Bernoulli Factory Algorithms

This version of the document is dated 2023-04-08.

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 sample probabilities that are irrational numbers, using only random bits, which is related to the Bernoulli factory problem. This page is focused on methods that *exactly* sample 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: 68W20, 60-08, 60-04.

1 Introduction

Suppose a coin shows heads with an unknown probability, λ. The goal is to use that coin (and possibly also a fair coin) to build a "new" coin that shows heads with a probability that depends on λ, call it $f(\lambda)$. This is the *Bernoulli factory problem*.

This page:

- Catalogs algorithms to solve the Bernoulli factory problem for a wide variety of functions, algorithms known as *Bernoulli factories*. For many of these algorithms, step-by-step instructions are provided. (Many of these algorithms were suggested in (Flajolet et al., 2010)[1], but without step-by-step instructions in many cases.)
- Contains algorithms to exactly sample probabilities that are irrational numbers, which is related to the Bernoulli factory problem. (An *irrational number* is a number that can't be written as a ratio of two integers.) Again, many of these algorithms were suggested in (Flajolet et al., 2010)[2].
- Assumes knowledge of **computer programming and**

mathematics, but **little or no familiarity with calculus**.
- Is focused on methods that *exactly* sample the probability described, without introducing rounding errors or other errors beyond those already present in the inputs (and assuming that a source of independent and unbiased random bits is available).

The Python module ***bernoulli.py*** includes implementations of several Bernoulli factories. For extra notes, see: **Supplemental Notes for Bernoulli Factory Algorithms**

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.

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

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

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

Comments on other aspects of this document are welcome.

2 Contents

- **Algorithms**
  - **Implementation Notes**
  - **Algorithms for General Functions of λ**
    - **Certain Polynomials**
    - **Certain Rational Functions**
    - **Certain Power Series**
    - **General Factory Functions**
  - **Algorithms for General Irrational Constants**
    - **Digit Expansions**
    - **Continued Fractions**
    - **Continued Logarithms**
    - **Certain Algebraic Numbers**
    - **Certain Converging Series**
  - **Other General Algorithms**
    - **Convex Combinations**
    - **Bernoulli Race and Generalizations**
    - **Flajolet's Probability Simulation Schemes**
    - **Integrals**
  - **Algorithms for Specific Functions of λ**
    - **ExpMinus ($\exp(-z)$)**
    - **LogisticExp ($1 - \text{expit}(z/2^{prec})$)**
    - **$\exp(-(\lambda * z))$**
    - **$\exp(-\exp(m + \lambda))$**
    - **$\exp(-(m + \lambda)^k)$**
    - **$\exp(\lambda)*(1-\lambda)$**
    - **$(1 - \exp(-(m + \lambda))) / (m + \lambda)$**
    - **$\text{expit}(z)$ or $1-1/(1+\exp(z))$ or $\exp(z)/(1+\exp(z))$ or $1/(1+\exp(-z))$**
    - **$\text{expit}(z)*2 - 1$ or $\tanh(z/2)$**
    - **$\lambda*\exp(z) / (\lambda*\exp(z) + (1 - \lambda))$ or $\lambda*\exp(z) / (1 + \lambda*(\exp(z) - 1))$**
    - **$(1 + \exp(z - w)) / (1 + \exp(z))$**
    - **$1/(2^{m*(k + \lambda)})$ or $\exp(-(k + \lambda)*\ln(2^m))$**
    - **$1/(2^{(x/y)*(\lambda)})$ or $\exp(-(\lambda)*\ln(2^{x/y}))$**
    - **Two-Coin Algorithm ($c * \lambda * \beta / (\beta * (c * \lambda + d * \mu) - (\beta - 1) * (c + d))$)**
    - **$c * \lambda / (c * \lambda + d)$ or $(c/d) * \lambda / (1 + (c/d) * \lambda))$**
    - **$(d + \lambda) / c$**
    - **$d / (c + \lambda)$**
    - **$(d + \mu) / (c + \lambda)$**

- $(d + \mu) / ((d + \mu) + (c + \lambda))$
- $d^k / (c + \lambda)^k$, or $(d / (c + \lambda))^k$
- $1/(1+\lambda)$
- $1/(2 - \lambda)$
- $1/(1+(m+\lambda)^2)$
- $1 / (1 + (x/y)*\lambda)$
- $\lambda^{x/y}$
- $\mathrm{sqrt}(\lambda)$
- $\arctan(\lambda) / \lambda$
- $\arctan(\lambda) / \pi$
- $\arctan(\lambda)$
- $\cos(\lambda)$
- $\sin(\lambda*\mathrm{sqrt}(c)) / (\lambda*\mathrm{sqrt}(c))$
- $\sin(\lambda)$
- $\ln(1+\lambda)$
- $\ln(c+\lambda)/(c+\lambda)$
- $\arcsin(\lambda) + \mathrm{sqrt}(1 - \lambda^2) - 1$
- $\tanh(z)$
- Expressions Involving Polylogarithms
- $\min(\lambda, 1/2)$ and $\min(\lambda, 1-\lambda)$

◦ Algorithms for Specific Functions of λ (Probability-Sensitive)

- $\lambda + \mu$
- $\lambda - \mu$
- ϵ / λ
- μ / λ
- $\lambda * x/y$
- $(\lambda * x/y)^i$
- Linear Bernoulli Factories
- λ^μ
- $(1-\lambda)/\cos(\lambda)$
- $(1-\lambda) * \tan(\lambda)$
- $\ln((c + d + \lambda)/c)$
- $\arcsin(\lambda) / 2$

◦ Other Factory Functions

◦ Algorithms for Specific Constants

- $1 / \varphi$ (1 divided by the golden ratio)
- $\mathrm{sqrt}(2) - 1$
- $1/\mathrm{sqrt}(2)$
- $\tanh(1/2)$ or $(\exp(1) - 1) / (\exp(1) + 1)$

3 About Bernoulli Factories

A *Bernoulli factory* (Keane and O'Brien 1994)[3] 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.

