrates the *ratio-of-uniforms* technique for sampling a distribution (Kinderman and Monahan 1977)[98]. It involves transforming the distribution's probability density function (PDF) into a compact shape.

This algorithm works for every univariate (one-variable) distribution as long as—

- $PDF(x)$ (either the distribution's PDF or a function proportional to the PDF) is continuous "almost everywhere" on its domain,
- both $PDF(x)$ and $x^2*PDF(x)$ have a maximum on that domain, and
- either—
  - the distribution's ratio-of-uniforms shape (the transformed PDF) is covered entirely by the rectangle [0, ceil(*d1*)]×[0, ceil(*d2*)], where *d1* is not less than the maximum of abs(x)*sqrt($PDF(x)$) on its domain, and *d2* is not less than the maximum of sqrt($PDF(x)$) on its domain, or
  - half of that shape is covered this way and the shape is symmetric about the v-axis.

The algorithm follows.

1. Generate two empty PSRNs, with a positive sign, an integer part of 0, and an empty fractional part. Call the PSRNs *p1* and *p2*.
2. Set S to *base*, where *base* is the base of digits to be stored by the PSRNs (such as 2 for binary or 10 for decimal). Then set *c1* to an integer in the interval [0, *d1*), chosen uniformly at random, then set *c2* to an integer in [0, *d2*), chosen uniformly at random, then set *d* to 1.
3. Multiply *c1* and *c2* each by *base* and add a digit chosen uniformly at random to that coordinate.
4. Run an **InShape** function that determines whether the transformed PDF is covered by the current box. In principle, this is the case when $z \le 0$ everywhere in the box, where u lies in [$c1/S$, ($c1$+1)/S], v lies in [$c2/S$, ($c2$+1)/S], and z is $v^2-PDF(u/v)$. **InShape** returns *YES* if the box is fully inside the transformed PDF, *NO* if the box is fully outside it, and *MAYBE* in any other case, or if evaluating z fails for a given box (for example, because ln(0) would be calculated or v is 0). See the next section for implementation notes.
5. If **InShape** as described in step 4 returns *YES*, then do the following:
   1. Transfer *c1*'s least significant digits to *p1*'s fractional part, and

transfer *c2*'s least significant digits to *p2*'s fractional part. The variable *d* tells how many digits to transfer to each PSRN this way. Then set *p1*'s integer part to floor($c1/base^d$) and *p2*'s integer part to floor($c2/base^d$). (For example, if *base* is 10, *d* is 3, and *c1* is 7342, set *p1*'s fractional part to [3, 4, 2] and *p1*'s integer part to 7.)
    2. Run the **UniformDivide** algorithm on *p1* and *p2*, in that order.
    3. If the transformed PDF is symmetric about the *v*-axis, set the resulting PSRN's sign to positive or negative with equal probability. Otherwise, set the PSRN's sign to positive.
    4. Return the PSRN.
6. If **InShape** as described in step 4 returns *NO*, then go to step 2.
7. Multiply *S* by *base*, then add 1 to *d*, then go to step 3.

This algorithm appears here in the appendix rather than in the main text, because:

- Certain operations it uses can introduce numerical errors unless care is taken. These operations can include calculating upper and lower bounds of transcendental functions which, while it's possible to achieve in rational arithmetic (Daumas et al., 2007)[99], is less elegant than, say, the normal distribution sampler by Karney (2016)[100], which doesn't require calculating logarithms or other transcendental functions.
- This article is focused on algorithms that don't rely on calculations of irrational numbers.

