More Algorithms for Arbitrary-Precision Sampling

This version of the document is dated 2021-01-08.

Peter Occil

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

This page contains additional algorithms for arbitrary-precision sampling of continuous distributions, Bernoulli factory algorithms (biased-coin to biased-coin algorithms), and algorithms to simulate irrational probabilities. These samplers are designed to not rely on floating-point arithmetic. They may depend on algorithms given in the following pages:

- **Partially-Sampled Random Numbers for Accurate Sampling of Continuous Distributions**
- **Bernoulli Factory Algorithms**

1 Contents

- **Contents**
- **Bernoulli Factories and Irrational Probability Simulation**
  - **Certain Numbers Based on the Golden Ratio**
  - **Ratio of Lower Gamma Functions ($\gamma(m, n)/\gamma(m, 1)$).**
  - **Derivative (slope) of arctan(λ)**
  - **cosh(λ) − 1**
  - **sinh(λ)/2**
  - **tanh(λ)**
  - **Certain Piecewise Linear Functions**
- **General Arbitrary-Precision Samplers**
  - **Uniform Distribution Inside N-Dimensional Shapes**
  - **Building an Arbitrary-Precision Sampler**
  - **Mixtures**
- **Specific Arbitrary-Precision Samplers**
  - **Rayleigh Distribution**
  - **Sum of Exponential Random Numbers**
  - **Hyperbolic Secant Distribution**
  - **Reciprocal of Power of Uniform**
  - **Distribution of $U/(1-U)$**
  - **Arc-Cosine Distribution**
  - **Logistic Distribution**
  - **Cauchy Distribution**
  - **Exponential Distribution with Rate ln(x)**
  - **Lindley Distribution and Lindley-Like Mixtures**
- **Requests and Open Questions**
- **Notes**
- **Appendix**
  - **Ratio of Uniforms**
  - **Implementation Notes for Box/Shape Intersection**
  - **SymPy Code for Piecewise Linear Factory Functions**

- **License**

2 Bernoulli Factories and Irrational Probability Simulation

In the methods below, λ is the unknown probability of heads of the coin involved in the Bernoulli Factory problem.

2.1 Certain Numbers Based on the Golden Ratio

The following algorithm given by Fishman and Miller (2013)[1] finds the continued fraction expansion of certain numbers described as—

- $G(m, \ell) = (m + \text{sqrt}(m^2 + 4 * \ell))/2$
  or $(m - \text{sqrt}(m^2 + 4 * \ell))/2$,

whichever results in a real number greater than 1, where m is a positive integer and ℓ is either 1 or -1. In this case, $G(1, 1)$ is the golden ratio.

First, define the following operations:

- **Get the previous and next Fibonacci-based number given *k*, *m*, and ℓ**:
  1. If k is 0 or less, return an error.
  2. Set *g0* to 0, *g1* to 1, x to 0, and y to 0.
  3. Do the following k times: Set y to $m * g1 + \ell * g0$, then set x to *g0*, then set *g0* to *g1*, then set *g1* to y.
  4. Return x and y, in that order.
- **Get the partial denominator given *pos*, *k*, *m*, and ℓ** (this partial denominator is part of the continued fraction expansion found by Fishman and Miller):
  1. **Get the previous and next Fibonacci-based number given *k*, *m*, and ℓ**, call them p and n, respectively.
  2. If ℓ is 1 and k is odd, return $p + n$.
  3. If ℓ is -1 and *pos* is 0, return $n - p - 1$.
  4. If ℓ is 1 and *pos* is 0, return $(n + p) - 1$.
  5. If ℓ is -1 and *pos* is even, return $n - p - 2$. (The paper had an error here; the correction given here was verified by Miller via personal communication.)
  6. If ℓ is 1 and *pos* is even, return $(n + p) - 2$.
  7. Return 1.

