More Algorithms for Arbitrary-Precision Sampling

This version of the document is dated 2021-07-05.

Peter Occil

Abstract: 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. They supplement my pages on Bernoulli factory algorithms and partially-sampled random numbers.

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

1 Introduction

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**

2 Contents

3 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.

3.1 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. Do the following process repeatedly, until this run of the algorithm returns a value:
    1. With probability $kp/(1 + kp)$, return a number that is 1 with probability $1/kp$ and 0 otherwise.
    2. Do a separate run of the currently running algorithm, but with $pos = pos + 1$. If the separate run returns 1, return 0.

3.2 Ratio of Lower Gamma Functions (γ(*m*, *x*)/γ(*m*, 1)).

1. Set *ret* to the result of **kthsmallest** with the two parameters m and m. (Thus, *ret* is distributed as $u^{1/m}$ where u is a uniform(0, 1) random number; although **kthsmallest** accepts only integers, this formula works for any m greater than 0.)
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 x 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.

Derivation: See Formula 1 in the section "**Probabilities Arising from Certain Permutations**", where:

- `ECDF(x)` is the uniform(0,1) distribution's cumulative distribution function, namely x if x is in [0, 1], 0 if x is less than 0, and 1 otherwise.
- `DPDF(x)` is the probability density function for the maximum of m uniform(0,1) random numbers, namely $m*x^{m-1}$ if x is in [0, 1], and 0 otherwise.

3.3 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. (The remaining steps are done repeatedly, until the algorithm returns a value.) 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.

3.4 cosh(λ) − 1

There are two algorithms.

The first 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)**(2)**. (In this document, $n!$ = 1*2*3*...*n is known as n factorial.)

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].

The second algorithm is one I found that takes advantage of the convex combination method.

1. ("Geometric" random number n.) Generate unbiased random bits until a zero is generated this way. Set n to 2 plus the number of ones generated this way. (The number n is generated with probability $g(n)$, as given below.)
2. (The next two steps succeed with probability $w_n(\lambda)/g(n)$.) If n is odd, return 0. Otherwise, with probability $2^{n-1}/(n!)$, go to the next step. Otherwise, return 0.
3. Flip the input coin n times or until a flip returns 0, whichever happens first. Return 1 if all the flips, including the last, returned 1. Otherwise, return 0.

Derivation: Follows from rewriting cosh(λ)−1 as the following series: $\sum_{n=0,1,...} w_n(\lambda) = \sum_{n=0,1,...} g(n)*(w_n(\lambda)/g(n))$, where—

- $g(n)$ is $(1/2)*(1/2)^{n-2}$ if $n \geq 2$, or 0 otherwise, and
- $w_n(\lambda)$ is $\lambda^n/(n!)$ if $n \geq 2$ and n is even, or 0 otherwise.

3.5 exp(λ/4)/2

1. ("Geometric" random number n.) Generate unbiased random bits until a zero is generated this way. Set n to the number of ones generated this way. (The number n is generated with probability $g(n)$, as given below.)
2. (The next two steps succeed with probability $w_n(\lambda)/g(n)$.) With probability $1/(2^n*(n!))$, go to the next step. Otherwise, return 0.

3. Flip the input coin n times or until a flip returns 0, whichever happens first. Return 1 if all the flips, including the last, returned 1. Otherwise, return 0.

Derivation: Follows from rewriting $\exp(\lambda/4)/2$ in a similar manner to $\cosh(\lambda)-1$, where this time, $g(n)$ is $(1/2)*(1/2)^n$ (the "geometric" probabilities"), and $w_n(\lambda)$ is the appropriate term for n in the target function's Taylor series.

Additional functions:

To simulate:	**Follow this algorithm, except the probability in step 2 is:**
$\exp(\lambda)/4$.	$2^{n-1}/(n!)$.
$\exp(\lambda)/6$.	$2^n/(3*(n!))$.
$\exp(\lambda/2)/2$.	$1/(n!)$.

3.6 sinh(λ)/2

This algorithm involves an application of the general martingale algorithm to the Taylor series for $\sinh(\lambda)/2$, 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. Do the following process repeatedly, until this algorithm returns a value:
    1. If w is not 0, flip the input coin and multiply w by the result of the flip. Do this substep again.
    2. If w is 0, set u to ℓ and go to the fourth substep. (No more terms are added here.)
    3. 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 $\ell + err$.
    4. If ret is less than (or equal to) ℓ, return 1. If ret is less than u, go to the next substep. 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**.)
    5. Add 1 to n.

3.7 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: Generate an unbiased random bit. If that bit is 1 (which happens with probability 1/2), flip the input coin and return the result.
2. Do the following process repeatedly, until this run of the algorithm returns a value:
    1. 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.
    2. Do a separate run of the currently running algorithm, but with $k = k + 2$. If the separate run returns 1, return 0.

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.[3]
     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. (The remaining steps are done repeatedly, until the algorithm returns a value.) 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 $\text{abs}(b)*2^m*(2^m-1)/(m!)$.
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.

3.8 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)[4]. These polynomials can then be used to generate heads with probability $f(\lambda)$ using the algorithms given in "**General Factory Functions**".

In this section, **fbelow(n, k)** and **fabove(n, k)** are the k^{th} coefficients (with k starting at 0) of the lower and upper polynomials, respectively, in Bernstein form of degree n.