> **Note:** The ratio-of-uniforms shape is convex if and only if −1/sqrt(*PDF*(*x*)) is a concave function (loosely speaking, its "slope" never increases) (Leydold 2000)[101].
>
> **Examples:**
>
> 1. For the normal distribution, *PDF* is proportional to $\exp(-x^2/2)$, so that *z* after a logarithmic transformation (see next section) becomes $4*\ln(v) + (u/v)^2$, and since the distribution's ratio-of-uniforms shape is symmetric about the *v*-axis, the return value's sign is positive or negative with equal probability.
> 2. For the standard lognormal distribution (**Gibrat's distribution**), *PDF*(*x*) is proportional to $\exp(-(\ln(x))^2/2)/x$, so that *z* after a logarithmic transformation becomes $2*\ln(v)-$

> $(-\ln(u/v)^2/2 - \ln(u/v))$, and the returned PSRN has a positive sign.
> 3. For the gamma distribution with shape parameter $a > 1$, $PDF(x)$ is proportional to $x^{a-1}*\exp(-x)$, so that z after a logarithmic transformation becomes $2*\ln(v)-(a-1)*\ln(u/v)-(u/v)$, or 0 if u or v is 0, and the returned PSRN has a positive sign.

18.7 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, the uniform and exponential distributions are the only practical distributions of this kind. Details follow.

Let X be a random variate of the form `0.bbbbbbb...`, where each `b` is an independent random binary digit (0 or 1).

Let a_j be the probability that the digit at position j equals 1 (starting with $j = 1$ for the first digit after the point).

Then Kakutani's theorem (Kakutani 1948)[102] says that X has an *absolutely continuous*[103] distribution if and only if the sum of squares of $(a_j - 1/2)$ converges (so that the binary digits become less and less biased as they move farther and farther from the binary point). See also Marsaglia (1971)[104], Chatterji (1964)[105].

This kind of absolutely continuous distribution can thus be built if we can find an infinite sequence a_j that converges to 1/2, and set X's binary digits using those probabilities. However, as Marsaglia (1971)[106] showed, the absolutely continuous distribution can only be one of the following:

1. The distribution's probability density function (PDF) is zero somewhere in every open interval in (0, 1), without being 0 on all of [0, 1]. Thus, the PDF is not continuous.

2. The PDF is positive at 1/2, 1/4, 1/8, and so on, so the PDF is continuous and positive on all of (0, 1), and the sequence has the form—

$a_j = \exp(w/2^j)/(1 + \exp(w/2^j))$,

where w is a constant.

3. The PDF is not described in Case 2 above, but is positive on some open interval in (0, 1), so the PDF will be piecewise continuous, and X can be multiplied by an integer power of 2 so that the new variate's distribution has a PDF described in Case 2.

As Marsaglia also showed, similar results apply when the base of the random digits is other than 2 (binary). See also my **Stack Exchange question**.

Case 2 has several special cases, including:

- The uniform distribution ($w = 0$).
- The fractional part of an exponential random variate with rate 1 ($w = -1$; (Devroye and Gravel 2020)[107]).
- More general, the fractional part of an exponential variate with rate λ ($w = -\lambda$).
- 1 minus the fractional part of an exponential variate with rate w when $w > 0$.
- $a_j = y^{v/2^j}/(1 + y^{v/2^j})$, with $w = \ln(y)*v$ where $y > 0$ and v are constants.

19 License

---

1. Karney, C.F.F., 2016. Sampling exactly from the normal distribution. ACM Transactions on Mathematical Software (TOMS), 42(1), pp.1-14. Also: "**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. Karney, C.F.F., 2016. Sampling exactly from the normal distribution. ACM Transactions on Mathematical Software (TOMS), 42(1), pp.1-14. Also: "**Sampling exactly from the normal distribution**", arXiv:1303.6257v2 [physics.comp-ph], 2014.↩

4. Devroye, L., Gravel, C., "**Random variate generation using only finitely many unbiased, independently and identically distributed random bits**", arXiv:1502.02539v6 [cs.IT], 2020.↩

5. 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.↩

6. 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.↩

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

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

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

10. von Neumann, J., "Various techniques used in connection with random digits", 1951.↩

