# More Algorithms for Arbitrary-Precision Sampling

This version of the document is dated 2021-11-21.

**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 produce heads with an irrational probability. 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 produce heads with an irrational probability. 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

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

## 3.1 cosh($\lambda$) − 1, and Certain Other Convex Functions

The following algorithm, which takes advantage of the **convex combination method**, samples the probability cosh($\lambda$) − 1, and can serve as a framework for sampling probabilities equal to certain other functions.

1. (The first two steps generate a number $n$ that equals $i$ with probability $g(i)$, as given later.) Generate unbiased random bits (each bit is 0 or 1 with equal probability) until

a zero is generated this way. Set $n$ to the number of ones generated this way.
2. Set $n$ to 2*$n$ + 2.
3. (The next two steps succeed with probability $w_n(\lambda)/g(n)$.) Let $P$ be $2^{n/2}/(n!)$. With probability $P$, go to the next step. Otherwise, return 0.
4. (At this point, $n$ equals $i$ with probability $w_i(1)$.) 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 (or if $n$ is 0). Otherwise, return 0.

Derivation: Follows from rewriting cosh($\lambda$)−1 as the following series: $$\sum_{n\ge 0} w_n(\lambda) = \sum_{n\ge 0} g(n) \frac{w_n(\lambda)}{g(n)},$$ where:

- $g(n)$ is (1/2)*(1/2)$^{(n-2)/2}$ if $n$>0 and $n$ is even, or 1 otherwise. This serves to send nonzero probabilities to terms in the series with nonzero coefficients. For example, in the case of cosh($\lambda$) − 1, the nonzero terms are at 2, 4, 6, 8, and so on, so these terms are assigned the probabilities 1/2, 1/4, 1/8, 1/16, and so on, respectively.
- $w_n(\lambda)$ is $\lambda^n/(n!)$ if $n$>0 and $n$ is even, or 0 otherwise. This is a term of the Taylor series expansion at 0.

Additional functions can be simulated using this algorithm, by modifying it as in the following table. For convenience, the table also includes cosh($\lambda$)−1. (Note 1 at the end of this section describes what these functions have in common.)

| **Target function *f*($\lambda$)** | **Step 2 reads "Set *n* to ..."** | **Value of *P*** |
|---|---|---|
| exp($\lambda$/4)/2. | $n$. | 1/(2$^n$*($n$!)) |
| exp($\lambda$)/4. | $n$. | $2^{n-1}/(n!)$ |
| exp($\lambda$)/6. | $n$. | $2^n/(3*(n!))$ |
| exp($\lambda$/2)/2. | $n$. | $1/(n!)$ |
| (exp($\lambda$)−1)/2. | $n$ + 1. | $2^{n-1}/(n!)$ |
| sinh($\lambda$)/2 | 2*$n$ + 1. | $2^{(n-1)/2}/(n!)$ |
| cosh($\lambda$)/2 | 2*$n$. | $2^{n/2}/(n!)$ |
| cosh($\lambda$)−1 | 2*$n$ + 2. | $2^{n/2}/(n!)$ |

The table below shows functions shifted downward and shows the algorithm changes needed to simulate the modified function. In the table, $D$ is a rational number in the interval [0, $\varphi$(0)], where $\varphi$(.) is the original function.

| **Original function ($\varphi$($\lambda$))** | **Target function *f*($\lambda$)** | **Step 2 reads "Set *n* to ..."** | **Value of *P*** |
|---|---|---|---|
| exp($\lambda$)/4 | $\varphi(\lambda) - D$ | $n$. | (1/4−$D$)*2 or ($\varphi$(0)−$D$)*2 if $n$ = 0; $2^{n-1}/(n!)$ otherwise. |
| exp($\lambda$)/6 | $\varphi(\lambda) - D$ | $n$. | (1/6−$D$)*2 if $n$ = 0; $2^n/(3*(n!))$ otherwise. |
| exp($\lambda$/2)/2 | $\varphi(\lambda) - D$ | $n$. | (1/2−$D$)*2 if $n$ = 0; $1/(n!)$ otherwise. |
| cosh($\lambda$)/4 | $\varphi(\lambda) - D$. | 2*$n$. | (1/4−$D$)*2 if $n$ = 0; $2^{n/2}/(2*(n!))$ otherwise. |

The functions have similar derivations as follows:

- $g(n)$ is (1/2)*(1/2)$^{h(n)}$, where $h(n)$ is the inverse of the "Step 2" columns above. If a certain value for $n$, call it $i$, can't occur in the algorithm after step 2 is complete, then $g(i)$ is 1 instead. (For example, if the column reads "2*$n$ + 1", then $h(n)$ is $(n-1)/2$.)
- $w_n(\lambda)$ is the appropriate term for n in the target function's Taylor series expansion at 0. If a certain value for $n$, call it $i$, can't occur in the algorithm after step 2 is complete, then $w_i(\lambda)$ is 0 instead.

  **Notes:**

  1. All target functions in this section are infinite series that map the interval [0, 1] to [0, 1] and can be written as— $$f(\lambda)=a_0 \lambda^0 + ... + a_i \lambda^i + ..., $$ where the *coefficients* $a_i$ are 0 or greater. (This way of writing the function is called a *Taylor series expansion at 0*.)
     This kind of function—

     - is non-negative for every $\lambda$,
     - is either constant or monotone increasing, and
     - is *convex* (its "slope" or "velocity" doesn't decrease as $\lambda$ increases).

     To show the function is convex, find the "slope-of-slope" function of $f$ and show it's non-negative for every $\lambda$ in the domain. To do so, first find the "slope": omit the first term and for each remaining term (with $i\ge 1$), replace $a_i \lambda^i$ with $a_i i \lambda^{i-1}$. The resulting "slope" function is still an infinite series with coefficients 0 or greater. Hence, so will the "slope" of this "slope" function, so the result follows by induction.

  2. The target function $f(\lambda)$ is also a *probability generating function*, written as given in Note 1 where each $a_i$ is the probability that the $n$ value used in step 4 equals $i$ (cf. Dughmi et al. (2017)[^1]).

## 3.2 Certain Power Series

The following way to design Bernoulli factories covers a broad class of power series (see also the main Bernoulli Factory Algorithms article).[^2]

Let $f(\lambda)$ be a factory function that can be written as the following series expansion: $$f(\lambda) = \sum_{i\ge 0} a_i (g(\lambda))^i,$$ where $g(\lambda)$ is a factory function and each $a_i$, a *coefficient*, should be a rational number.

Suppose the following:

- There is a rational number $Z$ defined as follows. For every $\lambda$ in $[0, 1]$, $f(\lambda) \le Z \lt 1$.
- There is an even integer $m$ defined as follows. The series above can be split into two parts: the first part ($A$) is the sum of the first $m$ terms, and the second part ($C$) is the sum of the remaining terms. Moreover, both parts admit a Bernoulli factory algorithm. Specifically: $$C(\lambda) = \sum_{i\ge m} a_i (g(\lambda))^i, A(\lambda) = f(\lambda) - C(\lambda).$$ One way to satisfy the condition on $C$ is if $C$ is an alternating series (starting at $m$, even-indexed $a$'s are positive and odd-indexed are negative) and if $0 \le |a_{i+1}| \le |a_i| \le 1$ for every $i\ge m$ (that is, the coefficients starting with coefficient $m$ have absolute values that are 1 or less and form a nonincreasing sequence); such functions $C$ admit an algorithm given in Łatuszyński et al. (2019/2011)[^3]. ($C$ and $A$ admit a Bernoulli factory only if they map the interval [0, 1] to [0, 1], among other requirements.)

Then rewrite the function as— $$f(\lambda) = A(\lambda) + (g(\lambda))^{m} B(\lambda),$$ where—

- $A(\lambda) = f(\lambda) - C(\lambda) = \sum_{i=0}^{m-1} a_i (g(\lambda))^i$ is a polynomial in $g(\lambda)$ of degree $m-1$, and
- $B(\lambda) = C(\lambda) / (g(\lambda))^{m} = \sum_{i\ge m} a_{m+i} (g(\lambda))^i$.

Rewrite $A$ as a polynomial in Bernstein form, in the variable $g(\lambda)$. (One way to transform a polynomial to Bernstein form, given the "power" coefficients $a_0, ..., a_{m-1}$, is the so-called "matrix method" from Ray and Nataraj (2012)[^4].) Let $b_0, ..., b_{m-1}$ be the Bernstein-form polynomial's coefficients. Then if those coefficients all lie in $[0, 1]$, then the following algorithm simulates $f(\lambda)$.

**Algorithm 1:** Run a **linear Bernoulli factory**, with parameters $x=2$, $y=1$, and $\epsilon=1-Z$. Whenever the linear Bernoulli factory "flips the input coin", it runs the sub-algorithm below.

- **Sub-algorithm:** Generate an unbiased random bit. If that bit is 1, sample the polynomial $A$ as follows (Goyal and Sigman 2012)[^9]:

    1. Run a Bernoulli factory algorithm for $g(\lambda)$, $m-1$ times. Let $j$ be the number of runs that return 1.
    2. With probability $b_j$, return 1. Otherwise, return 0.

    If the bit is 0, do the following:

    1. Run a Bernoulli factory algorithm for $g(\lambda)$, $m$ times. Return 0 if any of the runs returns 0.
    2. Run a Bernoulli factory algorithm for $B(\lambda)$, and return the result.

---

The following examples show how this method leads to algorithms for simulating certain factory functions.

**Example 1:** Take $f(\lambda) = \sin(3\lambda)/2$, which is a power series.

- $f$ is bounded above by $Z=1/2 \lt 1$.
- $f$ satisfies $m=8$ since splitting the series at 8 leads to two functions that admit Bernoulli factories.
- Thus, $f$ can be written as— $$f(\lambda) = A(\lambda) + \lambda^8 \left(\sum_{i\ge 0} a_{8+i} \lambda^i\right),$$ where $a_i = \frac{3^i}{i! \times 2}(-1)^{(i-1)/2}$ if $i$ is odd and 0 otherwise.
- $A$ is rewritten from "power" form (with coefficients $a_0, ..., a_{m-1}$) to Bernstein form, with the following coefficients, in order: [0, 3/14, 3/7, 81/140, 3/5, 267/560, 81/280, 51/1120].
- Now, **Algorithm 1** can be used to simulate $f$ given a coin that shows heads (returns 1) with probability $\lambda$, where:
    - $g(\lambda) = \lambda$, so the Bernoulli factory algorithm for $g(\lambda)$ is simply to flip the coin for $\lambda$.
    - The coefficients $b_0, ..., b_{m-1}$, in order, are the Bernstein-form coefficients found for $A$.
    - The Bernoulli factory algorithm for $B(\lambda)$ is as follows: Let $h_i = a_i$. Then run the **general martingale algorithm** (in "Bernoulli Factory Algorithms") with $g(\lambda) = \lambda$ and $a_i = h_{m+i}$.

**Example 2:** Take $f(\lambda) = 1/2 + \sin(6\lambda)/4$, another power series.

- $f$ is bounded above by $Z=3/4 \lt 1$.
- $f$ satisfies $m=16$ since splitting the series at 16 leads to two functions that admit Bernoulli factories.
- Thus, $f$ can be written as— $$f(\lambda) = A(\lambda) + \lambda^{m} \left(\sum_{i\ge 0} a_{m+i} \lambda^i\right),$$ where $m=16$, and where $a_i$ is $1/2$ if $i = 0$; $\frac{6^i}{i! \times 4}(-1)^{(i-1)/2}$ if $i$ is odd; and 0 otherwise.
- $A$ is rewritten from "power" form (with coefficients $a_0, ..., a_{m-1}$) to Bernstein form, with the following coefficients, in order: [1/2, 3/5, 7/10, 71/91, 747/910, 4042/5005, 1475/2002, 15486/25025, 167/350, 11978/35035, 16869/70070, 167392/875875, 345223/1751750, 43767/175175, 83939/250250, 367343/875875].
- Now, **Algorithm 1** can be used to simulate $f$ in the same manner as for Example 1.

**Example 3:** Take $f(\lambda) = 1/2 + \sin(\pi\lambda)/4$. To simulate this probability:

1. Create a $\mu$ coin that does the following: "With probability 1/3, return 0. Otherwise, run the algorithm for ***π*/4** (in 'Bernoulli Factory Algorithms') and return the result." (Simulates $\pi/6$.)
2. Run the algorithm for $1/2 + \sin(6\lambda)/4$ in Example 2, using the $\mu$ coin.

**Example 4:** Take $f(\lambda) = 1/2 + \cos(6\lambda)/4$. This is as in Example 2, except —

- $Z=3/4$ and $m=16$;
- $a_i$ is $3/4$ if $i = 0$; $\frac{6^i}{i! \times 4}(-1)^{i/2}$ if $i$ is even and greater than 0; and 0 otherwise; and
- the Bernstein-form coefficients for $A$, in order, are [3/4, 3/4, 255/364, 219/364, 267/572, 1293/4004, 4107/20020, 417/2860, 22683/140140, 6927/28028, 263409/700700, 2523/4900, 442797/700700, 38481/53900, 497463/700700].