An application of the continued fraction algorithm is the following algorithm that generates 1 with probability $G(m, \ell)^{-k}$ and 0 otherwise, where k is an integer that is 1 or greater (see "Continued Fractions" in my page on Bernoulli factory algorithms). The algorithm starts with $pos = 0$, then the following steps are taken:

1. **Get the partial denominator given *pos*, *k*, *m*, and ℓ**, call it kp.
2. With probability $kp/(1 + kp)$, return a number that is 1 with probability $1/kp$ and 0 otherwise.
3. Run this algorithm recursively, but with $pos = pos + 1$. If the algorithm returns 1, return 0. Otherwise, go to step 2.

2.2 Ratio of Lower Gamma Functions (γ(*m*, *n*)/γ(*m*, 1)).

1. Set *ret* to the result of **kthsmallest** with the two parameters *m* and *m*.
2. Set *k* to 1, then set *u* to point to the same value as *ret*.
3. Generate a uniform(0, 1) random number *v*.
4. If *v* is less than *u*: Set *u* to *v*, then add 1 to *k*, then go to step 3.
5. If *k* is odd, return a number that is 1 if *ret* is less than *n* and 0 otherwise. (If *ret* is implemented as a uniform partially-sampled random number (PSRN), this comparison should be done via **URandLessThanReal**.) If *k* is even, go to step 1.

2.3 Derivative (slope) of arctan(*λ*)

This algorithm involves the series expansion of this function ($1 - \lambda^2 + \lambda^4 - ...$) and involves the general martingale algorithm.

1. Set *u* to 1, set *w* to 1, set ℓ to 0, and set *n* to 1.
2. Generate a uniform(0, 1) random number *ret*.
3. (Loop.) If *w* is not 0, flip the input coin and multiply *w* by the result of the flip. Do this step again.
4. If *n* is even, set *u* to $\ell + w$. Otherwise, set ℓ to $u - w$.
5. If *ret* is less than (or equal to) ℓ, return 1. If *ret* is less than *u*, go to the next step. If neither is the case, return 0. (If *ret* is a uniform PSRN, these comparisons should be done via the **URandLessThanReal algorithm**, which is described in my **article on PSRNs**.)
6. Add 1 to *n* and go to step 3.

2.4 cosh(*λ*) − 1

This algorithm involves an application of the general martingale algorithm to the Taylor series for cosh(*λ*)−1, which is $\lambda^2/(2!) + \lambda^4/(4!) + ...$ See (Łatuszyński et al. 2009/2011, algorithm 3)**(18)**.

1. Set *u* to 0, set *w* to 1, set ℓ to 0, and set *n* to 1.
2. Generate a uniform(0, 1) random number *ret*.
3. If *w* is not 0, flip the input coin and multiply *w* by the result of the flip. Do this step again.
4. If *w* is 0, set *u* to ℓ and go to step 6. (The estimate λ^{n*2} is 0, so no more terms are added and we use ℓ as the final estimate for cosh(*λ*)−1.)
5. Let *m* be (*n**2), let *α* be 1/(*m*!) (a term of the Taylor series), and let *err* be 2/((*m*+1)!) (the error term). Add *α* to ℓ, then set *u* to ℓ + *err*.
6. If *ret* is less than (or equal to) ℓ, return 1. If *ret* is less than *u*, go to the next step. If neither is the case, return 0. (If *ret* is a uniform PSRN, these comparisons should be done via the **URandLessThanReal algorithm**, which is described in my **article on PSRNs**.)
7. Add 1 to *n* and go to step 3.

In this algorithm, the error term, which follows from *Taylor's theorem*, has a numerator of 2 because 2 is higher than the maximum value that the function's slope, slope-of-slope, etc. functions can achieve anywhere in the interval [0, 1].

2.5 sinh(*λ*)/2

This algorithm involves an application of the general martingale algorithm to the Taylor series for sinh(λ)/2, which is $\lambda^1/(1!*2) + \lambda^3/(3!*2) + ...$, or as used here, $\lambda*(1/2 + \lambda^2/(3!*2) + \lambda^4/(5!*2) + ...)$.