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 degree n, **fbelow(n, k)** is $\min((k/n)*mult, 1-\varepsilon)$, and **fabove($n$, k)** is $\min((k/n)*y/x, y)$. (**fbelow** matches f because f is *concave* in the interval [0, 1], which roughly means that its rate of growth there never goes up.)
5. 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.

> **Note:** In Nacu and Peres (2005)**(5)**, the following polynomial sequences were suggested to simulate $\min(\lambda*2, 1 - 2*\varepsilon)$, provided $\varepsilon < 1/8$, where n is a power of 2. However, with these sequences, an extraordinary number of input coin flips is required to simulate this function each time.
>
> - **fbelow(n, k)** = $\min((k/n)*2, 1 - 2*\varepsilon)$.
> - **fabove(n, k)** = $\min((k/n)*2, 1 - 2*\varepsilon)$ + $(\max(0, k/n+3*\varepsilon - 1/2)/(\varepsilon/(1-\text{sqrt}(2)/2)))*\text{sqrt}(2/n)$ + $(72*\max(0, k/n-1/9)/(1-\exp(-2*\varepsilon*\varepsilon)))*\exp(-2*\varepsilon*\varepsilon*n)$.

My own algorithm for $\min(\lambda, 1/2)$ is as follows. See the **appendix** for the derivation of this algorithm.

1. Generate an unbiased random bit. If that bit is 1 (which happens with probability 1/2), flip the input coin and return the result.
2. Run the algorithm for $\min(\lambda, 1-\lambda)$ given later, and return the result of that run.

And the algorithm for $\min(\lambda, 1-\lambda)$ is as follows:

1. (Random walk.) Generate unbiased random bits until more zeros than ones are generated this way for the first time. Then set m to $(n-1)/2+1$, where n is the number of bits generated this way.
2. (Build a degree-$m*2$ polynomial equivalent to $(4*\lambda*(1-\lambda))^m/2$.) Let z be $(4^m/2)/\text{choose}(m*2,m)$. Define a polynomial of degree $m*2$ whose $(m*2)+1$ Bernstein coefficients are all zero except the m^{th} coefficient (starting at 0), whose value is z. Elevate the degree of this polynomial enough times so that all its coefficients are 1 or less (degree elevation increases the polynomial's degree without changing its shape or position; see the derivation in the appendix). Let d be the new polynomial's degree.
3. (Simulate the polynomial, whose degree is d (Goyal and Sigman 2012)**(6)**.) Flip the input coin d times and set h to the number of ones generated this way. Let a be the h^{th} Bernstein coefficient (starting at 0) of the new polynomial. With probability a, return 1. Otherwise, return 0.

I suspected that the required degree d would be $\text{floor}(m*2/3)+1$, as described in the appendix. With help from the **MathOverflow community**, steps 2 and 3 of the algorithm above can be described more efficiently as follows:

- (3.) Let r be $\text{floor}(m*2/3)+1$, and let d be $m*2+r$.
- (4.) (Simulate the polynomial, whose degree is d.) Flip the input coin d times and set h to the number of ones generated this way. Let a be (1/2) * $2^{m*2}*\text{choose}(r,h-m)/\text{choose}(d, h)$ (the polynomial's h^{th} Bernstein coefficient starting at 0; the first term is 1/2 because the polynomial being simulated has the value 1/2

at the point 1/2). With probability a, return 1. Otherwise, return 0.

The $\min(\lambda, 1-\lambda)$ algorithm can be used to simulate certain other piecewise linear functions with three breakpoints, and algorithms for those functions are shown in the following table. In the table, μ is the unknown probability of heads of a second input coin.

Breakpoints	**Algorithm**
0 at 0; 1/2 at 1/2; and μ at 1.	Flip the μ input coin. If it returns 1, flip the λ input coin and return the result. Otherwise, run the algorithm for $\min(\lambda, 1-\lambda)$ using the λ input coin, and return the result of that run.
0 at 0; $\mu/2$ at 1/2; and $\mu/2$ at 1.	Flip the μ input coin. If it returns 0, return 0. Otherwise, generate an unbiased random bit. If that bit is 1 (which happens with probability 1/2), flip the λ input coin and return the result. Otherwise, run the algorithm for $\min(\lambda, 1-\lambda)$ using the λ input coin, and return the result of that run.
μ at 0; 1/2 at 1/2; and 0 at 1.	Flip the μ input coin. If it returns 1, flip the λ input coin and return 1 minus the result. Otherwise, run the algorithm for $\min(\lambda, 1-\lambda)$ using the λ input coin, and return the result of that run.
1 at 0; 1/2 at 1/2; and μ at 1.	Flip the μ input coin. If it returns 0, flip the λ input coin and return 1 minus the result. Otherwise, run the algorithm for $\min(\lambda, 1-\lambda)$ using the λ input coin, and return 1 minus the result of that run.
μ at 0; 1/2 at 1/2; and 1 at 1.	Flip the μ input coin. If it returns 0, flip the λ input coin and return the result. Otherwise, run the algorithm for $\min(\lambda, 1-\lambda)$ using the λ input coin, and return 1 minus the result of that run.

3.9 Non-Negative Factories

The Bernoulli factory approach can be extended in two ways to produce random numbers beyond the interval [0, 1]. Both algorithms use a different *oracle* (black box) than coins that output heads or tails.

