Bernoulli Factory Algorithms

This version of the document is dated 2021-01-16.

Peter Occil

Abstract: This page catalogs algorithms to turn coins biased one way into coins biased another way, also known as *Bernoulli factories*. It provides step-by-step instructions to help programmers implement these Bernoulli factory algorithms. This page also contains algorithms to exactly simulate probabilities that are irrational numbers, using only random bits, which is related to the Bernoulli factory problem. This page is focused on sampling methods that *exactly* simulate a given probability without introducing new errors, assuming "truly random" numbers are available. The page links to a Python module that implements several Bernoulli factories.

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

1 Introduction

This page catalogs algorithms to turn coins biased one way into coins biased another way, also known as *Bernoulli factories*. Many of them were suggested in (Flajolet et al., 2010)[(1)], but without step-by-step instructions in many cases. This page provides these instructions to help programmers implement the Bernoulli factories they describe. The Python module ***bernoulli.py*** includes implementations of several Bernoulli factories.

This page also contains algorithms to exactly simulate probabilities that are irrational numbers, using only random bits, which is related to the Bernoulli factory problem. Again, many of these were suggested in (Flajolet et al., 2010)[(1)].

This page is focused on sampling methods that *exactly* simulate the probability described, without introducing rounding errors or other errors beyond those already present in the inputs (and assuming that we have a source of "truly" random numbers, that is, random numbers that are independent and identically distributed).

1.1 About This Document

This is an open-source document; for an updated version, see the source code or its rendering on GitHub. You can send comments on this document on the GitHub issues page. See "Requests and Open Questions" for a list of things about this document that I seek answers to.

I encourage readers to implement any of the algorithms given in this page, and report their implementation experiences. This may help improve this page.

2 Contents

- **Algorithms**
  - **Algorithms for Functions of λ**
    - **Certain Power Series**
    - **$\exp(-\lambda)$**
    - **$\exp(-(\lambda^k * x))$**
    - **$\exp(-(\lambda^k * (x + m)))$**
    - **$\exp(-(\lambda + m)^k)$**
    - **$\exp(\lambda)*(1-\lambda)$**
    - **$(1-\lambda)/\cos(\lambda)$**
    - **$(1-\lambda) * \tan(\lambda)$**
    - **$\exp(\lambda * c - c)$**
    - **$\exp(-\lambda - c)$**
    - **$1/(2^{k + \lambda})$ or $\exp(-(k + \lambda)*\ln(2))$**
    - **$1/(2^{m*(k + \lambda)})$ or $1/((2^m)*(k + \lambda))$ or $\exp(-(k + \lambda)*\ln(2^m))$**
    - **$1/(1+\lambda)$**
    - **$\lambda/(1+\lambda)$**
    - **$\ln(1+\lambda)$**
    - **$1 - \ln(1+\lambda)$**
    - **$c * \lambda * \beta / (\beta * (c * \lambda + d * \mu) - (\beta - 1) * (c + d))$**
    - **$c * \lambda / (c * \lambda + d)$ or $(c/d) * \lambda / (1 + (c/d) * \lambda))$**
    - **$1 / (c + \lambda)$**
    - **$(d + \lambda) / c$**
    - **$d / (c + \lambda)$**
    - **$(d + \mu) / (c + \lambda)$**
    - **$d^k / (c + \lambda)^k$, or $(d / (c + \lambda))^k$**
    - **$\lambda + \mu$**
    - **$\lambda - \mu$**
    - **$1/(c + \lambda)$**
    - **$1 - \lambda$**
    - **$\nu * \lambda + (1 - \nu) * \mu$**
    - **$\lambda + \mu - (\lambda * \mu)$**
    - **$(\lambda + \mu) / 2$**
    - **$\arctan(\lambda) /\lambda$**
    - **$\arctan(\lambda)$**
    - **$\cos(\lambda)$**
    - **$\sin(\lambda)$**
    - **$\lambda^{x/y}$**
    - **λ^{μ}**
    - **$\text{sqrt}(\lambda)$**
    - **$\arcsin(\lambda) + \text{sqrt}(1 - \lambda^2) - 1$**
    - **$\arcsin(\lambda) / 2$**
    - **$\lambda * \mu$**
    - **$\lambda * x/y$ (linear Bernoulli factories)**
    - **$(\lambda * x/y)^i$**
    - **ϵ / λ**
    - **Certain Rational Functions**
    - **Certain Polynomials in Bernstein Form**
    - **Certain Algebraic Functions**
    - **Expressions Involving Polylogarithms**
  - **Algorithms for Irrational Constants**
    - **Digit Expansions**

3 About Bernoulli Factories

A *Bernoulli factory* (Keane and O'Brien 1994)[2] is an algorithm that takes an input coin (a method that returns 1, or heads, with an unknown probability, or 0, or tails, otherwise) and returns 0 or 1 with a probability that depends on the input coin's probability of heads. For example, a Bernoulli factory algorithm can take a coin that returns heads with probability λ and produce a coin that returns heads with probability $\exp(-\lambda)$.

A *factory function* is a function that relates the old probability to the new one. Its domain is [0, 1] or a subset of [0, 1], and returns a probability in [0, 1]. There are certain requirements for factory functions. As shown by Keane and O'Brien (1994)[2], a function $f(\lambda)$ can serve as a factory function if and only if f, within its domain—

- is continuous everywhere,
- does not go to 0 or 1 exponentially fast in value, and
- either returns a constant value in [0, 1] everywhere, or returns a value in [0, 1] at each of the points 0 and 1 and a value in (0, 1) at each other point.

As one example, the function $f = 2*\lambda$ cannot serve as a factory function, since its graph touches 1 somewhere in the open interval (0, 1).[3]

If a function's graph touches 0 or 1 somewhere in (0, 1), papers have suggested dealing with this by modifying the function so it no longer touches 0 or 1 there (for example, $f = 2*\lambda$ might become $f = \min(2 * \lambda, 1 - \epsilon)$ where ϵ is in (0, 1/2) (Keane and O'Brien 1994)[2], (Huber 2014, introduction)[4]), or by somehow ensuring that λ does not come close to the point where the graph touches 0 or 1 (Nacu and Peres 2005, theorem 1)[5].

The next section will show algorithms for a number of factory functions, allowing different kinds of probabilities to be simulated from input coins.

4 Algorithms

In the following algorithms:

- λ is the unknown probability of heads of the input coin.
- choose(n, k) = $n!/(k! * (n - k)!)$ is a binomial coefficient. It can be calculated, for example, by calculating $i/(n-i+1)$ for each integer i in $[n-k+1, n]$, then multiplying the results (Manolopoulos 2002)[6]. Note that for all $m>0$, choose(m, 0) = choose(m, m) = 1 and choose(m, 1) = choose(m, $m-1$) = m.
- The instruction to "generate a uniform(0, 1) random number" can be implemented—
  - by creating a **uniform partially-sampled random number (PSRN)** with a positive sign, an integer part of 0, and an empty fractional part (most accurate), or
  - by generating `RNDRANGEMaxExc(0, 1)` or `RNDINT(1000)` (less accurate).
- The instruction to "generate an exponential random number" can be implemented—
  - by creating an empty **exponential PSRN** (most accurate), or
  - by getting the result of the **ExpRand** or **ExpRand2** algorithm (described in my article on PSRNs) with a rate of 1, or
  - by generating `-ln(1/RNDRANGEMinExc(0, 1))` (less accurate).
- To **sample from a random number *u*** means to generate a number that is 1 with probability u and 0 otherwise.
  - If the number is a uniform PSRN, call the **SampleGeometricBag** algorithm with the PSRN and take the result of that call (which will be 0 or 1) (most accurate). (**SampleGeometricBag** is described in my **article on PSRNs**.)
  - Otherwise, this can be implemented by generating another uniform(0, 1) random number v and generating 1 if v is less than u or 0 otherwise (less

accurate).

- Where an algorithm says "if a is less than b", where a and b are random numbers, it means to run the **RandLess** algorithm on the two numbers (if they are both PSRNs), or do a less-than operation on a and b, as appropriate. (**RandLess** is described in my **article on PSRNs**.)
- Where a step in the algorithm says "with probability x" to refer to an event that may or may not happen, then this can be implemented in one of the following ways:
    - Convert x to a rational number y/z, then call `ZeroOrOne(y, z)`. The event occurs if the call returns 1. (Most accurate.) For example, if an instruction says "With probability 3/5, return 1", then implement it as "Call `ZeroOrOne(3, 5)`. If the call returns 1, return 1." `ZeroOrOne` is described in my article on **random sampling methods**.
    - Generate a uniform(0, 1) random number v. The event occurs if v is less than x. (Less accurate.)
- For best results, the algorithms should be implemented using exact rational arithmetic (such as `Fraction` in Python or `Rational` in Ruby). Floating-point arithmetic is discouraged because it can introduce errors due to fixed-precision calculations, such as rounding and cancellations.

The algorithms as described here do not always lead to the best performance. An implementation may change these algorithms as long as they produce the same results as the algorithms as described here.

The algorithms assume that a source of independent and unbiased random bits is available, in addition to the input coins. But it's possible to implement these algorithms using nothing but those coins as a source of randomness. See the **appendix** for details.

Bernoulli factory algorithms that simulate $f(\lambda)$ are equivalent to unbiased estimators of $f(\lambda)$. See the **appendix** for details.

4.1 Algorithms for Functions of λ

4.1.1 Certain Power Series

Mendo (2019)[7] gave a Bernoulli factory algorithm for certain functions that can be rewritten as a series of the form—

$$1 - (c[0] * (1 - \lambda) + ... + c[i] * (1 - \lambda)^{i + 1} + ...),$$

where $c[i] >= 0$ are the coefficients of the series and sum to 1. (According to Mendo, this implies that the series is differentiable — its graph has no "sharp corners" — and takes on a value that approaches 0 or 1 as λ approaches 0 or 1, respectively). The algorithm follows:

1. Let v be 1 and let *result* be 1.
2. Set *dsum* to 0 and i to 0.
3. Flip the input coin. If it returns v, return *result*.
4. If i is equal to or greater than the number of coefficients, set *ci* to 0. Otherwise, set *ci* to $c[i]$.
5. With probability $ci/(1 - dsum)$, return 1 minus *result*.
6. Add *ci* to *dsum*, add 1 to i, and go to step 3.

As pointed out in Mendo (2019)[7], variants of this algorithm work for power series of the

form—

1. ($c[0] * (1 - \lambda) + ... + c[i] * (1 - \lambda)^{i+1} + ...$), or
2. ($c[0] * \lambda + ... + c[i] * \lambda^{i+1} + ...$), or
3. $1 - (c[0] * \lambda + ... + c[i] * \lambda^{i+1} + ...)$.

In the first two cases, replace "let *result* be 1" in the algorithm with "let *result* be 0". In the last two cases, replace "let v be 1" with "let v be 0". Also, as pointed out by Mendo, the $c[i]$ can also sum to less than 1, in which case if the algorithm would return 1, instead it returns a number that is 1 with probability equal to the sum of all $c[i]$, and 0 otherwise.

(Łatuszyński et al. 2009/2011)**(8)** gave an algorithm that works for a wide class of series and other constructs that converge to the desired probability from above and from below.

One of these constructs is an alternating series of the form—

$$d[0] - d[1] * \lambda + d[2] * \lambda^2 - ...,$$

where $d[i]$ are all in the interval [0, 1] and form a nonincreasing sequence of coefficients.

The following is the general algorithm for this kind of series, called the **general martingale algorithm**. It takes a list of coefficients and an input coin, and returns 1 with the probability given by the series above, and 0 otherwise.

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

If the alternating series has the form—

$$d[0] - d[1] * \lambda^2 + d[2] * \lambda^4 - ...,$$

then modify the general martingale algorithm by adding the following after step 3: "3a. Repeat step 3 once." (Examples of this kind of series are found in $\sin(\lambda)$ and $\cos(\lambda)$.)

(Nacu and Peres 2005, proposition 16)**(5)**. This algorithm simulates a function of the form—

$$d[0] + d[1] * \lambda + d[2] * \lambda^2 - ...,$$

where each $d[i]$ is 0 or greater, and takes two parameters t and ε, where t must be chosen such that t is in (0, 1], $f(t) < 1$, and $\lambda < t - 2*\varepsilon$.

1. Create a ν input coin that does the following: "(1) Set n to 0. (2) With probability ε/t, go to the next substep. Otherwise, add 1 to n and repeat this substep. (3) With probability $1 - d[n]*t^n$, return 0. (4) Call the **2014 algorithm**, the **2016 algorithm**, or the **2019 algorithm**, described later, n times, using the (λ) input coin, $x/y = 1/(t - \varepsilon)$, $i = 1$ (for the 2019 algorithm), and $\epsilon = \varepsilon$. If any of these calls returns 0, return 0. Otherwise, return 1."

2. Call the **2014 algorithm**, the **2016 algorithm**, or the **2019 algorithm** once, using the ν input coin described earlier, $x/y = t/\_\varepsilon$, $i = 1$ (for the 2019 algorithm), and $\epsilon = \varepsilon$, and return the result.

4.1.2 exp(−λ)

This algorithm converges quickly everywhere in (0, 1). (In other words, the algorithm is *uniformly fast*, meaning the average running time is bounded from above by the same constant for all choices of λ and other parameters (Devroye 1986, esp. p. 717)[(9)].[(10)]) This algorithm is adapted from the general martingale algorithm (in "Certain Power Series", above), and makes use of the fact that $\exp(-\lambda)$ can be rewritten as $1 - \lambda + \lambda^2/2 - \lambda^3/6 + \lambda^4/24 - ...$, which is an alternating series whose coefficients are 1, 1, 1/(2!), 1/(3!), 1/(4!), ...

1. Set u to 1, set w to 1, set ℓ to 0, and set n to 1.
2. Generate a uniform(0, 1) random number *ret*.
3. If w is not 0, flip the input coin, multiply w by the result of the flip, and divide w by n. (This is changed from the general martingale algorithm to take account of the factorial more efficiently in the second and later coefficients.)
4. If n is even, set u to $\ell + w$. Otherwise, set ℓ to $u - w$.
5. If *ret* is less than (or equal to) ℓ, return 1. If *ret* is less than u, go to the next step. If neither is the case, return 0. (If *ret* is a uniform PSRN, these comparisons should be done via the **URandLessThanReal algorithm**, which is described in my **article on PSRNs**.)
6. Add 1 to n and go to step 3.

See the appendix for other algorithms.

4.1.3 exp(−(λ^k * x))

In the following algorithm, which applies the general martingale algorithm, k is an integer 0 or greater, and x is a rational number in the interval [0, 1]. It represents the series $1 - \lambda^{k}*x + \lambda^{2*k}*x/2! - \lambda^{3*k}*x/3!$, ..., and the coefficients are 1, x, $x/(2!)$, $x/(3!)$, ...

1. Special cases: If x is 0, return 1. If k is 0, run the **algorithm for exp(−x/y)** (given later in this page) with $x/y = x$, and return the result.
2. Set u to 1, set w to 1, set ℓ to 0, and set n to 1.
3. Generate a uniform(0, 1) random number *ret*.
4. If w is not 0, flip the input coin k times or until the flip returns 0. If any of the flips returns 0, set w to 0, or if all the flips return 1, divide w by n. Then, multiply w by a number that is 1 with probability x and 0 otherwise.
5. If n is even, set u to $\ell + w$. Otherwise, set ℓ to $u - w$.
6. If *ret* is less than (or equal to) ℓ, return 1. If *ret* is less than u, go to the next step. If neither is the case, return 0. (If *ret* is a uniform PSRN, these comparisons should be done via the **URandLessThanReal algorithm**, which is described in my **article on PSRNs**.)
7. Add 1 to n and go to step 4.

4.1.4 exp(−(λ^k * (x + m)))

In the following algorithm, k and m are both integers 0 or greater, and x is a rational number in the interval [0, 1].

1. Call the **algorithm for exp(−(λ^k * x))** m times with $k = k$ and $x = 1$. If any of these

calls returns 0, return 0.
2. If x is 0, return 1.
3. Call the **algorithm for exp(−(λ^k * x))** once, with k = k and x = x. Return the result of this call.

4.1.5 exp(−(λ + m)^k)

In the following algorithm, m and k are both integers 0 or greater.

1. If k is 0, run the **algorithm for exp(−x/y)** (given later on this page) with x/y = 1/1, and return the result.
2. If k is 1 and m is 0, run the **algorithm for exp(−λ)** and return the result.
3. Run the **algorithm for exp(−x/y)** with x/y = m^k / 1. If the algorithm returns 0, return 0.
4. Run the **algorithm for exp(−(λ^k * x))**, with k = k and x = 1. If the algorithm returns 0, return 0.
5. If m is 0, return 1.
6. Set i to 1, then while i < k:
    1. Set z to choose(k, i) * m^{k-i}.
    2. Run the **algorithm for exp(−(λ^k * x))** z times, with k = i and x = 1. If any of these calls returns 0, return 0.
    3. Add 1 to i.
7. Return 1.

4.1.6 exp(λ)*(1−λ)

(Flajolet et al., 2010)**(1)**:

1. Set k and w each to 0.
2. Flip the input coin. If it returns 0, return 1.
3. Generate a uniform(0, 1) random number U.
4. If k > 0 and w is less than U, return 0.
5. Set w to U, add 1 to k, and go to step 2.

4.1.7 (1−λ)/cos(λ)

(Flajolet et al., 2010)**(1)**:

1. Flip the input coin until the flip returns 0. Then set G to the number of times the flip returns 1 this way.
2. If G is **odd**, return 0.
3. Generate a uniform(0, 1) random number U, then set i to 1.
4. While i is less than G:
    1. Generate a uniform(0, 1) random number V.
    2. If i is odd and V is less than U, return 0.
    3. If i is even and U is less than V, return 0.
    4. Add 1 to i, then set U to V.
5. Return 1.