1. Flip the input coin. If it returns 0, return 0.
2. Set u to 0, set w to 1, set ℓ to 1/2 (the first term is added already), and set n to 1.
3. Generate a uniform(0, 1) random number ret.
4. If w is not 0, flip the input coin and multiply w by the result of the flip. Do this step again.
5. If w is 0, set u to ℓ and go to step 7. (No more terms are added here.)
6. Let m be (n*2+1), let α be 1/(m!*2) (a term of the Taylor series), and let err be 1/((m+1)!) (the error term). Add α to ℓ, then set u to ℓ + err.
7. If ret is less than (or equal to) ℓ, return 1. If ret is less than u, go to the next step. If neither is the case, return 0. (If ret is a uniform PSRN, these comparisons should be done via the **URandLessThanReal algorithm**, which is described in my **article on PSRNs**.)
8. Add 1 to n and go to step 4.

2.6 tanh(λ)

There are two algorithms.

The first takes advantage of the so-called Lambert's continued fraction for tanh(.), as well as Bernoulli Factory algorithm 3 for continued fractions. The algorithm begins with k equal to 1. Then the following steps are taken.

1. If k is 1: With probability 1/2, flip the input coin and return the result.
2. If k is greater than 1, then do the following with probability $k/(1+k)$:
    - Flip the input coin twice. If any of these flips returns 0, return 0. Otherwise, return a number that is 1 with probability $1/k$ and 0 otherwise.
3. Run this algorithm recursively, but with $k = k + 2$. If the result is 1, return 0. Otherwise, go to step 1.

The second algorithm involves an alternating series expansion of tanh(.) and involves the general martingale algorithm.

First, define the following operation:

- **Get the m^{th} Bernoulli number**:
    1. If m is 0, 1, 2, 3, or 4, return 1, $-1/2$, 1/6, 0, or $-1/30$, respectively. Otherwise, if m is odd, return 0.
    2. Set i to 2 and v to $1 - (m+1)/2$.
    3. While i is less than m:
        1. **Get the i^{th} Bernoulli number**, call it b. Add b*choose(m+1, i) to v.[(2)]
        2. Add 2 to i.
    4. Return $-v/(m+1)$.

The algorithm is then as follows:

1. Flip the input coin. If it returns 0, return 0.
2. Set u to 1, set w to 1, set ℓ to 0, and set n to 1.
3. Generate a uniform(0, 1) random number ret.
4. (Loop.) If w is not 0, flip the input coin. If the flip returns 0, set w to 0. Do this step again.
5. (Calculate the next term of the alternating series for tanh.) Let m be 2*(n+1). **Get**

the m^{th} Bernoulli number, call it b. Let t be abs(b)*2^m*(2^m-1)/(m!).
6. If n is even, set u to $\ell + w * t$. Otherwise, set ℓ to $u - w * t$.
7. If *ret* is less than (or equal to) ℓ, return 1. If *ret* is less than u, go to the next step. If neither is the case, return 0. (If *ret* is a uniform PSRN, these comparisons should be done via the **URandLessThanReal algorithm**, which is described in my **article on PSRNs**.)
8. Add 1 to n and go to step 4.

2.7 Certain Piecewise Linear Functions

Let $f(\lambda)$ be a function of the form $\min(\lambda*mult, 1-\varepsilon)$. (This is a piecewise linear function with two pieces: a rising linear part and a constant part.) This section describes how to calculate the Bernstein coefficients for polynomials that converge from above and below to f, based on Thomas and Blanchet (2012)**(3)**. These polynomials can then be used to generate heads with probability $f(\lambda)$ via the algorithms given in "**General Factory Functions**".

The code in the **appendix** uses the computer algebra library SymPy to calculate a list of parameters for a sequence of polynomials converging from above. The method to do so is called `calc_linear_func(eps, mult, count)`, where `eps` is ε, `mult` = *mult*, and `count` is the number of polynomials to generate. Each item returned by `calc_linear_func` is a list of two items: the degree of the polynomial, and a *Y parameter*. The procedure to calculate the required polynomials is then logically as follows (as written, it runs very slowly, though):

1. Set i to 1.
2. Run `calc_linear_func(eps, mult, i)` and get the degree and *Y parameter* for the last listed item, call them n and y, respectively.
3. Set x to $-((y-(1-\varepsilon))/\varepsilon)^5/mult + y/mult$. (This exact formula doesn't appear in the Thomas and Blanchet paper; rather it comes from the **supplemental source code** uploaded by A. C. Thomas at my request.
4. For the i^{th} upper polynomial, the k^{th} Bernstein coefficient (starting at 0) is $\min((k/n)*y/x, y)$.
5. For the i^{th} lower polynomial, the k^{th} Bernstein coefficient (starting at 0) is $\min((k/n)*mult, 1-\varepsilon)$. (This matches f because f is *concave* in the interval [0, 1], which roughly means that its rate of growth there never goes up.)
6. Add 1 to i and go to step 2.