11. As noted by von Neumann (1951), a uniform random variate bounded by 0 and 1 can be produced by "juxtapos[ing] enough random binary digits". In this sense, the variate is $X1/\mathtt{B}^1 + X2/\mathtt{B}^2 + \ldots$, (where $\mathtt{B}$ is the digit base 2, and $X1$, $X2$, etc. are independent uniform random integers in the interval $[0, \mathtt{B})$), 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.↩

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

13. Karney, C.F.F., 2016. Sampling exactly from the normal distribution. ACM Transactions on Mathematical Software (TOMS), 42(1), pp.1-14. Also: "**Sampling exactly from the normal distribution**", arXiv:1303.6257v2 [physics.comp-ph], 2014.↩

14. Karney, C.F.F., 2016. Sampling exactly from the normal distribution. ACM Transactions on Mathematical Software (TOMS), 42(1), pp.1-14. Also: "**Sampling exactly from the normal distribution**", arXiv:1303.6257v2 [physics.comp-ph], 2014.↩

15. Karney, C.F.F., 2016. Sampling exactly from the normal distribution. ACM Transactions on Mathematical Software (TOMS), 42(1), pp.1-14. Also: "**Sampling exactly from the normal distribution**", arXiv:1303.6257v2 [physics.comp-ph], 2014.↩

16. Yusong Du, Baoying Fan, and Baodian Wei, "**An Improved Exact Sampling Algorithm for the Standard Normal Distribution**", arXiv:2008.03855 [cs.DS], 2020.↩

17. This means that every zero-volume (Lebesgue measure zero) subset of the distribution's domain (such as a finite set of points) has zero probability. Equivalently, it means the distribution has a probability density function.↩

18. Oberhoff, Sebastian, "**Exact Sampling and Prefix Distributions**", *Theses and Dissertations*, University of Wisconsin Milwaukee, 2018.↩

19. Hill, T.P. and Schürger, K., 2005. Regularity of digits and significant digits of random variables. *Stochastic processes and their applications*, 115(10), pp.1723-1743.↩

20. A *non-discrete distribution* is a probability distribution taking on values that each can't be mapped to a different integer. An example is a distribution taking on any real number between 0 and 1.↩

21. Devroye, L., Gravel, C., "**Random variate generation using only finitely many unbiased, independently and identically distributed random bits**", arXiv:1502.02539v6 [cs.IT], 2020.↩

22. Oberhoff, Sebastian, "**Exact Sampling and Prefix Distributions**", *Theses and Dissertations*, University of Wisconsin Milwaukee, 2018.↩

23. 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.↩

24. J.F. Williamson, "Random selection of points distributed on curved surfaces", *Physics in Medicine & Biology* 32(10), 1987.↩

25. Boehm, Hans-J. "Towards an API for the real numbers." In Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 562-576. 2020.↩

26. Hans-J. Boehm. 1987. Constructive Real Interpretation of Numerical Programs. In Proceedings of the SIGPLAN ’87 Symposium on Interpreters and Interpretive Techniques. 214-221↩

27. Goubault-Larrecq, Jean, Xiaodong Jia, and Clément Théron. "**A Domain-Theoretic Approach to Statistical Programming Languages**", arXiv:2106.16190 (2021) (especially sec. 12.3).↩

28. Karney, C.F.F., 2016. Sampling exactly from the normal distribution. ACM Transactions on Mathematical Software (TOMS), 42(1), pp.1-14. Also: "**Sampling exactly from the normal distribution**", arXiv:1303.6257v2 [physics.comp-ph], 2014.↩

29. Devroye, L., Gravel, C., "**Random variate generation using only finitely many unbiased, independently and identically distributed random bits**", arXiv:1502.02539v6 [cs.IT], 2020.↩

30. Karney, C.F.F., 2016. Sampling exactly from the normal distribution. ACM Transactions on Mathematical Software (TOMS), 42(1), pp.1-14. Also: "**Sampling exactly from the normal distribution**", arXiv:1303.6257v2 [physics.comp-ph], 2014.↩