4.1.8 (1−λ) * tan(λ)

(Flajolet et al., 2010)**(1)**:

1. Flip the input coin until the flip returns 0. Then set G to the number of times the flip

returns 1 this way.
2. If G is **even**, return 0.
3. Generate a uniform(0, 1) random number U, then set i to 1.
4. While i is less than G:
    1. Generate a uniform(0, 1) random number V.
    2. If i is odd and V is less than U, return 0.
    3. If i is even and U is less than V, return 0.
    4. Add 1 to i, then set U to V.
5. Return 1.

4.1.9 exp(λ * *c* − *c*)

Used in (Dughmi et al. 2017)[11] to apply an exponential weight (here, c) to an input coin.

1. Generate a Poisson(c) random integer, call it N.
2. Flip the input coin until the flip returns 0 or the coin is flipped N times, whichever comes first. Return 1 if all the coin flips, including the last, returned 1 (or if N is 0); or return 0 otherwise.

4.1.10 exp(−λ − *c*)

To the best of my knowledge, I am not aware of any article or paper by others that presents this particular Bernoulli factory. In this algorithm, c is an integer that is 0 or greater.

1. Run the **algorithm for exp(−*c*/1)** described later in this document. Return 0 if the algorithm returns 0.
2. Return the result of the **algorithm for exp(−λ)**.

4.1.11 1/(2$^{k + \lambda}$) or exp(−(*k* + λ)*ln(2))

This new algorithm uses the base-2 logarithm $k + \lambda$, where k is an integer 0 or greater, and is useful when this logarithm is very large.

1. If $k > 0$, generate unbiased random bits until a zero bit or k bits were generated this way, whichever comes first. If a zero bit was generated this way, return 0.
2. Create an input coin μ that does the following: "Flip the input coin, then run the **algorithm for ln(2)** (given later). If both the call and the flip return 1, return 1. Otherwise, return 0."
3. Run the **algorithm for exp(−μ)** using the μ input coin, and return the result.

4.1.12 1/(2$^{m*(k + \lambda)}$) or 1/((2^{m})*(*k* + λ)) or exp(−(*k* + λ)*ln(2^{m}))

An extension of the previous algorithm. Here, m is an integer greater than 0.

1. If $k > 0$, generate unbiased random bits until a zero bit or $k*m$ bits were generated this way, whichever comes first. If a zero bit was generated this way, return 0.
2. Create an input coin μ that does the following: "Flip the input coin, then run the **algorithm for ln(2)** (given later). If both the call and the flip return 1, return 1. Otherwise, return 0."
3. Run the **algorithm for exp(−μ)** m times, using the μ input coin. If any of the calls returns 0, return 0. Otherwise, return 1.

4.1.13 1/(1+λ)

This algorithm is a special case of the two-coin Bernoulli factory of (Gonçalves et al., 2017)[12] and is uniformly fast. It will be called the **two-coin special case** in this document.[13]

1. With probability 1/2, return 1. (For example, generate either 0 or 1 with equal probability, that is, an unbiased random bit, and return 1 if that bit is 1.)
2. Flip the input coin. If it returns 1, return 0. Otherwise, go to step 1.

4.1.14 λ/(1+λ)

Return 1 minus the result of the **algorithm for 1/(1+λ)**.

4.1.15 ln(1+λ)

(Flajolet et al., 2010)[1]:

1. Generate a uniform(0, 1) random number u.
2. Flip the input coin. If it returns 0, flip the coin again and return the result.
3. **Sample from the number u**. If the result is 0, flip the input coin and return the result.
4. Flip the input coin. If it returns 0, return 0.
5. **Sample from the number u**. If the result is 0, return 0. Otherwise, go to step 2.

Observing that the even-parity construction used in the Flajolet paper is equivalent to the two-coin special case, which is uniformly fast for all λ parameters, the algorithm above can be made uniformly fast as follows:

1. With probability 1/2, flip the input coin and return the result.
2. Generate a uniform(0, 1) random number u, if u wasn't generated yet.
3. **Sample from the number u**, then flip the input coin. If the call and the flip both return 1, return 0. Otherwise, go to step 1.

4.1.16 1 − ln(1+λ)

Invert the result of the algorithm for ln(1+λ) (make it 1 if it's 0 and vice versa).[14]

4.1.17 c * λ * β / (β * (c * λ + d * μ) − (β − 1) * (c + d))

This is the general two-coin algorithm of (Gonçalves et al., 2017)[12] and (Vats et al. 2020)[15]. It takes two input coins that each output heads (1) with probability λ or μ, respectively. It also takes a parameter β in the interval [0, 1], which is a so-called "portkey" or early rejection parameter (when $\beta = 1$, the formula simplifies to $c * \lambda / (c * \lambda + d * \mu)$).

1. With probability β, go to step 2. Otherwise, return 0. (For example, call `ZeroOrOne` with β's numerator and denominator, and return 0 if that call returns 0, or go to step 2 otherwise. `ZeroOrOne` is described in my article on **random sampling methods**.)
2. With probability $c / (c + d)$, flip the λ input coin. Otherwise, flip the μ input coin. If the λ input coin returns 1, return 1. If the μ input coin returns 1, return 0. If the corresponding coin returns 0, go to step 1.

4.1.18 c * λ / (c * λ + d) or (c/d) * λ / (1 + (c/d) * λ))

This algorithm, also known as the **logistic Bernoulli factory** (Huber 2016)[16], (Morina

et al., 2019)[17], is a special case of the two-coin algorithm above, but this time uses only one input coin.

1. With probability d / (c + d), return 0.
2. Flip the input coin. If the flip returns 1, return 1. Otherwise, go to step 1.

(Note that Huber [2016] specifies this Bernoulli factory in terms of a Poisson point process, which seems to require much more randomness on average.)

4.1.19 1 / (*c* + *λ*)

In this algorithm, c must be 1 or greater. For example, this algorithm can simulate a probability of the form 1 / z, where z is greater than 0 and made up of an integer part (c) and a fractional part (λ) that can be simulated by a Bernoulli factory. See also the algorithms for continued fractions.

1. With probability c / (1 + c), return a number that is 1 with probability 1/c and 0 otherwise.
2. Flip the input coin. If the flip returns 1, return 0. Otherwise, go to step 1.

4.1.20 (*d* + *λ*) / *c*

This algorithm currently works only if d and c are integers and 0 <= d < c.

1. Generate an integer in [0, c) uniformly at random, call it i.
2. If i < d, return 1. If i = d, flip the input coin and return the result. If neither is the case, go to step 1.

4.1.21 *d* / (*c* + *λ*)

In this algorithm, c must be 1 or greater and d must be in the interval [0, c]. See also the algorithms for continued fractions.

1. With probability c / (1 + c), return a number that is 1 with probability d/c and 0 otherwise.
2. Flip the input coin. If the flip returns 1, return 0. Otherwise, go to step 1.

4.1.22 (*d* + *μ*) / (*c* + *λ*)

Combines the algorithms in the previous two sections. This algorithm currently works only if d and c are integers and 0 <= d < c.

1. With probability c / (1 + c), do the following:
    1. Generate an integer in [0, c) uniformly at random, call it i.
    2. If i < d, return 1. If i = d, flip the μ input coin and return the result. If neither is the case, go to the previous substep.
2. Flip the λ input coin. If the flip returns 1, return 0. Otherwise, go to step 1.

4.1.23 d^k / (*c* + *λ*)^k, or (*d* / (*c* + *λ*))^k

In this algorithm, c must be 1 or greater, d must be in the interval [0, c], and k must be an integer 0 or greater.

1. Set i to 0.
2. If k is 0, return 1.

3. With probability c / (1 + c), do the following:
    1. With probability d/c, add 1 to i and then either return 1 if i is now k or greater, or abort these substeps and go to step 2 otherwise.
    2. Return 0.
4. Flip the input coin. If the flip returns 1, return 0. Otherwise, go to step 2.

4.1.24 λ + μ

(Nacu and Peres 2005, proposition 14(iii))**(5)**. This algorithm takes two input coins that simulate λ or μ, respectively, and a parameter ϵ, which must be greater than 0 and chosen such that $\lambda + \mu < 1 - \epsilon$.

1. Create a ν input coin that does the following: "With probability 1/2, flip the λ input coin and return the result. Otherwise, flip the μ input coin and return the result."
2. Call the **2014 algorithm**, the **2016 algorithm**, or the **2019 algorithm**, described later, using the ν input coin, x/y = 2/1, i = 1 (for the 2019 algorithm), and $\epsilon = \epsilon$, and return the result.

4.1.25 $\lambda - \mu$

(Nacu and Peres 2005, proposition 14(iii-iv))**(5)**. This algorithm takes two input coins that simulate λ or μ, respectively, and a parameter ϵ, which must be greater than 0 and chosen such that $\lambda - \mu > \epsilon$ (and should be chosen such that ϵ is slightly less than $\lambda - \mu$).

1. Create a ν input coin that does the following: "With probability 1/2, flip the λ input coin and return **1 minus the result**. Otherwise, flip the μ input coin and return the result."
2. Call the **2014 algorithm**, the **2016 algorithm**, or the **2019 algorithm**, described later, using the ν input coin, x/y = 2/1, i = 1 (for the 2019 algorithm), and $\epsilon = \epsilon$, and return 1 minus the result.

4.1.26 1/(c + λ)

Works only if $c > 0$.

1. With probability c/(1 + c), return a number that is 1 with probability 1/c and 0 otherwise.
2. Flip the input coin. If the flip returns 1, return 0. Otherwise, go to step 1.

4.1.27 1 − λ

(Flajolet et al., 2010)**(1)**: Flip the λ input coin and return 0 if the result is 1, or 1 otherwise.

4.1.28 ν * λ + (1 − ν) * μ

(Flajolet et al., 2010)**(1)**: Flip the ν input coin. If the result is 0, flip the λ input coin and return the result. Otherwise, flip the μ input coin and return the result.

4.1.29 λ + μ − (λ * μ)

(Flajolet et al., 2010)**(1)**: Flip the λ input coin and the μ input coin. Return 1 if either flip returns 1, and 0 otherwise.

4.1.30 (λ + μ) / 2

(Nacu and Peres 2005, proposition 14(iii))[5]; (Flajolet et al., 2010)[1]: With probability 1/2, flip the λ input coin and return the result. Otherwise, flip the μ input coin and return the result.

4.1.31 arctan(λ) /λ

(Flajolet et al., 2010)[1]:

1. Generate a uniform(0, 1) random number u.
2. **Sample from the number u** twice, and flip the input coin twice. If any of these calls or flips returns 0, return 1.
3. **Sample from the number u** twice, and flip the input coin twice. If any of these calls or flips returns 0, return 0. Otherwise, go to step 2.

Observing that the even-parity construction used in the Flajolet paper is equivalent to the two-coin special case, which is uniformly fast for all λ parameters, the algorithm above can be made uniformly fast as follows:

1. With probability 1/2, return 1.
2. Generate a uniform(0, 1) random number u, if it wasn't generated yet.
3. **Sample from the number u** twice, and flip the input coin twice. If all of these calls and flips return 1, return 0. Otherwise, go to step 1.

4.1.32 arctan(λ)

(Flajolet et al., 2010)[1]: Call the **algorithm for arctan(λ) /λ** and flip the input coin. Return 1 if the call and flip both return 1, or 0 otherwise.

4.1.33 cos(λ)

This algorithm adapts the general martingale algorithm for this function's series expansion. In fact, this is a special case of Algorithm 3 of (Łatuszyński et al. 2009/2011) [8] (which is more general than Proposition 3.4, the general martingale algorithm). The series expansion for cos(λ) is $1 - \lambda^2/(2!) + \lambda^4/(4!) - \ldots$, which is an alternating series except the exponent is increased by 2 (rather than 1) with each term. The coefficients are thus 1, 1/(2!), 1/(4!), ... A *lower truncation* of the series is a truncation of that series that ends with a minus term, and the corresponding *upper truncation* is the same truncation but without the last minus term. This series expansion meets the requirements of Algorithm 3 because—

- the lower truncation is less than or equal to its corresponding upper truncation almost surely,
- the lower and upper truncations are in the interval [0, 1],
- each lower truncation is greater than or equal to the previous lower truncation almost surely,
- each upper truncation is less than or equal to the previous upper truncation almost surely, and
- the lower and upper truncations have an expected value that approaches λ from below and above.

The algorithm to simulate cos(λ) follows.

1. Set u to 1, set w to 1, set ℓ to 0, set n to 1, and set *fac* to 2.

2. Generate a uniform(0, 1) random number *ret*.
3. If *w* is not 0, flip the input coin. If the flip returns 0, set *w* to 0. Do this step again. (Note that in the general martingale algorithm, only one coin is flipped in this step. Up to two coins are flipped instead because the exponent increases by 2 rather than 1.)
4. If *n* is even, set *u* to $\ell + w / fac$. Otherwise, set ℓ to $u - w / fac$. (Here we divide by the factorial of 2-times-*n*.)
5. If *ret* is less than (or equal to) ℓ, return 1. If *ret* is less than *u*, go to the next step. If neither is the case, return 0. (If *ret* is a uniform PSRN, these comparisons should be done via the **URandLessThanReal algorithm**, which is described in my **article on PSRNs**.)
6. Add 1 to *n*, then multiply *fac* by $(n * 2 - 1) * (n * 2)$, then go to step 3.

4.1.34 sin(λ)

This algorithm is likewise a special case of Algorithm 3 of (Łatuszyński et al. 2009/2011)[8]. $\sin(\lambda)$ can be rewritten as $\lambda * (1 - \lambda^2/(3!) + \lambda^4/(5!) - ...)$, which includes an alternating series where the exponent is increased by 2 (rather than 1) with each term. The coefficients are thus 1, 1/(3!), 1/(5!), ... This series expansion meets the requirements of Algorithm 3 for the same reasons as the cos(λ) series does.

The algorithm to simulate $\sin(\lambda)$ follows.

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

4.1.35 $\lambda^{x/y}$

In the algorithm below, the case where *x*/*y* is in (0, 1) is due to recent work by Mendo (2019)[7]. The algorithm works only when *x*/*y* is 0 or greater.

1. If *x*/*y* is 0, return 1.
2. If *x*/*y* is equal to 1, flip the input coin and return the result.
3. If *x*/*y* is greater than 1:
    1. Set *ipart* to floor(*x*/*y*) and *fpart* to `rem(x, y)`.
    2. If *fpart* is greater than 0, subtract 1 from *ipart*, then call this algorithm recursively with *x* = floor(*fpart*/2) and *y* = *y*, then call this algorithm, again recursively, with *x* = *fpart* − floor(*fpart*/2) and *y* = *y*. Return 0 if either call returns 0. (This is done rather than the more obvious approach in order to avoid calling this algorithm with fractional parts very close to 0, because the algorithm runs much more slowly than for fractional parts closer to 1.)
    3. If *ipart* is 1 or greater, flip the input coin *ipart* many times. Return 0 if any of these flips returns 1.
    4. Return 1.
4. *x*/*y* is less than 1, so set *i* to 1.
5. Flip the input coin; if it returns 1, return 1.
6. With probability *x*/(*y***i*), return 0.

7. Add 1 to i and go to step 5.

   Note: When x/y is less than 1, the minimum number of coin flips needed, on average, by this algorithm will grow without bound as λ approaches 0. In fact, no fast Bernoulli factory algorithm can avoid this unbounded growth without additional information on λ (Mendo 2019)[7]. See also the appendix, which also shows an alternative way to implement this and other Bernoulli factory algorithms using partially-sampled random numbers (PSRNs), which exploits knowledge of λ but is not the focus of this article since it involves arithmetic.

4.1.36 λ^{μ}

This algorithm is based on the previous one, but changed to accept a second input coin (which outputs heads with probability μ) rather than a fixed value for the exponent. To the best of my knowledge, I am not aware of any article or paper by others that presents this particular Bernoulli factory.

1. Set i to 1.
2. Flip the input coin that simulates the base, λ; if it returns 1, return 1.
3. Flip the input coin that simulates the exponent, μ; if it returns 1, return 0 with probability $1/i$.
4. Add 1 to i and go to step 1.

4.1.37 sqrt(λ)

Use the algorithm for $\lambda^{1/2}$.

4.1.38 arcsin(λ) + sqrt($1 - \lambda^2$) − 1

(Flajolet et al., 2010)[1]. The algorithm given here uses the special two-coin case rather than the even-parity construction.

1. Generate a uniform(0, 1) random number u.
2. Create a secondary coin μ that does the following: "**Sample from the number u** twice, and flip the input coin twice. If all of these calls and flips return 1, return 0. Otherwise, return 1."
3. Call the **algorithm for $\mu^{1/2}$** using the secondary coin μ. If it returns 0, return 0.
4. With probability 1/2, flip the input coin and return the result.
5. **Sample from the number u** once, and flip the input coin once. If both the call and flip return 1, return 0. Otherwise, go to step 4.

4.1.39 arcsin(λ) / 2

The Flajolet paper doesn't explain in detail how arcsin(λ)/2 arises out of arcsin(λ) + sqrt($1 - \lambda^2$) − 1 via Bernoulli factory constructions, but here is an algorithm.[18] However, the number of input coin flips is expected to grow without bound as λ approaches 1.

1. With probability 1/2, run the **algorithm for arcsin(λ) + sqrt($1 - \lambda^2$) − 1** and return the result.
2. Create a secondary coin μ that does the following: "Flip the input coin twice. If both flips return 1, return 0. Otherwise, return 1." (The coin simulates $1 - \lambda^2$.)
3. Call the **algorithm for $\mu^{1/2}$** using the secondary coin μ. If it returns 0, return 1;

otherwise, return 0. (This step effectively cancels out the sqrt($1 - \lambda^2$) $-$ 1 part and divides by 2.)

4.1.40 λ * μ

(Flajolet et al., 2010)[1]: Flip the λ input coin and the μ input coin. Return 1 if both flips return 1, and 0 otherwise.