It would be interesting to find general formulas to find the appropriate polynomials (degrees and *Y parameters*) given only the values for *mult* and ε, rather than find them "the hard way" via `calc_linear_func`. For this procedure, the degrees and *Y parameters* can be upper bounds, as long as the sequence of degrees is monotonically increasing and the sequence of Y parameters is nonincreasing.

3 General Arbitrary-Precision Samplers

3.1 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 boundary and a nonzero finite volume, and must assign zero probability to any 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 [**appendix** 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: 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 the shape, so multiply *S* by *base*, then add 1 to *d*, then go to step 3.

Notes:

- See (Li and El Gamal 2016)**(4)** and (Oberhoff 2018)**(5)** for related work on

encoding random points uniformly distributed in a shape.

- 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 area the shape takes up in the bounding box, the higher the acceptance rate.
- Devroye (1986, chapter 8, section 3)**(6)** 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.)
- Besides a grid, another useful data structure is a *mapped regular paving* (Harlow et al. 2012)**(7)**, 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 (e.g., 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.

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 a full diamond, step 5.3 in the algorithm is done for all *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 a full

hypersphere with radius 1, step 5.3 in the algorithm is done for all N dimensions. In the case of a 2-dimensional circle, this algorithm thus adapts the well-known rejection technique of generating X and Y coordinates until $X^2+Y^2 < 1$ (e.g., (Devroye 1986, p. 230 et seq.)**(6)**).

- 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 a full astroid, step 5.3 in the algorithm is done for all N dimensions.

3.2 Building an Arbitrary-Precision Sampler

In many cases, if a continuous distribution—

- has a probability density function (PDF), or a function proportional to the PDF, with a known symbolic form,
- has a cumulative distribution function (CDF) with a known symbolic form,
- takes on only values 0 or greater, and
- has a PDF that has an infinite tail to the right, is bounded from above (that is, *PDF(0)* is other than infinity), and decreases monotonically,

it may be possible to describe an arbitrary-precision sampler 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 a positive-sign zero-integer-part uniform PSRN, *ret*.
6. Create an input coin that calls **SampleGeometricBag** on the PSRN *ret*. Run a Bernoulli factory algorithm that simulates the probability $C(i, \lambda)$, using the input coin (here, λ is the probability built up in *ret* via **SampleGeometricBag**, and lies in the interval [0, 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 in the interval $[PDF(intval), \max(1, PDF(intval))]$, 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 monotonically 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.
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 numbers**, 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 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)[6]; an exception is that instead of generating a uniform random number 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.

3.3 Mixtures

A *mixture* involves sampling one of several distributions, where each distribution has a separate probability of being sampled. In general, an arbitrary-precision sampler is possible if all of the following conditions are met:

- There is a finite number of distributions to choose from.
- The probability of sampling each distribution is a rational number, or it can be expressed as a function for which a **Bernoulli factory algorithm** exists.
- For each distribution, an arbitrary-precision sampler exists.

> **Example:** One example of a mixture is two beta distributions, with separate parameters. One beta distribution is chosen with probability $\exp(-3)$ (a probability for which a Bernoulli factory algorithm exists) and the other is chosen with the opposite probability. For the two beta distributions, an arbitrary-precision sampling algorithm exists (see my article on **partially-sampled random numbers (PSRNs)** for details).