Algorithm 1. Say we have an oracle that produces independent random numbers in the interval $[a, b]$, and these numbers have an unknown mean of μ. The goal is now to produce non-negative random numbers that average to $f(\mu)$. Unless f is constant, this is possible if and only if—

- f is continuous on $[a, b]$, and
- $f(\mu)$ is bounded from below by ε*min($(\mu - a)^n$, $(b - \mu)^n$) for some integer n and some ε greater than 0 (loosely speaking, f is non-negative and neither touches 0 inside (a, b) nor moves away from 0 more slowly than a polynomial)

(Jacob and Thiery 2015)[7]. (Here, a and b are both rational numbers and may be less than 0.)

In the algorithm below, let κ be a rational number greater than the maximum value of f in the interval $[a, b]$, and let $g(\lambda) = f(a + (b-a)*\lambda)/\kappa$.

1. Create a λ input coin that does the following: "Take a number from the oracle, call it x. With probability $(x-a)/(b-a)$ (see note below), return 1. Otherwise, return 0."
2. Run a Bernoulli factory algorithm for $g(\lambda)$, using the λ input coin. Then return κ times the result.

   Note: The check "With probability $(x-a)/(b-a)$" is exact if the oracle produces only rational numbers. If the oracle can produce irrational numbers (such as numbers that follow a beta distribution or another continuous distribution),

then the code for the oracle should use uniform **partially-sampled random numbers (PSRNs)**. In that case, the check can be implemented as follows. Let x be a uniform PSRN representing a number generated by the oracle. Set y to **RandUniformFromReal**($b-a$), then the check succeeds if **URandLess**(y, **UniformAddRational**(x, $-a$)) returns 1, and fails otherwise.

Example: Suppose an oracle produces random numbers in the interval [3, 13] with unknown mean μ, and we seek to use the oracle to produce non-negative random numbers with mean $f(\mu) = -319/100 + \mu*103/50 - \mu^2*11/100$, which is a polynomial with Bernstein coefficients [2, 9, 5] in the given interval. Then since 8 is greater than the maximum of f in that interval, $g(\lambda)$ is a degree-2 polynomial with Bernstein coefficients [2/8, 9/8, 5/8] in the interval [0, 1]. g can't be simulated as is, though, but by increasing g's degree to 3 we get the Bernstein coefficients [1/4, 5/6, 23/24, 5/8], which are all less than 1 so we can proceed with the following algorithm (see "**Certain Polynomials**"):

1. Set *heads* to 0.
2. Generate three random numbers from the oracle (which must produce random numbers in the interval [3, 13]). For each number x: With probability $(x-3)/(10-3)$, add 1 to *heads*.
3. Depending on *heads*, return 8 (that is, 1 times the upper bound) with the given probability, or 0 otherwise: *heads*=0 → probability 1/4; 1 → 5/6; 2 → 23/24; 3 → 5/8.

Algorithm 2. This algorithm takes an oracle and produces non-negative random numbers that average to the mean of $f(X)$, where X is a number produced by the oracle. The algorithm appears in the appendix, however, because it requires applying an arbitrary function (here, f) to a potentially irrational number.

3.10 Pushdown Automata for Square-Root-Like Functions

The following algorithm extends the square-root construction of Flajolet et al. (2010)**(8)**, takes an input coin with probability of heads λ, and returns 1 with probability—

- $f(\lambda) = (1 - \lambda)/\text{sqrt}(1 + 4*\lambda*g(\lambda)*(g(\lambda) - 1))$, or equivalently,
- $f(\lambda) = (1 - \lambda) * \sum_{n=0,1,...} \lambda^n*g(\lambda)^n*(1 - g(\lambda))^n*\text{choose}(2*n, n) = (1 - \lambda) * \sum_{n=0,1,...} (\lambda*g(\lambda)*(1 - g(\lambda)))^n*\text{choose}(2*n, n)$, or equivalently,
- $f(\lambda) = (1 - \lambda) * \text{OGF}(\lambda*g(\lambda)*(1 - g(\lambda)))$,

and 0 otherwise, where—

- $g(\lambda)$ is a continuous function that maps the half-open interval [0, 1) to the closed interval [0, 1] and admits a Bernoulli factory, and
- $\text{OGF}(x) = \sum_{n=0,1,...} x^n*\text{choose}(2*n, n)$ is the algorithm's ordinary generating function.

If g is a rational function (a ratio of two polynomials) with rational coefficients, then f is an algebraic function and can be simulated by a *pushdown automaton* (a state machine that keeps a stack of symbols) (Mossel and Peres 2005)**(9)**, as in the algorithm below. But this algorithm will still work even if g is not a rational function.

1. Set d to 0.
2. Do the following process repeatedly until this run of the algorithm returns a value:
    1. Flip the input coin. If it returns 1, go to the next substep. Otherwise, return

either 1 if d is 0, or 0 otherwise.
2. Run a Bernoulli factory algorithm for $g(\lambda)$. If the run returns 1, add 1 to d. Otherwise, subtract 1 from d. Do this substep again.