**Example 5:** Take $f(\lambda) = 1/2 + \cos(\pi\lambda)/4$. This is as in Example 3, except step 2 runs the algorithm for $1/2 + \cos(6\lambda)/4$ in Example 4.

**Example:** The following functions can be written as power series that satisfy the **general martingale algorithm** (in "Bernoulli Factory Algorithms"). In the table, $B(i)$ is the $i$th *Bernoulli number* (see the note after the table).

| **Function $f(\lambda)$** | **Coefficients** | **Value of $d_0$** |
|---|---|---|
| $\lambda/(\exp(\lambda)-1)$ | $a_i = B(i)/(i!)$ | 1. |
| $\tanh(\lambda)$ | $a_i = \frac{B(i+1) 2^{i+1} (2^{i+1}-1)}{(i+1)!}$ if $i$ is odd, or 0 otherwise. | 1. |
| $\cos(\sqrt \lambda)$ | $a_i = \frac{(-1)^i}{(2i)!}$. | 1. |

To simulate a function in the table, run the **general martingale algorithm** with $g(\lambda) = \lambda$ and with the given coefficients and value of $d_0$ ($d_0$ is the first nonzero coefficient).

**Note:** Bernoulli numbers can be computed with the following algorithm, namely **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 $i^{\text{th}}$ Bernoulli number**, call it $b$. Add $b$*choose($m+1$, $i$) to $v$. [^5]
    2. Add 2 to $i$.
4. Return $-v/(m+1)$.

**Note:** The general martingale algorithm allows the sequence $(a_i)$ to sum to 1, but this appears to be possible only if the sequence's nonzero values have the form $(1, -z_0, z_0, -z_1, z_1, ..., -z_i, z_i, ...)$, where the $z_i$ are positive, are no greater than 1, and form a non-increasing sequence that is finite or converges to 0. Moreover, it appears that every power series with this sequence of coefficients is bounded above by $\lambda$.

## 3.3 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)[^6]. 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 $k^{\text{th}}$ 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 $\varepsilon$, `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 $\varepsilon$, 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)[^7], the following polynomial sequences were suggested to simulate $\min(2\lambda, 1-2\varepsilon)$, provided $\varepsilon \lt 1/8$, 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($n$, $k$)** = $\min(2(k/n), 1-2\varepsilon)$.
- **fabove($n$, $k$)** = $\min(2(k/n), 1-2\varepsilon)+$ $ \frac{2\times\max(0, k/n+3\varepsilon - 1/2)}{\varepsilon(2-\sqrt{2})} \sqrt{2/n}+$ $\frac{72\times\max(0,k/n-1/9)}{1-\exp(-2\times\varepsilon^2)} \exp(-2n\times\varepsilon^2)$.

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)/choose($m$*2,$m$). Define a polynomial of degree $m$*2 whose ($m$*2)+1 Bernstein coefficients are all zero except the $m^{\text{th}}$ 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)[^8].) Flip the input coin $d$ times and set $h$ to the number of ones generated this way. Let $a$ be the $h^{\text{th}}$ 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 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 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) * $2^{m*2}$*choose($r$,$h-m$)/choose($d$, $h$) (the polynomial's $h^{\text{th}}$ 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, $\mu$ is the unknown probability of heads of a second input coin.

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

| | |
|---|---|
| 1/2 at 1/2; and $\mu$/2 at 1. | Otherwise, flip the $\mu$ input coin. If it returns 1, flip the $\lambda$ input coin and return the result. Otherwise, flip the $\lambda$ input coin and return 1 minus the result. |
| 0 at 0; $\mu$/2 at 1/2; and $\mu$/2 at 1. | Flip the $\mu$ 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 $\lambda$ input coin and return the result. Otherwise, run the algorithm for $\min(\lambda, 1-\lambda)$ using the $\lambda$ input coin, and return the result of that run. |
| $\mu$ at 0; 1/2 at 1/2; and 0 at 1. | Flip the $\mu$ input coin. If it returns 1, flip the $\lambda$ input coin and return 1 minus the result. Otherwise, run the algorithm for $\min(\lambda, 1-\lambda)$ using the $\lambda$ input coin, and return the result of that run. |
| 1 at 0; 1/2 at 1/2; and $\mu$ at 1. | Flip the $\mu$ input coin. If it returns 0, flip the $\lambda$ input coin and return 1 minus the result. Otherwise, run the algorithm for $\min(\lambda, 1-\lambda)$ using the $\lambda$ input coin, and return 1 minus the result of that run. |
| $\mu$ at 0; 1/2 at 1/2; and 1 at 1. | Flip the $\mu$ input coin. If it returns 0, flip the $\lambda$ input coin and return the result. Otherwise, run the algorithm for $\min(\lambda, 1-\lambda)$ using the $\lambda$ input coin, and return 1 minus the result of that run. |
| $B$ at 0; $B$+($A$/2) at 1/2; and $B$+($A$/2) at 1. | ($A \le 1$ and $B \le 1-A$ are rational numbers.) With probability $1-A$, return a number that is 1 with probability $B/(1-A)$ and 0 otherwise. Otherwise, generate an unbiased random bit. If that bit is 1, flip the $\lambda$ input coin and return the result. Otherwise, run the algorithm for $\min(\lambda, 1-\lambda)$ using the $\lambda$ input coin, and return the result of that run. |

## 3.4 Sampling Distributions Using Incomplete Information

The Bernoulli factory is a special case of the problem of **sampling a probability distribution with unknown parameters**. This problem can be described as sampling from a new distribution using an *oracle* (black box) that produces numbers of an incompletely known distribution. In the Bernoulli factory problem, this oracle is a *coin that shows heads or tails where the probability of heads is unknown*. The rest of this section deals with oracles that go beyond coins.

**Algorithm 1.** Say we have an oracle that produces independent random variates in the interval [$a$, $b$], and these numbers have an unknown mean of $\mu$. The goal is now to produce non-negative random variates whose expected ("average") value is $f(\mu)$. 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 $\varepsilon * \min((\mu - a)^n, (b - \mu)^n)$ for some integer $n$ and some $\varepsilon$ 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)[^9]. (Here, $a$ and $b$ are both rational numbers and may be less than 0.)

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

1. Create a $\lambda$ 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 $\lambda$ input coin. Then return $K$ 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 variates in the interval [3, 13] with unknown mean $\mu$, and we seek to use the oracle to produce non-negative random variates 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 variates from the oracle (which must produce random variates 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 variates whose expected ("average") value is 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.

**Algorithm 3.** For this algorithm, see the appendix.

**Algorithm 4.** Say there is an oracle in the form of an $n$-sided fair die ($n \geq 2$) with an unknown number of faces, where each face shows a different integer in the interval [0, $n$). The question arises: Which probability distributions based on $n$ can be sampled with this oracle? This question was studied in the French-language dissertation of R. Duvignau (2015, section 5.2)[^10], and the following are four of these distributions.

***Bernoulli 1/n.*** It's trivial to generate a Bernoulli variate that is 1 with probability $1/n$ and 0 otherwise: just take a number from the oracle and return either 1 if that number is 0, or 0 otherwise. Alternatively, take two numbers from the oracle and return either 1 if both are the same, or 0 otherwise (Duvignau 2015, p. 153)[^10].

***Random variate with mean n.*** Likewise, it's trivial to generate variates with a mean of $n$: Do "Bernoulli 1/n" trials as described above until a trial returns 0, then return the number of trials done this way. (This is often called 1 plus a "geometric" random variate, and has a mean of $n$.)

***Binomial with parameters n and 1/n.*** Using the oracle, the following algorithm generates a binomial variate of this kind (Duvignau 2015, Algorithm 20)[^10]:

1. Take items from the oracle until the same item is taken twice.
2. Create a list consisting of the items taken in step 1, except for the last item taken, then shuffle that list.
3. In the shuffled list, count the number of items that didn't change position after being

shuffled, then return that number.

***Binomial with parameters n and k/n.*** Duvignau 2015 also includes an algorithm (Algorithm 25) to generate a binomial variate of this kind using the oracle (where $k$ is a known integer such that $0 < k$ and $k \leq n$):

1. Take items from the oracle until $k$ different items were taken this way. Let $U$ be a list of these $k$ items, in the order in which they were first taken.
2. Create an empty list $L$.
3. For each integer $i$ in [0, $k$):
    1. Create an empty list $M$.
    2. Take an item from the oracle. If the item is in $U$ at a position **less than *i*** (positions start at 0), repeat this substep. Otherwise, if the item is not in $M$, add it to $M$ and repeat this substep. Otherwise, go to the next substep.
    3. Shuffle the list $M$, then add to $L$ each item that didn't change position after being shuffled (if not already present in $L$).
4. For each integer $i$ in [0, $k$):
    1. Let $P$ be the item at position $i$ in $U$.
    2. Take an item from the oracle. If the item is in $U$ at position ***i* or less** (positions start at 0), repeat this substep.
    3. If the last item taken in the previous substep is in $U$ at a position **greater than *i***, add $P$ to $L$ (if not already present).
5. Return the number of items in $L$.

    **Note:** Duvignau proved a result (Theorem 5.2) that answers the question: Which probability distributions based on the unknown $n$ can be sampled with the oracle?[^11] The result applies to a family of (discrete) distributions with the same unknown parameter $n$, starting with either 1 or a greater integer. Let Supp($m$) be the set of values taken on by the distribution with parameter equal to $m$. Then that family can be sampled using the oracle if and only if:

    - There is a computable function $f(k)$ that outputs a positive number.
    - For each $n$, Supp($n$) is included in Supp($n$+1).
    - For every $k$ and for every $n$ starting with the greater of 2 or the first $n$ for which $k$ is in Supp($n$), the probability of seeing $k$ given parameter $n$ is at least $(1/n)^{f(k)}$ (roughly speaking, the probability doesn't decay at a faster than polynomial rate as $n$ increases).

## 3.5 Pushdown Automata for Square-Root-Like Functions

A *pushdown automaton* is a state machine that keeps a stack of symbols. In this document, the input for this automaton is a stream of flips of a coin that shows heads with probability $\lambda$, and the output is 0 or 1 depending on which state the automaton ends up in when it empties the stack (Mossel and Peres 2005)[^12]. That paper shows that a pushdown automaton, as defined here, can simulate only *algebraic functions*, that is, functions that can be a solution of a system of polynomial equations. The **appendix** defines these machines in more detail and has proofs on which algebraic functions are possible with pushdown automata.

The following algorithm extends the square-root construction of Flajolet et al. (2010) [^13], takes an input coin with probability of heads $\lambda$, 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 − $g(\lambda)$))$^n$*choose(2*$n$, $n$), or equivalently,
- $f(\lambda)$ = (1 − $\lambda$) * 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. If $g$ is a rational function (a ratio of two polynomials) with rational coefficients, then $f$ is algebraic and can be simulated by a *pushdown automaton*, as in the algorithm below. But this algorithm will still work even if $g$ is not a rational function.
- OGF($x$) = $\sum_{n=0,1,...} x^n$*choose(2*$n$, $n$) is the algorithm's ordinary generating function (also known as counting generating 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 (except the "Do this substep again" part) can be expressed as follows. 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 *transition rules* 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 transition rules to the automaton to simulate $g(\lambda)$ by a finite-state machine (only possible if $g$ is rational with rational coefficients (Mossel and Peres 2005)[^12]). 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— $$f(\lambda)=(1-\lambda) \sum_{n\ge 0} \lambda^{Hn} g(\lambda)^n (1-g\lambda)^{(H-1)n} {Hn \choose n},$$ where ${n \choose m}$ = choose($n$, $m$) is a binomial coefficient; $H \ge 2$ 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: This substep is not done again.)

The following algorithm simulates the probability— $$ f(\lambda) = (1-\lambda) \sum_{n\ge 0} \lambda^n \left( \sum_{m\ge 0} W(n,m) g(\lambda)^m (1-g(\lambda))^{n-m} {n \choose m}\right)$$ $$= (1-\lambda) \sum_{n\ge 0} \lambda^n \left( \sum_{m\ge 0} V(n,m) g(\lambda)^m (1-g(\lambda))^{n-m}\right),$$ where $g$ has the same meaning as earlier; $W(n, m)$ is 1 if $m*H$ equals $(n-m)*T$, or 0 otherwise; and $H \ge 1$ and $T \ge 1$ 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 $\lambda^n$, 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: This substep is not done again.)

# 4 Irrational Probabilities

## 4.1 Certain Numbers Based on the Golden Ratio

The following algorithm given by Fishman and Miller (2013)[^14] 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 $\ell$ 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 $\ell$**:
    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 $\ell$** (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 $\ell$**, call them $p$ and $n$, respectively.
    2. If $\ell$ is 1 and $k$ is odd, return $p + n$.
    3. If $\ell$ is $-1$ and $pos$ is 0, return $n - p - 1$.
    4. If $\ell$ is 1 and $pos$ is 0, return $(n + p) - 1$.
    5. If $\ell$ 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 $\ell$ 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 $\ell$**, 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.