4.1.41 λ * x/y (linear Bernoulli factories)

In general, this function will touch 0 or 1 somewhere in [0, 1], when $x/y > 1$. This makes the function relatively non-trivial to simulate in this case.

Huber has suggested several algorithms for this function over the years.

The first algorithm is called the **2014 algorithm** in this document (Huber 2014)[4]. It uses three parameters: x, y, and ϵ, such that $x/y > 0$ and ϵ is greater than 0. When x/y is greater than 1, the ϵ parameter has to be chosen such that $\lambda * x/y < 1 - \epsilon$, in order to bound the function away from 0 and 1. As a result, some knowledge of λ has to be available to the algorithm. (In fact, as simulation results show, the choice of ϵ is crucial to this algorithm's performance; for best results, ϵ should be chosen such that $\lambda * x/y$ is slightly less than $1 - \epsilon$.) The algorithm as described below also includes certain special cases, not mentioned in Huber, to make it more general.

1. Special cases: If x is 0, return 0. Otherwise, if x equals y, flip the input coin and return the result. Otherwise, if x is less than y, then: (a) With probability x/y, flip the input coin and return the result; otherwise (b) return 0.
2. Set c to x/y, and set k to 23 / (5 * ϵ).
3. If ϵ is greater than 644/1000, set ϵ to 644/1000.
4. Set i to 1.
5. Flip the input coin. If it returns 0, then generate numbers that are each 1 with probability ($c - 1$) / c and 0 otherwise, until 0 is generated this way, then add 1 to i for each number generated this way (including the last).
6. Subtract 1 from i, then if i is 0, return 1.
7. If i is less than k, go to step 5.
8. If i is k or greater:
    1. Generate i numbers that are each 1 with probability 2 / (ϵ + 2) or 0 otherwise. If any of those numbers is 0, return 0.
    2. Multiply c by 2 / (ϵ + 2), divide ϵ by 2, and multiply k by 2.
9. If i is 0, return 1. Otherwise, go to step 5.

The second algorithm is called the **2016 algorithm** (Huber 2016)[16] and uses the same parameters x, y, and ϵ, and its description uses the same special cases. The difference here is that it involves a so-called "logistic Bernoulli factory", which is replaced in this document with a different one that simulates the same function. When x/y is greater than 1, the ϵ parameter has to be chosen such that $\lambda * x/y <= 1 - \epsilon$.

1. The same special cases as for the 2014 algorithm apply.
2. Set m to ceil(1 + 9 / (2 * ϵ)).
3. Set β to 1 + 1 / ($m - 1$).
4. **Algorithm A** is what Huber calls this step. Set s to 1, then while s is greater than 0 and less than m:
    1. Run the **logistic Bernoulli factory** algorithm with $c/d = \beta * x/y$.
    2. Set s to $s - z * 2 + 1$, where z is the result of the logistic Bernoulli factory.
5. If s is other than 0, return 0.

6. With probability $1/\beta$, return 1.
7. Run this algorithm recursively, with $x/y = \beta * x/y$ and $\epsilon = 1 - \beta * (1 - \epsilon)$. If it returns 0, return 0.
8. The **high-power logistic Bernoulli factory** is what Huber calls this step. Set s to 1, then while s is greater than 0 and less than or equal to m minus 2:
    1. Run the **logistic Bernoulli factory** algorithm with $c/d = \beta * x/y$.
    2. Set s to $s + z * 2 - 1$, where z is the result of the logistic Bernoulli factory.
9. If s is equal to m minus 1, return 1.
10. Subtract 1 from m and go to step 7.

The paper that presented the 2016 algorithm also included a third algorithm, described below, that works only if $\lambda * x / y$ is known to be less than 1/2. This third algorithm takes three parameters: x, y, and m, and m has to be chosen such that $\lambda * x / y <= m < 1/2$.

1. The same special cases as for the 2014 algorithm apply.
2. Run the **logistic Bernoulli factory** algorithm with $c/d = (x/y) / (1 - 2 * m)$. If it returns 0, return 0.
3. With probability $1 - 2 * m$, return 1.
4. Run the 2014 algorithm or 2016 algorithm with $x/y = (x/y) / (2 * m)$ and $\epsilon = 1 - m$.

4.1.42 $(\lambda * x/y)^i$

(Huber 2019)[19]. This algorithm, called the **2019 algorithm** in this document, uses four parameters: x, y, i, and ϵ, such that $x/y > 0$, $i >= 0$ is an integer, and ϵ is greater than 0. When x/y is greater than 1, the ϵ parameter has to be chosen such that $\lambda * x/y < 1 - \epsilon$. It also has special cases not mentioned in Huber 2019.

1. Special cases: If i is 0, return 1. If x is 0, return 0. Otherwise, if x equals y and i equals 1, flip the input coin and return the result.
2. Special case: If x is less than y and $i = 1$, then: (a) With probability x/y, flip the input coin and return the result; otherwise (b) return 0.
3. Special case: If x is less than y, then create a secondary coin μ that does the following: "(a) With probability x/y, flip the input coin and return the result; otherwise (b) return 0", then run the **algorithm for $(\mu^{i/1})$** (described earlier) using this secondary coin.
4. Set t to 355/100 and c to x/y.
5. If i is 0, return 1.
6. While $i = t / \epsilon$:
    1. Set β to $(1 - \epsilon / 2) / (1 - \epsilon)$.
    2. Run the **algorithm for $(1/\beta)^i$** (the algorithm labeled x^y and given in the irrational constants section). If it returns 0, return 0.
    3. Multiply c by β, then divide ϵ by 2.
7. Run the **logistic Bernoulli factory** with $c/d = c$, then set z to the result. Set i to $i + 1 - z * 2$, then go to step 5.

4.1.43 ϵ / λ

(Lee et al. 2014)[20] This algorithm, in addition to the input coin, takes a parameter ϵ, which must be greater than 0 and be chosen such that ϵ is less than λ.

1. Set β to $\max(\epsilon, 1/2)$ and set γ to $1 - (1 - \beta) / (1 - (\beta / 2))$.
2. Create a μ input coin that flips the input coin and returns 1 minus the result.
3. With probability ϵ, return 1.
4. Run the **2014 algorithm**, **2016 algorithm**, or **2019 algorithm**, with the μ input coin, $x/y = 1 / (1 - \epsilon)$, $i = 1$ (for the 2019 algorithm), and $\epsilon = \gamma$. If the result is 0,

return 0. Otherwise, go to step 3. (Note that running the algorithm this way simulates the probability $(\lambda - \epsilon)/(1 - \epsilon)$ or $1 - (1 - \lambda)/(1 - \epsilon)$).

4.1.44 Certain Rational Functions

According to (Mossel and Peres 2005)[21], a function can be simulated by a finite-state machine (equivalently, a "probabilistic regular grammar" (Smith and Johnson 2007)[22], (Icard 2019)[23]) if and only if the function can be written as a rational function (ratio of polynomials) with rational coefficients, that takes in an input λ in some subset of (0, 1) and outputs a number in the interval (0, 1).

The following algorithm is suggested from the Mossel and Peres paper and from (Thomas and Blanchet 2012)[24]. It assumes the rational function is of the form $D(\lambda)/E(\lambda)$, where—

- $D(\lambda) = \Sigma_{i = 0, ..., n}\ \lambda^i * (1 - \lambda)^{n - i} * d[i]$,
- $E(\lambda) = \Sigma_{i = 0, ..., n}\ \lambda^i * (1 - \lambda)^{n - i} * e[i]$,
- every $d[i]$ is less than or equal to the corresponding $e[i]$, and
- each $d[i]$ and each $e[i]$ is an integer or rational number in the interval [0, choose(n, i)], where the upper bound is the total number of n-bit words with i ones.

Here, $d[i]$ is akin to the number of "passing" n-bit words with i ones, and $e[i]$ is akin to that number plus the number of "failing" n-bit words with i ones.

The algorithm follows.

1. Flip the input coin n times, and let j be the number of times the coin returned 1 this way.
2. Call **WeightedChoice**(**NormalizeRatios**([$e[j] - d[j]$, $d[j]$, choose(n, j) $- e[j]$])), where **WeightedChoice** and **NormalizeRatios** are given in "**Randomization and Sampling Methods**". If the call returns 0 or 1, return that result. Otherwise, go to step 1.

   Notes:

   1. In the formulas above—

      - $d[i]$ can be replaced with $\delta[i]$ * choose(n,i), where $\delta[i]$ is a rational number in the interval [0, 1] (and thus expresses the probability that a given word is a "passing" word among all n-bit words with i ones), and
      - $e[i]$ can be replaced with $\eta[i]$ * choose(n,i), where $\eta[i]$ is a rational number in the interval [0, 1] (and thus expresses the probability that a given word is a "passing" or "failing" word among all n-bit words with i ones),

      and then $\delta[i]$ and $\eta[i]$ can be seen as control points for two different 1-dimensional **Bézier curves**, where the δ curve is always on or "below" the η curve. For each curve, λ is the relative position on that curve, the curve begins at $\delta[0]$ or $\eta[0]$, and the curve ends at $\delta[n]$ or $\eta[n]$. See also the next section.

   2. This algorithm could be modified to avoid additional randomness besides the input coin flips by packing the coin flips into an n-bit word and looking up whether that word is "passing", "failing", or neither, among all n-bit words with j ones, but this is not so trivial to do (especially because in

general, a lookup table first has to be built in a setup step, which can be impractical unless 2^n is relatively small). Moreover, this approach works only if $d[i]$ and $e[i]$ are integers (or if $d[i]$ is replaced with floor($d[i]$) and $e[i]$ with ceil($e[i]$) (Holtz et al. 2011)[25], but this, of course, suffers from rounding error when done in this algorithm). See also (Thomas and Blanchet 2012)[24].

4.1.45 Certain Polynomials in Bernstein Form

A polynomial can be written in *Bernstein form* as $\Sigma_{i = 0, ..., n}$ choose(n, i) * λ^i * $(1 - \lambda)^{n - i}$ * $a[i]$, where n is the polynomial's degree and $a[i]$ are its n plus one *Bernstein coefficients*. According to (Goyal and Sigman 2012)[26], a function can be simulated with a fixed number of input coin flips if and only if the function is a polynomial that has Bernstein coefficients all in the interval [0, 1] (see also (Wästlund 1999, section 4)[27]; (Qian and Riedel 2008)[28]). They also give an algorithm for simulating these polynomials, which is given below.

1. Flip the input coin n times, and let j be the number of times the coin returned 1 this way.
2. With probability $a[j]$, return 1. Otherwise, return 0.

Notes:

1. Each $a[i]$ acts as a control point for a 1-dimensional **Bézier curve**, where λ is the relative position on that curve, the curve begins at $a[0]$, and the curve ends at $a[n]$. For example, given control points 0.2, 0.3, and 0.6, the curve is at 0.2 when $\lambda = 0$, and 0.6 when $\lambda = 1$. (The curve, however, is not at 0.3 when $\lambda = 1/2$; in general, Bézier curves do not cross their control points other than the first and the last.)
2. The problem of simulating polynomials in Bernstein form is related to *stochastic logic*, which involves simulating probabilities that arise out of Boolean functions (functions that use only AND, OR, NOT, and XOR operations) that take a fixed number of bits as input, where each bit has a separate probability of being 1 rather than 0, and output a single bit (for further discussion see (Qian et al. 2011)[29]).
3. This algorithm can serve as an approximate way to simulate any factory function f. In this case, $a[j]$ is calculated as $f(j/n)$, so that the resulting polynomial closely approximates the function; the higher n is, the better this approximation.

Example: Take the following parabolic function discussed in (Thomas and Blanchet 2012)[24]: $(1-4*(\lambda-1/2)^2)*c$, where c is in the interval (0, 1). This is a polynomial that can be rewritten as $-4*c*\lambda^2+4*c*\lambda$, so that this *power form* has coefficients (0, $4*c$, $-4*c$) and a degree (n) of 2. Using the matrix method by Ray and Nataraj (2012)[30], we get Bernstein coefficients (0, $2*c$, 0). Thus, for this polynomial, $a[0]$ is 0, $a[1]$ is $2*c$, and $a[2]$ is 0. Thus, if c is in the interval (0, 1/2], we can simulate this function as follows: "Flip the input coin twice. If exactly one of the flips returns 1, return a number that is 1 with probability $2*c$ and 0 otherwise. Otherwise, return 0." For other values of c, the algorithm requires computing the Bernstein coefficients, then elevating the polynomial's degree enough times so that those Bernstein coefficients all lie in [0, 1]; the required degree approaches infinity as c approaches 1.[31]

Niazadeh et al. (2020)[32] describes monomials (involving one or more coins) of the form $\Pi_{i = 1, ..., n} \lambda[i]^{a[i]} * (1-\lambda[i])^{b[i]}$, where there are n coins, $\lambda[i]$ is the probability of heads of coin i, and $a[i] >= 0$ and $b[i] >= 0$ are parameters for coin i (specifically, of $a+b$ flips, the first a flips must return heads and the rest must return tails to succeed).

1. For each i in [1, n]:
    1. Flip the $\lambda[i]$ input coin $a[i]$ times. If any of the flips returns 0, return 0.
    2. Flip the $\lambda[i]$ input coin $b[i]$ times. If any of the flips returns 1, return 0.
2. Return 1.

The same paper also describes polynomials that are weighted sums of this kind of monomials, namely polynomials of the form $P = \Sigma_{j = 1, ..., k} c[j]*M[j](\boldsymbol{\lambda})$, where there are k monomials, $M[j](.)$ identifies monomial j, $\boldsymbol{\lambda}$ identifies the coins' probabilities of heads, and $c[j] >= 0$ is the weight for monomial j. (If there is only one coin, these polynomials are in Bernstein form if $c[j]$ is $\alpha[j]$*choose($k-1, j-1$) where $\alpha[j]$ is a Bernstein coefficient in the interval [0, 1], and if $a[1] = j-1$ and $b[1] = k-j$ for each monomial j.)

Let C be the sum of all $c[j]$. To simulate the probability P/C, choose one of the monomials with probability proportional to its weight (see "**A Note on Weighted Choice Algorithms**"), then run the algorithm above on that monomial (see also "**Convex Combinations**", later).

4.1.46 Certain Algebraic Functions

(Flajolet et al., 2010)[1] showed how certain functions can be simulated by generating a bitstring and determining whether that bitstring belongs to a certain class of bitstrings. The rules for determining whether a bitstring belongs to that class are called a *binary stochastic grammar*, which uses an alphabet of only two "letters", or more generally a *stochastic grammar*. The functions belong to a class called *algebraic functions* (functions that can be a solution of a polynomial system).

According to (Mossel and Peres 2005)[21], a factory function can be simulated by a pushdown automaton only if that function can be a solution of a polynomial system with rational coefficients.[33]

The following algorithm simulates the following algebraic function:

- $\Sigma_{k = 0, 1, 2, ...} (\lambda^k * (1 - \lambda) * W(k) / \beta^k)$, or alternatively,
- $(1 - \lambda) * OGF(\lambda/\beta)$,

where—

- $W(k)$ is a number in the interval [0, β^k] and is the number of k-letter words that can be produced by the stochastic grammar in question,
- β is the alphabet size, or the number of "letters" in the alphabet (e.g., 2 for the cases discussed in the Flajolet paper), and is an integer 2 or greater,
- the *ordinary generating function* $OGF(x) = W(0) + W(1) * x + W(2) * x^2 + W(3) * x^3 + ...$, and
- the second formula incorporates a correction to Theorem 3.2 of the paper[34].

(Here, the k^{th} coefficient of $OGF(x)$ corresponds to $W(k)$.) The algorithm follows.

1. Set g to 0.
2. With probability λ, add 1 to g and repeat this step. Otherwise, go to step 3.

3. Return a number that is 1 with probability W(g)/β^g, and 0 otherwise. (In the Flajolet paper, this is done by generating a g-letter word uniformly at random and "parsing" that word using a binary stochastic grammar to determine whether that word can be produced by that grammar. Note that this determination can be made this way as each of the word's "letters" is generated.)

An extension to this algorithm, not mentioned in the Flajolet paper, is the use of stochastic grammars with a bigger alphabet than two "letters". For example, in the case of *ternary stochastic grammars*, the alphabet size is 3 and β is 3 in the algorithm above. In general, for *β-ary stochastic grammars*, the alphabet size is β, which can be any integer 2 or greater.

Examples:

1. The following is an example from the Flajolet paper. A g-letter binary word can be "parsed" as follows to determine whether that word encodes a ternary tree: "3. If g is 0, return 0. Otherwise, set i to 1 and d to 1.; 3a. Generate an unbiased random bit (that is, either 0 or 1, chosen with equal probability), then subtract 1 from d if that bit is 0, or add 2 to d otherwise.; 3b. Add 1 to i. Then, if $i < g$ and $d > 0$, go to step 3a.; 3c. Return 1 if d is 0 and i is g, or 0 otherwise."

2. If W(g), the number of g-letter words that can be produced by the stochastic grammar in question, has the form—

   - choose(g, g/t) * $(\beta-1)^{g-g/t}$ (the number of g-letter words with exactly g/t A's, for an alphabet size of β) if g is divisible by t[(35)], and
   - 0 otherwise,

   where t is an integer 2 or greater and β is the alphabet size and is an integer 2 or greater, step 3 of the algorithm can be done as follows: "3. If g is not divisible by t, return 0. Otherwise, generate g uniform random integers in the interval 0, β) (e.g., g unbiased random bits if β is 2), then return 1 if exactly g/t zeros were generated this way, or 0 otherwise." If β = 2, then this reproduces another example from the Flajolet paper.