As a pushdown automaton, this algorithm can be given as follows (except the "Do this substep again" part). Let the stack have the single symbol EMPTY, and start at the state POS-S1. Based on the current state, the last coin flip (HEADS or TAILS), and the symbol on the top of the stack, set the new state and replace the top stack symbol with zero, one, or two symbols. These *transitions* can be written as follows:

- (POS-S1, HEADS, *topsymbol*) → (POS-S2, {*topsymbol*}) (set state to POS-S2, keep *topsymbol* on the stack).
- (NEG-S1, HEADS, *topsymbol*) → (NEG-S2, {*topsymbol*}).
- (POS-S1, TAILS, EMPTY) → (ONE, {}) (set state to ONE, pop the top symbol from the stack).
- (NEG-S1, TAILS, EMPTY) → (ONE, {}).
- (POS-S1, TAILS, X) → (ZERO, {}).
- (NEG-S1, TAILS, X) → (ZERO, {}).
- (ZERO, *flip*, *topsymbol*) → (ZERO, {}).
- (POS-S2, *flip*, *topsymbol*) → Add enough transitions to the automaton to simulate $g(\lambda)$ by a finite-state machine (only possible if g is rational with rational coefficients). Transition to POS-S2-ZERO if the machine outputs 0, or POS-S2-ONE if the machine outputs 1.
- (NEG-S2, *flip*, *topsymbol*) → Same as before, but the transitioning states are NEG-S2-ZERO and NEG-S2-ONE, respectively.
- (POS-S2-ONE, *flip*, *topsymbol*) → (POS-S1, {*topsymbol*, X}) (replace top stack symbol with *topsymbol*, then push X to the stack).
- (POS-S2-ZERO, *flip*, EMPTY) → (NEG-S1, {EMPTY, X}).
- (POS-S2-ZERO, *flip*, X) → (POS-S1, {}).
- (NEG-S2-ZERO, *flip*, *topsymbol*) → (NEG-S1, {*topsymbol*, X}).
- (NEG-S2-ONE, *flip*, EMPTY) → (POS-S1, {EMPTY, X}).
- (NEG-S2-ONE, *flip*, X) → (NEG-S1, {}).

The machine stops when it removes EMPTY from the stack, and the result is either ZERO (0) or ONE (1).

For the following algorithm, which extends the end of Note 1 of the Flajolet paper, the probability is—

- $f(\lambda) = (1 - \lambda) * \sum_{n=0,1,\ldots} \lambda^{H*n} * g(\lambda)^n * (1 - g(\lambda))^{(H-1)*n} * \text{choose}(H*n, n)$,

where $H \geq 2$ is an integer, and g has the same meaning as earlier.

1. Set d to 0.
2. Do the following process repeatedly until this run of the algorithm returns a value:
    1. Flip the input coin. If it returns 1, go to the next substep. Otherwise, return either 1 if d is 0, or 0 otherwise.
    2. Run a Bernoulli factory algorithm for $g(\lambda)$. If the run returns 1, add $(H-1)$ to d. Otherwise, subtract 1 from d. (Note that this substep is not done again.)

The following algorithm simulates the probability—

- $f(\lambda) = (1 - \lambda) * \sum_{n=0,1,\ldots} \lambda^n * (\sum_{m=0,1,\ldots,n} W(n, m) * g(\lambda)^m * (1 - g(\lambda))^{n-m} * \text{choose}(n, m))$
  $= (1 - \lambda) * \sum_{n=0,1,\ldots} \lambda^n * (\sum_{m=0,1,\ldots,n} V(n, m) * g(\lambda)^m * (1 - g(\lambda))^{n-m})$,

where g has the same meaning as earlier; $W(n, m)$ is 1 if $m*H$ equals $(n-m)*T$, or 0

otherwise; and $H \geq 1$ and $T \geq 1$ are integers. (In the first formula, the sum in parentheses is a polynomial in Bernstein form, in the variable $g(\lambda)$ and with only zeros and ones as coefficients. Because of the λ^n, the polynomial gets smaller as n gets larger. $V(n, m)$ is the number of n-letter words that have m heads *and* describe a walk that ends at the beginning.)

1. Set d to 0.
2. Do the following process repeatedly until this run of the algorithm returns a value:
    1. Flip the input coin. If it returns 1, go to the next substep. Otherwise, return either 1 if d is 0, or 0 otherwise.
    2. Run a Bernoulli factory algorithm for $g(\lambda)$. If the run returns 1 ("heads"), add H to d. Otherwise ("tails"), subtract T from d. (Note that this substep is not done again.)

4 General Arbitrary-Precision Samplers

4.1 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 $[0, d1] \times [0, d2] \times \ldots \times [0, dN]$ 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: 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 the shape, so multiply *S* by *base*, then add 1 to *d*, then go to step 3.

Notes:

- See (Li and El Gamal 2016)**(10)** and (Oberhoff 2018)**(11)** 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)**(12)** 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)**(13)**, 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 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 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 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.)**(12)**).
- 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 each of the N dimensions.

4.2 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)[12]; 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.

4.3 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).

4.4 Weighted Choice Involving PSRNs

Given n uniform PSRNs, called *weights*, with labels starting from 0 and ending at $n-1$, the following algorithm chooses an integer in [0, n) with probability proportional to its weight. Each weight's sign must be positive.

1. Create an empty list, then for each weight starting with weight 0, add the weight's integer part plus 1 to that list. For example, if the weights are [2.22...,0.001...,1.3...], in that order, the list will be [3, 1, 2], corresponding to integers 0, 1, and 2, in that order. Call the list just created the *rounded weights list*.
2. Choose an integer i with probability proportional to the weights in the rounded weights list. This can be done, for example, by taking the result of **WeightedChoice**(*list*), where *list* is the rounded weights list and **WeightedChoice** is given in "**Randomization and Samping Methods**".
3. Run **URandLessThanReal**(w, rw), where w is the original weight for integer i, and rw is the rounded weight for integer i in the rounded weights list. That algorithm returns 1 if w turns out to be less than rw. If the result is 1, return i. Otherwise, go to step 2.

5 Specific Arbitrary-Precision Samplers

5.1 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 first or third algorithm for **exp(−(λ^k * c))** using the same input coin, c = 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. 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 (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 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.