## 4.2 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 random variate in [0, 1]; 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 variate $v$.
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 probability that a uniform random variate in [0, 1] is $x$ or less, 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 random variates in [0, 1], namely $m*x^{m-1}$ if $x$ is in [0, 1], and 0 otherwise.

## 4.3 1/(exp(1) + *c* − 2)

Involves the continued fraction expansion and Bernoulli Factory algorithm 3 for continued fractions. In this algorithm, $c \geq 1$ is a rational number.

The algorithm begins with *pos* equal to 1. Then the following steps are taken.

- Do the following process repeatedly until this run of the algorithm returns a value:
    1. If *pos* is divisible by 3 (that is, if rem(*pos*, 3) equals 0): Let $k$ be (*pos*/3)*2. With probability $k/(1+k)$, return a number that is 1 with probability $1/k$ and 0 otherwise.
    2. If *pos* is 1: With probability $c/(1+c)$, return a number that is 1 with probability $1/c$ and 0 otherwise.
    3. If *pos* is greater than 1 and not divisible by 3: Generate an unbiased random bit. If that bit is 1 (which happens with probability 1/2), return 1.
    4. Do a separate run of the currently running algorithm, but with $pos = pos + 1$. If the separate run returns 1, return 0.

## 4.4 exp(1) − 2

Involves the continued fraction expansion and Bernoulli Factory algorithm 3 for continued fractions. Run the algorithm for **1/(exp(1)+*c*−2)** above with $c = 1$, except the algorithm begins with *pos* equal to 2 rather than 1 (because the continued fractions are almost the same).

## 4.5 Euler–Mascheroni Constant $\gamma$

As **I learned**, the fractional part of $1/U$, where $U$ is a uniform random variate in (0, 1), has a mean equal to 1 minus the Euler–Mascheroni constant $\gamma$, about 0.5772.[^15] This leads to the following algorithm to sample a probability equal to $\gamma$:

1. Generate a PSRN for the reciprocal of a uniform random variate, as described in **another page of mine**.
2. Set the PSRN's integer part to 0, then run **SampleGeometricBag** on that PSRN. Return 0 if the run returns 1, or 1 otherwise.

## 4.6 $\pi/4$

The following algorithm to sample the probability $\pi/4$ is based on the section "**Uniform Distribution Inside N-Dimensional Shapes**", especially its Note 5.

1. Set $S$ to 2. Then set *c1* and *c2* to 0.
2. Do the following process repeatedly, until the algorithm returns a value:
    1. Set *c1* to 2**c1* plus an unbiased random bit (either 0 or 1 with equal probability). Then, set *c2* to 2**c2* plus an unbiased random bit.
    2. If $((c1+1)^2 + (c2+1)^2) < S^2$, return 1. (Point is inside the quarter disk, whose area is $\pi/4$.)
    3. If $((c1)^2 + (c2)^2) > S^2$, return 0. (Point is outside the quarter disk.)
    4. Multiply $S$ by 2.

## 4.7 $\pi/4 - 1/2$ or $(\pi - 2)/4$

Follows the $\pi/4$ algorithm, except it samples from a quarter disk with an area equal to 1/2 removed.

1. Set $S$ to 2. Then set *c1* and *c2* to 0.
2. Do the following process repeatedly, until the algorithm returns a value:
    1. Set *c1* to 2**c1* plus an unbiased random bit (either 0 or 1 with equal probability). Then, set *c2* to 2**c2* plus an unbiased random bit.
    2. Set *diamond* to *MAYBE* and *disk* to *MAYBE*.
    3. If $((c1+1) + (c2+1)) < S$, set *diamond* to *YES*.
    4. If $((c1) + (c2)) > S$, set *diamond* to *NO*.
    5. If $((c1+1)^2 + (c2+1)^2) < S^2$, set *disk* to *YES*.
    6. If $((c1)^2 + (c2)^2) > S^2$, set *disk* to *NO*.
    7. If *disk* is *YES* and *diamond* is *NO*, return 1. Otherwise, if *diamond* is *YES* or *disk* is *NO*, return 0.
    8. Multiply $S$ by 2.

## 4.8 $(\pi - 3)/4$

Follows the $\pi/4$ algorithm, except it samples from a quarter disk with enough boxes removed from it to total an area equal to 3/4.

1. Set $S$ to 32. Then set $c1$ to a uniform random integer in the half-open interval 0, $S$) and $c2$ to another uniform random integer in [0, $S$).
2. (Retained boxes.) If $c1$ is 0 and $c2$ is 0, or if $c1$ is 0 and $c2$ is 1, return 1.
3. (Removed boxes.) If $((c1+1)^2 + (c2+1)^2) < 1024$, return 0.
4. Multiply $S$ by 2.
5. (Sample the modified quarter disk.) Do the following process repeatedly, until the algorithm returns a value:
    1. Set $c1$ to $2*c1$ plus an unbiased random bit (either 0 or 1 with equal probability). Then, set $c2$ to $2*c2$ plus an unbiased random bit.
    2. If $((c1+1)^2 + (c2+1)^2) < S^2$, return 1. (Point is inside the quarter disk, whose area is $\pi/4$.)
    3. If $((c1)^2 + (c2)^2) > S^2$, return 0. (Point is outside the quarter disk.)
    4. Multiply $S$ by 2.

## 4.9 $\pi - 3$

Similar to the $\pi/4$ algorithm. First it samples a point inside an area covering 1/4 of the unit square, then inside that area, it determines whether that point is inside another area covering $(\pi - 3)/4$ of the unit square. Thus, the algorithm acts as though it samples $((\pi - 3)/4) / (1/4) = \pi - 3$.

1. Set $S$ to 2. Then set $c1$ and $c2$ to 0.
2. Do the following process repeatedly, until the algorithm aborts it or returns a value:
    1. Set $S$ to 32. Then set $c1$ to a uniform random integer in the half-open interval [0, $S$) and $c2$ to another uniform random integer in [0, $S$).
    2. (Return 1 if in retained boxes.) If $c1$ is 0 and $c2$ is 0, or if $c1$ is 0 and $c2$ is 1, return 1.
    3. (Check if outside removed boxes.) If $((c1+1)^2 + (c2+1)^2) >= 1024$, abort this process and go to step 3. (Otherwise, $c1$ and $c2$ are rejected and this process continues.)
3. Set $S$ to 64.
4. (Sample the modified quarter disk.) Do the following process repeatedly, until the algorithm returns a value:
    1. Set $c1$ to $2*c1$ plus an unbiased random bit (either 0 or 1 with equal probability). Then, set $c2$ to $2*c2$ plus an unbiased random bit.
    2. If $((c1+1)^2 + (c2+1)^2) < S^2$, return 1. (Point is inside the quarter disk, whose area is $\pi/4$.)
    3. If $((c1)^2 + (c2)^2) > S^2$, return 0. (Point is outside the quarter disk.)
    4. Multiply $S$ by 2.

    **Note:** Only a limited set of ($c1$, $c2$) pairs, including (0, 0) and (0, 1), will pass step 2 of this algorithm. Thus it may be more efficient to choose one of them uniformly at random, rather than do step 2 as shown. If (0, 0) or (0, 1) is chosen this way, the algorithm returns 1.

## 4.10 4/(3*$\pi$)

Given that the point ($x$, $y$) has positive coordinates and lies inside a disk of radius 1 centered at (0, 0), the mean value of $x$ is $4/(3*\pi)$. This leads to the following algorithm to sample that probability:

1. Generate two PSRNs in the form of a uniformly chosen point inside a 2-dimensional quarter hypersphere (see "[**Uniform Distribution Inside N-Dimensional Shapes**" below, as well as the examples).
2. Let *x* be one of those PSRNs. Run **SampleGeometricBag** on that PSRN and return the result (which will be either 0 or 1).

   **Note:** The mean value 4/(3*π) can be derived as follows. The relative probability that *x* is "close" to *z* is *p*(*z*) = sqrt(1 − *z***z*), where *z* is in the interval [0, 1]. Now find the area under the graph of *z***p*(*z*)/*c* (where *c*=π/4 is the area under the graph of *p*(*z*)). The result is the mean value 4/(3*π). The following Python code prints this mean value using the SymPy computer algebra library: `p=sqrt(1-z*z); c=integrate(p,(z,0,1)); print(integrate(z*p/c,(z,0,1)));`.

## 4.11 (1 + exp(*k*)) / (1 + exp(*k* + 1))

This algorithm simulates this probability by computing lower and upper bounds of exp(1), which improve as more and more digits are calculated. These bounds are calculated through an algorithm by Citterio and Pavani (2016)[^16]. Note the use of the methodology in Łatuszyński et al. (2009/2011, algorithm 2)[^17] in this algorithm. In this algorithm, *k* must be an integer 0 or greater.

1. If *k* is 0, run the **algorithm for 2 / (1 + exp(2))** and return the result. If *k* is 1, run the **algorithm for (1 + exp(1)) / (1 + exp(2))** and return the result.
2. Generate a uniform(0, 1) random variate, call it *ret*.
3. If *k* is 3 or greater, return 0 if *ret* is greater than 38/100, or 1 if *ret* is less than 36/100. (This is an early return step. If *ret* is implemented as a uniform PSRN, these comparisons should be done via the **URandLessThanReal algorithm**, which is described in my **article on PSRNs**.)
4. Set *d* to 2.
5. Calculate a lower and upper bound of exp(1) (*LB* and *UB*, respectively) in the form of rational numbers whose numerator has at most *d* digits, using the Citterio and Pavani algorithm. For details, see later.
6. Set *rl* to $(1+LB^k) / (1+UB^{k+1})$, and set *ru* to $(1+UB^k) / (1+LB^{k+1})$; both these numbers should be calculated using rational arithmetic.
7. If *ret* is greater than *ru*, return 0. If *ret* is less than *rl*, return 1. (If *ret* is implemented as a uniform PSRN, these comparisons should be done via **URandLessThanReal**.)
8. Add 1 to *d* and go to step 5.

The following implements the parts of Citterio and Pavani's algorithm needed to calculate lower and upper bounds for exp(1) in the form of rational numbers.

Define the following operations:

- **Setup:** Set *p* to the list `[0, 1]`, set *q* to the list `[1, 0]`, set *a* to the list `[0, 0, 2]` (two zeros, followed by the integer part for exp(1)), set *v* to 0, and set *av* to 0.
- **Ensure *n*:** While *v* is less than or equal to *n*:
  1. (Ensure partial denominator *v*, starting from 0, is available.) If *v* + 2 is greater than or equal to the size of *a*, append 1, *av*, and 1, in that order, to the list *a*, then add 2 to *av*.
  2. (Calculate convergent *v*, starting from 0.) Append *a*[*n*+2] * *p*[*n*+1]+*p*[*n*] to the list *p*, and append *a*[*n*+2] * *q*[*n*+1]+*q*[*n*] to the list *q*. (Positions in lists start at 0. For example, *p*[0] means the first item in *p*; *p*[1] means the second; and so on.)
  3. Add 1 to *v*.

- **Get the numerator for convergent *n*:** Ensure $n$, then return $p[n+2]$.
- **Get convergent *n*:** Ensure $n$, then return $p[n+2]/q[n+2]$.
- **Get semiconvergent *n* given *d*:**
  1. Ensure $n$, then set $m$ to floor(((10^$d$)−1−$p[n+1]$)/$p[n+2]$).
  2. Return ($p[n+2] * m + p[n+1]$) / ($q[n+2] * m + q[n+1]$).

Then the algorithm to calculate lower and upper bounds for exp(1), given $d$, is as follows:

1. Set $i$ to 0, then run the **setup**.
2. **Get the numerator for convergent *i***, call it $c$. If $c$ is less than $10^d$, add 1 to $i$ and repeat this step. Otherwise, go to the next step.
3. **Get convergent *i* − 1** and **get semiconvergent *i* − 1 given *d***, call them *conv* and *semi*, respectively.
4. If ($i$ − 1) is odd, return *semi* as the lower bound and *conv* as the upper bound. Otherwise, return *conv* as the lower bound and *semi* as the upper bound.