4 Specific Arbitrary-Precision Samplers

4.1 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. With probability $\exp(-(k * 2 + 1)/y)$, go to step 3. Otherwise, add 1 to k and repeat this step. (The probability check should be done with the **exp(−x/y) algorithm** in "**Bernoulli Factory Algorithms**", with $x/y = (k * 2 + 1)/y$.)
3. (Now we sample the piece located at $[k, k + 1)$.) Create a positive-sign zero-integer-part uniform PSRN, and create an input coin that returns the result of **SampleGeometricBag** on that uniform PSRN.
4. Set ky to $k * k / y$.
5. (At this point, we simulate $\exp(-U^2/y)$, $\exp(-k^2/y)$, $\exp(-U*k*2/y)$, as well as a scaled-down version of $U + k$, where U is the number built up by the uniform PSRN.) Call the **exp(−x/y) algorithm** with $x/y = ky$, then call the **exp(−(λ^k * x)) algorithm** using the input coin from step 2, $x = 1/y$, and $k = 2$, then call the **exp(−(λ^k * (x+m)))** algorithm using the same input coin, $x+m = \text{floor}(k * 2 / y)$, and $k = 1$, 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. 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 $\text{rem}(k / 2^{z - 1 - N}, 2)$. (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 (e.g., 0 or 1 for base 2), set the item at that position to a digit chosen uniformly at random (e.g., either 0 or 1 for base 2), increasing the 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 $\text{rem}(k / 2^{z - 1 - i}, base)$, 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 next, etc.).
    2. Set da to the item at that position, and set db to a digit chosen uniformly at random (that is, an integer in the interval $[0, base)$).
    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.

4.2 Sum of Exponential Random Numbers

An arbitrary-precision sampler for the sum of n exponential random numbers (also known as the Erlang(n) or gamma(n) distribution) is doable via partially-sampled uniform random numbers, though it is obviously inefficient for large values of n.

1. Generate n exponential random numbers with a rate of 1 via the **ExpRand** or **ExpRand2** algorithm described in my article on **partially-sampled random numbers (PSRNs)**. These numbers will be uniform PSRNs; this algorithm won't work for exponential PSRNs (e-rands), described in the same article, because the sum of two e-rands may follow a subtly wrong distribution. By contrast, generating exponential random numbers 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 numbers generated in step 1 by applying the **UniformAdd** algorithm given in another document.

4.3 Hyperbolic Secant Distribution

The following algorithm adapts the rejection algorithm from p. 472 in (Devroye 1986)[6] for arbitrary-precision sampling.

1. Generate a uniform PSRN, call it *ret*, and turn it into an exponential random number with a rate of 1, using an algorithm that employs rejection from the uniform distribution.
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.

4.4 Reciprocal of Power of Uniform

The following algorithm generates a PSRN of the form $1/U^{1/x}$, where U is a uniform random number in [0, 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 a positive-sign zero-integer-part uniform PSRN, *ret*.
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 lies in the interval [0, 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 number can't lie in the interval [0, 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 $intval^{x+1}/(i+\lambda)^{x+1}$.

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

The following algorithm generates a PSRN of the form $U/(1-U)$, where U is a uniform random number in [0, 1].

1. 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 a positive-sign zero-integer-part uniform PSRN, *ret*.
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 lies in the interval [0, 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 $(intval+1)^2/(i+\lambda+1)^2$.

4.6 Arc-Cosine Distribution

Here we reimplement an example from Devroye's book *Non-Uniform Random Variate Generation* (Devroye 1986, pp. 128–129)**(6)**. The following arbitrary-precision sampler generates a random number from a distribution with the following cumulative distribution function (CDF): `1 - cos(pi*x/2)`. The random number will be in the interval [0, 1]. Note that the result is the same as applying acos(U)*2/π, where U is a uniform [0, 1] random number, as pointed out by Devroye. The algorithm follows.

1. Call the **kthsmallest** algorithm with `n = 2` and `k = 2`, but without filling it with digits at the last step. Let *ret* be the result.
2. Set m to 1.
3. Call the **kthsmallest** algorithm with `n = 2` and `k = 2`, but without filling it with digits at the last step. Let u be the result.
4. With probability 4/(4*m*m + 2*m), call the **URandLess** algorithm with parameters u

and *ret* in that order, and if that call returns 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, we incorporate an erratum in the algorithm on page 129 of the book.)