- The Greek letter lambda (λ) represents the unknown probability of heads.
- The Bernoulli factory's outputs are statistically independent, and so are those of the input coin.
- Many Bernoulli factories also use a *fair coin* in addition to the input coin. A fair coin shows heads or tails with equal probability, and represents a source of randomness outside the input coin.
- A *factory function* is a known function that relates the old probability to the new one. Its domain is the *closed* interval [0, 1] or a subset of that interval, and maps an input in that interval to an output in that interval.

> **Example:** A Bernoulli factory algorithm can take a coin that returns heads with probability λ and produce a coin that returns heads with probability $\exp(-\lambda)$. In this example, $\exp(-\lambda)$ is the factory function.

Keane and O'Brien (1994)[4] showed that a function f that maps [0, 1] (or a subset of it) to [0, 1] admits a Bernoulli factory if and only if—

- f is constant on its domain, or
- f is continuous and polynomially bounded on its domain (polynomially bounded means that both $f(\lambda)$ and $1-f(\lambda)$ are not less than $\min(\lambda^n, (1-\lambda)^n)$ for some integer n).

The following shows some functions that are factory functions and some that are not. In the table below, ϵ is a number greater than 0 and less than 1/2.

Function $f(\lambda)$	Domain	Can f be a factory function?
0	[0, 1]	Yes; constant.
1	[0, 1]	Yes; constant.
1/2	(0, 1)	Yes; constant.
1/4 if $\lambda<1/2$,		

and 3/4 elsewhere	(0, 1)	No; discontinuous.
$2*\lambda$	[0, 1] or [0, 1/2)	No; not polynomially bounded since $f(\lambda)$ approaches 1 as λ approaches 1/2 (as opposed to 0 or 1).[5].
$1-2*\lambda$	[0, 1] or [0, 1/2)	No; not polynomially bounded since $f(\lambda)$ approaches 0 as λ approaches 1/2.
$2*\lambda$	$[0, 1/2-\epsilon]$	Yes; continuous and polynomially bounded on domain (Keane and O'Brien 1994)[6].
$\min(2 * \lambda, 1 - \epsilon)$	[0, 1]	Yes; continuous and polynomially bounded on domain (Huber 2014, introduction)[7].
0 if $\lambda = 0$, or $\exp(-1/\lambda)$ otherwise	(0, 1)	No; not polynomially bounded since it moves away from 0 more slowly than any polynomial.
ϵ if $\lambda = 0$, or $\exp(-1/\lambda) + \epsilon$ otherwise	(0, 1)	Yes; continuous, minimum greater than 0, maximum less than 1.

If f's domain includes 0 and/or 1 (so that the input coin is allowed to return 0 every time or 1 every time, respectively), then f can be a factory function only if—

1. the function is constant on its domain, or is continuous and polynomially bounded on its domain, and
2. $f(0)$ equals 0 or 1 whenever 0 is in the function's domain, and
3. $f(1)$ equals 0 or 1 whenever 1 is in the function's domain,

unless outside randomness (besides the input coin) is available.

4 Algorithms

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

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.

> **Notes:**
>
> 1. Most of the algorithms assume that a source of independent and unbiased random bits is available, in addition to the input coins. But in many cases, they can be implemented using nothing but those coins as a source of randomness. See the **appendix** for details.
> 2. Bernoulli factory algorithms that sample the probability $f(\lambda)$ act as unbiased estimators of $f(\lambda)$ (their "long run average" equals $f(\lambda)$). See the **appendix** for details.

4.1 Implementation Notes

This section shows implementation notes that apply to the algorithms in this article. They should be followed to avoid introducing error in the algorithms.

In the following algorithms:

- The Greek letter lambda (λ) represents the unknown probability of heads of the input coin.
- choose(n, k) = (1*2*3*...*n)/((1*...*k)*(1*...*($n-k$))) = n!/(k! * ($n-k$)!) $={n \choose k}$ is a *binomial coefficient*, or the number of ways to choose k out of n labeled items. It can be calculated, for example, by calculating $i/(n-i+1)$ for each integer i in $[n-k+1, n]$, then multiplying the results (Manolopoulos 2002)[8]. For every $m>0$, choose(m, 0) = choose(m, m) = 1 and choose(m, 1) = choose(m, $m-1$) = m; also, in this document, choose(n, k) is 0 when k is less than 0 or greater than n.
- n! = 1*2*3*...*n is also known as n factorial; in this document, (0!) = 1.
- *Summation notation*, involving the Greek capital sigma (Σ), is a way to write the sum of one or more terms of similar form. For example, $\sum_{k=0}^n g(k)$ means $g(0)+g(1)+...+g(n)$, and $\sum_{k\ge 0} g(k)$ means $g(0)+g(1)+...$.
- The instruction to "generate a uniform random variate between 0 and 1" 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 a uniform random variate greater than 0 and less than 1 (for example, `RNDRANGEMinMaxExc(0, 1)` in "**Randomization and Sampling Methods**" (less accurate).