3. If W(g) has the form—
   choose(g * α, g) * $(\beta-1)^{g*\alpha-g}$ / $\beta^{g*\alpha-g}$,
   where α is an integer 1 or greater and β is the alphabet size and is an integer 2 or greater, step 3 of the algorithm can be done as follows: "3. Generate g * α uniform random integers in the interval [0, β) (e.g., g * α unbiased random bits if β is 2), then return 1 if exactly g zeros were generated this way, or 0 otherwise." If α = 2 and β = 2, then this expresses the *square-root construction* sqrt(1 − λ), mentioned in the Flajolet paper. If α is 1, the modified algorithm simulates the following probability: (β* ($\lambda-1$))/($\lambda-\beta$). And interestingly, I have found that if α is 2 or greater, the probability simplifies to involve a hypergeometric function. Specifically, the probability becomes—

   - $(1 - \lambda) * {}_{\alpha-1}F_{\alpha-2}(1/\alpha, 2/\alpha, ..., (\alpha-1)/\alpha; 1/(\alpha-1), ..., (\alpha-2)/(\alpha-1); \lambda * \alpha^\alpha/((\alpha-1)^{\alpha-1} * 2^\alpha))$ if $\beta = 2$, or more generally,
   - $(1 - \lambda) * {}_{\alpha-1}F_{\alpha-2}(1/\alpha, 2/\alpha, ..., (\alpha-1)/\alpha; 1/(\alpha-1), ..., (\alpha-2)/(\alpha-1); \lambda*\alpha^\alpha* (\beta-1)^{\alpha-1}/((\alpha-1)^{\alpha-1} * \beta^\alpha))$.

   The ordinary generating function for this modified algorithm is thus—

OGF(z) = ${}_{\alpha-1}F_{\alpha-2}(1/\alpha, 2/\alpha, ..., (\alpha-1)/\alpha; 1/(\alpha-1), ..., (\alpha-2)/(\alpha-1); z*\alpha^{\alpha}*(\beta-1)^{\alpha-1}/((\alpha-1)^{\alpha-1} * \beta^{\alpha-1}))$.

4. The probability involved in example 2 likewise involves hypergeometric functions:
    - $(1 - \lambda) * {}_{t-1}F_{t-2}(1/t, 2/t, ..., (t-1)/t; 1/(t-1), ..., (t-2)/(t-1); \lambda^t*t^t*(\beta-1)^{t-1}/((t-1)^{t-1} * \beta^t))$.

4.1.47 Expressions Involving Polylogarithms

The following algorithm simulates the expression $\mathrm{Li}_r(\lambda) * (1 / \lambda - 1)$, where r is an integer 1 or greater. If λ is 1/2, this expression simplifies to $\mathrm{Li}_r(1/2)$. See also (Flajolet et al., 2010)[**(1)**]. However, even with a relatively small r such as 6, the expression quickly approaches a straight line.

1. Flip the input coin until it returns 0, and let t be 1 plus the number of times the coin returned 1 this way.
2. Return a number that is 1 with probability $1/t^r$ and 0 otherwise.

4.2 Algorithms for Irrational Constants

The following algorithms generate heads with a probability equal to an irrational number. (On the other hand, probabilities that are *rational* constants are trivial to simulate. If fair coins are available, the `ZeroOrOne` method, which is described in my article on **random sampling methods**, should be used. If coins with unknown bias are available, then a ***randomness extraction*** method should be used to turn them into fair coins.)

4.2.1 Digit Expansions

Probabilities can be expressed as a digit expansion (of the form `0.dddddd...`). The following algorithm returns 1 with probability `p` and 0 otherwise, where `p` is a probability in the interval 0, 1). Note that the number 0 is also an infinite digit expansion of zeros, and the number 1 is also an infinite digit expansion of base-minus-ones. Irrational numbers always have infinite digit expansions, which must be calculated "on-the-fly".

In the algorithm (see also (Brassard et al., 2019)[**(36)**, (Devroye 1986, p. 769)[**(9)**]), `BASE` is the digit base, such as 2 for binary or 10 for decimal.

1. Set `u` to 0 and `k` to 1.
2. Set `u` to `(u * BASE) + v`, where `v` is a random integer in the interval [0, `BASE`) (such as `RNDINTEXC(BASE)`, or simply an unbiased random bit if `BASE` is 2). Calculate `pa`, which is an approximation to `p` such that abs(`p`−`pa`) <= $\mathtt{BASE}^{-\mathtt{k}}$. Set `pk` to `pa`'s digit expansion up to the `k` digits after the point. Example: If `p` is $\pi/4$, `BASE` is 10, and `k` is 5, then `pk` = `78539`.
3. If `pk + 1 <= u`, return 0. If `pk - 2 >= u`, return 1. If neither is the case, add 1 to `k` and go to step 2.

4.2.2 Continued Fractions

The following algorithm simulates a probability expressed as a simple continued fraction of the following form: 0 + 1 / (a[1] + 1 / (a[2] + 1 / (a[3] + ...))). The a[i] are the *partial denominators*, none of which may have an absolute value less than 1. Inspired by (Flajolet

et al., 2010, "Finite graphs (Markov chains) and rational functions")[1], I developed the following algorithm.

Algorithm 1. This algorithm works only if each $a[i]$'s absolute value is 1 or greater and $a[1]$ is positive, but otherwise, each $a[i]$ may be negative and/or a non-integer. The algorithm begins with *pos* equal to 1. Then the following steps are taken.

1. Set k to $a[pos]$.
2. If the partial denominator at *pos* is the last, return a number that is 1 with probability $1/k$ and 0 otherwise.
3. If $a[pos]$ is less than 0, set kp to $k - 1$ and s to 0. Otherwise, set kp to k and s to 1. (This step accounts for negative partial denominators.)
4. With probability $kp/(1+kp)$, return a number that is 1 with probability $1/kp$ and 0 otherwise.
5. Run this algorithm recursively, but with $pos = pos + 1$. If the result is s, return 0. Otherwise, go to step 4.

A *generalized continued fraction* has the form $0 + b[1] / (a[1] + b[2] / (a[2] + b[3] / (a[3] + ...)))$. The $a[i]$ are the same as before, but the $b[i]$ are the *partial numerators*. The following are two algorithms to simulate a probability in the form of a generalized continued fraction.

Algorithm 2. This algorithm works only if each $b[i]/a[i]$ is 1 or less, but otherwise, each $b[i]$ and each $a[i]$ may be negative and/or a non-integer. This algorithm employs an equivalence transform from generalized to simple continued fractions. The algorithm begins with *pos* and r both equal to 1. Then the following steps are taken.

1. Set r to $1 / (r * b[pos])$, then set k to $a[pos] * r$. (k is the partial denominator for the equivalent simple continued fraction.)
2. If the partial numerator/denominator pair at *pos* is the last, return a number that is 1 with probability $1/\text{abs}(k)$ and 0 otherwise.
3. Set kp to $\text{abs}(k)$ and s to 1.
4. Set *r2* to $1 / (r * b[pos + 1])$. If $a[pos + 1] * r2$ is less than 0, set kp to $kp - 1$ and s to 0. (This step accounts for negative partial numerators and denominators.)
5. With probability $kp/(1+kp)$, return a number that is 1 with probability $1/kp$ and 0 otherwise.
6. Run this algorithm recursively, but with $pos = pos + 1$ and $r = r$. If the result is s, return 0. Otherwise, go to step 5.

Algorithm 3. This algorithm works only if each $b[i]/a[i]$ is 1 or less and if each $b[i]$ and each $a[i]$ is greater than 0, but otherwise, each $b[i]$ and each $a[i]$ may be a non-integer. The algorithm begins with *pos* equal to 1. Then the following steps are taken.

1. If the partial numerator/denominator pair at *pos* is the last, return a number that is 1 with probability $b[pos]/a[pos]$ and 0 otherwise.
2. With probability $a[pos]/(1 + a[pos])$, return a number that is 1 with probability $b[pos]/a[pos]$ and 0 otherwise.
3. Run this algorithm recursively, but with $pos = pos + 1$. If the result is 1, return 0. Otherwise, go to step 2.

See the appendix for a correctness proof of Algorithm 3.

Notes:

- If any of these algorithms encounters a probability outside the interval [0, 1], the entire algorithm will fail for that continued fraction.

- These algorithms will work for continued fractions of the form "1 − ..." (rather than "0 + ...") if—
    - before running the algorithm, the first partial numerator and denominator have their sign removed, and
    - after running the algorithm, 1 minus the result (rather than just the result) is taken.
- These algorithms are designed to allow the partial numerators and denominators to be calculated "on the fly".
- The following is an alternative way to write Algorithm 1, which better shows the inspiration because it shows how the "even parity construction" (or the two-coin special case) as well as the "1 − x" construction can be used to develop rational number simulators that are as big as their continued fraction expansions, as suggested in the cited part of the Flajolet paper. However, it only works if the size of the continued fraction expansion (here, *size*) is known in advance.

    1. Set i to *size*.
    2. Create an input coin that does the following: "Return a number that is 1 with probability 1/a[*size*] or 0 otherwise".
    3. While i is 1 or greater:
        1. Set k to $a[i]$.
        2. Create an input coin that takes the previous input coin and k and does the following: "(a) With probability k/(1+k), return a number that is 1 with probability 1/k and 0 otherwise; (b) Flip the previous input coin. If the result is 1, return 0. Otherwise, go to step (a)". (The probability k/(1+k) is related to λ/(1+λ) = 1 − 1/(1+λ), which involves the even-parity construction—or the two-coin special case—for 1/(1+λ) as well as complementation for "1 − x".)
        3. Subtract 1 from i.
    4. Flip the last input coin created by this algorithm, and return the result.

4.2.3 Continued Logarithms

The *continued logarithm* (Gosper 1978)[37], (Borwein et al., 2016)[38] of a number in (0, 1) has the following continued fraction form: 0 + (1 / $2^{c[1]}$) / (1 + (1 / $2^{c[2]}$) / (1 + ...)), where $c[i]$ are the coefficients of the continued logarithm and all 0 or greater. I have come up with the following algorithm that simulates a probability expressed as a continued logarithm expansion.

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

1. If the coefficient at *pos* is the last, return a number that is 1 with probability 1/($2^{c[pos]}$) and 0 otherwise.
2. With probability 1/2, return a number that is 1 with probability 1/($2^{c[pos]}$) and 0 otherwise.
3. Run this algorithm recursively, but with *pos* = *pos* + 1. If the result is 1, return 0. Otherwise, go to step 2.

For a correctness proof, see the appendix.

4.2.4 1 / φ (1 divided by the golden ratio)

This algorithm uses the algorithm described in the section on continued fractions to simulate 1 divided by the golden ratio, whose continued fraction's partial denominators are 1, 1, 1, 1, ...

1. With probability 1/2, return 1.
2. Run this algorithm recursively. If the result is 1, return 0. Otherwise, go to step 1.

4.2.5 sqrt(2) − 1

Another example of a continued fraction is that of the fractional part of the square root of 2, where the partial denominators are 2, 2, 2, 2, ... The algorithm to simulate this number is as follows:

1. With probability 2/3, generate an unbiased random bit and return that bit.
2. Run this algorithm recursively. If the result is 1, return 0. Otherwise, go to step 1.

4.2.6 1/sqrt(2)

This third example of a continued fraction shows how to simulate a probability $1/z$, where $z > 1$ has a known simple continued fraction expansion. In this case, the partial denominators are as follows: floor(z), $a[1]$, $a[2]$, ..., where the $a[i]$ are z's partial denominators (not including z's integer part). In the example of 1/sqrt(2), the partial denominators are 1, 2, 2, 2, ..., where 1 comes first since floor(sqrt(2)) = 1. The algorithm to simulate 1/sqrt(2) is as follows:

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

1. If *pos* is 1, return 1 with probability 1/2. If *pos* is greater than 1, then with probability 2/3, generate an unbiased random bit and return that bit.
2. Run this algorithm recursively, but with $pos = pos + 1$. If the result is 1, return 0. Otherwise, go to step 1.

4.2.7 tanh(1/2) or (exp(1) − 1) / (exp(1) + 1)

The algorithm begins with k equal to 2. Then the following steps are taken.

1. With probability $k/(1+k)$, return a number that is 1 with probability $1/k$ and 0 otherwise.
2. Run this algorithm recursively, but with $k = k + 4$. If the result is 1, return 0. Otherwise, go to step 1.

4.2.8 arctan(*x*/*y*) * *y*/*x*

(Flajolet et al., 2010)[1]:

1. Generate a uniform(0, 1) random number u.
2. Generate a number that is 1 with probability $x * x/(y * y)$, or 0 otherwise. If the number is 0, return 1.
3. **Sample from the number *u*** twice. If either of these calls returns 0, return 1.
4. Generate a number that is 1 with probability $x * x/(y * y)$, or 0 otherwise. If the number is 0, return 0.
5. **Sample from the number *u*** twice. If either of these calls returns 0, return 0. Otherwise, go to step 2.

Observing that the even-parity construction used in the Flajolet paper is equivalent to the two-coin special case, which is uniformly fast, the algorithm above can be made uniformly

fast as follows:

1. With probability 1/2, return 1.
2. Generate a uniform(0, 1) random number u, if it wasn't generated yet.
3. With probability $x * x/(y * y)$, **sample from the number *u*** twice. If both of these calls return 1, return 0.
4. Go to step 1.

4.2.9 *π* / 12

Two algorithms:

- First algorithm: Use the algorithm for **arcsin(1/2) / 2**. Where the algorithm says to "flip the input coin", instead generate an unbiased random bit.
- Second algorithm: With probability 2/3, return 0. Otherwise, run the algorithm for **π / 4** and return the result.

4.2.10 *π* / 4

(Flajolet et al., 2010)[(1)]:

1. Generate a random integer in the interval 0, 6), call it n.
2. If n is less than 3, return the result of the **algorithm for arctan(1/2) * 2**. Otherwise, if n is 3, return 0. Otherwise, return the result of the **algorithm for arctan(1/3) * 3**.

4.2.11 1 / *π*

(Flajolet et al., 2010)[(1)]:

1. Set t to 0.
2. With probability 1/4, add 1 to t and repeat this step. Otherwise, go to step 3.
3. With probability 1/4, add 1 to t and repeat this step. Otherwise, go to step 4.
4. With probability 5/9, add 1 to t.
5. Generate $2*t$ unbiased random bits (that is, either 0 or 1, chosen with equal probability), and return 0 if there are more zeros than ones generated this way or more ones than zeros. (Note that this condition can be checked even before all the bits are generated this way.) Do this step two more times.
6. Return 1.

For a sketch of how this algorithm is derived, see the appendix.

4.2.12 $(a/b)^{x/y}$

In the algorithm below, a, b, x, and y are integers, and the case where x/y is in (0, 1) is due to recent work by Mendo (2019)[(7)]. This algorithm works only if—

- x/y is 0 or greater and a/b is in the interval [0, 1], or
- x/y is less than 0 and a/b is 1 or greater.

The algorithm follows.

1. If x/y is less than 0, swap a and b, and remove the sign from x/y. If a/b is now no longer in the interval [0, 1], return an error.
2. If x/y is equal to 1, return 1 with probability a/b and 0 otherwise.

3. If x is 0, return 1. Otherwise, if a is 0, return 0. Otherwise, if a equals b, return 1.
4. If x/y is greater than 1:
    1. Set $ipart$ to floor(x/y) and $fpart$ to `rem(x, y)`.
    2. If $fpart$ is greater than 0, subtract 1 from $ipart$, then call this algorithm recursively with x = floor($fpart$/2) and $y = y$, then call this algorithm, again recursively, with $x = fpart$ − floor($fpart$/2) and $y = y$. Return 0 if either call returns 0. (This is done rather than the more obvious approach in order to avoid calling this algorithm with fractional parts very close to 0, because the algorithm runs much more slowly than for fractional parts closer to 1.)
    3. If $ipart$ is 1 or greater, generate a random number that is 1 with probability a^{ipart}/b^{ipart} or 0 otherwise. (Or generate $ipart$ many random numbers that are each 1 with probability a/b or 0 otherwise, then multiply them all into one number.) If that number is 0, return 0.
    4. Return 1.
5. Set i to 1.
6. With probability a/b, return 1.
7. Otherwise, with probability $x/(y*i)$, return 0.
8. Add 1 to i and go to step 6.

4.2.13 exp(−x/y)

This algorithm takes integers $x >= 0$ and $y > 0$ and outputs 1 with probability `exp(-x/y)` or 0 otherwise. It originates from (Canonne et al. 2020)[39].

1. Special case: If x is 0, return 1. (This is because the probability becomes `exp(0) = 1`.)
2. If `x > y` (so x/y is greater than 1), call this algorithm (recursively) `floor(x/y)` times with $x = y = 1$ and once with $x = x$ − floor(x/y) * y and $y = y$. Return 1 if all these calls return 1; otherwise, return 0.
3. Set r to 1 and i to 1.
4. Return r with probability ($y * i - x$) / ($y * i$).
5. Set r to 1 − r, add 1 to i, and go to step 4.

4.2.14 exp(−z)

This algorithm is similar to the previous algorithm, except that the exponent, z, can be any real number 0 or greater, as long as z can be rewritten as the sum of one or more components whose fractional parts can each be simulated by a Bernoulli factory algorithm that outputs heads with probability equal to that fractional part. (This makes use of the identity exp(−a) = exp(−b) * exp(−c).)

More specifically:

1. Decompose z into $n > 0$ positive components that sum to z. For example, if $z = 3.5$, it can be decomposed into only one component, 3.5 (whose fractional part is trivial to simulate), and if $z = \pi$, it can be decomposed into four components that are all (π / 4), which has a not-so-trivial simulation described earlier on this page.
2. For each component $LC[i]$ found this way, let $LI[i]$ be floor($LC[i]$) and let $LF[i]$ be $LC[i]$ − floor($LC[i]$) ($LC[i]$'s fractional part).

The algorithm is then as follows:

- For each component $LC[i]$, call the **algorithm for exp(− $LI[i]$/1)**, and call the **general martingale algorithm** adapted for **exp(−λ)** using the input coin that simulates $LF[i]$. If any of these calls returns 0, return 0; otherwise, return 1. (See also (Canonne et al. 2020)[39].)