5. If *m* is odd, 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, go to step 1.

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

```
def example_4_2_1(rg, bern, precision=53):
    while True:
       ret=rg.kthsmallest_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
```

4.7 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(*k*))/(1+exp(*k*+1))** described in "**Bernoulli Factory Algorithms**"). 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.) Generate a uniform(0, 1) random number, call it f.
4. (Steps 4 through 7 succeed with probability $\exp(-(f+k))/(1+\exp(-(f+k)))^2$.) With probability 1/2, go to step 3.
5. Run the **algorithm for exp(−*k*/1)** (described in "Bernoulli Factory Algorithms"), then **sample from the number *f*** (e.g., call **SampleGeometricBag** on f if f is implemented as a uniform PSRN). If any of these calls returns 0, go to step 4.
6. 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. Run the **algorithm for exp(−*k*/1)** and **sample from the number *f*** (e.g., call **SampleGeometricBag** on f if f is implemented as a uniform PSRN). If both calls return 1, go to step 3. Otherwise, go to step 6.

4.8 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 **UniformDivision** algorithm (described in the article on PSRNs) 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.

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

1. (Samples the integer part of the random number.) Generate a random number that expresses the number of failed trials before the first success, where each trial succeeds with probability $1-1/x$. Set *k* to that random number. (This is also known as a "geometric random number", but this terminology is avoided because it has conflicting meanings in academic works. If *x* is 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.)
2. (The rest of the algorithm samples the fractional part.) Generate a uniform (0, 1) random number, call it *f*.
3. Create a μ input coin that does the following: "**Sample from the number *f*** (e.g., call **SampleGeometricBag** on *f* if *f* is implemented as a uniform PSRN), then run the **algorithm for ln(2)** (described in "Bernoulli Factory Algorithms"). If both calls return 1, return 1. Otherwise, return 0." (This simulates the probability $\lambda = f*\ln(2)$.) If *x* is not a power of 2, also create a ν input coin that does the following: "**Sample from the number *f***, then run the **algorithm for ln(1 + *y*/*z*)** (described in "Bernoulli Factory Algorithms") with $y/z = (x-2^b)/2^b$. If both calls return 1, return 1. Otherwise, return 0."
4. Run the **algorithm for exp(−*λ*)** (described in "Bernoulli Factory Algorithms") *b* times, using the μ input coin. If *x* is not a power of 2, 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 number with rate ln(*x*) and bounded by *m* has a similar algorithm to this one. Step 1 is changed to read as follows: "Set *k* to a bounded-geometric($1-1/x$, *m*) random number (Bringmann and Friedrich 2013)**(8)**, or more simply, the lesser of *m* or the number of failed trials before

the first success, where each trial succeeds with probability $1-1/x$. (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 (note that 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 number in step 2, a uniform(0,μ) random number 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 in the interval (0, 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 number. Unlike the similar algorithm of Ahrens and Dieter (1972)[9], this one doesn't require a table of probability values.

1. (Samples the integer part of the random number. 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.) Generate a uniform (0, 1) random number, call it f.
3. Create an input coin that does the following: "**Sample from the number *f*** (e.g., call **SampleGeometricBag** on f if f is implemented as a uniform PSRN), then run the **algorithm for ln(2)** (described in "Bernoulli Factory Algorithms"). If both calls return 1, return 1. Otherwise, return 0." (This simulates the probability $\lambda = f*\ln(2)$.)
4. Run the **algorithm for exp(−*λ*)** (described in "Bernoulli Factory Algorithms"), 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.

4.10 Lindley Distribution and Lindley-Like Mixtures

A random number that follows the Lindley distribution (Lindley 1958)[10] with parameter θ (a rational number greater than 0) can be generated as follows:

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

For the Garima distribution (Shanker 2016)[11], $w = (1+\theta)/(2+\theta)$.

For the i-Garima distribution (Singh and Das 2020)[12], $w = (2+\theta)/(3+\theta)$.

For the mixture-of-weighted-exponential-and-weighted-gamma distribution in (Iqbal and Iqbal 2020)[13], two exponential random numbers (rather than one) are generated in step 1, and three (rather than two) are generated in step 2.