- The instruction to "choose [integers] with probability proportional to [*weights*]" can be implemented in one of the following ways:

    - If the weights are rational numbers, take the result of **WeightedChoice**(**NormalizeRatios**(*weights*))), where **WeightedChoice** and **NormalizeRatios** are given in "**Randomization and Sampling Methods**".
    - If the weights are uniform PSRNs, use the algorithm given in "**Weighted Choice Involving PSRNs**".

    For example, "Choose 0, 1, or 2 with probability proportional to the weights [A, B, C]" means to choose 0, 1, or 2 at random so that 0 is chosen with probability A/(A+B+C), 1 with probability B/(A+B+C), and 2 with probability C/(A+B+C).

- Where an algorithm says "if a is less than b", where a and b are random variates, 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 an algorithm says "if a is less than (or equal to) b", where a and b are random variates, it means to run the **RandLess** algorithm on the two numbers (if they are both PSRNs), or do a less-than-or-equal operation on a and b, as appropriate.

- To **sample from a 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 a uniform random variate between 0 and 1 v (see above) and generating 1 if v is less than u (see above) or 0 otherwise.

- 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:
    - Generate a uniform random variate between 0 and 1 v (see above). The event occurs if v is less than x (see above).
    - Convert x to a rational number y/z, then call `ZeroOrOne(y, z)`. The event occurs if the call returns 1. 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**. If x is not a rational number, then rounding error will result, however.
- 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.

4.2 Algorithms for General Functions of λ

This section describes general-purpose algorithms for sampling probabilities that are polynomials, rational functions, or functions in general.

4.2.1 Certain Polynomials

Any polynomial can be written in *Bernstein form* as—

$${n\choose 0}\lambda^0 (1-\lambda)^{n-0} a[0] + {n\choose 1}\lambda^1 (1-\lambda)^{n-1} a[1] + ... + {n\choose n}\lambda^n (1-\lambda)^{n-n} a[n],$$

where n is the polynomial's *degree* and $a[0]$, $a[1]$, ..., $a[n]$ are its n plus one *coefficients*.

But a polynomial admits a Bernoulli factory only if each of its coefficients is 0 or greater and less than 1 (once the polynomial is written in Bernstein form), and a function can be simulated with a

fixed number of coin flips only if it's a polynomial of that kind (Goyal and Sigman 2012[9]; Qian et al. 2011)[10]; see also Wästlund 1999, section 4[11]).

Goyal and Sigman 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.[12]
2. Return a number that is 1 with probability $a[j]$, or 0 otherwise.

For certain polynomials with duplicate coefficients, the following is an optimized version of this algorithm, not given by Goyal and Sigman:

1. Set j to 0 and i to 0. If n is 0, return 0.
2. If i is n or greater, or if the coefficients $a[k]$, with k in the interval $[j, j+(n-i)]$, are all equal, return a number that is 1 with probability $a[j]$, or 0 otherwise.
3. Flip the input coin. If it returns 1, add 1 to j.
4. Add 1 to i and go to step 2.

And here is another optimized algorithm:

1. Set j to 0 and i to 0. If n is 0, return 0. Otherwise, generate a uniform random variate between 0 and 1, call it u.
2. If u is less than a lower bound of the lowest coefficient, return 1. Otherwise, if u is less than (or equal to) an upper bound of the highest coefficient, go to the next step. Otherwise, return 0.
3. If i is n or greater, or if the coefficients $a[k]$, with k in the interval $[j, j+(n-i)]$, are all equal, return a number that is 1 if u is less than $a[j]$, or 0 otherwise.
4. Flip the input coin. If it returns 1, add 1 to j.
5. Add 1 to i and go to step 3.

Because the coefficients $a[i]$ must be 0 or greater, but not greater than 1, some or all of them can themselves be coins with unknown probability of heads. In that case, the first algorithm can read as follows:

1. Flip the input coin n times, and let j be the number of times the coin returned 1 this way.
2. If $a[j]$ is a coin, flip it and return the result. Otherwise, return a number that is 1 with probability $a[j]$, or 0 otherwise.

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 exclusive-OR 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)[13], Qian and Riedel 2008[14]).
3. These algorithms can serve as an approximate way to simulate any function f that maps the interval [0, 1] to [0, 1], whether continuous or not. In this case, $a[j]$ is calculated as $f(j/n)$, so that the resulting polynomial closely approximates the function. In fact, if the function is continuous, it's possible to choose the polynomial degree (n) high enough to achieve a given maximum error (this is a result of the so-called "Weierstrass approximation theorem"). For more information, see my **Supplemental Notes on Bernoulli Factories**.

Examples:

1. Take the following parabolic function discussed in Thomas and Blanchet (2012)[15]: $(1-4*(\lambda-1/2)^2)*c$, where $0 < c < 1$. This is a polynomial of degree 2 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. Rewriting the polynomial in Bernstein form (such as via the matrix method by Ray and Nataraj (2012)[16]) leads to 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 $0 < c \le 1/2$, this function can be simulated 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."