31. Karney, C.F.F., 2016. Sampling exactly from the normal distribution. ACM Transactions on Mathematical Software (TOMS), 42(1), pp.1-14. Also: "**Sampling exactly from the normal distribution**", arXiv:1303.6257v2 [physics.comp-ph], 2014.↩

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

33. Note that $ak * \beta^{-(i+1)}$ is not just within $\beta^{-(i+1)}$ of its "true" result's absolute value, but also not more than that value. Hence $ak >= bk + 1$ rather than $ak >= bk + 2$.↩

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

35. Karney, C.F.F., 2016. Sampling exactly from the normal distribution. ACM Transactions on Mathematical Software (TOMS), 42(1), pp.1-14. Also: "**Sampling exactly from the normal distribution**", arXiv:1303.6257v2 [physics.comp-ph], 2014.↩

36. 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.↩

37. 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.↩

38. In effect, for each supported integer n, the tree gives the probabilities of getting a value in $[n*p, (n+1)*p]$, where p is the resolution of the tree such as 1/100000.↩

39. Devroye, L., ***Non-Uniform Random Variate Generation***, 1986.↩

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

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

42. Oberhoff, Sebastian, "**Exact Sampling and Prefix Distributions**", *Theses and Dissertations*, University of Wisconsin Milwaukee, 2018.↩

43. Devroye, L., ***Non-Uniform Random Variate Generation***, 1986.↩

44. Devroye, L., ***Non-Uniform Random Variate Generation***, 1986.↩

45. Devroye, L., ***Non-Uniform Random Variate Generation***, 1986.↩

46. Oberhoff, Sebastian, "**Exact Sampling and Prefix Distributions**", *Theses and Dissertations*, University of Wisconsin Milwaukee, 2018.↩

47. von Neumann, J., "Various techniques used in connection with random digits", 1951.↩

48. Karney, C.F.F., 2016. Sampling exactly from the normal distribution. ACM Transactions on Mathematical Software (TOMS), 42(1), pp.1-14. Also: "**Sampling exactly from the normal distribution**", arXiv:1303.6257v2 [physics.comp-ph], 2014.↩

49. Fan, Baoying et al. “On Generating Exponentially Distributed Variates by Using Early Rejection.” *2019 IEEE 5th International Conference on Computer and Communications (ICCC)* (2019): 1307-1311.↩

50. C.T. Li, A. El Gamal, "**A Universal Coding Scheme for Remote Generation of Continuous Random Variables**", arXiv:1603.05238v1 [cs.IT], 2016.↩

51. Oberhoff, Sebastian, "**Exact Sampling and Prefix Distributions**", *Theses and Dissertations*, University of Wisconsin Milwaukee, 2018.↩

52. Devroye, L., ***Non-Uniform Random Variate Generation***, 1986.↩

53. Harlow, J., Sainudiin, R., Tucker, W., "Mapped Regular Pavings", *Reliable Computing* 16 (2012).↩

54. Duff, T., "Interval Arithmetic and Recursive Subdivision for Implicit Functions and Constructive Solid Geometry", Computer Graphics 26(2), July 1992.↩

55. Daumas, M., Lester, D., Muñoz, C., "**Verified Real Number Calculations: A Library for Interval Arithmetic**", arXiv:0708.3721 [cs.MS], 2007.↩

56. I thank D. Eisenstat from the *Stack Overflow* community for leading me to this insight.↩

57. Devroye, L., ***Non-Uniform Random Variate Generation***, 1986.↩

58. Devroye, L., ***Non-Uniform Random Variate Generation***, 1986.↩

59. Devroye, L., ***Non-Uniform Random Variate Generation***, 1986.↩

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

61. More specifically, the *essential supremum*, that is, the function's highest point in [0, 1] ignoring zero-volume, or measure-zero, sets. However, the mode is also correct here, since discontinuous PDFs don't admit Bernoulli factories, as required by step 2.↩