5 Requests and Open Questions

We would like to see new implementations of the following:

- Algorithms that implement **InShape** for specific closed curves, specific closed surfaces, and specific signed distance functions. Recall that **InShape** determines whether a box lies inside, outside, or partly inside or outside a given curve or surface.
- Descriptions of new arbitrary-precision algorithms that use the skeleton given in the section "Building an Arbitrary-Precision Sampler".

The appendix contains implementation notes for **InShape**, which determines whether a box is outside or partially or fully inside a shape. However, practical implementations of **InShape** will generally only be able to evaluate a shape pointwise. What are necessary and/or sufficient conditions that allow an implementation to correctly classify a box just by evaluating the shape pointwise?

6 Notes

- (1) Fishman, D., Miller, S.J., "Closed Form Continued Fraction Expansions of Special Quadratic Irrationals", ISRN Combinatorics Vol. 2013, Article ID 414623 (2013).
- (2) choose(n, k) = $n!/(k! * (n - k)!)$ is a binomial coefficient. It can be calculated, for example, by calculating $i/(n-i+1)$ for each integer i in 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). Note that for all $m>0$, choose(m, 0) = choose(m, m) = 1 and choose(m, 1) = choose(m, $m-1$) = m.
- (3) Thomas, A.C., Blanchet, J., "**A Practical Implementation of the Bernoulli Factory**", arXiv:1106.2508v3 [stat.AP], 2012.
- (4) C.T. Li, A. El Gamal, "**A Universal Coding Scheme for Remote Generation of Continuous Random Variables**", arXiv:1603.05238v1 [cs.IT], 2016
- (5) Oberhoff, Sebastian, "**Exact Sampling and Prefix Distributions**", *Theses and Dissertations*, University of Wisconsin Milwaukee, 2018.
- (6) Devroye, L., ***Non-Uniform Random Variate Generation***, 1986.
- (7) Harlow, J., Sainudiin, R., Tucker, W., "Mapped Regular Pavings", *Reliable Computing* 16 (2012).
- (8) 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).
- (9) Ahrens, J.H., and Dieter, U., "Computer methods from sampling from the exponential and normal distributions", *Communications of the ACM* 15, 1972.
- (10) Lindley, D.V., "Fiducial distributions and Bayes' theorem", *Journal of the Royal Statistical Society Series B*, 1958.
- (11) Shanker, R., "Garima distribution and its application to model behavioral science data", *Biom Biostat Int J.* 4(7), 2016.
- (12) Singh, B.P., Das, U.D., "**On an Induced Distribution and its Statistical Properties**", arXiv:2010.15078 [stat.ME], 2020.
- (13) 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.
- (14) 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.
- (15) Daumas, M., Lester, D., Muñoz, C., "**Verified Real Number Calculations: A Library for Interval Arithmetic**", arXiv:0708.3721 [cs.MS], 2007.
- (16) Karney, C.F.F., "**Sampling exactly from the normal distribution**", arXiv:1303.6257v2

[physics.comp-ph], 2014.

- (17) I thank D. Eisenstat from the *Stack Overflow* community for leading me to this insight.
- (18) 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.

7 Appendix

7.1 Ratio of Uniforms

The Cauchy sampler given earlier demonstrates the *ratio-of-uniforms* technique for sampling a distribution (Kinderman and Monahan 1977)**(14)**. It involves transforming the distribution's density function (PDF) into a compact shape. The ratio-of-uniforms method appears here in the appendix, particularly since it can involve calculating upper and lower bounds of transcendental functions which, while it's possible to achieve in rational arithmetic (Daumas et al., 2007)**(15)**, is less elegant than, say, the normal distribution sampler by Karney (2014)**(16)**, which doesn't require calculating logarithms or other transcendental functions.

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

- for all x, $PDF(x) < \infty$ and $PDF(x)*x^2 < \infty$, where *PDF* is the distribution's PDF or a function proportional to the PDF,
- *PDF* is continuous almost everywhere, 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 highest value of x*sqrt(*PDF*(x)) anywhere, and *d2* is not less than the highest value of sqrt(*PDF*(x)) anywhere, 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 $z <= 0$ 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 (e.g., 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 **UniformDivision** algorithm (described in the article on PSRNs) 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.