    - If $1/2 < c < 1$, the algorithm requires rewriting the polynomial in Bernstein form, then elevating the degree of the rewritten polynomial enough times to bring its coefficients in [0, 1]; the required degree approaches infinity as c approaches 1.[17]

2. The *conditional* construction, mentioned in Flajolet et al. (2010)[18], has the form—
   $(\lambda) * a[0] + (1 - \lambda) * a[1]$.
   This is a degree-1 polynomial in Bernstein form with variable λ and coefficients $a[0]$ and $a[1]$. It has the following algorithm: "Flip the λ input coin. If the result is 0, flip the $a[0]$ input coin and return the result. Otherwise, flip the $a[1]$ input coin and return the result." Special cases of the conditional construction include complement, mean, product, and logical OR; see "**Other Factory Functions**".

Multiple coins. Niazadeh et al. (2021)[19] describes monomials (involving one or more coins) of the form $\lambda[1]^{a[1]} * (1-\lambda[1])^{b[1]} * \lambda[2]^{a[2]} * (1-\lambda[2])^{b[2]} * \ldots * \lambda[n]^{a[n]} * (1-\lambda[n])^{b[n]}$, where there are n coins, $\lambda[i]$ is the probability of heads of coin i, and $a[i] \geq 0$ and $b[i] \geq 0$ 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 $P =$ $\sum{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 $c[j] \geq 0$ is the weight for 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 "**Weighted Choice With Replacement**"), then run the algorithm

above on that monomial (see also "**Convex Combinations**", later).

The following is a special case:

- If there is only one coin, the polynomials P are in Bernstein form if $c[j]$ is $\alpha[j]$*choose($k-1, j-1$) where $\alpha[j]$ is a coefficient 0 or greater, but not greater than 1, and if $a[1] = j-1$ and $b[1] = k-j$ for each monomial j.

4.2.2 Certain Rational Functions

A *rational function* is a ratio of polynomials.

According to Mossel and Peres (2005)[20], a function that maps the open interval (0, 1) to itself can be simulated by a finite-state machine if and only if the function can be written as a rational function whose coefficients are rational numbers.

The following algorithm is suggested from the Mossel and Peres paper and from (Thomas and Blanchet 2012)[21]. It assumes the rational function is written as $D(\lambda)/E(\lambda)$, where—

- $D(\lambda)$ = $\sum_{i=0}^n$ $\lambda^i * (1 - \lambda)^{n - i} * d[i]$,
- $E(\lambda)$ = $\sum_{i=0}^n$ $\lambda^i * (1 - \lambda)^{n - i} * e[i]$,
- 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. (Because of the assumptions, D and E are polynomials that map the closed interval [0, 1] to itself.)

The algorithm follows.

1. Flip the input coin n times, and let *heads* be the number of times the coin returned 1 this way.
2. Choose 0, 1, or 2 with probability proportional to these weights: [$e[heads] - d[heads]$, $d[heads]$, choose(n, $heads$) $- e[heads]$]. If 0 or 1 is chosen this way, return it. 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 can be impractical (in general, a lookup table of size 2^n first has to be built in a setup step; as n grows, the table size grows exponentially). 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($e[i]$) (Nacu and Peres 2005)[22], but this suffers from rounding error when done in this algorithm). See also (Thomas and Blanchet 2012)[23].

3. As with polynomials, this algorithm (or the one given later) can serve as an approximate way to simulate any factory function, via a rational function that closely approximates that function. The higher n is, the better this approximation, and in general, a degree-n rational function approximates a given function better than a degree-n polynomial. However, to achieve a given error tolerance with a rational function, the degree n as well as $d[i]$ and $e[i]$ have to be optimized. This is unlike the polynomial case where only the degree n has to be optimized.

> **Example**: Take the function $f(\lambda) = 1/(\lambda-2)^2$. This is a rational function, in this case a ratio of two polynomials that are both nonnegative on the interval [0, 1]. One algorithm to simulate this function follows.
> (1) Flip the input coin twice, and let *heads* be the number of times the coin returned 1 this way.
> (2) Depending on *heads*, choose 0, 1, or 2 with probability proportional to the following weights: *heads*=0 → [3, 1, 0], *heads*=1 → [1, 1, 2], *heads*=2 → [0, 1, 3]; if 0 or 1 is chosen this way, return it; otherwise, go to step 1.
> Here is how f was prepared to derive this algorithm:
> (1) Take the numerator 1, and the denominator $(\lambda-2)^2$. Rewrite the denominator as $1*\lambda^2 - 4*\lambda + 4$.
> (2) Rewrite the numerator and denominator into homogeneous polynomials (polynomials whose terms have the same degree) of degree 2; see the "homogenizing" section in "**Preparing Rational Functions**". The result is (1, 2, 1) and (4, 4, 1) respectively.
> (3) Divide both polynomials (actually their coefficients) by the same value so that both polynomials are 1 or less. An easy (but not always best) choice is to divide them by their maximum coefficient, which is 4 in this case. The result is d = (1/4, 1/2, 1/4), e = (1, 1, 1/4).
> (4) Prepare the weights as given in step 2 of the original algorithm. The result is [3/4, 1/4, 0], [1/2, 1/2, 1], and [0, 1/4, 3/4], for different counts of heads. Because the weights in this case are multiples of 1/4, they can be simplified to integers without affecting the algorithm: [3, 1, 0], [1, 1, 2], [0, 1, 3], respectively.