## 4.12 Other Probabilities

Algorithms in bold are given either in this page or in the "**Bernoulli Factory Algorithms**" page.

| To simulate: | Follow this algorithm: |
|---|---|
| 3 − exp(1) | Run the algorithm for **exp(1) − 2**, then return 1 minus the result. |
| 1/(exp(1)−1) | Run the algorithm for **1/(exp(1)+*c*−2)** with $c$ = 1. |
| 1/(1+exp(1)) | Run the algorithm for **1/(exp(1)+*c*−2)** with $c$ = 3. |
| $n/\pi$ | ($n$ is 1, 2, or 3.)<br>Create $\lambda$ coin for algorithm **π − 3**.<br>Run algorithm for **(*d* + *λ*) / (*c* + *λ*)** with $d$=$n$ and $c$=3. |
| $r/\pi$ | ($r$ is a rational number in open interval (0, 3).)<br>Create $\lambda$ coin for algorithm **π − 3**.<br>Create $\mu$ coin that does: "With probability $r$ − floor($r$), return 1; otherwise return 0."<br>Run algorithm for **(*d* + *μ*) / (*c* + *λ*)** with $d$=floor($r$) and $c$=3. |
| exp(1)/$\pi$ | Create $\mu$ coin for algorithm **exp(1) − 2**.<br>Create $\lambda$ coin for algorithm **π − 3**.<br>Run algorithm for **(*d* + *μ*) / (*c* + *λ*)** with $d$=2 and $c$=3. |
| exp(1)/4 | Follow the algorithm for **exp(λ/4)/2**, except the probability in step 2 is $2^{n-1}/(n!)$, $c$ is 0, and step 3 is replaced with "Return 1." |

# 5 General Arbitrary-Precision Samplers

## 5.1 Uniform Distribution Inside N-Dimensional Shapes

The following is a general way to describe an arbitrary-precision sampler for generating a point uniformly at random inside a geometric shape located entirely in the hypercube [0, $d1$]×[0, $d2$]×...×[0,$dN$] in $N$-dimensional space, where $d1$, ..., $dN$ are integers greater

than 0. The algorithm will generally work if the shape is reasonably defined; the technical requirements are that the shape must have a zero-volume (Lebesgue measure zero) boundary and a nonzero finite volume, and must assign zero probability to 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 or fully outside the shape, so multiply $S$ by *base*, then add 1 to $d$, then go to step 3.

    **Notes:**

    1. See (Li and El Gamal 2016)[^18] and (Oberhoff 2018)[^19] for related work on encoding random points uniformly distributed in a shape.

2. Rejection sampling on a shape is subject to the "curse of dimensionality", since typical shapes of high dimension will tend to cover much less volume than their bounding boxes, so that it would take a lot of time on average to accept a high-dimensional box. Moreover, the more area the shape takes up in the bounding box, the higher the acceptance rate.
3. Devroye (1986, chapter 8, section 3)[^20] describes grid-based methods to optimize random point generation. In this case, the space is divided into a grid of boxes each with size $1/base^k$ in all dimensions; the result of **InShape** is calculated for each such box and that box labeled with the result; all boxes labeled *NO* are discarded; and the algorithm is modified by adding the following after step 2: "2a. Choose a precalculated box uniformly at random, then set *c1*, ..., *cN* to that box's coordinates, then set *d* to *k* and set *S* to $base^k$. If a box labeled *YES* was chosen, follow the substeps in step 5. If a box labeled *MAYBE* was chosen, multiply *S* by *base* and add 1 to *d*." (For example, if *base* is 10, *k* is 1, *N* is 2, and *d1* = *d2* = 1, the space could be divided into a 10×10 grid, made up of 100 boxes each of size (1/10)×(1/10). Then, **InShape** is precalculated for the box with coordinates ((0, 0), (1, 1)), the box ((0, 1), (1, 2)), and so on [the boxes' coordinates are stored as just given, but **InShape** instead uses those coordinates divided by $base^k$, or $10^1$ in this case], each such box is labeled with the result, and boxes labeled *NO* are discarded. Finally the algorithm above is modified as just given.)
4. Besides a grid, another useful data structure is a *mapped regular paving* (Harlow et al. 2012)[^21], 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.
5. The algorithm can be adapted to return 1 with probability equal to its acceptance rate (which equals the shape's volume divided by the hyperrectangle's volume), and return 0 with the opposite probability. In this case, replace steps 5 and 6 with the following: "5. If the result of **InShape** is *YES*, return 1.; 6. If the result of **InShape** is *NO*, return 0." (I thank BruceET of the Cross Validated community for leading me to this insight.)

**Examples:**

- The following example generates a point inside a quarter diamond (centered at (0, ..., 0), "radius" *k* where *k* is an integer greater than 0): Let *d1*, ..., *dN* be *k*. Let **InShape** return *YES* if ((*c1*+1) + ... + (*cN*+1)) < *S***k*; *NO* if (*c1* + ... + *cN*) > *S***k*; and *MAYBE* otherwise. For *N*=2, the

acceptance rate (see note 5) is 1/2. For a full diamond, step 5.3 in the algorithm is done for each of the $N$ dimensions.

- The following example generates a point inside a quarter hypersphere (centered at (0, ..., 0), radius $k$ where $k$ is an integer greater than 0): Let $d1$, ..., $dN$ be $k$. Let **InShape** return *YES* if $((c1+1)^2 + ... + (cN+1)^2) < (S*k)^2$; *NO* if $(c1^2 + ... + cN^2) > (S*k)^2$; and *MAYBE* otherwise. For $N$=2, the acceptance rate (see note 5) is $\pi/4$. For a full hypersphere with radius 1, step 5.3 in the algorithm is done for each of the $N$ dimensions. In the case of a 2-dimensional disk, this algorithm thus adapts the well-known rejection technique of generating X and Y coordinates until $X^2+Y^2 < 1$ (e.g., (Devroye 1986, p. 230 et seq.)[^20]).
- The following example generates a point inside a quarter *astroid* (centered at (0, ..., 0), radius $k$ where $k$ is an integer greater than 0): Let $d1$, ..., $dN$ be $k$. Let **InShape** return *YES* if $((sk-c1-1)^2 + ... + (sk-cN-1)^2) > sk^2$; *NO* if $((sk-c1)^2 + ... + (sk-cN)^2) < sk^2$; and *MAYBE* otherwise, where $sk = S*k$. For $N$=2, the acceptance rate (see note 5) is $1 - \pi/4$. For a full astroid, step 5.3 in the algorithm is done for each of the $N$ dimensions.

## 5.2 Building an Arbitrary-Precision Sampler

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 $ret$, a uniform PSRN with a positive sign and an integer part of 0.
6. Create an input coin that calls **SampleGeometricBag** on the PSRN $ret$. Run a Bernoulli factory algorithm that simulates the probability $C(i, \lambda)$, using the input coin

(here, $\lambda$ 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 $\lambda$ 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 variates**, as well as the algorithms for the Rayleigh distribution and for the reciprocal of power of uniform, both given later.

Perhaps the most difficult part of describing an arbitrary-precision sampler with this skeleton is finding the appropriate Bernoulli factory 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)[^20]; an exception is that instead of generating a uniform random variate and comparing it to calculations of a CDF, this algorithm uses conditional probabilities of choosing a given piece, probabilities labeled $A$ and $B$. This approach was taken so that the CDF of the distribution in question is never directly calculated in the course of the algorithm, which furthers the goal of sampling with arbitrary precision and without using floating-point arithmetic.

## 5.3 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).

## 5.4 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.

## 5.5 Cumulative Distribution Functions

Suppose we have a real number $z$ (which might be a uniform PSRN or a rational number). If a continuous distribution—

- has a probability density function (PDF) (as with the normal or exponential distribution), and
- has an arbitrary-precision sampler that returns a uniform PSRN $X$,

then it's possible to generate 1 with the same probability as the sampler returns an $X$ that is less than or equal to $z$, as follows:

1. Run the arbitrary-precision sampler to generate $X$, a uniform PSRN.
2. Run **URandLess** (if $z$ is a uniform PSRN) or **URandLessThanReal** (if $z$ is a real number) with parameters $X$ and $z$, in that order, and return the result.

Specifically, the probability of returning 1 is the *cumulative distribution function* (CDF) for the distribution of $X$.

**Notes:**

1. Although step 2 of the algorithm checks whether $X$ is merely less than $z$, this is still correct; because the distribution of $X$ has a PDF, $X$ is less than $z$ with the same probability as $X$ is less than or equal to $z$.
2. All probability distributions have a CDF, not just those with a PDF, but also discrete ones such as Poisson or binomial.

# 6 Specific Arbitrary-Precision Samplers

## 6.1 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, $c = \text{floor}(k * 2 / y)$, and $k = 1$, then call the **sub-algorithm** given later with the uniform PSRN and $k = k$. If all of these calls return 1, the uniform PSRN was accepted. Otherwise, remove all digits from the uniform PSRN's fractional part and go to step 4.
6. If the uniform PSRN, call it *ret*, was accepted by step 5, set *ret*'s integer part to $k$, then optionally fill *ret* with uniform random digits as necessary to give its fractional part the desired number of digits (similarly to **FillGeometricBag**), and return *ret*.

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

For base 2:

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

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

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

## 6.2 Hyperbolic Secant Distribution

The following algorithm adapts the rejection algorithm from p. 472 in (Devroye 1986) [^20] for arbitrary-precision sampling.

1. Generate *ret*, an exponential random variate 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.

## 6.3 Reciprocal of Power of Uniform

The following algorithm generates a PSRN of the form $1/U^{1/x}$, where $U$ is a uniform random variate 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 *ret*, a uniform PSRN with a positive sign and an integer part of 0.
5. Create an input coin that calls **SampleGeometricBag** on the PSRN *ret*. Call the **algorithm for $d^k$ / $(c + \lambda)^k$** in "**Bernoulli Factory Algorithms**", using the input coin, where $d = intval$, $c = i$, and $k = x + 1$ (here, $\lambda$ 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 variate 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 $intval^{x+1}/(i+\lambda)^{x+1}$.

## 6.4 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 *ret*, a uniform PSRN with a positive sign and an integer part of 0.
6. Create an input coin that calls **SampleGeometricBag** on the PSRN *ret*. Call the **algorithm for $d^k$ / $(c + \lambda)^k$** in "**Bernoulli Factory Algorithms**", using the input coin, where $d = intval + 1$, $c = i + 1$, and $k = 2$ (here, $\lambda$ 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 $(intval+1)^2/(i+\lambda+1)^2$.

## 6.5 Arc-Cosine Distribution

Here we reimplement an example from Devroye's book *Non-Uniform Random Variate Generation* (Devroye 1986, pp. 128–129)[^20]. The following arbitrary-precision sampler generates a random variate from a distribution with the following cumulative distribution function (CDF): `1 - cos(pi*x/2)`. The random variate will be in the interval [0, 1]. This algorithm's result is the same as applying acos(*U*)*2/π, where *U* is a uniform [0, 1] random variate, 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. The code uses floating-point arithmetic only at the end, to convert the result to a convenient form, and it relies on methods from *randomgen.py* and *bernoulli.py*.

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

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

## 6.6 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 variate, 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.

## 6.7 Cauchy Distribution

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

1. Generate two empty PSRNs, with a positive sign, an integer part of 0, and an empty fractional part. Call the PSRNs *p1* and *p2*.
2. Set $S$ to *base*, where *base* is the base of digits to be stored by the PSRNs (such as 2 for binary or 10 for decimal). Then set *c1* and *c2* each to 0. Then set $d$ to 1.
3. Multiply *c1* and *c2* each by *base* and add a digit chosen uniformly at random to that coordinate.
4. If $((c1+1)^2 + (c2+1)^2) < S^2$, then do the following:
    1. Transfer *c1*'s least significant digits to *p1*'s fractional part, and transfer *c2*'s least significant digits to *p2*'s fractional part. The variable $d$ tells how many digits to transfer to each PSRN this way. (For example, if *base* is 10, $d$ is 3, and *c1* is 342, set *p1*'s fractional part to [3, 4, 2].)
    2. Run the **UniformDivide** algorithm (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.

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

Exponential random variates can be generated using an input coin of unknown probability of heads of $\lambda$ (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 $\lambda$. 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)[^20]. The algorithm follows:

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

Notice that the algorithm's average running time increases as $\lambda$ decreases.

## 6.9 Exponential Distribution with Rate ln(x)

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

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

The algorithm follows.