4.2.15 $(a/b)^z$

This algorithm is similar to the previous algorithm for powering, except that the exponent, z, can be any real number 0 or greater, as long as z can be rewritten as the sum of one or more components whose fractional parts can each be simulated by a Bernoulli factory algorithm that outputs heads with probability equal to that fractional part. This algorithm makes use of a similar identity as for exp and works only if z is 0 or greater and a/b is in the interval [0, 1].

Decompose z into $LC[i]$, $LI[i]$, and $LF[i]$ just as for the **exp(− z)** algorithm. The algorithm is then as follows.

- If z is 0, return 1. Otherwise, if a is 0, return 0. Otherwise, for each component $LC[i]$ (until the algorithm returns a number):
  1. Call the **algorithm for $(a/b)^{LI[i]/1}$**. If it returns 0, return 0.
  2. Set j to 1.
  3. Generate a random number that is 1 with probability a/b and 0 otherwise. If that number is 1, abort these steps and move on to the next component or, if there are no more components, return 1.
  4. Flip the input coin that simulates $LF[i]$ (which is the exponent); if it returns 1, return 0 with probability $1/j$.
  5. Add 1 to j and go to substep 2.

4.2.16 1 / 1 + exp(x / (y * 2^{prec})) (LogisticExp)

This is the probability that the bit at $prec$ (the $prec^{\text{th}}$ bit after the point) is set for an exponential random number with rate x/y. This algorithm is a special case of the **logistic Bernoulli factory**.

1. With probability 1/2, return 1.
2. Call the **algorithm for exp(− x/(y * 2^{prec}))**. If the call returns 1, return 1. Otherwise, go to step 1.

4.2.17 1 / 1 + exp(z / 2^{prec})) (LogisticExp)

This is similar to the previous algorithm, except that z can be any real number described in the **algorithm for exp(−z)**.

Decompose z into $LC[i]$, $LI[i]$, and $LF[i]$ just as for the **exp(−z)** algorithm. The algorithm is then as follows.

1. For each component $LC[i]$, create an input coin that does the following: "(a) With probability $1/(2^{prec})$, return 1 if the input coin that simulates $LF[i]$ returns 1; (b) Return 0".
2. Return 0 with probability 1/2.
3. Call the **algorithm for exp(− x/y)** with $x = \Sigma_i LI[i]$ and $y = 2^{prec}$. If this call returns 0, go to step 2.
4. For each component $LC[i]$, call the **algorithm for exp(−λ)**, using the corresponding input coin for $LC[i]$ created in step 1. If any of these calls returns 0, go to step 2. Otherwise, return 1.

4.2.18 Polylogarithmic Constants

The following algorithm simulates a polylogarithmic constant of the form $\mathrm{Li}_r(1/2)$, where r is an integer 1 or greater. See (Flajolet et al., 2010)**(1)** and "Convex Combinations" (the algorithm works by decomposing the series forming the polylogarithmic constant into $g(i) = (1/2)^i$, which sums to 1, and $h_i() = i^r$, where $i >= 1$).

1. Set t to 1.
2. With probability 1/2, add 1 to t and repeat this step. Otherwise, go to step 3.
3. Return a number that is 1 with probability $1/t^r$ and 0 otherwise.

4.2.19 ζ(3) * 3 / 4 and Other Zeta-Related Constants

(Flajolet et al., 2010)**(1)**. It can be seen as a triple integral whose integrand is $1/(1 + a * b * c)$, where a, b, and c are uniform(0, 1) random numbers. This algorithm is given below, but using the two-coin special case instead of the even-parity construction. Note that the triple integral in section 5 of the paper is $\zeta(3) * 3 / 4$, not $\zeta(3) * 7 / 8$. (Here, $\zeta(x)$ is the Riemann zeta function.)

1. Generate three uniform(0,1) random numbers.
2. With probability 1/2, return 1.
3. **Sample from each of the three numbers** generated in step 1. If all three calls return 1, return 0. Otherwise, go to step 2. (This implements a triple integral involving the uniform random numbers.)

This can be extended to cover any constant of the form $\zeta(k) * (1 - 2^{-(k - 1)})$ where $k >= 2$ is an integer, as suggested slightly by the Flajolet paper when it mentions $\zeta(5) * 31 / 32$ (which should probably read $\zeta(5) * 15 / 16$ instead), using the following algorithm.

1. Generate k uniform(0,1) random numbers.
2. With probability 1/2, return 1.
3. **Sample from each of the *k* numbers** generated in step 1. If all k calls return 1, return 0. Otherwise, go to step 2.

4.2.20 erf(*x*)/erf(1)

In the following algorithm, x is a real number in the interval [0, 1].

1. Generate a uniform(0, 1) random number, call it *ret*.
2. Set u to point to the same value as *ret*, and set k to 1.
3. (In this and the next step, we create v, which is the maximum of two uniform [0, 1] random numbers.) Generate two uniform(0, 1) random numbers, call them a and b.
4. If a is less than b, set v to b. Otherwise, set v to a.
5. If v is less than u, set u to v, then add 1 to k, then go to step 3.
6. If k is odd, return 1 if *ret* is less than x, or 0 otherwise. (If *ret* is implemented as a uniform PSRN, this comparison should be done via the **URandLessThanReal algorithm**, which is described in my **article on PSRNs**.)
7. Go to step 1.

In fact, this algorithm takes advantage of a theorem related to the Forsythe method of random sampling (Forsythe 1972)**(40)**. See the section "**Probabilities Arising from Certain Permutations**" in the appendix for more information.

> **Note:** If the last step in the algorithm reads "Return 0" rather than "Go to step 1", then the algorithm simulates the probability erf(x)*sqrt(π)/2 instead.

4.2.21 2 / (1 + exp(2)) or (1 + exp(0)) / (1 + exp(1))

This algorithm takes advantage of formula 2 mentioned in the section "**Probabilities Arising from Certain Permutations**" in the appendix. Here, the relevant probability is rewritten as $1 - (\int_{(-\infty,\ 1)} (1 - \exp(-\max(0,\ \min(1,\ z)))) * \exp(-z)\ dz) / (\int_{(-\infty,\ \infty)} (1 - \exp(-\max(0,\ \min(1,\ z))) * \exp(-z)\ dz)$.

1. Generate an **exponential** random number *ex*, then set *k* to 1.
2. Set *u* to point to the same value as *ex*.
3. Generate a **uniform(0,1)** random number *v*.
4. Set *stop* to 1 if *u* is less than *v*, and 0 otherwise.
5. If *stop* is 1 and *k* **is even**, return a number that is 0 if *ex* is **less than 1**, and 1 otherwise. Otherwise, if *stop* is 1, go to step 1.
6. Set *u* to *v*, then add 1 to *k*, then go to step 3.

4.2.22 (1 + exp(1)) / (1 + exp(2))

This algorithm takes advantage of the theorem mentioned in the section "**Probabilities Arising from Certain Permutations**" in the appendix. Here, the relevant probability is rewritten as $1 - (\int_{(-\infty,\ 1/2)} \exp(-\max(0,\ \min(1,\ z))) * \exp(-z)\ dz) / (\int_{(-\infty,\ \infty)} \exp(-\max(0,\ \min(1,\ z)) * \exp(-z)\ dz)$.

1. Generate an **exponential** random number *ex*, then set *k* to 1.
2. Set *u* to point to the same value as *ex*.
3. Generate a **uniform(0,1)** random number *v*.
4. Set *stop* to 1 if *u* is less than *v*, and 0 otherwise.
5. If *stop* is 1 and *k* **is odd**, return a number that is 0 if *ex* is **less than 1/2**, and 1 otherwise. Otherwise, if *stop* is 1, go to step 1.
6. Set *u* to *v*, then add 1 to *k*, then go to step 3.

4.2.23 (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 by an algorithm by Citterio and Pavani (2016)**(41)**. Note the use of the methodology in (Łatuszyński et al. 2009/2011, algorithm 2)**(8)** 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 number, 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 the appendix.
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.

4.2.24 Euler's Constant γ

The following algorithm to simulate Euler's constant γ is due to Mendo (2020)[42]. This solves an open question given in (Flajolet et al., 2010)[1]. The series used was given by Sondow (2005)[43]. An algorithm for γ appears here even though it is not yet known whether this constant is irrational.

1. Set ϵ to 1, then set n, $lamunq$, lam, s, k, and $prev$ to 0 each.
2. Add 1 to k, then add $s/(2^k)$ to lam.
3. If $lamunq+\epsilon <= lam + 1/(2^k)$, go to step 8.
4. If $lamunq > lam + 1/(2^k)$, go to step 8.
5. If $lamunq > lam + 1/(2^{k+1})$ and $lamunq+\epsilon < 3/(2^{k+1})$, go to step 8.
6. (This step adds a term of the series for γ to $lamunq$, and sets ϵ to an upper bound on the error that results if the series is truncated after summing this and the previous terms.) If n is 0, add 1/2 to $lamunq$ and set ϵ to 1/2. Otherwise, add $B(n)/(2*n*(2*n+1)*(2*n+2))$ to $lamunq$ and set ϵ to $\min(prev, (2+B(n)+(1/n))/(16*n*n))$, where $B(n)$ is the minimum number of bits needed to store n (or the smallest $b>=1$ such that $n < 2^b$).
7. Add 1 to n, then set $prev$ to ϵ, then go to step 3.
8. Let $bound$ be $lam+1/(2^k)$. If $lamunq+\epsilon <= bound$, set s to 0. Otherwise, if $lamunq > bound$, set s to 2. Otherwise, set s to 1.
9. With probability 1/2, go to step 2. Otherwise, return a number that is 0 if s is 0, 1 if s is 2, or an unbiased random bit (either 0 or 1 with equal probability) otherwise.

4.2.25 exp(−x/y) * z/t

This algorithm is again based on an algorithm due to Mendo (2020)[42]. In this algorithm, x, y, z, and t are integers greater than 0, except x and/or z may be 0, and must be such that $\exp(-x/y) * z/t$ is in the interval [0, 1].

1. If z is 0, return 0. If x is 0, return a number that is 1 with probability z/t and 0 otherwise.
2. Set ϵ to 1, then set n, $lamunq$, lam, s, and k to 0 each.
3. Add 1 to k, then add $s/(2^k)$ to lam.
4. If $lamunq+\epsilon <= lam + 1/(2^k)$, go to step 9.
5. If $lamunq > lam + 1/(2^k)$, go to step 9.
6. If $lamunq > lam + 1/(2^{k+1})$ and $lamunq+\epsilon < 3/(2^{k+1})$, go to step 8.
7. (This step adds two terms of $\exp(-x/y)$'s alternating series, multiplied by z/t, to $lamunq$, and sets ϵ to an upper bound on how close the current sum is to the desired probability.) Let m be $n*2$. Set ϵ to $z*x^m/(t*(m!)*y^m)$. If m is 0, add $z*(y-x)/(t*y)$ to $lamunq$. Otherwise, add $z*x^m*(m*y-x+y) / (t*y^{m+1}*((m+1)!))$ to $lamunq$.
8. Add 1 to n and go to step 4.
9. Let $bound$ be $lam+1/(2^k)$. If $lamunq+\epsilon <= bound$, set s to 0. Otherwise, if $lamunq > bound$, set s to 2. Otherwise, set s to 1.
10. With probability 1/2, go to step 3. Otherwise, return a number that is 0 if s is 0, 1 if s is 2, or an unbiased random bit (either 0 or 1 with equal probability) otherwise.

4.2.26 ln(2)

A special case of the algorithm for $\ln(1+\lambda)$ given earlier.

1. With probability 1/2, return 1.
2. Generate a uniform(0, 1) random number u, if it wasn't generated yet.
3. **Sample from the number u**. If the result is 1, return 0. Otherwise, go to step 1.

4.2.27 ln(1+y/z)

See also the algorithm given earlier for ln(1+λ). In this algorithm, y/z is a rational number in the interval [0, 1].

1. If y/z is 0, return 0.
2. With probability 1/2, return a number that is 1 with probability y/z and 0 otherwise.
3. Generate a uniform(0, 1) random number u, if u wasn't generated yet.
4. **Sample from the number u**, then generate a number that is 1 with probability y/z and 0 otherwise. If the call returns 1 and the number generated is 1, return 0. Otherwise, go to step 2.

4.3 General Algorithms

4.3.1 Convex Combinations

Assume we have one or more input coins $h_i(\lambda)$ that returns heads with a probability that depends on λ. (The number of coins may be infinite.) The following algorithm chooses one of these coins at random then flips that coin. Specifically, the algorithm generates 1 with probability equal to the following weighted sum: $g(0) * h_0(\lambda) + g(1) * h_1(\lambda) + ...$, where $g(i)$ is the probability that coin i will be chosen, h_i is the function simulated by coin i, and all the $g(i)$ sum to 1. See (Wästlund 1999, Theorem 2.7)[27]. (Alternatively, the algorithm can be seen as returning heads with probability $\mathbf{E}[h_X(\lambda)]$, that is, the expected or average value of h_X where X is the number that identifies the randomly chosen coin.)

1. Generate a random integer X in some way. For example, it could be a uniform random integer in [1, 6], or it could be a Poisson random number. (Specifically, the number X is generated with probability $g(X)$.)
2. Flip the coin represented by X and return the result.

   Examples:

   1. Generate a Poisson(μ) random number X, then flip the input coin. With probability 1/(1+X), return the result of the coin flip; otherwise, return 0. This corresponds to $g(i)$ being the Poisson(μ) probabilities and $h_i()$ returning 1 with probability 1/(1+i), and 0 otherwise. The probability that this method returns 1 is $\mathbf{E}[1/(1+X)]$, or $(\exp(\mu)-1)/(\exp(\mu)*\mu)$.
   2. Generate a Poisson(μ) random number X and return 1 if X is 0, or 0 otherwise. This is a Bernoulli factory for exp(−μ) mentioned earlier, and corresponds to $g(i)$ being the Poisson(μ) probabilities and $h_i()$ returning 1 if i is 0, and 0 otherwise.
   3. Generate a Poisson(μ) random number X, run the **algorithm for exp(−z)** with $z = X$, and return the result. The probability of returning 1 this way is $\mathbf{E}[\exp(-X)]$, or $\exp(\mu*\exp(-1)-\mu)$. The following Python code uses the computer algebra library SymPy to find this probability: `from sympy.stats import *; E(exp(-Poisson('P', x))).simplify()`.
   4. *Bernoulli Race* (Dughmi et al. 2017)[11]: Say we have n coins, then choose

one of them uniformly at random and flip that coin. If the flip returns 1, return X; otherwise, repeat this algorithm. This algorithm chooses a random coin based on its probability of heads. Each iteration corresponds to $g(i)$ being $1/n$ and $h_i()$ being the probability for the corresponding coin i.

5. (Wästlund 1999)**(27)**: Generate a Poisson(1) random number X, then flip the input coin X times. Return 0 if any of the flips returns 1, or 1 otherwise. This is a Bernoulli factory for $\exp(-\lambda)$, and corresponds to $g(i)$ being the Poisson(1) probabilities, namely $1/(i!*\exp(1))$, and $h_i()$ being $(1-\lambda)^i$.

4.3.2 Simulating the Probability Generating Function

The following algorithm is a special case of the convex combination method. It generates heads with probability $\mathbf{E}[\lambda^X]$, that is, the expected or average value of λ^X. $\mathbf{E}[\lambda^X]$ is the *probability generating function*, also known as *factorial moment generating function*, for the distribution of X (Dughmi et al. 2017)**(11)**.

1. Generate a random integer X in some way. For example, it could be a uniform random integer in [1, 6], or it could be a Poisson random number.
2. Flip the input coin until the flip returns 0 or the coin is flipped X times, whichever comes first. Return 1 if all the coin flips, including the last, returned 1 (or if X is 0); or return 0 otherwise.

4.3.3 Integrals

(Flajolet et al., 2010)**(1)** showed how to turn an algorithm that simulates $f(\lambda)$ into an algorithm that simulates the following probability:

- $(1/\lambda)\ \int_{[0,\ \lambda]} f(u)\ du$, or equivalently,
- $\int_{[0,\ 1]} f(u * \lambda)\ du$ (an integral).

This can be done by modifying the algorithm as follows:

- Generate a uniform(0, 1) random number u at the start of the algorithm.
- Instead of flipping the input coin, flip a coin that does the following: "Flip the input coin, then **sample from the number *u***. Return 1 if both the call and the flip return 1, and return 0 otherwise."

I have found that it's possible to simulate the following integral, namely—

- $\int_{[a,\ b]} f(u)\ du$,

where $[a, b]$ is [0, 1] or a closed interval therein, using different changes to the algorithm, namely:

- Add the following step at the start of the algorithm: "Generate a uniform(0, 1) random number u at the start of the algorithm. Then if u is less than a or is greater than b, repeat this step. (If u is a uniform PSRN, these comparisons should be done via the **URandLessThanReal** algorithm.)"
- Instead of flipping the input coin, flip a coin that does the following: "**Sample from the number *u*** and return the result."
- If the algorithm would return 1, it instead returns a number that is 1 with probability $b - a$ and 0 otherwise.

Note: If a is 0 and b is 1, the probability simulated by this algorithm will be monotonically increasing (will keep going up), have a slope no greater than 1, and equal 0 at the point 0.

4.3.4 Certain Converging Series

The algorithm for Euler's constant is one example of a general algorithm given by Mendo (2020)[(42)] for simulating any probability in (0, 1), as long as it can be rewritten as a converging series—

- that has the form $a[0] + a[1] + ...$, where $a[n]$ are all positive rational numbers, and
- for which a sequence $err[0]$, $err[1]$, ..., is available that is nonincreasing and converges to 0, where $err[n]$ is an upper bound on the error from truncating the series a after summing the first $n+1$ terms.

The algorithm follows.