5.2 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.

5.3 Hyperbolic Secant Distribution

The following algorithm adapts the rejection algorithm from p. 472 in (Devroye 1986)[12] for arbitrary-precision sampling.

1. Generate *ret*, an exponential random number with a rate of 1 via the **ExpRand** or **ExpRand2** algorithm described in my article on **PSRNs**. 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.

5.4 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 $intval^{x+1}/(i+\lambda)^{x+1}$.

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

The following algorithm generates a PSRN distributed as $U/(1-U)$, where U is a uniform random variate in [0, 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 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 $(intval+1)^2/(i+\lambda+1)^2$.

5.6 Arc-Cosine Distribution

Here we reimplement an example from Devroye's book *Non-Uniform Random Variate Generation* (Devroye 1986, pp. 128–129)**(12)**. 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
```

5.7 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 $\exp(-(f+k))/(1+\exp(-(f+k)))^2$.) Generate an unbiased random bit. If that bit is 1 (which happens 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. 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. 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.

5.8 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.

5.9 Exponential Distribution with Unknown Rate λ, Lying in (0, 1]

Exponential random numbers can be generated using an input coin of unknown probability of heads of λ (which can be in the interval (0, 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)**(12)**. The algorithm follows:

1. Generate an exponential(1) random number 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 number 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.

5.10 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. 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 number.) 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 number", 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: "**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 $\lambda = f*\ln(2)$.) Then:
    - If x is a rational number, but 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."
    - 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: "**Sample from the number 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. Run the **algorithm for exp(−λ)** (described in "Bernoulli Factory Algorithms") b times, using the μ input coin. 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 number 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 number, which an algorithm of Bringmann and Friedrich (2013)**(14)** 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 (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)**(15)**, 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.

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.

5.11 Symmetric Geometric Distribution

Samples from the symmetric geometric distribution from (Ghosh et al. 2012)[16], with parameter λ, in the form of an input coin with unknown probability of heads of λ.

1. Flip the input coin until it returns 0. Set n to the number of times the coin returned 1 this way.
2. Run a **Bernoulli factory algorithm for 1/(2−λ)**, using the input coin. If the run returns 1, return n. Otherwise, return $-1 - n$.

This is similar to an algorithm mentioned in an appendix in Li (2021)[17], in which the input coin—

- has $\lambda = 1-\exp(-\varepsilon)$, and
- can be built as follows using another input coin with probability of heads ε: "Run a **Bernoulli factory algorithm for exp(−λ)** using the ε input coin, then return 1 minus the result."

5.12 Lindley Distribution and Lindley-Like Mixtures

A random number that follows the Lindley distribution (Lindley 1958)[18] with parameter θ (a real 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)[19], $w = (1+\theta)/(2+\theta)$.

For the i-Garima distribution (Singh and Das 2020)**(20)**, $w = (2+\theta)/(3+\theta)$.

For the mixture-of-weighted-exponential-and-weighted-gamma distribution in (Iqbal and Iqbal 2020)**(21)**, two exponential random numbers (rather than one) are generated in step 1, and three (rather than two) are generated in step 2.

> **Note:** 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.

6 Requests and Open Questions

1. 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".
2. 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?
3. Take a polynomial $f(\lambda)$ of even degree n of the form $\text{choose}(n,n/2)*\lambda^{n/2}*(1-\lambda)^{n/2}*k$, where k is greater than 1 (thus all f's Bernstein coefficients are 0 except for the middle one, which equals k). Suppose $f(1/2)$ lies in the interval (0, 1). If we do the degree elevation, described in the appendix, enough times (at least r times), then f's Bernstein coefficients will all lie in [0, 1]. The question is: how many degree elevations are enough? A **MathOverflow answer** showed that r is at least $m = (n/f(1/2)^2)/(1-f(1/2)^2)$, but is it true that $\text{floor}(m)+1$ elevations are enough?

7 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) 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.
- (3) $\text{choose}(n, k) = (1*2*3*...*n)/((1*...*k)*(1*...*(n-k))) = n!/(k! * (n - k)!)$ 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). Note that for every $m>0$, $\text{choose}(m, 0) = \text{choose}(m, m) = 1$ and $\text{choose}(m, 1) = \text{choose}(m, m-1) = m$; also, in this document, $\text{choose}(n, k)$ is 0 when k is less than 0 or greater than n.
- (4) Thomas, A.C., Blanchet, J., "**A Practical Implementation of the Bernoulli Factory**", arXiv:1106.2508v3 [stat.AP], 2012.