62. Devroye, L., ***Non-Uniform Random Variate Generation***, 1986.↩

63. choose(n, k) = (1*2*3*...*n)/((1*...*k)*(1*...*(n−k))) = n!/(k! * (n − k)!) is a *binomial coefficient*, or the number of ways to choose k out of n labeled items. It can be calculated, for example, by calculating i/(n−i+1) for each integer i in the interval [n−k+1, n], then multiplying the results (Yannis Manolopoulos. 2002. "**Binomial coefficient computation: recursion or iteration?**", SIGCSE Bull. 34, 4 (December 2002), 65–67). For every m>0, choose(m, 0) = choose(m, m) = 1 and choose(m, 1) = choose(m, m−1) = m; also, in this document, choose(n, k) is 0 when k is less than 0 or greater than n.↩

64. n! = 1*2*3*...*n is also known as n factorial; in this document, (0!) = 1.↩

65. Goyal, V. and Sigman, K., 2012. On simulating a class of Bernstein polynomials. ACM Transactions on Modeling and Computer Simulation (TOMACS), 22(2), pp.1-5.↩

66. Goyal, V. and Sigman, K., 2012. On simulating a class of Bernstein polynomials. ACM Transactions on Modeling and Computer Simulation (TOMACS), 22(2), pp.1-5.↩

67. S. Ray, P.S.V. Nataraj, "A Matrix Method for Efficient Computation of Bernstein Coefficients", *Reliable Computing* 17(1), 2012.↩

68. Devroye, L., ***Non-Uniform Random Variate Generation***, 1986.↩

69. "x is odd" means that x is an integer and not divisible by 2. This is true if x − 2*floor(x/2) equals 1, or if x is an integer and the least significant bit of abs(x) is 1.↩

70. "x is even" means that x is an integer and divisible by 2. This is true if x − 2*floor(x/2) equals 0, or if x is an integer and the least significant bit of abs(x) is 0.↩

71. Devroye, L., ***Non-Uniform Random Variate Generation***, 1986.↩

72. Bringmann, K. and Friedrich, T., 2013, July. "Exact and efficient generation of geometric random variates and random graphs", in *International Colloquium on Automata, Languages, and Programming* (pp. 267-278).↩

73. Ahrens, J.H., and Dieter, U., "Computer methods for sampling from the exponential and normal distributions", *Communications of the ACM* 15, 1972.↩

74. Lindley, D.V., "Fiducial distributions and Bayes' theorem", *Journal of the Royal Statistical Society Series B*, 1958.↩

75. Shanker, R., "Garima distribution and its application to model behavioral science data", *Biom Biostat Int J.* 4(7), 2016.↩

76. Singh, B.P., Das, U.D., "**On an Induced Distribution and its Statistical Properties**", arXiv:2010.15078 [stat.ME], 2020.↩

77. Iqbal, T. and Iqbal, M.Z., 2020. On the Mixture Of Weighted Exponential and Weighted Gamma Distribution. International Journal of Analysis and Applications, 18(3), pp.396-408.↩

78. Sen, S., et al., "The xgamma distribution: statistical properties and application", 2016.↩

79. Sen, Subhradev, Suman K. Ghosh, and Hazem Al-Mofleh. "The Mirra distribution for modeling time-to-event data sets." In Strategic Management, Decision Theory, and Decision Science, pp. 59-73. Springer, Singapore, 2021.↩

80. Shanker, Rama, Shambhu Sharma, Uma Shanker, and Ravi Shanker. "Sushila distribution and its application to waiting times data." International Journal of Business Management 3, no. 2 (2013): 1-11.↩

81. Chouia, Sarra, and Halim Zeghdoudi. "The xlindley distribution: Properties and application." Journal of Statistical Theory and Applications 20, no. 2 (2021): 318-327.↩