1. Set ϵ to 1, then set n, $lamunq$, lam, s, and k to 0 each.
2. Add 1 to k, then add $s/(2^k)$ to lam.
3. If $lamunq+\epsilon <= lam + 1/(2^k)$, go to step 8.
4. If $lamunq > lam + 1/(2^k)$, go to step 8.
5. If $lamunq > lam + 1/(2^{k+1})$ and $lamunq+\epsilon < 3/(2^{k+1})$, go to step 8.
6. Add $a[n]$ to $lamunq$ and set ϵ to $err[n]$.
7. Add 1 to n, then go to step 3.
8. Let $bound$ be $lam+1/(2^k)$. If $lamunq+\epsilon <= bound$, set s to 0. Otherwise, if $lamunq > bound$, set s to 2. Otherwise, set s to 1.
9. With probability 1/2, go to step 2. Otherwise, return a number that is 0 if s is 0, 1 if s is 2, or an unbiased random bit (either 0 or 1 with equal probability) otherwise.

If a, given above, is instead a sequence that converges to the *base-2 logarithm* of a probability in (0, 1), the following algorithm I developed simulates that probability. For simplicity's sake, even though logarithms for such probabilities are negative, all the $a[i]$ must be 0 or greater (and thus are the negated values of the already negative logarithm approximations) and must form a nondecreasing sequence, and all the $err[i]$ must be 0 or greater.

1. Set $intinf$ to floor(max(0, abs($a[0]$))). (This is the absolute integer part of the first term in the series, or 0, whichever is greater.)
2. If $intinf$ is greater than 0, generate unbiased random bits until a zero bit or $intinf$ bits were generated this way. If a zero was generated this way, return 0.
3. Generate an exponential random number E with rate ln(2). This can be done, for example, by using the algorithm given in "**More Algorithms for Arbitrary-Precision Sampling**". (We take advantage of the exponential distribution's *memoryless property*: given that an exponential random number E is greater than $intinf$, E minus $intinf$ has the same distribution.)
4. Set n to 0.
5. Set inf to max(0, $a[n]$), then set sup to min(0, $inf+err[n]$).
6. If E is less than $inf+intinf$, return 0. If E is less than $sup+intinf$, go to the next step. If neither is the case, return 1.
7. Set n to 1, then go to step 5.

The case when a converges to a *natural logarithm* rather than a base-2 logarithm is trivial by comparison. Again for this algorithm, all the $a[i]$ must be 0 or greater and form a nondecreasing sequence, and all the $err[i]$ must be 0 or greater.

1. Generate an exponential random number E (with rate 1).
2. Set n to 0.
3. Set inf to max(0, $a[n]$), then set sup to min(0, $inf+err[n]$).
4. If E is less than $inf+intinf$, return 0. If E is less than $sup+intinf$, go to the next step. If neither is the case, return 1.
5. Set n to 1, then go to step 5.

Example:

- Let $f(\lambda) = \cosh(1)-1$. The first algorithm in this section can simulate this constant if step 6 is modified to read: "Let m be ((n+1)*2), and let α be 1/(m!) (a term of the Taylor series). Add α to $lamunq$ and set ϵ to 2/((m+1)!) (the error term).".**(44)**
- Logarithms can form the basis of efficient algorithms to simulate the probability z = choose(n, k)/2^n when n can be very large (e.g., as large as 2^{30}), without relying on floating-point arithmetic. In this example, the trivial algorithm for choose(n, k), the binomial coefficient, will generally require a growing amount of storage that depends on n and k. On the other hand, any constant can be simulated using up to two unbiased random bits on average, and even slightly less than that for the constants at hand here (Kozen 2014)**(45)**. Instead of calculating the binomial coefficient directly, a series can be calculated that converges to that coefficient's logarithm, such as ln(choose(n, k)), which is economical in space even for large n and k. Then the algorithm above can be used with that series to simulate the probability z. A similar approach has been implemented (see **interval.py** and **betadist.py**). See also an appendix in (Bringmann et al. 2014)**(46)**.

4.3.5 General Factory Functions

A coin with unknown probability of heads of λ can be turned into a coin with probability of heads of $f(\lambda)$, where f is any factory function, via an algorithm that works with two sequences of polynomials:

- One sequence of polynomials must be non-decreasing and converge from below to f, and the other sequence must be non-increasing and converge from above to f.
- For both sequences, there must be a way to calculate their polynomials' Bernstein coefficients.
- For each sequence, the difference between one polynomial and its previous one must have non-negative Bernstein coefficients (once the previous polynomial is transformed to have the same degree as the other).

This section sets forth two algorithms to simulate factory functions via polynomials. In both algorithms:

- **fbelow**(n, k) is a lower bound of the k^{th} Bernstein coefficient for a degree-n polynomial that approximates f from below, where k is in the interval [0, n]. For example, this can be $f(k/n)$ minus a constant that depends on n. (See note 6 below.)
- **fabove**(n, k) is an upper bound of the k^{th} Bernstein coefficient for a degree-n polynomial that approximates f from above. For example, this can be $f(k/n)$ plus a constant that depends on n. (See note 6.)

The first algorithm implements the reverse-time martingale framework (Algorithm 4) in Łatuszyński et al. (2009/2011)**(8)** and the degree-doubling suggestion in Algorithm I of

Flegal and Herbei (2012)[47], although an error in Algorithm I is noted below. The first algorithm follows.

1. Generate a uniform(0, 1) random number, call it *ret*.
2. Set ℓ and ℓt to 0. Set *u* and *ut* to 1. Set *lastdegree* to 0, and set *ones* to 0.
3. Set *degree* so that the first pair of polynomials has degree equal to *degree* and has Bernstein coefficients all lying in [0, 1]. For example, this can be done as follows: Let **fbound**(*n*) be the minimum value for **fbelow**(*n*, *k*) and the maximum value for **fabove**(*n*,*k*) for any *k* in the interval [0, *n*]; then set *degree* to 1; then while **fbound**(*degree*) returns an upper or lower bound that is less than 0 or greater than 1, multiply *degree* by 2; then go to the next step.
4. Set *startdegree* to *degree*.
5. (Loop.) Flip the input coin *t* times, where *t* is *degree* − *lastdegree*. For each time the coin returns 1 this way, add 1 to *ones*.
6. Set ℓ to **fbelow**(*degree*, *ones*), set *u* to **fabove**(*degree*, *ones*), and set *lastdegree* to *degree*.
7. (This step and the next find the expected values of the previous ℓ and *u* given the current coin flips.) If *degree* equals *startdegree*, set ℓs to 0 and *us* to 1. (Algorithm I of Flegal and Herbei 2012 doesn't take this into account.)
8. If *degree* is greater than *startdegree*: Let *nh* be choose(*degree*, *ones*), and let *od* be *degree*/2. Set ℓs to $\Sigma_{j=0,\ldots,ones}$ **fbelow**(*od*,*j*)*choose(*degree*−*od*, *ones*−*j*)*choose(*od*,*j*)/*nh*, and set *us* to $\Sigma_{j=0,\ldots,ones}$ **fabove**(*od*,*j*)*choose(*degree*−*od*, *ones*−*j*)*choose(*od*,*j*)/*nh*.
9. Let *m* be (*ut*−ℓt)/(*us*−ℓs). Set ℓt to ℓt+(ℓ−ℓs)*m, and set *ut* to *ut*−(*us*−*u*)*m.
10. If *ret* is less than (or equal to) ℓt, return 1. If *ret* is less than *ut*, go to the next step. If neither is the case, return 0. (If *ret* is a uniform PSRN, these comparisons should be done via the **URandLessThanReal algorithm**, which is described in my **article on PSRNs**.)
11. (Find the next pair of polynomials and restart the loop.) Increase *degree* so that the next pair of polynomials has degree equal to a higher value of *degree* and gets closer to the target function (for example, multiply *degree* by 2). Then, go to step 5.

The second algorithm was given in Thomas and Blanchet (2012)[24]; it assumes the same sequences of polynomials are available as in the previous algorithm. An algorithm equivalent to that algorithm is given below.

1. Set *ones* to 0, and set *lastdegree* to 0.
2. Set *degree* so that the first pair of polynomials has degree equal to *degree* and has Bernstein coefficients all lying in [0, 1]. For example, this can be done as follows: Let **fbound**(*n*) be the minimum value for **fbelow**(*n*, *k*) and the maximum value for **fabove**(*n*,*k*) for any *k* in the interval [0, *n*]; then set *degree* to 1; then while **fbound**(*degree*) returns an upper or lower bound that is less than 0 or greater than 1, multiply *degree* by 2; then go to the next step.
3. Set *startdegree* to *degree*.
4. (Loop.) Flip the input coin *t* times, where *t* is *degree* − *lastdegree*. For each time the coin returns 1 this way, add 1 to *ones*.
5. Set *c* to choose(*degree*, *ones*).
6. Calculate *a*[*degree*,*ones*] = floor(**fbelow**(*degree*, *ones*)*c) and set *acount* to it, then calculate *b*[*degree*,*ones*] = floor((1−**fabove**(*degree*, *ones*))*c) and set *bcount* to it, then subtract (*acount* + *bcount*) from *c*.
7. If *degree* is greater than *startdegree*, then:
    1. Let *diff* be *degree*−*lastdegree*, let *u* be max(0, *ones*−*lastdegree*), and let *v* be min(*ones*, *diff*). (The following substep removes outcomes from *acount* and *bcount* that would have terminated the algorithm earlier. The procedure differs from step (f) of section 3 of the paper, which appears to be incorrect, and the

procedure was derived from the **supplemental source code** uploaded by A. C. Thomas at my request.)

2. For each integer k in the interval [u, v]:
    1. Set d to choose($diff$, k).
    2. Subtract (a[$lastdegree$, $ones$−k]*d) from $acount$. Here, a[s,t] is calculated as floor(**fbelow**(s, t)*choose(s, t)), and may be stored for later use.
    3. Subtract (b[$lastdegree$, $ones$−k]*d) from $bcount$. Here, b[s,t] is calculated as floor((1−**fabove**(s, t))*choose(s, t)), and may be stored for later use.

8. Call **WeightedChoice**([$acount$, $bcount$, c]), where **WeightedChoice** is given in "**Randomization and Sampling Methods**". (This generates a number that is 0, 1, or 2 with probability proportional to each of the given weights.)
9. If the number generated by the previous step is 0, return 1. If the number generated by that step is 1, return 0.
10. (Find the next pair of polynomials and restart the loop.) Set $lastdegree$ to $degree$, then increase $degree$ so that the next pair of polynomials has degree equal to a higher value of $degree$ and gets closer to the target function (for example, multiply $degree$ by 2). Then, go to step 4.

Notes:

1. The efficiency of these two algorithms depends, among other things, on how "smooth" f is, and on how easy it is to calculate the appropriate values for **fbelow** and **fabove**. The best way to implement **fbelow** and **fabove** will require a deep mathematical analysis of f.

2. If f is known to be *concave* in the interval [0, 1] (which roughly means that its rate of growth there never goes up), then **fbelow**(n, k) can equal f(k/n), thanks to Jensen's inequality.

3. If f is known to be *convex* in the interval [0, 1] (which roughly means that its rate of growth there never goes down), then **fabove**(n, k) can equal f(k/n), thanks to Jensen's inequality. One example is f(λ) = exp(−λ/4).

4. The following method (see Powell 1981)**(48)** implements **fabove** and **fbelow** if f(λ), in the interval [0, 1]—

    - has continuous "slope" and "slope-of-slope" functions (in other words, f is C^2 continuous there), and
    - either—
        - has a minimum of greater than 0 and a maximum of less than 1, or
        - is convex and has a minimum of greater than 0, or
        - is concave and has a maximum of less than 1.

    Let m be an upper bound of the highest value of abs(f''(x)) for any x in [0, 1], where f'' is the "slope-of-slope" function of f. Then:

    - **fbelow**(n, k) = f(k/n) + m/(n*8) (or f(k/n) if f is concave; see note 2).
    - **fabove**(n, k) = f(k/n) + m/(n*8) (or f(k/n) if f is convex; see note 3).

    The SymPy code in the **appendix** can calculate the necessary values for **fbound(*n*)** and m, given f.

5. In some cases, a single pair of polynomial sequences may not converge quickly to the desired function f, especially when f is not C^2 continuous. An intriguing suggestion from Thomas and Blanchet (2012)**(24)** is to use

multiple pairs of polynomial sequences that converge to f, where each pair is optimized for particular ranges of λ: first flip the input coin several times to get a rough estimate of λ, then choose the pair that's optimized for the estimated λ, and run either algorithm in this section on that pair.

6. If $f(k/n)$ is not a rational number, then it should be calculated in **fabove** and **fbelow** with an accuracy that improves as n (the polynomial degree) gets larger. In that case, **fabove** should calculate an upper bound of $f(k/n)$, and **fbelow** a lower bound. Also, it's often convenient to implement **fabove** and **fbelow** with the same code routine and with rational interval arithmetic (such as the one described in Daumas et al. (2007)[49]), since both bounds would then be available at once.

Examples:

1. If $f(\lambda) = \min(\lambda, c)$ with c in the interval (0, 1), then the following implementations can be used (Lorentz 1953)[50]:
   - **fbelow**(n, k) = $f(k/n)$. This is possible because f is concave.
   - **fabove**(n, k) = $f(k/n) + S/\text{sqrt}(n)$, where S = (4306+837*sqrt(6))/5832 is Sikkema's constant (Sikkema 1961)[51] and has an upper bound of 1.08989.
   - **fbound**(n) = [0, **fabove**(n, n)].
2. If $f(\lambda) = \sin(2*\lambda)/2$, then note 4 suggests the following:
   - **fbelow**(n, k) = $\sin(2*k/n)/2$. This is possible because f is concave.
   - **fabove**(n, k) = $\sin(2*k/n)/2 + 2 / (n*8)$.
   - **fbound**(n) = [0, (1/2) + 1/(4*n)].
3. If $f(\lambda) = \sin(3*\lambda)/2$, then notes 4 suggests the following:
   - **fbelow**(n, k) = $\sin(3*k/n)/2$. This is possible because f is concave.
   - **fabove**(n, k) = $\sin(3*k/n)/2 + (9/16) / (n*8)$.
   - **fbound**(n) = [0, (1/2) + 9/(16*n)].

5 Requests and Open Questions

1. See the open questions found in the section "**Probabilities Arising from Certain Permutations**" in the appendix.

2. I request expressions of mathematical functions that can be expressed in any of the following ways:

   - Series expansions for continuous functions that equal 0 or 1 at the points 0 and 1. These are required for Mendo's algorithm for **certain power series**.

   - Series expansions for alternating power series whose coefficients are all in the interval [0, 1] and form a nonincreasing sequence. This is required for another class of power series.

   - Series expansions with non-negative coefficients and for which bounds on the truncation error are available.

   - Upper and lower bound approximations that converge to a given constant or function. These upper and lower bounds must be nonincreasing or nondecreasing, respectively.

   - To apply the algorithms for **general factory functions**, what is needed are two

sequences of polynomials in Bernstein form, one of which converges from above to a given function, the other from below. These sequences must be nonincreasing or nondecreasing, respectively, and the polynomials must be of increasing degree and have Bernstein coefficients that are all rational numbers lying in [0, 1], but the polynomials in each sequence may start closer to the function at some points than at others.

Especially helpful would be an automated procedure to compute such sequences, in terms of their Bernstein coefficients, for a large class of factory functions (such as min(λ, c) where c is a constant in (0, 1)). (This is in the sense that when given only information about the desired function, such as the coordinates of the function's piecewise linear graph, the procedure can automatically compute the appropriate sequences without further user intervention.)

I have found **several methods** to compute such sequences, but most of them have issues that I seek clarification on. For example, the method of Holtz et al. (2011)**(25)** requires knowing the function's smoothness class and requires the function to be bounded away from 0 and 1; moreover the method uses several constants, namely s, θ_α, and D, with no easy lower bounds. As another example, Gal's method (1989)**(52)** produces polynomials that converge too slowly to be practical.

See also my questions on *Mathematics Stack Exchange*:

- **Computing converging polynomials**.
- **Bounds of Bernstein coefficients**.

- Simple **continued fractions** that express useful constants.

All these expressions should not rely on floating-point arithmetic or the direct use of irrational constants (such as π or sqrt(2)), but may rely on rational arithmetic. For example, a series expansion that *directly* contains the constant π is not desired; however, a series expansion that converges to a fraction of π is.

3. Is there a simpler or faster way to implement the base-2 or natural logarithm of binomial coefficients? See the example in the section "**Certain Converging Series**".

4. According to (Mossel and Peres 2005)**(21)**, a pushdown automaton can take a coin with unknown probability of heads of λ and turn it into a coin with probability of heads of $f(\lambda)$ only if f is a factory function and can be a solution of a polynomial system with rational coefficients. (See "**Certain Algebraic Functions**".) Are there any results showing whether the converse is true; namely, can a pushdown automaton simulate *any* f of this kind? Note that this question is not quite the same as the question of which algebraic functions can be simulated by a context-free grammar (either in general or restricted to those of a certain ambiguity and/or alphabet size), and is not quite the same as the question of which *probability generating functions* can be simulated by context-free grammars or pushdown automata, although answers to those questions would be nice. (See also Icard 2019**(23)**. Answering this question might involve ideas from analytic combinatorics; e.g., see the recent works of Cyril Banderier and colleagues.)

6 Correctness and Performance Charts

Charts showing the correctness and performance of some of these algorithms are found in a **separate page**.

7 Acknowledgments

I acknowledge Luis Mendo, who responded to one of my open questions, as well as C. Karney.