- (5) Nacu, Şerban, and Yuval Peres. "**Fast simulation of new coins from old**", The Annals of Applied Probability 15, no. 1A (2005): 93-115.

- [6] 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.
- [7] Jacob, P.E., Thiery, A.H., "On nonnegative unbiased estimators", Ann. Statist., Volume 43, Number 2 (2015), 769-784.
- [8] Flajolet, P., Pelletier, M., Soria, M., "**On Buffon machines and numbers**", arXiv:0906.5560 [math.PR], 2010
- [9] Mossel, Elchanan, and Yuval Peres. New coins from old: computing with unknown bias. Combinatorica, 25(6), pp.707-724, 2005.
- [10] C.T. Li, A. El Gamal, "**A Universal Coding Scheme for Remote Generation of Continuous Random Variables**", arXiv:1603.05238v1 [cs.IT], 2016
- [11] Oberhoff, Sebastian, "**Exact Sampling and Prefix Distributions**", *Theses and Dissertations*, University of Wisconsin Milwaukee, 2018.
- [12] Devroye, L., ***Non-Uniform Random Variate Generation***, 1986.
- [13] Harlow, J., Sainudiin, R., Tucker, W., "Mapped Regular Pavings", *Reliable Computing* 16 (2012).
- [14] 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).
- [15] Ahrens, J.H., and Dieter, U., "Computer methods for sampling from the exponential and normal distributions", *Communications of the ACM* 15, 1972.
- [16] Ghosh, A., Roughgarden, T., and Sundararajan, M., "Universally Utility-Maximizing Privacy Mechanisms", *SIAM Journal on Computing* 41(6), 2012.
- [17] Li, L., 2021. Bayesian Inference on Ratios Subject to Differentially Private Noise (Doctoral dissertation, Duke University).
- [18] Lindley, D.V., "Fiducial distributions and Bayes' theorem", *Journal of the Royal Statistical Society Series B*, 1958.
- [19] Shanker, R., "Garima distribution and its application to model behavioral science data", *Biom Biostat Int J.* 4(7), 2016.
- [20] Singh, B.P., Das, U.D., "**On an Induced Distribution and its Statistical Properties**", arXiv:2010.15078 [stat.ME], 2020.
- [21] 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.
- [22] 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.
- [23] Daumas, M., Lester, D., Muñoz, C., "**Verified Real Number Calculations: A Library for Interval Arithmetic**", arXiv:0708.3721 [cs.MS], 2007.
- [24] Karney, C.F.F., "**Sampling exactly from the normal distribution**", arXiv:1303.6257v2 [physics.comp-ph], 2014.
- [25] I thank D. Eisenstat from the *Stack Overflow* community for leading me to this insight.
- [26] Brassard, G., Devroye, L., Gravel, C., "Remote Sampling with Applications to General Entanglement Simulation", *Entropy* 2019(21)(92), **https://doi.org/10.3390/e21010092** .
- [27] Devroye, L., Gravel, C., "**Random variate generation using only finitely many unbiased, independently and identically distributed random bits**", arXiv:1502.02539v6 [cs.IT], 2020.
- [28] Keane, M. S., and O'Brien, G. L., "A Bernoulli factory", *ACM Transactions on Modeling and Computer Simulation* 4(2), 1994.
- [29] Wästlund, J., "**Functions arising by coin flipping**", 1999.
- [30] Dale, H., Jennings, D. and Rudolph, T., 2015, "Provable quantum advantage in randomness processing", *Nature communications* 6(1), pp. 1-4.
- [31] Tsai, Yi-Feng, Farouki, R.T., "Algorithm 812: BPOLY: An Object-Oriented Library of Numerical Algorithms for Polynomials in Bernstein Form", *ACM Trans. Math. Softw.* 27(2), 2001.
- [32] Lee, A., Doucet, A. and Łatuszyński, K., 2014. "**Perfect simulation using atomic regeneration with application to Sequential Monte Carlo**", arXiv:1407.5770v1 [stat.CO].

8 Appendix

8.1 Ratio of Uniforms

The Cauchy sampler given earlier demonstrates the *ratio-of-uniforms* technique for sampling a distribution (Kinderman and Monahan 1977)**(22)**. 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)**(23)**, is less elegant than, say, the normal distribution sampler by Karney (2014)**(24)**, which doesn't require calculating logarithms or other transcendental functions.

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

- for every 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 $z \le 0$ 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 (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.

8.2 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 $f(t1, ..., tN) \le 0$, **InShape** should use rational interval arithmetic (such as the one given in (Daumas et al., 2007)**(23)**), 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**(25)** 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 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.

8.3 Probability Transformations

The following algorithm takes a uniform partially-sampled random number (PSRN) as a "coin" and flips that "coin" using **SampleGeometricBag** (a method described in my **article on PSRNs**). 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)**(26)**, (Devroye 1986, p. 769)**(12)**, (Devroye and Gravel 2020) **(27)**. 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 bounded from below by $\min(U^n, (1-U)^n)$ for some integer n (Keane and O'Brien 1994)**(28)**).
  - Otherwise, $f(U)$ must map the interval [0, 1] to [0, 1] and be continuous everywhere except at a countable number of points.

  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)**(2)**.

The algorithm follows.

1. Set v to 0 and k to 1.
2. (v acts as a uniform(0, 1) random number 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:

1. 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.
2. This section appears in the appendix because this article is focused on algorithms that don't rely on calculations of irrational numbers.

8.4 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
    bino=v[i] if i<len(v) else v[n-i]
    ret+=func.subs(x,S(i)/n)*bino*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
   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)
```