"Dice Enterprise" special case. The following algorithm implements a special case of the "Dice Enterprise" method of Morina et al. (2022)[24]. The algorithm returns one of m outcomes (namely X, an integer in [0, m)) with probability $P_X(\lambda)$ / ($P_0(\lambda)$ + $P_1(\lambda)$ + ... + $P_{m-1}(\lambda)$), where λ is the input coin's probability of heads and m is 2 or greater. Specifically, the probability is a *rational function*, or ratio of polynomials. Here, all the $P_k(\lambda)$ are in the form of polynomials as follows:

- The polynomials are *homogeneous*, that is, they are written as $\sum_{i=0}^n $ $\lambda^i * (1 - \lambda)^{n-i} * a[i]$, where n is the polynomial's

degree and $a[i]$ is a coefficient.

- The polynomials have the same degree (namely n) and all $a[i]$ are 0 or greater.
- The sum of j^{th} coefficients is greater than 0, for each j starting at 0 and ending at n, except that the list of sums may begin and/or end with zeros. Call this list R. For example, this condition holds true if R is (2, 4, 4, 2) or (0, 2, 4, 0), but not if R is (2, 0, 4, 3).

Any rational function that admits a Bernoulli factory can be brought into the form just described, as detailed in the appendix under "**Preparing Rational Functions**". In this algorithm, let $R[j]$ be the sum of j^{th} coefficients of the polynomials (with j starting at 0). First, define the following operation:

- **Get the new state given *state*, *b*, *u*, and *n***:
  1. If *state* > 0 and *b* is 0, return either *state*−*1* if *u* is less than (or equal to) *PA*, or *state* otherwise, where *PA* is $R[state-1]/\max(R[state], R[state-1])$.
  2. If *state* < *n* and *b* is 1, return either *state*+*1* if *u* is less than (or equal to) *PB*, or *state* otherwise, where *PB* is $R[state+1]/\max(R[state], R[state+1])$.
  3. Return *state*.

Then the algorithm is as follows:

1. Create two empty lists: *blist* and *ulist*.
2. Set *state1* to the position of the first non-zero item in R. Set *state2* to the position of the last non-zero item in R. In both cases, positions start at 0. If all the items in R are zeros, return 0.
3. Flip the input coin and append the result (which is 0 or 1) to the end of *blist*. Generate a uniform random variate between 0 and 1 and append it to the end of *ulist*.
4. (Monotonic coupling from the past (Morina et al., 2022)[25], (Propp and Wilson 1996)[26].) Set i to the number of items in *blist* minus 1, then while i is 0 or greater:
   1. Let b be the item at position i (starting at 0) in *blist*, and let u be the item at that position in *ulist*.
   2. **Get the new state given *state1*, *b*, *u*, and *n***, and set *state1* to the new state.
   3. **Get the new state given *state2*, *b*, *u*, and *n***, and set *state2* to the new state.
   4. Subtract 1 from i.

5. If *state1* and *state2* are not equal, go to step 2.
6. Let $b(j)$ be coefficient $a[state1]$ of the polynomial for j. Choose an integer in $[0, m)$ with probability proportional to these weights: $[b(0), b(1), ..., b(m-1)]$. Then return the chosen integer.

> **Notes:**
>
> 1. If there are only two outcomes, then this is the special Bernoulli factory case; the algorithm would then return 1 with probability $P_1(\lambda) / (P_0(\lambda) + P_1(\lambda))$.
> 2. If $R[j] = \text{choose}(n, j)$, steps 1 through 5 have the same effect as counting the number of ones from n input coin flips (which would be stored in *state1* in this case), but unfortunately, these steps wouldn't be more efficient. In this case, *PA* is equivalent to "1 if *state* is greater than floor(n/2), and *state*/(n+1−*state*) otherwise", and *PB* is equivalent to "1 if *state* is less than floor(n/2), and (n−*state*)/(*state*+1) otherwise".
>
> **Example:** Let $P_0(\lambda) = 2*\lambda*(1-\lambda)$ and $P_1(\lambda) = (4*\lambda*(1-\lambda))^2/2$. The goal is to produce 1 with probability $P_1(\lambda) / (P_0(\lambda) + P_1(\lambda))$. **Preparing this function** (along with noting that the maximum degree is $n = 4$) results in the coefficient sums $R = (0, 2, 12, 2, 0)$. Since R begins and ends with 0, step 2 of the algorithm sets *state1* and *state2*, respectively, to the position of the first or last nonzero item, namely 1 or 3. (Alternatively, because R begins and ends with 0, a third polynomial is included, namely the constant $P_2(\lambda) = 0.001$, so that the new coefficient sums would be $R' = (0.001, 10.004, 12.006, 2.006, 0.001)$ [formed by adding the coefficient 0.001*choose(n, i) to the sum at i, starting at i = 0]. Now run the algorithm using R', and if it returns 2 [meaning that the constant polynomial was chosen], try again until the algorithm no longer returns 2.)

4.2.3 Certain Power Series

Some functions can be written as— $$f(\lambda) = a_0 (g(\lambda))^0 + a_1 (g(\lambda))^1 + ... + a_i (g(\lambda))^i + ...,\tag{1}$$ where a_i are *coefficients* and $g(\lambda)$ is a function in the variable λ. The right-hand side of (1) is called a *power series* as long as $g(\lambda) = \lambda$. A function writable as (1) will be called a *generalized power series* here. Not all power

series sum to a definite value, but all generalized power series that matter in this section do, and they must be Bernoulli factory functions. (In particular, $g(\lambda)$ must be a Bernoulli factory function, too.)

Depending on the coefficients, different algorithms can be built to simulate a generalized power series:

- The coefficients are arbitrary, but can be split into two parts.
- The coefficients alternate in sign, and their absolute values form a decreasing sequence.
- The coefficients are nonnegative and sum to 1 or less.
- The coefficients are nonnegative and may sum to 1 or greater.