8 Notes

- [1] Flajolet, P., Pelletier, M., Soria, M., "**On Buffon machines and numbers**", arXiv:0906.5560 [math.PR], 2010.
- [2] Keane, M. S., and O'Brien, G. L., "A Bernoulli factory", *ACM Transactions on Modeling and Computer Simulation* 4(2), 1994.
- [3] There is an analogue to the Bernoulli factory problem called the *quantum Bernoulli factory*, with the same goal of simulating functions of unknown probabilities, but this time with algorithms that employ quantum-mechanical operations (unlike *classical* algorithms that employ no such operations). However, quantum-mechanical programming is far from being accessible to most programmers at the same level as classical programming, and will likely remain so for the foreseeable future. For this reason, the *quantum Bernoulli factory* is outside the scope of this document, but it should be noted that more factory functions can be "constructed" using quantum-mechanical operations than by classical algorithms. For example, a factory function defined in [0, 1] has to meet the requirements proved by Keane and O'Brien except it can touch 0 and/or 1 at a finite number of points in the domain, but its value still cannot go to 0 or 1 exponentially fast (Dale, H., Jennings, D. and Rudolph, T., 2015, "Provable quantum advantage in randomness processing", *Nature communications* 6(1), pp. 1-4).
- [4] Huber, M., "**Nearly optimal Bernoulli factories for linear functions**", arXiv:1308.1562v2 [math.PR], 2014.
- [5] Nacu, Şerban, and Yuval Peres. "**Fast simulation of new coins from old**", The Annals of Applied Probability 15, no. 1A (2005): 93-115.
- [6] Yannis Manolopoulos. 2002. "Binomial coefficient computation: recursion or iteration?", SIGCSE Bull. 34, 4 (December 2002), 65–67. DOI: **https://doi.org/10.1145/820127.820168.**
- [7] Mendo, Luis. "An asymptotically optimal Bernoulli factory for certain functions that can be expressed as power series." Stochastic Processes and their Applications 129, no. 11 (2019): 4366-4384.
- [8] 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.
- [9] Devroye, L., ***Non-Uniform Random Variate Generation***, 1986.
- [10] Another algorithm for exp(−λ) involves the von Neumann schema described in the appendix, but unfortunately, it converges slowly as λ approaches 1.
- [11] 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).
- [12] Gonçalves, F. B., Łatuszyński, K. G., Roberts, G. O. (2017). Exact Monte Carlo likelihood-based inference for jump-diffusion processes.
- [13] There are two other algorithms for this function, but they both converge very slowly when λ is very close to 1. One is the general martingale algorithm, since when λ is in [0, 1], this function is an alternating series of the form `1 - x + x^2 - x^3 + ...`, whose coefficients are 1, 1, 1, 1, ... The other is the so-called "even-parity" construction from Flajolet et al. 2010: "(1) Flip the input coin. If it returns 0, return 1. (2) Flip the input coin. If it returns 0, return 0. Otherwise, go to step 1."
- [14] Another algorithm for this function uses the general martingale algorithm, but uses more bits on average as λ approaches 1. Here, the alternating series is `1 - x + x^2/2 - x^3/3 + ...`, whose coefficients are 1, 1, 1/2, 1/3, ...
- [15] Vats, D., Gonçalves, F. B., Łatuszyński, K. G., Roberts, G. O. "**Efficient Bernoulli factory MCMC**

for intractable likelihoods", arXiv:2004.07471 [stat.CO], 2020.

- (16) Huber, M., "**Optimal linear Bernoulli factories for small mean problems**", arXiv:1507.00843v2 [math.PR], 2016.
- (17) Morina, G., Łatuszyński, K., et al., "**From the Bernoulli Factory to a Dice Enterprise via Perfect Sampling of Markov Chains**", arXiv:1912.09229 [math.PR], 2019/2020.
- (18) One of the only implementations I could find of this, if not the only, was a **Haskell implementation**.
- (19) Huber, M., "**Designing perfect simulation algorithms using local correctness**", arXiv:1907.06748v1 [cs.DS], 2019.
- (20) Lee, A., Doucet, A. and Łatuszyński, K., 2014. "**Perfect simulation using atomic regeneration with application to Sequential Monte Carlo**", arXiv:1407.5770v1 [stat.CO].
- (21) Mossel, Elchanan, and Yuval Peres. New coins from old: computing with unknown bias. Combinatorica, 25(6), pp.707-724.
- (22) Smith, N. A. and Johnson, M. (2007). Weighted and probabilistic context-free grammars are equally expressive. Computational Linguistics, 33(4):477–491.
- (23) Icard, Thomas F., "Calibrating generative models: The probabilistic Chomsky–Schützenberger hierarchy." Journal of Mathematical Psychology 95 (2020): 102308.
- (24) Thomas, A.C., Blanchet, J., "**A Practical Implementation of the Bernoulli Factory**", arXiv:1106.2508v3 [stat.AP], 2012.
- (25) Holtz, O., Nazarov, F., Peres, Y., "New Coins from Old, Smoothly", *Constructive Approximation* 33 (2011).
- (26) 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.
- (27) Wästlund, J., "**Functions arising by coin flipping**", 1999.
- (28) Qian, W. and Riedel, M.D., 2008, June. The synthesis of robust polynomial arithmetic with stochastic logic. In 2008 45th ACM/IEEE Design Automation Conference (pp. 648-653). IEEE.
- (29) Weikang Qian, Marc D. Riedel, Ivo Rosenberg, "Uniform approximation and Bernstein polynomials with coefficients in the unit interval", *European Journal of Combinatorics* 32(3), 2011, **https://doi.org/10.1016/j.ejc.2010.11.004** **http://www.sciencedirect.com/science/article/pii/S0195669810001666**
- (30) S. Ray, P.S.V. Nataraj, "A Matrix Method for Efficient Computation of Bernstein Coefficients", Reliable Computing 17(1), 2012.
- (31) And this shows that the polynomial couldn't be simulated if c were allowed to be 1, since the required degree would be infinity; in fact, the polynomial would touch 1 at the point 0.5 in this case, ruling out its simulation by any algorithm (see "About Bernoulli Factories", earlier).
- (32) Niazadeh, R., Leme, R.P., Schneider, J., "**Combinatorial Bernoulli Factories: Matchings, Flows, and Polytopes**", arXiv:2011.03865v1 [cs.DS], Nov. 7, 2020.
- (33) A *pushdown automaton*, as used here, is defined in Mossel and Peres 2005 and is described as a machine that maintains a stack of symbols and transitions from one state to another based on the current state, the symbol at the top of the stack, and the outcome of a biased coin flip. With each state transition, the machine adds symbols to the stack or removes symbols from it. When the stack is empty, the machine halts, and the result is 0 or 1 depending on the machine's state at that time.
- (34) The probability given in Theorem 3.2 of the Flajolet paper, namely just "$\Sigma_{k = 0, 1, 2, \ldots} (W(k) * (\lambda/2)^k)$", appears to be incorrect in conjunction with Figure 4 of that paper.
- (35) Here, "choose(g, g/t)" means that out of g letters, g/t of them must be A's, and "$(\beta-1)^{g-g/t}$" is the number of words that have $g-g/t$ letters other than A, given that the remaining letters were A's.
- (36) Brassard, G., Devroye, L., Gravel, C., "Remote Sampling with Applications to General Entanglement Simulation", *Entropy* 2019(21)(92), **https://doi.org/10.3390/e21010092** .
- (37) Bill Gosper, "Continued Fraction Arithmetic", 1978.
- (38) Borwein, J. et al. “Continued Logarithms and Associated Continued Fractions.” *Experimental Mathematics* 26 (2017): 412 - 429.
- (39) Canonne, C., Kamath, G., Steinke, T., "**The Discrete Gaussian for Differential Privacy**", arXiv:2004.00010 [cs.DS], 2020.
- (40) Forsythe, G.E., "Von Neumann's Comparison Method for Random Sampling from the Normal and

Other Distributions", *Mathematics of Computation* 26(120), October 1972.
- (41) Citterio, M., Pavani, R., "A Fast Computation of the Best *k*-Digit Rational Approximation to a Real Number", *Mediterranean Journal of Mathematics* 13 (2016).
- (42) Mendo, L., "**Simulating a coin with irrational bias using rational arithmetic**", arXiv:2010.14901 [math.PR], 2020.
- (43) Sondow, Jonathan. "New Vacca-Type Rational Series for Euler's Constant and Its 'Alternating' Analog ln 4/*π*.", 2005.
- (44) The error term, which follows from *Taylor's theorem*, has a numerator of 2 because 2 is higher than the maximum value at the point 1 (in cosh(1)) that *f*'s slope, slope-of-slope, etc. functions can achieve.
- (45) Kozen, D., **"Optimal Coin Flipping"**, 2014.
- (46) K. Bringmann, F. Kuhn, et al., "Internal DLA: Efficient Simulation of a Physical Growth Model." In: *Proc. 41st International Colloquium on Automata, Languages, and Programming (ICALP'14)*, 2014.
- (47) Flegal, J.M., Herbei, R., "Exact sampling from intractible probability distributions via a Bernoulli factory", *Electronic Journal of Statistics* 6, 10-37, 2012.
- (48) Powell, M.J.D., *Approximation Theory and Methods*, 1981.
- (49) Daumas, M., Lester, D., Muñoz, C., "**Verified Real Number Calculations: A Library for Interval Arithmetic**", arXiv:0708.3721 [cs.MS], 2007.
- (50) Lorentz, G.G., *Bernstein Polynomials*, 1953.
- (51) Sikkema, P.C., "Der Wert einiger Konstanten in der Theorie der Approximation mit Bernstein-Polynomen", Numer. Math. 3 (1961).
- (52) Gal, S.G., "Constructive approximation by monotonous polynomial sequences in LipMα, with α ∈ (0, 1]", *Journal of Approximation Theory* 59 (1989).
- (53) von Neumann, J., "Various techniques used in connection with random digits", 1951.
- (54) Pae, S., "Random number generation using a biased source", dissertation, University of Illinois at Urbana-Champaign, 2005.
- (55) Peres, Y., "Iterating von Neumann's procedure for extracting random bits", Annals of Statistics 1992,20,1, p. 590-597.
- (56) Estimating λ as λ', then finding $f(\lambda')$, is not necessarily an unbiased estimator of $f(\lambda)$, even if λ' is an unbiased estimator. Indeed, even though standard deviation equals the square root of variance, taking the square root of the bias-corrected sample variance does not lead to an unbiased estimator of the standard deviation.
- (57) Glynn, P.W., "Exact simulation vs exact estimation", *Proceedings of the 2016 Winter Simulation Conference*, 2016.
- (58) Devroye, L., Gravel, C., "**Random variate generation using only finitely many unbiased, independently and identically distributed random bits**", arXiv:1502.02539v6 [cs.IT], 2020.
- (59) Flajolet, P., Sedgewick, R., *Analytic Combinatorics*, Cambridge University Press, 2009.
- (60) Monahan, J.. "Extensions of von Neumann's method for generating random variables." Mathematics of Computation 33 (1979): 1065-1069.

9 Appendix

9.1 Randomized vs. Non-Randomized Algorithms

A *non-randomized algorithm* is a simulation algorithm that uses nothing but the input coin as a source of randomness (in contrast to *randomized algorithms*, which do use other sources of randomness) (Mendo 2019)**(7)**. Instead of generating outside randomness, a randomized algorithm can implement a ***randomness extraction*** procedure to generate that randomness using the input coins themselves. In this way, the algorithm becomes a *non-randomized algorithm*. For example, if an algorithm implements the **two-coin special case** by generating a random bit in step 1, it could replace generating that bit

with flipping the input coin twice until the flip returns 0 then 1 or 1 then 0 this way, then taking the result as 0 or 1, respectively (von Neumann 1951)[53]. A non-randomized algorithm works only if the probability of heads of any of the input coins is known to lie in the interval (0, 1).

In fact, there is a lower bound on the average number of coin flips needed to turn a coin with one probability of heads of (λ) into a coin with another ($\tau = f(\lambda)$). It's called the *entropy bound* (see, e.g., (Pae 2005)[54], (Peres 1992)[55]) and is calculated as—

$$((\tau - 1) * \ln(1 - \tau) - \tau * \ln(\tau)) / ((\lambda - 1) * \ln(1 - \lambda) - \lambda * \ln(\lambda)).$$

For example, if $f(\lambda)$ is a constant, non-randomized algorithms will generally require a growing number of coin flips to simulate that constant if the input coin is strongly biased towards heads or tails (the probability of heads is λ). Note that this formula only works if nothing but coin flips is allowed as randomness.

For certain values of λ, Kozen (2014)[45] showed a tighter lower bound of this kind, but this bound is generally non-trivial and assumes λ is known. However, if λ is 1/2 (the input coin is unbiased), this bound is simple: at least 2 flips of the input coin are needed on average to simulate a known constant τ, except when τ is a multiple of $1/(2^n)$ for any integer n.

9.2 Simulating Probabilities vs. Estimating Probabilities

A Bernoulli factory or another algorithm that produces heads with a given probability acts as an unbiased estimator for that probability that produces estimates in [0, 1] almost surely (Łatuszyński et al. 2009/2011)[8]. As a result, the probability $f(\lambda)$ can be simulated in theory by—

1. finding in some way an unbiased estimate of $f(\lambda)$, where $f(.)$ is a factory function and λ is the input coin's probability of heads;[56]
2. generating a uniform random number in [0,1], call it u; and
3. returning 1 if u is less than v, or 0 otherwise.

In practice, however, this method is prone to numerous errors, and they include errors due to the use of fixed precision in steps 1 and 2, such as rounding and cancellations. For this reason and also because "exact sampling" is the focus of this page, this page does not cover algorithms that directly estimate λ or $f(\lambda)$. See also (Mossel and Peres 2005, section 4.3)[21].

As also shown in (Łatuszyński et al. 2009/2011)[8], however, if $f(\lambda)$ can't serve as a factory function, it's not possible to build an unbiased estimator of that function which produces estimates in [0, 1] almost surely, since simulating that function isn't possible. For example, function A can't serve as a factory function, so no simulator for that function (and no unbiased estimator of the kind just given) is possible. This *is* possible for function B, however (Keane and O'Brien 1994)[2].

- Function A: $2 * \lambda$, when λ lies in (0, 1/2).
- Function B: $2 * \lambda$, when λ lies in $(0, 1/2 - \epsilon)$, where ϵ is in (0, 1/2).

Glynn (2016)[57] distinguishes between—

- *exact simulation*, or generating random numbers with the same *distribution* as that of $g(X)$ (same "shape", location, and scale of probabilities) in almost surely finite time, where $g(X)$ is a random value that follows the desired distribution, based on random numbers X, and
- *exact estimation*, or generating random numbers with the same *expected value* as that of $g(X)$ (that is, building an estimator of $g(X)$ that is *unbiased* and not merely *consistent* or *asymptotically unbiased*) in almost surely finite time.

Again, the focus of this page is "exact sampling" (*exact simulation*), not "exact estimation", but the input coin with probability of heads of λ can be any "exact estimator" of λ (as defined above) that outputs either 0 or 1.

9.3 Convergence of Bernoulli Factories

The following Python code illustrates how to test a Bernoulli factory algorithm for convergence to the correct probability, as well as the speed of this convergence. In this case, we are testing the Bernoulli factory algorithm of $x^{y/z}$, where x is in the interval (0, 1) and y/z is greater than 0. Depending on the parameters x, y, and z, this Bernoulli factory converges faster or slower.

```
<h1>Parameters for the Bernoulli factory x**(y/z)</h1>

x=0.005 # x is the input coin's probability of heads
y=2
z=3
<h1>Print the desired probability</h1>

print(x**(y/z))
passp = 0
failp = 0
<h1>Set cumulative probability to 1</h1>

cumu = 1
iters=4000
for i in range(iters):
  # With probability x, the algorithm returns 1 (heads)
  prob=(x);prob*=cumu; passp+=prob; cumu-=prob
  # With probability (y/(z*(i+1))), the algorithm returns 0 (tails)
  prob=(y/(z*(i+1)));prob*=cumu; failp+=prob; cumu-=prob
  # Output the current probability in this iteration,
  # but only for the first 30 and last 30 iterations
  if i<30 or i>=iters-30: print(passp)
```

As this code shows, as x (the probability of heads of the input coin) approaches 0, the convergence rate gets slower and slower, even though the probability will eventually converge to the correct one. In fact, when y/z is less than 1:

- The average number of coin flips needed by this algorithm will grow without bound as x approaches 0, and Mendo (2019)[7] showed that this is a lower bound; that is, no Bernoulli factory algorithm can do much better without knowing more information on x.
- $x^{y/z}$ has a slope that tends to a vertical slope near 0, so that the so-called ***Lipschitz condition*** is not met at 0. And (Nacu and Peres 2005, propositions 10 and 23)[5] showed that the Lipschitz condition is necessary for a Bernoulli factory to have an upper bound on the average running time.

Thus, a practical implementation of this algorithm may have to switch to an alternative

implementation (such as the one described in the next section) when it detects that the first few digits (after the point) of the uniform random number's fractional part are zeros.

9.4 Alternative Implementation of Bernoulli Factories

Say we have a Bernoulli factory algorithm that takes a coin with probability of heads of p and outputs 1 with probability $f(p)$. If this algorithm takes a uniform partially-sampled random number (PSRN) as the input coin and flips that coin using **SampleGeometricBag** (a method described in my **article on PSRNs**), the algorithm could instead be implemented as follows in order to return 1 with probability $f(U)$, where U is the number represented by the uniform PSRN (see also (Brassard et al., 2019)**(36)**, (Devroye 1986, p. 769)**(9)**, (Devroye and Gravel 2020)**(58)**. This algorithm assumes the uniform PSRN's sign is positive and its integer part is 0.

1. Set v to 0 and k to 1.
2. 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 $\text{abs}(fmin - fmax) <= 2 * b^{-k}$, 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 π, b is 10, and k is 2, pk is 314.
5. If $pk + 1 <= v$, return 0. If $pk - 2 >= v$, return 1. If neither is the case, add 1 to k and go to step 2.

However, the focus of this article is on algorithms that don't rely on calculations of irrational numbers, which is why this section is in the appendix.