8.5 Derivation of My Algorithm for min(λ, 1/2)

The following explains how the algorithm is derived.

The function min(λ, 1/2) can be rewritten as $A + B$ where—

- $A = (1/2) * \lambda$, and
- $B = (1/2) * \min(\lambda, 1-\lambda)$
  $= (1/2) * ((1-\text{sqrt}(1-4*\lambda*(1-\lambda)))/2)$
  $= (1/2) * \sum_{k = 1, 2, ...} g(k) * h_k(\lambda)$,

revealing that the function is a **convex combination**, and B is itself a convex combination where—

- $g(k) = \text{choose}(2*k,k)/((2*k-1)*2^{2*k})$, and
- $h_k(\lambda) = (4*\lambda*(1-\lambda))^k / 2 = (\lambda*(1-\lambda))^k * 4^k / 2$

(see also Wästlund (1999)[29]; Dale et al. (2015)[30]). The right-hand side of h, which is the polynomial built in step 3 of the algorithm, is a polynomial of degree $k*2$ with Bernstein coefficients—

- $z = (4^v/2) / \text{choose}(v*2,v)$ at $v=k$, and
- 0 elsewhere.

Unfortunately, z is generally greater than 1, so that the polynomial can't be simulated, as is, using the Bernoulli factory algorithm for **polynomials in Bernstein form**. Fortunately, the polynomial's degree can be elevated to bring the Bernstein coefficients to 1 or less (for degree elevation and other algorithms, see (Tsai and Farouki 2001)**(31)**). Moreover, due to the special form of the Bernstein coefficients in this case, the degree elevation process can be greatly simplified. Given an even degree d as well as z (as defined above), the degree elevation is as follows:

1. Set r to floor(d/3) + 1. (This starting value is because when this routine finishes, r/d appears to converge to 1/3 as d gets large, for the polynomial in question.) Let c be choose(d,d/2).
2. Create a list of $d+r+1$ Bernstein coefficients, all zeros.
3. For each integer i in the interval [0, $d+r$]:
    - If d/2 is in the interval [max(0, $i-r$), min(d,i)], set the i^{th} Bernstein coefficient (starting at 0) to z*c*choose(r,$i-d$/2)* / choose($d+r$, i).
4. If all the Bernstein coefficients are 1 or less, return them. Otherwise, add d/2 to r and go to step 2.

8.6 More Algorithms for Non-Negative Factories

Algorithm 2. Say we have an *oracle* that produces independent random real numbers that average to a known or unknown mean. The goal is now to produce non-negative random numbers that average to the mean of $f(X)$, where X is a number produced by the oracle. This is possible whenever f has a finite minimum and maximum and the mean of $f(X)$ is not less than δ, where δ is a known rational number greater than 0. The algorithm to do so follows (see Lee et al. 2014)**(32)**:

1. Let m be a rational number equal to or greater than the maximum value of abs($f(\mu)$) anywhere. Create a ν input coin that does the following: "Take a number from the oracle, call it x. With probability abs($f(x)$)/m, return a number that is 1 if $f(x) < 0$ and 0 otherwise. Otherwise, repeat this process."
2. Use one of the **linear Bernoulli factories** to simulate 2*ν (2 times the ν coin's probability of heads), using the ν input coin, with $\epsilon = \delta/m$. If the factory returns 1, return 0. Otherwise, take a number from the oracle, call it ξ, and return abs($f(\xi)$).

    Example: An example from Lee et al. (2014)**(32)**. Say the oracle produces uniform random numbers in [0, 3*π], and let $f(\nu) = \sin(\nu)$. Then the mean of $f(X)$ is 2/(3*π), which is greater than 0 and found in SymPy by `sympy.stats.E(sin(sympy.stats.Uniform('U',0,3*pi)))`, so the algorithm can produce non-negative random numbers that average to that mean.

    Notes:

    1. Averaging to the mean of $f(X)$ (that is, $\mathbf{E}[f(X)]$ where $\mathbf{E}[.]$ means expected or average value) is not the same as averaging to $f(\mu)$ where μ is the mean of the oracle's numbers (that is, $f(\mathbf{E}[X])$). For example, if X is 0 or 1 with equal probability, and $f(\nu) = \exp(-\nu)$, then $\mathbf{E}[f(X)] = \exp(0) + (\exp(-1) - \exp(0))*(1/2)$, and $f(\mathbf{E}[X]) = f(1/2) = \exp(-1/2)$.

    2. (Lee et al. 2014, Corollary 4)**(32)**: If $f(\mu)$ is known to return only values in the interval [a, c], the mean of $f(X)$ is not less than δ, $\delta > b$, and δ and b are known numbers, then Algorithm 2 can be modified as follows:

        - Use $f(\nu) = f(\nu) - b$, and use $\delta = \delta - b$.
        - m is taken as max($b-a$, $c-b$).

    - When Algorithm 2 finishes, add b to its return value.

3. The check "With probability abs($f(x)$)/m" is exact if the oracle produces only rational numbers *and* if $f(x)$ outputs only rational numbers. If the oracle or f can produce irrational numbers (such as numbers that follow a beta distribution or another continuous distribution), then this check should be implemented using uniform **partially-sampled random numbers (PSRNs)**.

Algorithm 3. Say we have an *oracle* that produces independent random real numbers that are all greater than or equal to a (which is a known rational number), whose mean (μ) is unknown, and whose variance should be finite. The goal is to use the oracle to produce non-negative random numbers with mean $f(\mu)$. This is possible only if f is 0 or greater everywhere in the interval _a_, ∞) and is nondecreasing in that interval (Jacob and Thiery 2015)[(7)]. This can be done using the algorithm below. In the algorithm:

- $f(\mu)$ must be a function that can be written as the following infinite series expansion: $c[0]*z^0 + c[1]*z^1 + ...$, where $z = \mu - a$ and all $c[i]$ are 0 or greater.
- ψ is a rational number close to 1, such as 95/100. (The exact choice is arbitrary and can be less or greater for efficiency purposes, but must be greater than 0 and less than 1.)

The algorithm follows.

1. Set *ret* to 0, *prod* to 1, k to 0, and w to 1. (w is the probability of generating k or more random numbers in a single run of the algorithm.)
2. If k is greater than 0: Generate a number from the oracle, call it x, and multiply *prod* by $x - a$.
3. Add $c[k]*prod/w$ to *ret*.
4. Multiply w by ψ and add 1 to k.
5. With probability ψ, go to step 2. Otherwise, return *ret*.