> **Note:** In theory, the series (1) can contain coefficients that are irrational numbers or sum to an irrational number, but the algorithms for such series can be inexact in practice. Also, not all generalized power series that admit a Bernoulli factory are covered by the algorithms in this section. They include:
>
> - Series with coefficients that alternate in sign, but do not satisfy the **general martingale algorithm** or **Algorithm 1** below. This includes nearly all such series that equal 0 at 0 and 1 at 1, or equal 0 at 1 and 1 at 0. (An example is $\sin(\lambda\pi/2)$.)
> - Series with negative and positive coefficients that do not eventually alternate in sign (ignoring zeros).

Certain Alternating Series:

Suppose the following holds true for a generalized power series $f(\lambda)$:

- f is written as in equation (1).
- Suppose (a_i) is the sequence formed from the coefficients of the series.
- Let (d_j) be the sequence formed from (a_i) by deleting the zero coefficients. Then suppose that:
  - d_0 is greater than 0, and the elements in (d_j) alternate in sign (example: 1/2, -1/3, 1/4, -1/5, ...).
  - The absolute values of (d_j)'s elements are 1 or less and form a nowhere increasing sequence that is finite or converges to 0.

In addition, the coefficients should be rational numbers.

> **Example:** Let $f(\lambda) = (1/2)\lambda^0 - (1/4)\lambda^2 + (1/8)\lambda^4 - ...$. Then $(a_i) = (1/2, 0, -1/4, 0, 1/8, ...)$ (for example, $a_0 = 1/2$) and deleting the zeros leads to $(d_i) = (1/2, -1/4, 1/8, ...)$ (for example, $d_0 = 1/2$), which meets the requirements above.

Then the algorithm below, based on an algorithm by Łatuszyński et al. (2009/2011, especially section 3.1)[27], simulates $f(\lambda)$ given a coin that shows heads (returns 1) with probability $g(\lambda)$.

General martingale algorithm:

1. Set u to abs(d_0) (d_0 is the value of the first nonzero coefficient in the sequence (a_i)), set w to 1, set ℓ to 0, and set n to 1.
2. Generate a uniform random variate between 0 and 1 *ret*.
3. Do the following process repeatedly, until this algorithm returns a value:
    1. If w is not 0, run a Bernoulli factory algorithm for $g(\lambda)$ (if $g(\lambda) = \lambda$, this is done by flipping the input coin), then multiply w by the result of the run.
    2. If a_n is greater than 0: Set u to $\ell + w * a_n, then, if no further nonzero coefficients follow a_n, set ℓ to u.
    3. If a_n is less than 0: Set ℓ to $u - w * abs($a_n$), then, if no further nonzero coefficients follow a_n, set u to ℓ.
    4. If *ret* is less than (or equal to) ℓ, return 1. Otherwise, if *ret* is less than u, add 1 to n. Otherwise, return 0. (If *ret* is a uniform partially-sampled random number [PSRN], these comparisons should be done via the **URandLessThanReal algorithm**, which is described in my **article on PSRNs**.)

> **Notes:**
>
> 1. The **general martingale algorithm**, as it's called in this article, supports more functions than in section 3.1 of Łatuszyński et al. (2019/2011), which supports only functions writable as a power series whose coefficients alternate in sign and decrease in absolute value, with no zeros in between nonzero coefficients. However, the general martingale algorithm uses that paper's framework. A proof of its correctness is given in the appendix.
> 2. The **general martingale algorithm** allows the sequence (a_i) to sum to 1, but in this case, it seems that the

> sequence's nonzero values must have the form $(1, -z_0, z_0, -z_1, z_1, ..., -z_i, z_i, ...)$, where the z_i are positive, are no greater than 1, and form a nowhere increasing sequence that is finite or converges to 0. Moreover, it appears that every power series with this sequence of coefficients is less than or equal to λ.

General Power Series:

Suppose the following for a generalized power series $f(\lambda)$:

- f is written as in equation (1).
- There is a rational number Z defined as follows. For every λ that satisfies $0 \le \lambda \le 1$, it is true that $0 \le f(\lambda) \le Z \lt 1$.
- There is an even integer m defined as follows. The series in equation (1) can be split into two parts: the first part (A) is the sum of the first m terms, and the second part (C) is the sum of the remaining terms. Moreover, both parts admit a Bernoulli factory algorithm (see "**About Bernoulli Factories**" in the "Bernoulli Factory Algorithms" article). Specifically: $$C(\lambda) = \sum_{i\ge m} a_i (g(\lambda))^i, A(\lambda) = f(\lambda) - C(\lambda).$$ As an example, if C is a generalized power series described in the section "Certain Alternating Series", above, then C admits a Bernoulli factory algorithm, namely the **general martingale algorithm**.

In addition, the algorithm will be simpler if each coefficient a_i is a rational number.

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

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

Rewrite A as a polynomial in Bernstein form, in the variable $g(\lambda)$. (One way to transform a polynomial to Bernstein form, given the "power" coefficients $a_0, ..., a_{m-1}$, is the so-called "matrix method" from Ray and Nataraj (2012)[28].) Let $b_0, ..., b_{m-

1}$ be the Bernstein-form polynomial's coefficients. Then if those coefficients all lie in $[0, 1]$, then the following algorithm simulates $f(\lambda)$.