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.

7.2 Implementation Notes for Box/Shape Intersection

The "**Uniform Distribution Inside N-Dimensional Shapes**" algorithm 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 $f(t1, ..., tN) <= 0$, **InShape** should use rational interval arithmetic (such as the one given in (Daumas et al., 2007)**(15)**), where the two bounds of each interval are rational numbers with arbitrary-precision numerators and denominators. Then, **InShape** should build one interval for each dimension of the box and evaluate f using those intervals**(17)** 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 $y^2 <= \exp(-(x/y)^2/2)$, the inequality becomes $2*\ln(y) <= -(x/y)^2/2$ and thus becomes $2*\ln(y) + (x/y)^2/2 <= 0$ and thus becomes $4*\ln(y) + (x/y)^2 <= 0$.

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 $\text{sqrt}(N)/(S*2)$, 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 shape 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.

7.3 SymPy Code for Piecewise Linear Factory Functions

```
def bernstein_n(func, x, n, pt=None):
  # Bernstein operator.
  # Create a polynomial that approximates func, which in turn uses
  # the symbol x.  The polynomial's degree is n and is evaluated
  # at the point pt (or at x if not given).
  if pt==None: pt=x
  ret=0
  v=[binomial(n,j) for j in range(n//2+1)]
  for i in range(0, n+1):
    oldret=ret
    bin=v[i] if i<len(v) else v[n-i]
    ret+=func.subs(x,S(i)/n)*bin*pt**i*(1-pt)**(n-i)
    if pt!=x and ret==oldret and ret>0: break
  return ret

def inflec(y,eps=S(2)/10,mult=2):
  # Calculate the inflection point (x) given y, eps, and mult.
  # The formula is not found in the paper by Thomas and
  # Blanchet 2012, but in
  # the supplemental source code uploaded by
  # A.C. Thomas.
  po=5 # Degree of y-to-x polynomial curve
  eps=S(eps)
  mult=S(mult)
  x=-((y-(1-eps))/eps)**po/mult + y/mult
  return x

def xfunc(y,sym,eps=S(2)/10,mult=2):
  # Calculate Bernstein "control polygon" given y,
  # eps, and mult.
  return Min(sym*y/inflec(y,eps,mult),y)

def calc_linear_func(eps=S(5)/10, mult=1, count=10):
   # Calculates the degrees and Y parameters
   # of a sequence of polynomials that converge
   # from above to min(x*mult, 1-eps).
   # eps must be in the interval (0, 1).
   # Default is 10 polynomials.
   polys=[]
```

```
  eps=S(eps)
  mult=S(mult)
  count=S(count)
  bs=20
  ypt=1-(eps/4)
  x=symbols('x')
  tfunc=Min(x*mult,1-eps)
  tfn=tfunc.subs(x,(1-eps)/mult).n()
  xpt=xfunc(ypt,x,eps=eps,mult=mult)
  bits=5
  #oldbx=None
  i=0
  lastbxn = 1
  diffs=[]
  while i<count:
    bx=bernstein_n(xpt,x,bits,(1-eps)/mult)
    bxn=bx.n()
    if bxn > tfn and bxn < lastbxn:
      # Dominates target function
      #if oldbx!=None:
      #   diffs.append(bx)
      #   diffs.append(oldbx-bx)
      #oldbx=bx
      oldxpt=xpt
      lastbxn = bxn
      polys.append([bits,ypt])
      print("    [%d,%s]," % (bits,ypt))
      # Find y2 such that y2 < ypt and
      # bernstein_n(oldxpt,x,bits,inflec(y2, ...)) >= y2,
      # so that next Bernstein expansion will go
      # underneath the previous one
      while True:
        ypt-=(ypt-(1-eps))/4
        xpt=inflec(ypt,eps=eps,mult=mult).n()
        bxs=bernstein_n(oldxpt,x,bits,xpt).n()
        if bxs>=ypt.n():
           break
      xpt=xfunc(ypt,x,eps=eps,mult=mult)
      bits+=20
      i+=1
    else:
      bits=int(bits*200/100)
  return polys

calc_linear_func(count=8)
```

8 License