9.5 SymPy Code for Parameters to Simulate C^2 Functions

A note in the section "**General Factory Functions**" gave a method to find polynomials that converge from above and below to a function with continuous slope and slope-of-slope functions, also known as a C^2 continuous function. The following Python code uses the SymPy computer algebra library to calculate three needed parameters given a C^2 continuous function `func` that uses the variable `x`: m (`m`) and the two bounds for **fbound(*n*)** (`bound1` and `bound2`, respectively).

```
def rminimum(f,x):
  return -rmaximum(-f,x)

def rmaximum(f,x): # Try maximum, and fall back if it fails
  try:
     return maximum(f,x,Interval(0, 1))
  except:
     # 0.1 is added below as a bias
     return (f).subs(x, nsolve(diff(f), (0,1)))+0.1
```

```
d=diff(diff(func))
m=Max(rmaximum(-d,x),rmaximum(d,x))
bound1=rminimum(func,x)-m/(n*8)
bound2=rmaximum(func,x)+m/(n*8)
```

9.6 Correctness Proof for the Continued Logarithm Simulation Algorithm

Theorem. *The algorithm given in "Continued Logarithms" returns 1 with probability exactly equal to the number represented by the continued logarithm c, and 0 otherwise.*

Proof. This proof of correctness takes advantage of Huber's "fundamental theorem of perfect simulation" (Huber 2019)**(19)**. Using Huber's theorem requires proving two things:

- First, we note that the algorithm clearly halts almost surely, since step 1 will stop the algorithm if it reaches the last coefficient, and step 2 always gives a chance that the algorithm will return a value, even if it's called recursively or the number of coefficients is infinite. Thus, the chance the algorithm has to be called recursively or with more iterations shrinks and shrinks as the algorithm does more recursions and iterations.
- Second, we show the algorithm is locally correct when the recursive call in step 3 is replaced with an oracle that simulates the correct "continued sub-logarithm". If step 1 reaches the last coefficient, the algorithm obviously passes with the correct probability. Otherwise, we will be simulating the probability $(1 / 2^{c[i]}) / (1 + x)$, where x is the "continued sub-logarithm" and will be at most 1 by construction. Steps 2 and 3 define a loop that divides the probability space into three pieces: the first piece takes up one half, the second piece (step 3) takes up a portion of the other half (which here is equal to $x/2$), and the last piece is the "rejection piece" that reruns the loop. Since this loop changes no variables that affect later iterations, each iteration acts like an acceptance/rejection algorithm already proved to be a perfect simulator by Huber. The algorithm will pass at step 2 with probability $p = (1 / 2^{c[i]}) / 2$ and fail either at step 2 with probability $f1 = (1 - 1 / 2^{c[i]}) / 2$, or at step 3 with probability $f2 = x/2$ (all these probabilities are relative to the whole iteration). Finally, dividing the passes by the sum of passes and fails ($p / (p + f1 + f2)$) leads to $(1 / 2^{c[i]}) / (1 + x)$, which is the probability we wanted.

Since both conditions of Huber's theorem are satisfied, this completes the proof. □

9.7 Correctness Proof for Continued Fraction Simulation Algorithm 3

Theorem. *Suppose a generalized continued fraction's partial numerators are b[i] and all greater than 0, and its partial denominators are a[i] and all greater than 0, and suppose further that each b[i]/a[i] is 1 or less. Then the algorithm given as Algorithm 3 in "Continued Fractions" returns 1 with probability exactly equal to the number represented by that continued fraction, and 0 otherwise.*

Proof. We use Huber's "fundamental theorem of perfect simulation" again in the proof of correctness.

- The algorithm halts almost surely for the same reason as the similar continued logarithm simulator.

- If the call in step 3 is replaced with an oracle that simulates the correct "sub-fraction", the algorithm is locally correct. If step 1 reaches the last element of the continued fraction, the algorithm obviously passes with the correct probability. Otherwise, we will be simulating the probability $b[i] / (a[i] + x)$, where x is the "continued sub-fraction" and will be at most 1 by assumption. Steps 2 and 3 define a loop that divides the probability space into three pieces: the first piece takes up a part equal to $h = a[i]/(a[i] + 1)$, the second piece (step 3) takes up a portion of the remainder (which here is equal to $x * (1 - h)$), and the last piece is the "rejection piece". The algorithm will pass at step 2 with probability $p = (b[i] / a[pos]) * h$ and fail either at step 2 with probability $f1 = (1 - b[i] / a[pos]) * h$, or at step 3 with probability $f2 = x * (1 - h)$ (all these probabilities are relative to the whole iteration). Finally, dividing the passes by the sum of passes and fails leads to $b[i] / (a[i] + x)$, which is the probability we wanted, so that both of Huber's conditions are satisfied and we are done. □

9.8 The von Neumann Schema

(Flajolet et al., 2010)[1] describes what it calls the *von Neumann schema* (sec. 2). Although the von Neumann schema is used in several Bernoulli factories given here, it's not a Bernoulli factory itself since it could produce random numbers other than 0 and 1, which is why this section appears in the appendix. Given a permutation class and an input coin, the von Neumann schema generates a random integer n, 0 or greater, with probability equal to—

- $(\lambda^n * \mathrm{V}(n) / n!) / \mathrm{EGF}(\lambda)$,

where—

- $\mathrm{EGF}(\lambda) = \Sigma_{k = 0, 1, ...} (\lambda^k * \mathrm{V}(k) / k!)$ (the *exponential generating function* or EGF, which completely determines a permutation class), and
- $\mathrm{V}(n)$ is a number in the interval $[0, n!]$ and is the number of permutations of size n that meet the requirements of the permutation class in question.

Effectively, a random number G is generated by flipping the coin until it returns 0 and counting the number of ones (the paper calls G a *geometric*(λ) random number, but this terminology is avoided in this article because it has several conflicting meanings in academic works), and then accepted with probability $\mathrm{V}(G)/(G!)$ and rejected otherwise. The probability that r random numbers are rejected this way is $p*(1 - p)^r$, where $p = (1 - \lambda) * \mathrm{EGF}(\lambda)$.

Examples of permutation classes include—

- single-cycle permutations ($\mathrm{EGF}(\lambda) = \mathrm{Cyc}(\lambda) = \ln(1/(1 - \lambda))$; $\mathrm{V}(n) = (n - 1)!$)
- sorted permutations, or permutations whose numbers are sorted in descending order ($\mathrm{EGF}(\lambda) = \mathrm{Set}(\lambda) = \exp(\lambda)$; $\mathrm{V}(n) = 1$),
- all permutations ($\mathrm{EGF}(\lambda) = \mathrm{Seq}(\lambda) = 1/(1 - \lambda)$; $\mathrm{V}(n) = n!$),
- alternating permutations of even size ($\mathrm{EGF}(\lambda) = 1/\cos(\lambda)$; the $\mathrm{V}(n)$ starting at $n = 0$ is **A000364** in the *On-Line Encyclopedia of Integer Sequences*), and
- alternating permutations of odd size ($\mathrm{EGF}(\lambda) = \tan(\lambda)$; the $\mathrm{V}(n)$ starting at $n = 0$ is **A000182**),

using the notation in "Analytic Combinatorics" (Flajolet and Sedgewick 2009)[59].

The following algorithm generates a random number that follows the von Neumann schema.

1. Set r to 0. (This is the number of times the algorithm rejects a random number.)
2. Flip the input coin until the flip returns 0. Then set G to the number of times the flip returns 1 this way.
3. With probability $\mathrm{V}(G)/G!$, return G (or r if desired). (In practice, the probability check is done by generating G uniform(0, 1) random numbers and determining whether those numbers satisfy the given permutation class, or generating as many of those numbers as necessary to make this determination. This is especially because $G!$, the factorial of G, can easily become very large.)
4. Add 1 to r and go to step 2.

A variety of Bernoulli factory probability functions can arise from the von Neumann schema, depending on the EGF and which values of G and/or r the Bernoulli factory algorithm treats as heads or tails. The following Python functions use the SymPy computer algebra library to find probabilities and other useful information for applying the von Neumann schema, given a permutation class's EGF.

```
def coeffext(f, x, power):
    # Extract a coefficient from a generating function
    # NOTE: Can also be done with just the following line:
    # return diff(f,(x,power)).subs(x,0)/factorial(power)
    px = 2
    for i in range(10):
      try:
        poly=Poly(series(f, x=x, n=power+px).removeO())
        return poly.as_expr().coeff(x, power)
      except:
        px+=2
    # Failed, assume 0
    return 0

def number_n_prob(f, x, n):
    # Probability that the number n is generated
    # for the von Neumann schema with the given
    # exponential generating function (e.g.f.)
    # Example: number_n_prob(exp(x),x,1) --> x**exp(-x)
    return (x**n*coeffext(f, x, n))/f

def r_rejects_prob(f, x, r):
    # Probability that the von Neumann schema
    # with the given e.g.f. will reject r random numbers
    # before accepting the next one
    p=(1-x)*f
    return p*(1-p)**r

def valid_perm(f, x, n):
    # Number of permutations of size n that meet
    # the requirements of the permutation class
    # determined by the given e.g.f. for the
    # von Neumann schema
    return coeffext(f, x, n)*factorial(n)
```

> **Note:** The von Neumann schema can simulate any *power series distribution* (such as Poisson, negative binomial, geometric, and logarithmic series), given a suitable exponential generating function. However, because of step 2, the number of input coin flips required by the schema grows without bound as λ approaches 1.
>
> **Example:** Using the class of *sorted permutations*, we can generate a Poisson(λ) random number via the von Neumann schema, where λ is the probability of heads of the input coin. This would lead to an algorithm for $\exp(-\lambda)$ — return 1

if a Poisson(λ) random number is 0, or 0 otherwise — but for the reason given in the note, this algorithm converges slowly as λ approaches 1.

A variation on the von Neumann schema occurs if G is generated differently than given in step 2, but is still generated by flipping the input coin. In that case, the algorithm above will return n with probability—

- $(\kappa(n; \lambda)*\mathrm{V}(n)/(n!)) / p$,

where $p = (\Sigma_{k=0,1,\ldots} (\kappa(k; \lambda)*\mathrm{V}(k)/(k!)))$, and where $\kappa(n; \lambda)$ is the probability that G is n, with parameter λ or the input coin's probability of heads. Also, the probability that r random numbers are rejected by the modified algorithm is $p*(1 - p)^r$.

Example: If G is a Poisson($z^2/4$) random number and the sorted permutation class is used, the algorithm will return 0 with probability $1/I_0(z)$, where $I_0(.)$ is the modified Bessel function of the first kind.

9.9 Probabilities Arising from Certain Permutations

Certain interesting probability functions can arise from permutations, such as permutations that are sorted or permutations whose highest number appears first.

Inspired by the **von Neumann schema** given earlier in this appendix, we can describe an algorithm that produces a random number given a permutation class as follows:

1. Create an empty list.
2. Generate a uniform(0, 1) random number u, and append u to the end of the list.
3. Let n be the number of items in the list minus 1. If the items in the list do not form a permutation that meets the permutation class's requirements, return n. Otherwise, go to step 2.

This algorithm returns the number n with the following probability:

$$G(n) = (1 - V(n + 1)/(V(n) * (n + 1))) * (1 - \Sigma_{j = 0, \ldots, n - 1} G(j))$$
$$= (V(n) * (n + 1) - V(n + 1)) / (V(0) * (n + 1)!),$$

where $V(n)$ is the number of permutations of size n that meet the permutation class's requirements. For this algorithm, $V(n)$ must be in the interval (0, n!] (thus, for example, this formula won't work if there are 0 permutations of odd size). $V(n)$ can be a sequence associated with an *exponential generating function* (EGF) for the kind of permutation involved in the algorithm, and examples of EGFs were given in the section on the von Neumann schema. For example, the class of *alternating permutations* (permutations whose numbers alternate between low and high, that is, $X1 > X2 < X3 > \ldots$) uses the EGF $\tan(\lambda)+1/\cos(\lambda)$.

For this algorithm, the probability that the generated n—

- is odd is $1 - 1 / \mathrm{EGF}(1)$, or
- is even is $1 / \mathrm{EGF}(1)$, or
- is less than k is $(V(0) - V(k)/(k!)) / V(0)$.

This algorithm can also be used to produce continuous random numbers, which will depend on the EGF (permutation class), which return values of n we care about, and so on. Specifically, consider the following algorithm:

1. Create an empty list.

2. If the list is empty, generate a random number distributed as D, call it δ. Otherwise, generate a random number distributed as E. Either way, append the random number to the end of the list. (In this step, D and E are both continuous distributions.)
3. Let n be the number of items in the list minus 1. If the items in the list do not form a permutation that meets the permutation class's requirements, return n. Otherwise, go to step 2.

Then the algorithm's behavior is given in the tables below.

Permutation Class	**Distribution *D***	**Distribution *E***	**The algorithm returns *n* with this probability:**	**The probability that *n* is ...**
Numbers sorted in descending order	Uniform(0,1)	Uniform(0,1)	n / ((n + 1)!).	Odd is $1-\exp(-1)$. Even is $\exp(-1)$.
Numbers sorted in descending order	Any	Any	$(\int_{(-\infty,\infty)}$ DPDF(z) * (ECDF(z)$^{n-1}$/((n−1)!) − ECDF(z)^n/(n!)) dz), for all $n > 0$ (see also proof of Theorem 2.1 of (Devroye 1986, Chapter IV)**(9)**. DPDF and ECDF are defined later.	Odd is denominator of formula 1 below.
Alternating numbers	Uniform(0,1)	Uniform(0,1)	(a_n * (n + 1) − a_{n+1}) / (n + 1)!, where a_i is the integer at position i (starting at 0) of the sequence **A000111** in the *On-Line Encyclopedia of Integer Sequences*.	Odd is $1-\cos(1)/(\sin(1)+1)$. Even is $\cos(1)/(\sin(1)+1)$.
Any	Uniform(0,1)	Uniform(0,1)	$(\int_{[0,1]} 1$ * (z^{n-1}*V(n)/((n−1)!) − z^n*V(n+1)/(n!)) dz), for all $n > 0$.	Odd is $1 - 1 /$ EGF(1).

Permutation Class	**Distribution *D***	**Distribution *E***	**The probability that *δ* is less than *x* given that *n* is ...**
Numbers sorted in descending order	Any	Any	Odd is $\psi(x) = (\int_{(-\infty, x)} \exp(-\text{ECDF}(z)) * \text{DPDF}(z)\ dz) / (\int_{(-\infty, \infty)} \exp(-\text{ECDF}(z)) * \text{DPDF}(z)\ dz)$ (Formula 1; see Theorem 2.1(iii) of (Devroye 1986, Chapter IV)**(9)**; see also Forsythe 1972**(40)**). Here, DPDF is the probability density function (PDF) of D, and ECDF is the cumulative distribution function (CDF) of E. If x is uniform(0, 1), this probability becomes $\int_{[0,1]} \psi(z)\ dz$.
Numbers sorted in descending order	Any	Any	Even is $(\int_{(-\infty, x)} (1 - \exp(-\text{ECDF}(z))) * \text{DPDF}(z)\ dz) / (\int_{(-\infty, \infty)} (1 - \exp(-\text{ECDF}(z))) * \text{DPDF}(z)\ dz)$ (Formula 2; see also Monahan 1979**(60)**). DPDF and ECDF are as above.

Numbers sorted in descending order	Uniform(0,1)	Uniform(0,1)	Odd is ((1−exp(−x))−exp(1))/(1−exp(1)). Therefore, the distribution of δ is exponential(1) and "truncated" to the interval [0, 1] (von Neumann 1951)**(53)**.
Numbers sorted in descending order	Uniform(0,1)	Max. of two uniform(0,1)	Odd is erf(x)/erf(1) (uses Formula 1, where DPDF(z) = 1 and ECDF(z) = z^2 for z in [0, 1]; see also **erf(x)/erf(1)**).

> **Note:** All the functions possible for formulas 1 and 2 are nondecreasing functions. Both formulas express the cumulative distribution function $F_D(x \mid n$ is odd) or $F_D(x \mid n$ is even), respectively.

Open Question: How can the tables above be filled for other permutation classes and different combinations of distributions D and E?

9.10 Sketch of Derivation of the Algorithm for 1 / π

The Flajolet paper presented an algorithm to simulate 1 / π but provided no derivation. Here is a sketch of how this algorithm works.

The algorithm is an application of the **convex combination** technique. Namely, 1 / π can be seen as a convex combination of two components:

- $g(n)$: $2^{6 * n} * (6 * n + 1) / 2^{8 * n + 2} = 2^{-2 * n} * (6 * n + 1) / 4 = (6 * n + 1) / (2^{2 * n + 2})$, which is the probability that the sum of the following independent random numbers equals n:
  - Two random numbers that each express the number of failures before the first success, where the chance of a success is 1−1/4 (the paper calls these two numbers *geometric*(1/4) random numbers, but this terminology is avoided in this article because it has several conflicting meanings in academic works).
  - One Bernoulli(5/9) random number.

  This corresponds to step 1 of the convex combination algorithm and steps 2 through 4 of the 1 / π algorithm. (This also shows that there is an error in the identity for 1 / π given in the Flajolet paper: the "8 n + 4" should read "8 n + 2".)

- $h_n()$: (choose($n * 2$, n) / $2^{n * 2}$)[3], which is the probability of heads of the "coin" numbered n. This corresponds to step 2 of the convex combination algorithm and step 5 of the 1 / π algorithm.

  Notes:

  1. $9 * (n + 1) / (2^{2 * n + 4})$ is the probability that the sum of two independent random numbers equals n, where each of the two numbers expresses the number of failures before the first success and the chance of a success is 1−1/4.
  2. $p^m * (1 - p)^n *$ choose($n + m - 1$, $m - 1$) is the probability that the sum of m independent random numbers equals n (a *negative binomial distribution*), where each of the m numbers expresses the number of failures before the first success and the chance of a success is p.
  3. $f(z) * (1 - p) + f(z - 1) * p$ is the probability that the sum of two independent random numbers — a Bernoulli(p) number and an integer z

with probability mass function $f(.)$ — equals z.

9.11 Calculating Bounds for exp(1)

The following implements the parts of Citterio and Pavani's algorithm (2016)[41] 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 $\text{floor}(((10^d)-1-p[n+1])/p[n+2])$.
  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.

10 License