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

- **Sub-algorithm:** Generate an unbiased random bit. If that bit is 1, sample the polynomial A as follows (Goyal and Sigman 2012) [^MATMP2]:

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

  If the bit is 0, do the following:

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

Series with Non-Negative Coefficients Summing to 1 or Less:

Now, suppose $f(\lambda)$ can be written as in equation (1), at the beginning of this section, but this time, the *coefficients* a_i are 0 or greater and their sum is 1 or less.

If $g(\lambda) = \lambda$, this kind of function—

- satisfies $0\le f(\lambda)\le 1$ whenever $0 \le \lambda \le 1$,
- is either constant or strictly increasing, and
- is *convex* (its "slope" or "velocity" doesn't decrease as λ increases)[29].

Suppose f can be written as $f(\lambda)= f_0(g(\lambda))$, where— $$f_0(\lambda) = \sum_{n} a_n \lambda^n = \sum_{n} w(n) \frac{a_n}{w(n)}\lambda^n,$$ where each sum is taken over all nonnegative values of n where $a_n > 0$. (For notation details, see "**Implementation Notes**".)

Then the key to simulating $f(\lambda)$ is to "tuck" the values a_n under a function $w(n)$ such that—

- $1 \ge w(n)\ge a_n\ge 0$ for every allowed *n*, and
- $w(0)+w(1)+...=1$ (required for a valid distribution of integers 0 or greater).

> **Notes:**
>
> 1. Assuming $f_0(1)$ does not equal 0, an appropriate $w(n)$ is trivial to find — $w(n)=a_n/f_0(1)$ (because $a_n \le f_0(1)$ for every allowed n). But in general, this can make $w(n)$ an irrational number and thus harder to handle with arbitrary precision.
> 2. If the coefficients a_n sum to 1, then $w(n)$ can equal a_n. In this case, $f_0(\lambda)$ is what's called the *probability generating function* for getting X with probability a_X (or $w(X)$), and the expected value ("long-run average") of X equals the "slope" of $f_0(\lambda)$ at 1. See also (Dughmi et al. 2021)[30].
> 3. Assuming $f_0(1)$ is an irrational number, $w(n)$ can equal $a_n + c_n/2^n$, where c_n is the n-th base-2 digit after the point in the binary expansion of $1 - f_0(1)$ (or 0 if $n=0$). Here, a number's *binary expansion* is written as `0.bbbbb...` in base 2, where each `b` is a base-2 digit (either 0 or 1). See my **Stack Exchange question**.

Once a_n and $w(n)$ are found, the function $f(\lambda)$ can be simulated using the following algorithm, which takes advantage of the **convex combination method**.

Algorithm 2:

1. Choose at random an integer *n* that equals *i* with probability $w(i)$.
2. (The next two steps succeed with probability $\frac{a_n}{w(n)} (g(\lambda))^n$.) Let *P* be $a_n/w(n)$. With probability *P*, go to the next step. Otherwise, return 0.
3. (At this point, *n* equals *i* with probability a_i.) Run a Bernoulli factory algorithm for $g(\lambda)$, *n* times or until a run returns 0, whichever happens first. (For example, if $g(\lambda)=\lambda$, flip the input coin each time.) Return 1 if all the runs, including the last, returned 1 (or if *n* is 0). Otherwise, return 0.

Step 1 is rather general, and doesn't fully describe how to generate the value n at random. That depends on the function $w(n)$. See "**Power Series Examples**", later, for examples of generalized power series $f(\lambda)$ that can be simulated using Algorithm 2.

> **Note:** Part of **Algorithm 2** involves choosing X at random with probability $w(X)$, then doing X coin flips. Thus, the algorithm uses, on average, at least the number of unbiased random bits needed to generate X on average (Knuth and Yao 1976)[31].

Algorithm 2 covers an algorithm that was given by Luis Mendo (2019)[32] for simulating certain functions writable as power series, but that works only if the coefficients sum to 1 or less and only if coefficient 0 (a_0) is 0.

To get to an algorithm equivalent to Mendo's, first **Algorithm 2** is modified to simulate $f_0(\lambda)$/*CS* as follows, where *CS* is the sum of all coefficients a_i, starting with $i=1$. This shows Mendo's algorithm, like **Algorithm 2**, is actually a special case of the **convex combination algorithm**.

- Step 1 of **Algorithm 2** becomes: "(1a.) Set *dsum* to 0 and n to 1; (1b.) With probability a_n/(*CS* − *dsum*), go to step 2. Otherwise, add a_n to *dsum*; (1c.) Add 1 to *i* and go to step 1b." (Choose at random n with probability $w(n)=a_n$/*CS*.)
- Step 2 becomes "Go to step 3". (The *P* in **Algorithm 2** is not used; it's effectively $w(n)/\frac{a_n}{CS}=\frac{a_n}{CS}/\frac{a_n}{CS} = 1$.)
- In step 3, $g(\lambda)$ is either λ (flip the input coin) or $1-\lambda$ (flip the input coin and take 1 minus the flip).

Mendo's algorithm and extensions of it mentioned by him cover several variations of functions writable as power series as follows:

Type	Power Series	Algorithm
1	$f(\lambda)=1-f_0(1-\lambda)$	With probability *CS*, run the modified algorithm with $g(\lambda)=1-\lambda$ and return 1 minus the result. Otherwise, return 1.
		With probability *CS*, run the

2	$f(\lambda)=f_0(1-\lambda)$	modified algorithm with $g(\lambda)=1-\lambda$ and return the result. Otherwise, return 0.
3	$f(\lambda)=f_0(\lambda)$	With probability CS, run the modified algorithm with $g(\lambda)=\lambda$ and return the result. Otherwise, return 0.
4	$f(\lambda)=1-f_0(\lambda)$	With probability CS, run the modified algorithm with $g(\lambda)=\lambda$ and return 1 minus the result. Otherwise, return 1.