1. (Samples the integer part of the random variate.) Generate a number that is 1 with probability 1/$x$ and 0 otherwise, repeatedly until a zero is generated this way. Set $k$ to the number of ones generated this way. (This is also known as a "geometric random variate", but this terminology is avoided because it has conflicting meanings in academic works.)
    - If $x$ is a rational number and a power of 2, this step can be implemented by generating blocks of $b$ unbiased random bits until a **non-zero** block of bits is generated this way, then setting $k$ to the number of **all-zero** blocks of bits generated this way.
    - If $x$ is a uniform PSRN, this step is implemented as follows: Run the first subalgorithm (later in this section) repeatedly until a run returns 0. Set $k$ to the number of runs that returned 1 this way.
2. (The rest of the algorithm samples the fractional part.) Create $f$, a uniform PSRN with a positive sign, an empty fractional part, and an integer part of 0.
3. Create a $\mu$ 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(1+*y*/*z*)** (given in "Bernoulli Factory Algorithms") with $y/z$ = 1/1. If both calls return 1, return 1. Otherwise, return 0." (This simulates the probability $\lambda = f$*ln(2).) Then:

    - If $x$ is a rational number, but not a power of 2, also create a $\nu$ input coin that does the following: "**Sample from the number *f***, then run the **algorithm for ln(1 + *y*/*z*)** with $y/z = (x-2^b)/2^b$. If both calls return 1, return 1. Otherwise, return 0."
    - If $x$ is a uniform PSRN, also create a $\rho$ input coin that does the following: "Return the result of the second subalgorithm (later in this section), given $x$ and $b$", and a $\nu$ input coin that does the following: "**Sample from the number *f***, then run the **algorithm for ln(1 + λ)**, using the $\rho$ 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 $\mu$ input coin. If a $\nu$ input coin was created in step 3, run the same algorithm once, using the $\nu$ input coin. If all these calls return 1, accept $f$. If $f$ is accepted this way, set $f$'s integer part to $k$, then optionally fill $f$ with uniform random digits as necessary to give its fractional part the desired number of digits (similarly to **FillGeometricBag**), then return $f$.
5. If $f$ was not accepted by the previous step, go to step 2.

**Note**: A *bounded exponential* random variate with rate ln($x$) and bounded by $m$ has a similar algorithm to this one. Step 1 is changed to read as follows: "Do the following $m$ times or until a zero is generated, whichever happens first: 'Generate a number that is 1 with probability $1/x$ and 0 otherwise'. Then set $k$ to the number of ones generated this way. ($k$ is a so-called bounded-geometric($1-1/x$, $m$) random variate, which an algorithm of Bringmann and Friedrich (2013)[^22] can generate as well. If $x$ is a power of 2, this can be implemented by generating blocks of $b$ unbiased random bits until a **non-zero** block of bits or $m$ blocks of bits are generated this way, whichever comes first, then setting $k$ to the number of **all-zero** blocks of bits generated this way.) If $k$ is $m$, return $m$ (this $m$ is a constant, not a uniform PSRN; if the algorithm would otherwise return a uniform PSRN, it can return something else in order to distinguish this constant from a uniform PSRN)." Additionally, instead of generating a uniform(0,1) random variate in step 2, a uniform(0,$\mu$) random variate can be generated instead, such as a uniform PSRN generated via **RandUniformFromReal**, to implement an exponential distribution bounded by $m+\mu$ (where $\mu$ 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 variate. Unlike the similar algorithm of Ahrens and Dieter (1972)[^23], this one doesn't require a table of probability values.

1. (Samples the integer part of the random variate. This will be geometrically distributed with parameter 1/2.) Generate unbiased random bits until a zero is generated this way. Set $k$ to the number of ones generated this way.
2. (The rest of the algorithm samples the fractional part.) Generate a uniform (0, 1) random variate, 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(1+*y*/*z*)** (given in "Bernoulli Factory Algorithms") with $y/z = 1/1$. If both calls return 1, return 1. Otherwise, return 0." (This simulates the probability $\lambda = f*\ln(2)$.)
4. Run the **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.

## 6.10 Symmetric Geometric Distribution

Samples from the symmetric geometric distribution from (Ghosh et al. 2012)[^24], with parameter $\lambda$, in the form of an input coin with unknown probability of heads of $\lambda$.

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 $1/(2-\lambda)$**, 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)[^25], in which the input coin—

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

## 6.11 Lindley Distribution and Lindley-Like Mixtures

A random variate that follows the Lindley distribution (Lindley 1958)[^26] with parameter $\theta$ (a real number greater than 0) can be generated as follows:

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

For the Garima distribution (Shanker 2016)[^27], $w = (1+\theta)/(2+\theta)$.

For the i-Garima distribution (Singh and Das 2020)[^28], $w = (2+\theta)/(3+\theta)$.

For the mixture-of-weighted-exponential-and-weighted-gamma distribution in (Iqbal and Iqbal 2020)[^29], two exponential random variates (rather than one) are generated in step 1, and three (rather than two) are generated in step 2.

> **Note:** If $\theta$ is a uniform PSRN, then the check "With probability $w = \theta/(1+\theta)$"

can be implemented by running the Bernoulli factory algorithm for **(*d* + *μ*) / ((*d* + *μ*) + (*c* + *λ*))**, where $c$ is 1; $\lambda$ represents an input coin that always returns 0; $d$ is $\theta$'s integer part, and $\mu$ is an input coin that runs **SampleGeometricBag** on $\theta$'s fractional part. The check succeeds if the Bernoulli factory algorithm returns 1.

## 6.12 Gamma Distribution

The path to building an arbitrary-precision gamma sampler makes use of two algorithms: one for integer parameters of $a$, and another for rational parameters of $a$ in [0, 1). Both algorithms can be combined into an arbitrary-precision gamma generator for rational parameters $a>0$.

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

1. Generate $n$ exponential random variates with a rate of 1 via the **ExpRand** or **ExpRand2** algorithm 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 variates via rejection from the uniform distribution will allow unsampled digits to be sampled uniformly at random without deviating from the exponential distribution.
2. Generate the sum of the random variates generated in step 1 by applying the **UniformAdd** algorithm given in another document.

The second algorithm takes a parameter $a$, which must be a rational number in the interval (0, 1]. Adapted from Berman's gamma generator, as given in Devroye 1986, p. 419. Because of the power-of-uniform sub-algorithm this algorithm works only if the PSRN's fractional digits are binary (zeros or ones).

1. Create *ret*, a uniform PSRN with a positive sign and an integer part of 0. If $a$ is 1, instead generate an exponential random variate with a rate of 1 via the **ExpRand** or **ExpRand2** algorithm and return that variate.
2. Generate a PSRN *ret* using the **power-of-uniform sub-algorithm** (in the page on PSRNs) with $px/py = 1/a$.
3. (The following two steps succeed with probability $(1-ret)^{1-a}$.) Create an input coin that does the following: "Flip the input coin and return 1 minus the result."
4. Run the **algorithm for $\lambda^{x/y}$** with $x/y = 1-a$, using the input coin from step 3. If the run returns 0, go to step 1.
5. (At this point, *ret* is distributed as beta($a$, $2 - a$).) Generate two exponential random variates with a rate of 1 via **ExpRand** or **ExpRand2**, then generate their sum by applying the **UniformAdd** algorithm. Call the sum $z$.
6. Run the **UniformMultiply** algorithm on *ret* and $z$, and return the result of that algorithm.

The third algorithm combines both algorithms and works for any rational parameter $a>0$.

1. Let $n$ = floor($a$). Generate *ret*, a sum of $n$ exponential variates generated via the first algorithm in this section. If $n = a$, return *ret*.
2. Let *frac* be $a$ − floor($a$). Generate *ret2*, a gamma variate generated via the second algorithm in this section, with $a$ = *frac*.

3. Run the **UniformAdd** algorithm on *ret* and *ret2* and return the result of that algorithm.

## 6.13 One-Dimensional Epanechnikov Kernel

Adapted from Devroye and Györfi (1985, p. 236)[^30].

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

## 6.14 Uniform Distribution Inside Rectellipse

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

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

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

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

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

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

# 7 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 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 floor($m$)+1 elevations are enough?

4. A ***finite-state generator*** is a finite-state machine that generates a real number's base-2 expansion such as 0.110101100..., driven by flips of a coin. A *pushdown generator* is a finite-state generator with a stack of memory. Both generators produce real numbers with a given probability distribution. For example, a generator with a loop that outputs 0 or 1 at an equal chance produces a *uniform distribution*. The following questions ask what kinds of distributions are possible with these generators.

    1. Of the probability distributions that a finite-state generator can generate, what is the exact class of:
        - *Discrete distributions* (those that cover a finite or countably infinite set of values)?
        - *Absolutely continuous distributions* (those with a probability density function such as the uniform or triangular distribution)?
        - *Singular distributions* (covering an uncountable but zero-volume set)?
        - Absolutely continuous distributions with *continuous* density functions?
    2. Same question as 1, but for pushdown generators.
    3. Of the probability distributions that a pushdown generator can generate, what is the exact class of distributions with piecewise density functions whose pieces have infinitely many "slope" functions? (The answer is known for finite-state generators.)

# 8 Notes

[^1]: Shaddin Dughmi, Jason D. Hartline, Robert Kleinberg, and Rad Niazadeh. 2017. Bernoulli Factories and Black-Box Reductions in Mechanism Design. In *Proceedings of 49th Annual ACM SIGACT Symposium on the Theory of Computing*, Montreal, Canada, June 2017 (STOC'17).

[^2]: A blog post by John D. Cook, "A more powerful alternating series theorem", Nov. 7, 2021, played a major role in leading me to this idea for Bernoulli factory designs.

[^3]: 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.

[^4]: S. Ray, P.S.V. Nataraj, "A Matrix Method for Efficient Computation of Bernstein Coefficients", *Reliable Computing* 17(1), 2012.

[^5]: choose($n$, $k$) = (1*2*3*...*$n$)/((1*...*$k$)*(1*...*($n-k$))) = $n!/(k!$ * $(n-k)!)$ is a *binomial coefficient*, or the number of ways to choose $k$ out of $n$ labeled items. It can be calculated, for example, by calculating $i/(n-i+1)$ for each integer $i$ in the interval $[n-k+1, n]$, then multiplying the results (Yannis Manolopoulos. 2002. "**Binomial coefficient computation: recursion or iteration?**", SIGCSE Bull. 34, 4 (December 2002), 65–67). For every $m>0$, choose($m$, 0) = choose($m$, $m$) = 1 and choose($m$, 1) = choose($m$, $m-1$) = $m$; also, in this document, choose($n$, $k$) is 0 when $k$ is less than 0 or greater than $n$.

[^6]: Thomas, A.C., Blanchet, J., "**A Practical Implementation of the Bernoulli Factory**", arXiv:1106.2508v3 [stat.AP], 2012.

[^7]: Nacu, Şerban, and Yuval Peres. "**Fast simulation of new coins from old**", The Annals of Applied Probability 15, no. 1A (2005): 93-115.

[^8]: 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.

[^9]: Jacob, P.E., Thiery, A.H., "On nonnegative unbiased estimators", Ann. Statist., Volume 43, Number 2 (2015), 769-784.

[^10]: Duvignau, R., 2015. Maintenance et simulation de graphes aléatoires dynamiques (Doctoral dissertation, Université de Bordeaux).

[^11]: There are many distributions that can be sampled using the oracle, by first generating unbiased random bits via randomness extraction methods, but then these distributions won't use the unknown $n$ in general. Duvignau proved Theorem 5.2 for an oracle that outputs *arbitrary* but still distinct items, as opposed to integers, but this case can be reduced to the integer case (see section 4.1.3).

[^12]: Mossel, Elchanan, and Yuval Peres. New coins from old: computing with unknown bias. Combinatorica, 25(6), pp.707-724, 2005.

[^13]: Flajolet, P., Pelletier, M., Soria, M., "**On Buffon machines and numbers**", arXiv:0906.5560 [math.PR], 2010

[^14]: Fishman, D., Miller, S.J., "Closed Form Continued Fraction Expansions of Special Quadratic Irrationals", ISRN Combinatorics Vol. 2013, Article ID 414623 (2013).

[^15]: It can also be said that the area under the graph of $x - \text{floor}(1/x)$, where $x$ is in the closed interval [0, 1], equals 1 minus $\gamma$. See, for example, Havil, J., *Gamma: Exploring Euler's Constant*, 2003.

[^16]: Citterio, M., Pavani, R., "A Fast Computation of the Best k-Digit Rational Approximation to a Real Number", Mediterranean Journal of Mathematics 13 (2016).

[^17]: 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.

[^18]: C.T. Li, A. El Gamal, "**A Universal Coding Scheme for Remote Generation of Continuous Random Variables**", arXiv:1603.05238v1 [cs.IT], 2016

[^19]: Oberhoff, Sebastian, "**Exact Sampling and Prefix Distributions**", *Theses and Dissertations*, University of Wisconsin Milwaukee, 2018.

[^20]: Devroye, L., ***Non-Uniform Random Variate Generation***, 1986.

[^21]: Harlow, J., Sainudiin, R., Tucker, W., "Mapped Regular Pavings", *Reliable Computing* 16 (2012).

[^22]: 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).

[^23]: Ahrens, J.H., and Dieter, U., "Computer methods for sampling from the exponential and normal distributions", *Communications of the ACM* 15, 1972.

[^24]: Ghosh, A., Roughgarden, T., and Sundararajan, M., "Universally Utility-Maximizing Privacy Mechanisms", *SIAM Journal on Computing* 41(6), 2012.

[^25]: Li, L., 2021. Bayesian Inference on Ratios Subject to Differentially Private Noise (Doctoral dissertation, Duke University).

[^26]: Lindley, D.V., "Fiducial distributions and Bayes' theorem", *Journal of the Royal Statistical Society Series B*, 1958.