82. Devroye, L., Györfi, L., *Nonparametric Density Estimation: The L1 View*, 1985.↩

83. Awan, Jordan, and Aleksandra Slavković. "Differentially private inference for binomial data." arXiv:1904.00459 (2019).↩

84. Loaiza-Ganem, Gabriel, and John P. Cunningham. "The continuous Bernoulli: fixing a pervasive error in variational autoencoders." *Advances in Neural Information Processing Systems* 32 (2019).↩

85. Devroye, L., Gravel, C., "**Random variate generation using only finitely many unbiased, independently and identically distributed random bits**", arXiv:1502.02539v6 [cs.IT], 2020.↩

86. 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.↩

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

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

89. Glen, A.G., Leemis, L.M. and Drew, J.H., 2004. Computing the distribution of the product of two continuous random variables. Computational statistics & data analysis, 44(3), pp.451-464.↩

90. Glen, A.G., Leemis, L.M. and Drew, J.H., 2004. Computing the distribution of the product of two continuous random variables. Computational statistics & data analysis, 44(3), pp.451-464.↩

91. Oberhoff, Sebastian, "**Exact Sampling and Prefix Distributions**", *Theses and Dissertations*, University of Wisconsin Milwaukee, 2018.↩

92. Devroye, L., Gravel, C., "**Random variate generation using only finitely many unbiased, independently and identically distributed random bits**", arXiv:1502.02539v6 [cs.IT], 2020.↩

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

94. Devroye, L., ***Non-Uniform Random Variate Generation***, 1986.↩

95. Devroye, L., Gravel, C., "**Random variate generation using only finitely many unbiased, independently and identically distributed random bits**", arXiv:1502.02539v6 [cs.IT], 2020.↩

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

97. Łatuszyński, K., Kosmidis, I., Papaspiliopoulos, O., Roberts, G.O., "**Simulating events of unknown probabilities via reverse time martingales**", arXiv:0907.4018v2 [stat.CO], 2009/2011.↩

98. Kinderman, A.J., Monahan, J.F., "Computer generation of random variables using the ratio of uniform deviates", *ACM Transactions on Mathematical Software* 3(3), pp. 257-260, 1977.↩

99. Daumas, M., Lester, D., Muñoz, C., "**Verified Real Number Calculations: A Library for Interval Arithmetic**", arXiv:0708.3721 [cs.MS], 2007.↩

100. Karney, C.F.F., 2016. Sampling exactly from the normal distribution. ACM Transactions on Mathematical Software (TOMS), 42(1), pp.1-14. Also: "**Sampling exactly from the normal distribution**", arXiv:1303.6257v2 [physics.comp-ph], 2014.↩

101. Leydold, J., "**Automatic sampling with the ratio-of-uniforms method**", ACM Transactions on Mathematical Software 26(1), 2000.↩

102. S. Kakutani, "On equivalence of infinite product measures", *Annals of Mathematics* 1948.↩

103. This means that every zero-volume (Lebesgue measure zero) subset of the distribution's domain (such as a finite set of points) has zero probability. Equivalently, it means the distribution has a probability density function.↩

104. George Marsaglia. "Random Variables with Independent Binary Digits." Ann. Math. Statist. 42 (6) 1922 - 1929, December, 1971. **https://doi.org/10.1214/aoms/1177693058** .↩

105. Chatterji, S. D.. “Certain induced measures and the fractional dimensions of their “supports”.” Zeitschrift für Wahrscheinlichkeitstheorie und Verwandte Gebiete 3 (1964): 184-192.↩

106. George Marsaglia. "Random Variables with Independent Binary Digits." Ann. Math. Statist. 42 (6) 1922 - 1929, December, 1971. **https://doi.org/10.1214/aoms/1177693058** .↩

107. Devroye, L., Gravel, C., "**Random variate generation using only finitely many unbiased, independently and identically distributed random bits**", arXiv:1502.02539v6 [cs.IT], 2020.↩