Now, assume the oracle's numbers are all less than or equal to b (rather than greater than or equal to a), where b is a known rational number. Then f must be 0 or greater everywhere in $(-\infty, b]$ and be nonincreasing there (Jacob and Thiery 2015)[(7)], and the algorithm above can be used with the following modifications: (1) In the note on the infinite series, $z = b - \mu$; (2) in step 2, multiply *prod* by $b - x$ rather than $x - a$.

> **Note:** This algorithm is exact if the oracle produces only rational numbers *and* if all $c[i]$ are rational numbers. If the oracle can produce irrational numbers, then they should be implemented using uniform PSRNs. See also note 3 on Algorithm 2.

8.7 Pushdown automata

Proposition 0: *Let A be the class of algebraic functions that map the open interval (0, 1) to (0, 1) and can be simulated by a pushdown automaton that terminates with probability 1. Then:*

- *All rational functions with rational coefficients that map (0, 1) to (0, 1) are in A.*
- *The square-root function sqrt(λ) is in A.*
- *If functions f and g are in A, then so are their product and composition.*

Proposition 1: *If f(λ) and g(λ) are functions in the class A, then so is their product, namely f(λ)*g(λ).*

Proof: Let F be the pushdown automaton for f, let G be that for g, and assume that both machines' stacks start with the symbol EMPTY. First, rename each state of G as necessary so that the sets of states of F and of G are disjoint. Then, for each rule in F of the form—

(*state*, *flip*, EMPTY) → (*state2*, {}),

where *state2* is a final state of F associated with output 1, replace that rule with—

(*state*, *flip*, EMPTY) → (*gstart*, {EMPTY}),

where *gstart* is the starting state for G. Then take the final states of the combined machine as the union of the final states of F and G. □

Proposition 2: *If f(λ) and g(λ) are functions in the class A, then so is their composition, namely f(g(λ)) or f∘g(λ).*

Proof: Let F be the pushdown automaton for f, let G be that for g, and assume that both machines' stacks start with the symbol EMPTY. First, rename each state of G as necessary so that the sets of states of F and of G are disjoint. Then, add to F a new stack symbol EMPTY′ (or a name not found in the stack symbols of G, as the case may be). Then, for each pair of rules in F of the form—

(*state*, HEADS, *stacksymbol*) → (*state2heads*, *stackheads*), and
(*state*, TAILS, *stacksymbol*) → (*state2tails*, *stacktails*),

where *state* is an arbitrary state and the transitions of the two rules differ, add two new states $state_0$ and $state_1$ that correspond to *state* and have names different from all other states, and replace that rule with the following rules:

(*state*, HEADS, *stacksymbol*) → (*gstart*, {*stacksymbol*, EMPTY′}),
(*state*, TAILS, *stacksymbol*) → (*gstart*, {*stacksymbol*, EMPTY′}),
($state_0$, HEADS, *stacksymbol*) → (*state2heads*, *stackheads*),
($state_0$, TAILS, *stacksymbol*) → (*state2heads*, *stackheads*),
($state_1$, HEADS, *stacksymbol*) → (*state2tails*, *stacktails*), and
($state_1$, TAILS, *stacksymbol*) → (*state2tails*, *stacktails*),

where *gstart* is the starting state for G, and copy the rules of the automaton for G onto F, but with the following modifications:

- Replace the symbol EMPTY in G with EMPTY′.
- Replace each rule in G of the form (*state*, *flip*, EMPTY′) → (*state2*, {}), where *state2* is a final state of G associated with output 1, with the rule (*state*, *flip*, EMPTY′) → ($state_1$, {}).
- Replace each rule in G of the form (*state*, *flip*, EMPTY′) → (*state2*, {}), where *state2* is a final state of G associated with output 0, with the rule (*state*, *flip*, EMPTY′) → ($state_0$, {}).

Then, the final states of the new machine are the same as those for the original machine F. □

Proposition 2: *Every rational function with rational coefficients that maps (0, 1) to (0, 1) is in class A.*

Proof: These functions can be simulated by a finite-state machine (Mossel and Peres 2005)**(9)**. This corresponds to a pushdown automaton with no stack symbols other than

EMPTY and that never pushes symbols onto the stack, and such that, whenever the machine transitions to a final state of the finite-state machine, it pops the only symbol EMPTY from the stack. □

Lemma 1: *The square root function sqrt(λ) is in class A.*

Proof: See (Mossel and Peres 2005)**(9)**. □

Corollary 1: *The function* $f(\lambda) = \lambda^{m/(2^n)}$*, where* $n \geq 1$ *is an integer and where* $m \geq 1$ *is an integer, is in class A.*

Proof: Start with the case m=1. If n is 1, write f as sqrt(λ); if n is 2, write f as sqrt∘sqrt(λ); and for general n, write f as sqrt∘sqrt∘...∘sqrt(λ), with n instances of sqrt. Because this is a composition and sqrt can be simulated by a pushdown automaton, so can f.

For general m and n, write f as $(\text{sqrt}\circ\text{sqrt}\circ...\circ\text{sqrt}(\lambda))^m$, with n instances of sqrt. This involves doing m multiplications of sqrt∘sqrt∘...∘sqrt, and because this is an integer power of a function that can be simulated by a pushdown automaton, so can f. □

9 License