[^27]: Shanker, R., "Garima distribution and its application to model behavioral science data", *Biom Biostat Int J.* 4(7), 2016.

[^28]: Singh, B.P., Das, U.D., "**On an Induced Distribution and its Statistical Properties**", arXiv:2010.15078 [stat.ME], 2020.

[^29]: 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.

[^30]: Devroye, L., Györfi, L., *Nonparametric Density Estimation: The L1 View*, 1985.

[^31]: 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.

[^32]: Daumas, M., Lester, D., Muñoz, C., "**Verified Real Number Calculations: A Library for Interval Arithmetic**", arXiv:0708.3721 [cs.MS], 2007.

[^33]: Karney, C.F.F., "**Sampling exactly from the normal distribution**", arXiv:1303.6257v2 [physics.comp-ph], 2014.

[^34]: I thank D. Eisenstat from the *Stack Overflow* community for leading me to this insight.

[^35]: Brassard, G., Devroye, L., Gravel, C., "Remote Sampling with Applications to General Entanglement Simulation", *Entropy* 2019(21)(92), **https://doi.org/10.3390/e21010092** .

[^36]: Devroye, L., Gravel, C., "**Random variate generation using only finitely many unbiased, independently and identically distributed random bits**", arXiv:1502.02539v6 [cs.IT], 2020.

[^37]: Keane, M. S., and O'Brien, G. L., "A Bernoulli factory", *ACM Transactions on Modeling and Computer Simulation* 4(2), 1994.

[^38]: Wästlund, J., "**Functions arising by coin flipping**", 1999.

[^39]: Dale, H., Jennings, D. and Rudolph, T., 2015, "Provable quantum advantage in randomness processing", *Nature communications* 6(1), pp. 1-4.

[^40]: 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.

[^41]: Lee, A., Doucet, A. and Łatuszyński, K., 2014. "**Perfect simulation using atomic regeneration with application to Sequential Monte Carlo**", arXiv:1407.5770v1 [stat.CO].

[^42]: Icard, Thomas F., "Calibrating generative models: The probabilistic Chomsky–Schützenberger hierarchy", *Journal of Mathematical Psychology* 95 (2020): 102308.

[^43]: Etessami, K. and Yannakakis, M., "Recursive Markov chains, stochastic grammars, and monotone systems of nonlinear equations", *Journal of the ACM* 56(1), pp.1-66, 2009.

[^44]: Dughmi, Shaddin, Jason Hartline, Robert D. Kleinberg, and Rad Niazadeh. "Bernoulli Factories and Black-box Reductions in Mechanism Design." Journal of the ACM (JACM) 68, no. 2 (2021): 1-30.

[^45]: Esparza, J., Kučera, A. and Mayr, R., 2004, July. Model checking probabilistic pushdown automata. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science*, 2004. (pp. 12-21). IEEE.

[^46]: Elder, Murray, Geoffrey Lee, and Andrew Rechnitzer. "Permutations generated by a depth 2 stack and an infinite stack in series are algebraic." *Electronic Journal of Combinatorics* 22(1), 2015.

[^47]: Knuth, Donald E. and Andrew Chi-Chih Yao. "The complexity of nonuniform random variate generation", in *Algorithms and Complexity: New Directions and Recent Results*, 1976.

[^48]: Vatan, F., "Distribution functions of probabilistic automata", in *Proceedings of the thirty-third annual ACM symposium on Theory of computing (STOC '01)*, pp. 684-693, 2001.

[^49]: Kindler, Guy and D. Romik, "On distributions computable by random walks on graphs," *SIAM Journal on Discrete Mathematics* 17 (2004): 624-633.

[^50]: Vatan (2001) claims that a finite-state generator has a continuous `CDF` (unless it produces a single value with probability 1), but this is not necessarily true if the generator has a state that outputs 0 forever.

[^51]: Adamczewski, B., Cassaigne, J. and Le Gonidec, M., 2020. On the computational complexity of algebraic numbers: the Hartmanis–Stearns problem revisited. Transactions of the American Mathematical Society, 373(5), pp.3085-3115.

[^52]: Cobham, A., "On the Hartmanis-Stearns problem for a class of tag machines", in *IEEE Conference Record of 1968 Ninth Annual Symposium on Switching and Automata Theory* 1968.

[^53]: Adamczewski, B., Bugeaud, Y., "On the complexity of algebraic numbers I. Expansions in integer bases", *Annals of Mathematics* 165 (2007).

[^54]: Richman, F. (2012). Algebraic functions, calculus style. Communications in Algebra, 40(7), 2671-2683.

# 9 Appendix

## 9.1 Ratio of Uniforms

The Cauchy sampler given earlier demonstrates the *ratio-of-uniforms* technique for sampling a distribution (Kinderman and Monahan 1977)[^31]. 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)[^32], is less elegant than, say, the normal distribution sampler by Karney (2014)[^33], 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 $[0, \text{ceil}(d1)]\times[0, \text{ceil}(d2)]$, where $d1$ is not less than the highest value of $x*\text{sqrt}(PDF(x))$ anywhere, and $d2$ is not less than the highest value of $\text{sqrt}(PDF(x))$ 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 $\text{floor}(c1/base^d)$ and *p2*'s integer part to $\text{floor}(c2/base^d)$. (For example, if *base* is 10, $d$ is 3, and *c1* is 7342, set *p1*'s fractional part to [3, 4, 2] and *p1*'s integer part to 7.)
   2. Run the **UniformDivide** algorithm (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.

## 9.2 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 $f(t1, ..., tN) \leq 0$, **InShape** should use rational interval arithmetic (such as the one given in (Daumas et al., 2007) [^32]), 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[^34] 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 $\sigma$ is an upper bound for $\text{sqrt}(N)/(S*2)$, such as $1/S$);
    - *NO* if the signed distance (or a lower bound of such distance) at each of the box's corners is greater than $\sigma$; and
    - *MAYBE* in any other case, or if the function is unsure.

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

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

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

## 9.3 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)[^35], (Devroye 1986, p. 769)[^20], (Devroye and Gravel 2020)[^36]. 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)[^37]).
    - Otherwise, $f(U)$ must map the interval [0, 1] to [0, 1] and be continuous everywhere or "almost everywhere".

  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)[^3].

The algorithm follows.

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

    **Notes:**

    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.

## 9.4 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)
```

## 9.5 Derivation of My Algorithm for min($\lambda$, 1/2)

The following explains how the algorithm is derived.

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

- $A$ = (1/2) * $\lambda$, and
- $B$ = (1/2) * min($\lambda$, 1−$\lambda$)
  = (1/2) * ((1−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—

- $g(k)$ = 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)[^38]; Dale et al. (2015)[^39]). 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—

- $z = (4^v/2)$ / 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)[^40]). 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 $i^{\text{th}}$ 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.

## 9.6 Algorithm for sin($\lambda$*$\pi$/2)

The following algorithm returns 1 with probability sin($\lambda$*$\pi$/2) and 0 otherwise, given a coin that shows heads with probability $\lambda$. However, this algorithm appears in the appendix since it requires manipulating irrational numbers, particularly numbers involving $\pi$.

1. Choose at random an integer $n$ (0 or greater) with probability $(\pi/2)^{4*n+2}/((4*n+2)!) - (\pi/2)^{4*n+4}/((4*n+4)!)$.
2. Let $v = 16*(n+1)*(4*n+3)$.
3. Flip the input coin $4*n+4$ times. Let *tails* be the number of flips that returned 0 this way. (This is the number of heads if the probability $\lambda$ were $1 - \lambda$.)
4. If *tails* = $4*n+4$, return 0.
5. If *tails* = $4*n+3$, return a number that is 0 with probability $8*(4*n+3)/(v-\pi^2)$ and 1 otherwise.
6. If *tails* = $4*n+2$, return a number that is 0 with probability $8/(v-\pi^2)$ and 1 otherwise.
7. Return 1.

Derivation: Write— $$f(\lambda) = \sin(\lambda \pi/2) = 1-g(1-\lambda),$$ where— $$g(\mu) = 1-\sin((1-\mu) \pi/2)$$ $$= \sum_{n\ge 0} \frac{(\mu\pi/2)^{4n+2}}{(4n+2)!} - \frac{(\mu\pi/2)^{4n+4}}{(4n+4)!}$$ $$= \sum_{n\ge 0} w_n(\mu) = \sum_{n\ge 0} w_n(1) \frac{w_n(\mu)}{w_n(1)}.$$

This is a **convex combination** of $w_n(1)$ and $\frac{w_n(\mu)}{w_n(1)}$ — to simulate $g(\mu)$, first an integer $n$ is chosen with probability $w_n(1)$ and then a coin that shows heads with probability $\frac{w_n(\mu)}{w_n(1)}$ is flipped. Finally, to simulate $f(\lambda)$, the input coin is "inverted" ($\mu = 1-\lambda$), $g(\mu)$ is simulated using the "inverted" coin, and 1 minus the simulation result is returned.

As given above, each term $w_n(\mu)$ is a polynomial in $\mu$, and is monotone increasing and equals 1 or less everywhere on the interval $[0, 1]$, and $w_n(1)$ is a constant so that $\frac{w_n(\mu)}{w_n(1)}$ remains a polynomial. Each polynomial $\frac{w_n(\mu)}{w_n(1)}$ can be transformed into a polynomial in Bernstein form with the following coefficients: $$(0, 0, ..., 0, 8/(v-\pi^2), 8(4n+3)/(v-\pi^2), 1),$$ where the polynomial is of degree $4n+4$ and so has $4n+5$ coefficients, and $v = \frac{((4n+4)!)\times 2^{4n+4}}{((4n+2)!)\times 2^{4n+2}} = 16 (n+1) (4n+3)$. These are the coefficients used in steps 4 through 7 of the algorithm above.

> **Note:** sin($\lambda$*$\pi$/2) = cos((1−$\lambda$)*$\pi$/2).

## 9.7 Sampling Distributions Using Incomplete Information: Omitted Algorithms

**Algorithm 2.** Say we have an *oracle* that produces independent random real numbers whose expected ("average") value is a known or unknown mean. The goal is now to

produce non-negative random variates whose expected value is 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 $\delta$, where $\delta$ is a known rational number greater than 0. The algorithm to do so follows (see Lee et al. 2014)[^41]:

1. Let $m$ be a rational number equal to or greater than the maximum value of abs($f(\mu)$) anywhere. Create a $\nu$ 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*$\nu$ (2 times the $\nu$ coin's probability of heads), using the $\nu$ input coin, with $\epsilon = \delta/m$. If the factory returns 1, return 0. Otherwise, take a number from the oracle, call it $\xi$, and return abs($f(\xi)$).

   **Example:** An example from Lee et al. (2014)[^41]. Say the oracle produces uniform random variates in [0, 3*$\pi$], and let $f(\nu) = \sin(\nu)$. Then the mean of $f(X)$ is 2/(3*$\pi$), 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 variates whose expected ("average") value is 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 $\mu$ 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)[^41]: If $f(\mu)$ is known to return only values in the interval [$a$, $c$], the mean of $f(X)$ is not less than $\delta$, $\delta > b$, and $\delta$ 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 ($\mu$) is unknown. The goal is to use the oracle to produce non-negative random variates with mean $f(\mu)$. This is possible only if $f$ is 0 or greater everywhere in the interval [$a$, $\infty$) and is nondecreasing in that interval (Jacob and Thiery 2015)[^9]. 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.
- $\psi$ 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 variates in a single run of the algorithm.)
2. If $k$ is greater than 0: Take 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 $\psi$ and add 1 to $k$.
5. With probability $\psi$, 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)[^9], 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.

# 9.8 Pushdown Automata and Algebraic Functions

This section has mathematical proofs showing which kinds of algebraic functions (functions that can be a solution of a system of polynomial equations) can be simulated with a pushdown automaton (a state machine with a stack).

The following summarizes what can be established about these algebraic functions:

- sqrt($\lambda$) can be simulated.
- Every rational function with rational coefficients that maps the open interval (0, 1) to (0, 1) can be simulated.
- If $f(\lambda)$ can be simulated, so can any Bernstein-form polynomial in the variable $f(\lambda)$ with coefficients that can be simulated.
- If $f(\lambda)$ and $g(\lambda)$ can be simulated, so can $f(\lambda)$*$g(\lambda)$, $f(g(\lambda))$, and $g(f(\lambda))$.
- If a full-domain pushdown automaton (defined later) can generate words of a given length with a given probability (a *probability distribution* of word lengths), then the probability generating function for that distribution can be simulated, as well as for that distribution conditioned on a finite set or periodic infinite set of word lengths (e.g., all odd word lengths only).
- If a stochastic context-free grammar (defined later) can generate a probability distribution of word lengths, and terminates with probability 1, then the probability generating function for that distribution can be simulated.
- Every quadratic irrational in (0, 1) can be simulated.

It is not yet known whether the following functions can be simulated: $\lambda^{1/p}$ for prime numbers $p$ greater than 2, or $\min(\lambda, 1-\lambda)$.

---

The following definitions are used in this section:

1. A *pushdown automaton* has a finite set of *states* and a finite set of *stack symbols*, one of which is called EMPTY, and takes a coin that shows heads with an unknown probability. It starts at a given state and its stack starts with EMPTY. On each iteration:
    - The automaton flips the coin.
    - Based on the coin flip (HEADS or TAILS), the current state, and the top stack

symbol, it moves to a new state (or keeps it unchanged) and replaces the top stack symbol with zero, one or two symbols. Thus, there are three kinds of *transition rules*:

- (*state*, *flip*, *symbol*) → (*state2*, {*symbol2*}): move to *state2*, replace top stack symbol with same or different one.
- (*state*, *flip*, *symbol*) → (*state2*, {*symbol2*, *new*}): move to *state2*, replace top stack symbol with *symbol2*, then *push* a new symbol (*new*) onto the stack.
- (*state*, *flip*, *symbol*) → (*state2*, {}): move to *state2*, *pop* the top symbol from the stack.

When the stack is empty, the machine stops, and returns either 0 or 1 depending on the state it ends up at. (Because each left-hand side has no more than one possible transition, the automaton is *deterministic*.)

2. A *full-domain pushdown automaton* means a pushdown automaton that terminates with probability 1 given a coin with probability of heads $\lambda$, for every $\lambda$ in the open interval (0, 1).

3. **PDA** is the class of functions that map the open interval (0, 1) to (0, 1) and can be simulated by a full-domain pushdown automaton. **PDA** also includes the constant functions 0 and 1.

4. **ALGRAT** is the class of functions that map the open interval (0, 1) to (0, 1), are continuous, and are algebraic over the rational numbers (they satisfy a nonzero polynomial system whose coefficients are rational numbers). **ALGRAT** also includes the constant functions 0 and 1.

5. A *probability generating function* has the form $p_0*\lambda^0 + p_1*\lambda^1 + ...$, where $p_i$ (a *coefficient*) is the probability of getting $i$.

   **Notes:**

   1. Mossel and Peres (2005)[^12] defined pushdown automata to start with a non-empty stack of *arbitrary* size, and to allow each rule to replace the top symbol with an *arbitrary* number of symbols. Both cases can be reduced to the definition in this section.
   2. Pushdown automata, as defined here, are very similar to so-called *probabilistic right-linear indexed grammars* (Icard 2020)[^42] and can be translated to those grammars as well as to *probabilistic pushdown systems* (Etessami and Yannakakis 2009)[^43], as long as those grammars and systems use only transition probabilities that are rational numbers.

**Proposition 0** (Mossel and Peres 2005[^42], Theorem 1.2): *A full-domain pushdown automaton can simulate a function that maps (0, 1) to (0, 1) only if the function is in class **ALGRAT**.*

It is not known whether **ALGRAT** and **PDA** are equal, but the following can be established about **PDA**:

**Lemma 1A:** *Let g(λ) be a function in the class **PDA**, and suppose a pushdown automaton F has two rules of the form (`state`, HEADS, `stacksymbol`) → RHS1 and (`state`, TAILS, `stacksymbol`) → RHS2, where `state` and `stacksymbol` are a specific state/symbol pair among the left-hand sides of F's rules. Then there is a pushdown automaton that transitions to RHS1 with probability g(λ) and to RHS2 with probability 1−g(λ) instead.*

*Proof:* If RHS1 and RHS2 are the same, then the conclusion holds and nothing has to be

done. Thus assume RHS1 and RHS2 are different.

Let $G$ be the full-domain pushdown automaton for $g$. Assume that machines $F$ and $G$ stop when they pop EMPTY from the stack. If this is not the case, transform both machines by renaming the symbol EMPTY to EMPTY″, adding a new symbol EMPTY″ and new starting state X0, and adding rules (X0, *flip*, EMPTY) → (*start*, {EMPTY″}) and rule (*state*, *flip*, EMPTY) → (*state*, {}) for all states other than X0, where *start* is the starting state of $F$ or $G$, as the case may be.

Now, 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 the following two pairs of rules in $F$, namely —

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

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 state in $G$ with a name distinct from all other states in $F$.
- 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 $F$. □

**Lemma 1B:** *There are pushdown automata that simulate the probabilities 0 and 1.*

*Proof:* The probability 0 automaton has the rules (START, HEADS, EMPTY) → (START, {}) and (START, TAILS, EMPTY) → (START, {}), and its only state START is associated with output 0. The probability 1 automaton is the same, except START is associated with output 1. Both automata obviously terminate with probability 1. □

Because of Lemma 1A, it's possible to label each left-hand side of a pushdown automaton's rules with not just HEADS or TAILS, but also a rational number or another function in **PDA**, as long as for each state/symbol pair, the probabilities for that pair sum to 1. For example, rules like the following are now allowed:

(START, 1/2, EMPTY) → ..., (START, sqrt($\lambda$)/2, EMPTY) → ..., (START, (1 − sqrt($\lambda$))/2, EMPTY) → ....

**Proposition 1A:** *If f(λ) is in the class **PDA**, then so is every polynomial written as—*

$\sum_{i = 0, \ldots, n} \text{choose}(n, i) * f(\lambda)^i * (1 - f(\lambda))^{n - i} * a[i]$,

*where n is the polynomial's degree and a[i] is a function in the class **PDA**.*

*Proof Sketch*: This corresponds to a two-stage pushdown automaton that follows the algorithm of Goyal and Sigman (2012)[^8]: The first stage counts the number of "heads" shown when flipping the f(λ) coin, and the second stage flips another coin that has success probability $a[i]$, where $i$ is the number of "heads". The automaton's transitions take advantage of Lemma 1A. □

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

*Proof:* Special case of Proposition 1A with $n$=1, $f(\lambda)$=$f(\lambda)$, $a[0]$=0 (using Lemma 1B), and $a[1]$=$g(\lambda)$. □

**Corollary 1A:** *If f(λ), g(λ), and h(λ) are functions in the class **PDA**, then so is f(λ)*g(λ) + (1−f(λ))*h(λ).*

*Proof:* Special case of Proposition 1A with $n$=1, $f(\lambda)$=$f(\lambda)$, $a[0]$=$h(\lambda)$, and $a[1]$=$g(\lambda)$. □

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

*Proof:* Let $F$ be the full-domain pushdown automaton for $f$. For each state/symbol pair among the left-hand sides of $F$'s rules, apply Lemma 1A to the automaton $F$, using the function $g$. Then the new machine $F$ terminates with probability 1 because the original $F$ and the original automaton for $g$ do for every $\lambda$ in (0, 1), and because the automaton for $g$ maps to (0, 1) where $F$ terminates with probability 1. Moreover, $f$ is in class **PDA** by Theorem 1.2 of (Mossel and Peres 2005)[^12] because the machine is a full-domain pushdown automaton. □

**Proposition 3:** *Every rational function with rational coefficients that maps (0, 1) to (0, 1) is in class **PDA**.*

*Proof:* These functions can be simulated by a finite-state machine (Mossel and Peres 2005)[^12]. This corresponds to a full-domain pushdown automaton that has no stack symbols other than EMPTY, never pushes symbols onto the stack, and pops the only symbol EMPTY from the stack whenever it transitions to a final state of the finite-state machine. □

> **Note:** An unbounded stack size is necessary for a pushdown automaton to simulate functions that a finite-state machine can't. With a bounded stack size, there is a finite-state machine where each state not only holds the pushdown automaton's original state, but also encodes the contents of the stack (which is possible because the stack's size is bounded); each operation that would push, pop, or change the top symbol transitions to a state with the appropriate encoding of the stack instead.

**Proposition 4:** *If a full-domain pushdown automaton can generate words with the same letter such that the length of each word follows a probability distribution, then that distribution's probability generating function is in class **PDA**.*

*Proof:* Let $F$ be a full-domain pushdown automaton. Add one state FAILURE, then augment $F$ with a special "letter-generating" operation as follows. Add the following rule

that pops all symbols from the stack:

(FAILURE, *flip*, *stacksymbol*) → (FAILURE, {}),

and for each rule of the following form that transitions to a letter-generating operation (where S and T are arbitrary states):

(S, *flip*, *stacksymbol*) → (T, *newstack*),

add another state S′ (with a name that differs from all other states) and replace that rule with the following rules:

(S, *flip*, *stacksymbol*) → (S′, {*stacksymbol*}),
(S′, HEADS, *stacksymbol*) → (T, *newstack*), and
(S′, TAILS, *stacksymbol*) → (FAILURE, {}).

Then if the stack is empty upon reaching the FAILURE state, the result is 0, and if the stack is empty upon reaching any other state, the result is 1. By (Dughmi et al. 2021)[^44], the machine now simulates the distribution's probability generating function. Moreover, the function is in class **PDA** by Theorem 1.2 of (Mossel and Peres 2005)[^12] because the machine is a full-domain pushdown automaton. □

Define a *stochastic context-free grammar* as follows. The grammar consists of a finite set of *nonterminals* and a finite set of *letters*, and rewrites one nonterminal (the starting nonterminal) into a word. The grammar has three kinds of rules (in generalized Chomsky Normal Form (Etessami and Yannakakis 2009)[^43]):

- $X \to a$ (rewrite $X$ to the letter $a$).
- $X \to_p (a, Y)$ (with rational probability $p$, rewrite $X$ to the letter $a$ followed by the nonterminal $Y$). For the same left-hand side, all the $p$ must sum to 1.
- $X \to (Y, Z)$ (rewrite $X$ to the nonterminals $Y$ and $Z$ in that order).

Instead of a letter (such as $a$), a rule can use $\varepsilon$ (the empty string). (The grammar is *context-free* because the left-hand side has only a single nonterminal, so that no context from the word is needed to parse it.)

**Proposition 5:** *Every stochastic context-free grammar can be transformed into a pushdown automaton. If the automaton is a full-domain pushdown automaton and the grammar has a one-letter alphabet, the automaton can generate words such that the length of each word follows the same distribution as the grammar, and that distribution's probability generating function is in class* ***PDA****.*

*Proof Sketch:* In the equivalent pushdown automaton:

- $X \to a$ becomes the two rules—
  (START, HEADS, $X$) → (*letter*, {}), and
  (START, TAILS, $X$) → (*letter*, {}).
  Here, *letter* is either START or a unique state in $F$ that "detours" to a letter-generating operation for $a$ and sets the state back to START when finished (see Proposition 4). If $a$ is $\varepsilon$, *letter* is START and no letter-generating operation is done.
- $X \to_{p_i} (a_i, Y_i)$ (all rules with the same nonterminal $X$) are rewritten to enough rules to transition to a letter-generating operation for $a_i$, and swap the top stack symbol with $Y_i$, with probability $p_i$, which is possible with just a finite-state machine (see Proposition 4) because all the probabilities are rational numbers (Mossel and Peres 2005)[^12]. If $a_i$ is $\varepsilon$, no letter-generating operation is done.
- $X \to (Y, Z)$ becomes the two rules—

(START, HEADS, $X$) → (START, {$Z$, $Y$}), and
(START, TAILS, $X$) → (START, {$Z$, $Y$}).

Here, $X$ is the stack symbol EMPTY if $X$ is the grammar's starting nonterminal. Now, assuming the automaton is full-domain, the rest of the result follows easily. For a single-letter alphabet, the grammar corresponds to a system of polynomial equations, one for each rule in the grammar, as follows:

- $X \to a$ becomes $X = 1$ if $a$ is the empty string ($\varepsilon$), or $X = \lambda$ otherwise.
- For each nonterminal $X$, all $n$ rules of the form $X \to_{p_i} (a_i, Y_i)$ become the equation $X = p_1*\lambda_1*Y_1 + p_2*\lambda_2*Y_2 + ... + p_n*\lambda_n*Y_n$, where $\lambda_i$ is either 1 if $a_i$ is $\varepsilon$, or $\lambda$ otherwise.
- $X \to (Y, Z)$ becomes $X = Y*Z$.

Solving this system for the grammar's starting nonterminal, and applying Proposition 4, leads to the *probability generating function* for the grammar's word distribution. (See also Flajolet et al. 2010[^13], Icard 2020[^42].) □

**Example:** The stochastic context-free grammar—
$X \to_{1/2} (a, X1)$,
$X1 \to (X, X2)$,
$X2 \to (X, X)$,
$X \to_{1/2} (a, X3)$,
$X3 \to \varepsilon$,
which encodes ternary trees (Flajolet et al. 2010)[^13], corresponds to the equation $X = (1/2) * \lambda*X*X*X + (1/2)*\lambda*1$, and solving this equation for $X$ leads to the probability generating function for such trees, which is a complicated expression.

**Notes:**

1. A stochastic context-free grammar in which all the probabilities are 1/2 is called a *binary stochastic grammar* (Flajolet et al. 2010[^13]). If the starting nonterminal has $n$ rules of probability $1/n$, then the grammar can be called an "$n$-ary stochastic grammar". It is even possible for a nonterminal to have two rules of probability $\lambda$ and $(1-\lambda)$, which are used when the input coin returns 1 (HEADS) or 0 (TAILS), respectively.

2. If a pushdown automaton simulates the function $f(\lambda)$, then $f$ corresponds to a special system of equations, built as follows (Mossel and Peres 2005)[^12]; see also (Esparza et al. 2004)[^45]. For each state of the automaton (call the state $en$), include the following equations in the system based on the automaton's transition rules:

    - ($st$, $p$, $sy$) → ($s2$, {}) becomes either $\alpha_{st,sy,en} = p$ if $s2$ is $en$, or $\alpha_{st,sy,en} = 0$ otherwise.
    - ($st$, $p$, $sy$) → ($s2$, {$sy1$}) becomes $\alpha_{st,sy,en} = p * \alpha_{s2,sy1,en}$.
    - ($st$, $p$, $sy$) → ($s2$, {$sy1$, $sy2$}) becomes $\alpha_{st,sy,en} = p*\alpha_{s2,sy2,\sigma[1]}*\alpha_{\sigma[1],sy1,en} + ... + p*\alpha_{s2,sy2,\sigma[n]}*\alpha_{\sigma[n],sy1,en}$, where $\sigma[i]$ is one of the machine's $n$ states.

    (Here, $p$ is the probability of using the given transition rule; the special value HEADS becomes $\lambda$, and the special value TAILS becomes $1-\lambda$.) Now, each time multiple equations have the same left-hand side, combine them into one equation with the same left-hand side, but with the sum of their right-hand sides. Then, for any variable of the form $\alpha_{a,b,c}$ not yet present in

the system, include the equation $\alpha_{a,b,c} = 0$. Then, for each final state $fs$ that returns 1, solve the system for the variable $\alpha_{\text{START,EMPTY},fs}$ (where START is the automaton's starting state) to get a solution (a function) that maps (0, 1) to (0, 1). (Each solve can produce multiple solutions, but only one of them will map (0, 1) to (0, 1) assuming every $p$ is either HEADS or TAILS.) Finally, add all the solutions to get $f(\lambda)$.

3. Assume there is a pushdown automaton ($F$) that follows Definition 1 except it uses a set of $N$ input letters (and not simply HEADS or TAILS), accepts an input word if the stack is empty, and rejects the word if the machine reaches a configuration without a transition rule. Then a pushdown automaton in the full sense of Definition 1 ($G$) can be built. In essence:

    1. Add a new FAILURE state, which when reached, pops all symbols from the stack.
    2. For each pair (*state*, *stacksymbol*) for $F$, add a set of rules that generate one of the input letters (each letter $i$ generated with probability $f_i(\lambda)$, which must be a function in **PDA**), then use the generated letter to perform the transition stated in the corresponding rule for $F$. If there is no such transition, transition to the FAILURE state instead.
    3. When the stack is empty, output 0 if $G$ is in the FAILURE state, or 1 otherwise.

    Then $G$ returns 1 with the same probability as $F$ accepts an input word with letters randomly generated as in the second step. Also, one of the $N$ letters can be a so-called "end-of-string" symbol, so that a pushdown automaton can be built that accepts "empty strings"; an example is (Elder et al. 2015)[^46].

**Proposition 6:** *If a full-domain pushdown automaton can generate a distribution of words with the same letter, there is a full-domain pushdown automaton that can generate a distribution of such words conditioned on—*

1. *a finite set of word lengths, or*
2. *a periodic infinite set of word lengths.*

One example of a finite set of word lengths is {1, 3, 5, 6}, where only words of length 1, 3, 5, or 6 are allowed. A *periodic infinite set* is defined by a finite set of integers such as {1}, as well as an integer modulus such as 2, so that in this example, all integers congruent to 1 modulo 2 (that is, all odd integers) are allowed word lengths and belong to the set.

*Proof Sketch:*

1. As in Lemma 1A, assume that the automaton stops when it pops EMPTY from the stack. Let $S$ be the finite set (e.g., {1, 3, 5, 6}), and let $M$ be the maximum value in the finite set. For each integer $i$ in [0, $M$], make a copy of the automaton and append the integer $i$ to the name of each of its states. Combine the copies into a new automaton $F$, and let its start state be the start state for copy 0. Now, whenever $F$ generates a letter, instead of transitioning to the next state after the letter-generating operation (see Proposition 4), transition to the corresponding state for the next copy (e.g., if the operation would transition to copy 2's version of "XYZ", namely "2_XYZ", transition to "3_XYZ" instead), or if the last copy is reached, transition to the last copy's FAILURE state. If $F$ would transition to a failure state corresponding to a copy not in $S$ (e.g., "0_FAILURE", "2_FAILURE", "3_FAILURE" in

this example), first all symbols other than EMPTY are popped from the stack and then $F$ transitions to its start state (this is a so-called "rejection" operation). Now, all the final states (except FAILURE states) for the copies corresponding to the values in $S$ (e.g., copies 1, 3, 5, 6 in the example) are treated as returning 1, and all other states are treated as returning 0.

2. Follow (1), except as follows: (A) $M$ is equal to the integer modulus minus 1. (B) For the last copy of the automaton, instead of transitioning to the next state after the letter-generating operation (see Proposition 4), transition to the corresponding state for copy 0 of the automaton. □

**Proposition 7:** *Every constant function equal to a quadratic irrational number in the interval (0, 1) is in class **PDA**.*

A *continued fraction* is one way to write a real number. For purposes of the following proof, every real number in (0, 1) has the following *continued fraction expansion*: 0 + 1 / ($a$[1] + 1 / ($a$[2] + 1 / ($a$[3] + ... ))), where each $a[i]$, a *partial denominator*, is an integer greater than 0. A *quadratic irrational number* is an irrational number of the form ($b$+sqrt($c$))/$d$, where $b$, $c$, and $d$ are rational numbers.

*Proof:* By Lagrange's continued fraction theorem, every quadratic irrational number has a continued fraction expansion that is eventually periodic; the expansion can be described using a finite number of partial denominators, the last "few" of which repeat forever. The following example describes a periodic continued fraction expansion: [0; 1, 2, (5, 4, 3)], which is the same as [0; 1, 2, 5, 4, 3, 5, 4, 3, 5, 4, 3, ...]. In this example, the partial denominators are the numbers after the semicolon; the size of the period (`(5, 4, 3)`) is 3; and the size of the non-period (`1, 2`) is 2.

Given a periodic expansion, and with the aid of an algorithm for simulating **continued fractions**, a recursive Markov chain for the expansion (Etessami and Yannakakis 2009)[^43] can be described as follows. The chain's components are all built on the following template. The template component has one entry E, one inner node N, one box, and two exits X0 and X1. The box has one *call port* as well as two *return ports* B0 and B1.

- From E: Go to N with probability $x$, or to the box's call port with probability $1 - x$.
- From N: Go to X1 with probability $y$, or to X0 with probability $1 - y$.
- From B0: Go to E with probability 1.
- From B1: Go to X0 with probability 1.

Let $p$ be the period size, and let $n$ be the non-period size. Now the recursive Markov chain to be built has $n$+$p$ components:

- For each $i$ in [1, $n$+1], there is a component labeled $i$. It is the same as the template component, except $x = a[i]/(1 + a[i])$, and $y = 1/a[i]$. The component's single box goes to the component labeled $i$+1, *except* that for component $n$+$p$, the component's single box goes to the component labeled $n$+1.

According to Etessami and Yannakakis (2009)[^43], the recursive Markov chain can be translated to a pushdown automaton of the kind used in this section. Now all that's left is to argue that the recursive Markov chain terminates with probability 1. For every component in the chain, it goes from its entry to its box with probability 1/2 or less (because each partial numerator must be 1 or greater). Thus, the component is never more likely than not to recurse, and there are otherwise no probability-1 loops in each component, so the overall chain terminates with probability 1. □

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

*Proof:* See (Mossel and Peres 2005)[^12]. □

**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 **PDA**.*

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

Moreover, $f$ is in class **PDA** by Theorem 1.2 of (Mossel and Peres 2005)[^12] because the machine is a full-domain pushdown automaton. □

## 9.8.1 Finite-State and Pushdown Generators

Another interesting class of machines (called *pushdown generators* here) are similar to pushdown automata (see above), with the following exceptions:

1. Each transition rule can also, optionally, output a base-$N$ digit in its right-hand side. An example is: (*state*, *flip*, *sy*) → (*digit*, *state2*, {*sy2*}).
2. The machine must output infinitely many digits if allowed to run forever.
3. Rules that would pop the last symbol from the stack are not allowed.

The "output" of the machine is now a real number $X$ in the interval [0, 1], in the form of the base-$N$ digit expansion `0.dddddd...`, where `dddddd...` are the digits produced by the machine from left to right. In the rest of this section:

- `CDF(z)` is the cumulative distribution function of $X$, or the probability that $X$ is $z$ or less.
- `PDF(z)` is the probability density function of $X$, or the "slope" function of `CDF(z)`, or the relative probability of choosing a number "close" to $z$ at random.

A *finite-state generator* (Knuth and Yao 1976)[^47] is the special case where the probability of heads is 1/2, each digit is either 0 or 1, rules can't push stack symbols, and only one stack symbol is used. Then if `PDF(z)` has infinitely many "slope" functions on the open interval (0, 1), it must be a polynomial with rational coefficients and not equal 0 at any irrational point on (0, 1) (Vatan 2001)[^48], (Kindler and Romik 2004)[^49], and it can be shown that the mean of $X$ must be a rational number. [^50]

**Proposition 8.** *Suppose a finite-state generator can generate a probability distribution that takes on finitely many values. Then:*

1. *Each value occurs with a rational probability.*
2. *Each value is either rational or transcendental.*

A real number is *transcendental* if it can't be a root of a nonzero polynomial with integer coefficients. Thus, part 2 means, for example, that irrational, non-transcendental numbers such as 1/sqrt(2) and the golden ratio minus 1 can't be generated exactly.

Proving this proposition involves the following lemma, which shows that a finite-state generator is related to a machine with a one-way read-only input and a one-way write-only output:

**Lemma 2.** *A finite-state generator can fit the model of a one-way transducer-like k-*

*machine (as defined in Adamczewski et al. (2020)[^51] section 5.3), for some k equal to 2 or greater.*

*Proof Sketch:* There are two cases.

Case 1: If every transition rule of the generator outputs a digit, then $k$ is the number of unique inputs among the generator's transition rules (usually, there are two unique inputs, namely HEADS and TAILS), and the model of a finite-state generator is modified as follows:

1. A *configuration* of the finite-state generator consists of its current state together with either the last coin flip result or, if the coin wasn't flipped yet, the empty string.
2. The *output function* takes a configuration described above and returns a digit. If the coin wasn't flipped yet, the function returns an arbitrary digit (which is not used in proposition 4.6 of the Adamczewski paper).

Case 2: If at least one transition rule does not output a digit, then the finite-state generator can be transformed to a machine where HEADS/TAILS is replaced with 50% probabilities, then transformed to an equivalent machine whose rules always output one or more digits, as claimed in Lemma 5.2 of Vatan (2001)[^48]. In case the resulting generator has rules that output more than one digit, additional states and rules can be added so that the generator's rules output only one digit as desired. Now at this point the generator's probabilities will be rational numbers. Now transform the generator from probabilities to inputs of size $k$, where $k$ is the product of those probabilities, by adding additional rules as desired. □

*Proof of Proposition 8:* Let $n$ be an integer greater than 0. Take a finite-state generator that starts at state START and branches to one of $n$ finite-state generators (sub-generators) with some probability, which must be rational because the overall generator is a finite-state machine (Icard 2020, Proposition 13)[^42]. The branching process outputs no digit, and part 3 of the proposition follows from Corollary 9 of Icard (2020)[^42]. The $n$ sub-generators are special; each of them generates the binary expansion of a single real number in [0, 1] with probability 1.

To prove part 2 of the proposition, translate an arbitrary finite-state generator to a machine described in Lemma 2. Once that is done, all that must be shown is that there are two different non-empty sequences of coin flips that end up at the same configuration. This is easy using the pigeonhole principle, since the finite-state generator has a finite number of configurations. Thus, by propositions 5.11, 4.6, and AB of Adamczewski et al. (2020)[^51], the generator can generate a real number's binary expansion only if that number is rational or transcendental (see also Cobham (1968)[^52]; Adamczewski and Bugeaud (2007)[^53]). □

**Proposition 9.** *If the distribution function generated by a finite-state generator is continuous and algebraic on the open interval (0, 1), then that function is a piecewise polynomial function.*

The proof follows from combining Kindler and Romik (2004, Theorem 2)[^49] and Knuth and Yao (1976)[^47] with Richman (2012)[^54], who proved that a continuous algebraic function on an open interval is piecewise analytic ("analytic" is a stronger statement than having infinitely many "slope" functions).

# 10 License