The conditions on f given above mean that—

- for series of type 1, $f(0) = 1-CS$ and $f(1) = 1$ (series of type 1 with CS=1 is the main form in Mendo's paper),
- for series of type 2, $f(0) = CS$ and $f(1) = 0$,
- for series of type 3, $f(0) = 0$ and $f(1) = CS$, and
- for series of type 4, $f(0) = 1$ and $f(1) = 1-CS$.

Series with General Non-Negative Coefficients:

If f is written as equation (1), in the beginning of this section, but—

- each of the coefficients is positive or zero, and
- the coefficients sum to greater than 1,

then Nacu and Peres (2005, proposition 16)[33] gave an algorithm which takes the following parameters:

- t is a rational number such that $B < t \leq 1$ and $f(t) < 1$.
- ϵ is a rational number such that $0 < \epsilon \leq (t - B)/2$.

B is not a parameter, but is the maximum allowed value for $g(\lambda)$ (probability of heads), and is greater than 0 and less than 1. The following algorithm is based on that algorithm, but runs a Bernoulli factory for $g(\lambda)$ instead of flipping the input coin with probability of heads λ.

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 − $a_n\cdot t^n$, return 0. (4) Run a **linear Bernoulli factory** n times, $x/y = 1/(t - \epsilon)$, and $\epsilon = \epsilon$. If the linear Bernoulli factory would flip the input coin, the coin is 'flipped' by running a Bernoulli factory for $g(\lambda)$. If any run of the linear Bernoulli factory returns 0, return 0. Otherwise, return 1."

2. Run a **linear Bernoulli factory** once, using the ν input coin described earlier, $x/y = t/\epsilon$, and $\epsilon = \epsilon$, and return the result.

Power Series Examples:

Examples 1 to 4 show how **Algorithm 1** leads to algorithms for simulating certain factory functions.

> **Note:** In the SymPy computer algebra library, the `series(func, x, n=20)` method computes the terms of a function's power series up to the term with x^{19}. An example is: `series(sin(x), x, n=20)`.

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

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

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

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

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

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

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

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

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

Examples 6: The following functions can be written as power series that satisfy the **general martingale algorithm**. In the table, $B(i)$ is the ith *Bernoulli number* (see the note after the table), and ${n \choose m}$ = choose(n, m) is a binomial coefficient.

Function $f(\lambda)$	Coefficients	Value of d_0
$\lambda/(\exp(\lambda)-1)$	$a_i = -1/2$ if $i=1$, or $B(i)/(i!)$ otherwise.	1.
Hyperbolic tangent: $\tanh(\lambda)$	$a_i = \frac{B(i+1) 2^{i+1} (2^{i+1}-1)}{(i+1)!}$ if i is odd[34], or 0 otherwise.	1.
$\cos(\sqrt \lambda)$	$a_i = \frac{(-1)^i}{(2i)!}$.	1.
$\sum_{i\ge 0} a_i x^i$ (**source**)	$a_i = \frac{(-1)^i 4^i}{(2i+1)^2 {2i \choose i}}$.	1.

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

> **Note:** Bernoulli numbers can be computed with the following algorithm, namely **Get the m^{th} Bernoulli number**:
>
> 1. If m is 0, 1, 2, 3, or 4, return 1, 1/2, 1/6, 0, or −1/30, respectively. Otherwise, if m is odd[35], return 0.
> 2. Set i to 2 and v to $1 - (m+1)/2$.
> 3. While i is less than m:
>   1. **Get the i^{th} Bernoulli number**, call it b. Add b*choose(m+1, i) to v.
>   2. Add 2 to i.
> 4. Return $-v/(m+1)$.

Examples 7 to 9 use **Algorithm 2** to simulate generalized power series where the coefficients a_0 are nonnegative.

Example 7: The hyperbolic cosine minus 1, denoted as $\cosh(\lambda)-1$, can be written as follows: $$f(\lambda)=\cosh(\lambda)-1 = \sum_{n} a_n \lambda^n = \sum_{n} w(n) \frac{a_n \lambda^n}{w(n)},$$ where:

- Each sum given above is taken over all values of n that can occur

after step 1 is complete (in this case, all values of n that are even and greater than 0).

- a_n is $1/(n!)$.[36]
- The coefficients a_n are tucked under a function $w(n)$, which in this case is $\frac{1}{2^{n-2}}$ if n>0 and n is even[37], or 0 otherwise.

For this particular function:

- Step 1 of **Algorithm 2** can read: "(1a.) Generate unbiased random bits (each bit is 0 or 1 with equal probability) until a zero is generated this way, then set n to the number of ones generated this way; (1b.) Set n to 2*n + 2."
- In step 2, P is $a_n/w(n) = \frac{1}{n!} / \frac{1}{2^{n-2}} = \frac{2^{n/2}}{n!}$ for each allowed n.
- In step 3, $g(\lambda)$ is simply λ.

Examples 8: cosh(λ)−1 and additional target functions are shown in the following table. (In the table below, $w(n)=1/(2^{z^{-1}(n)+1})$ where $z^{-1}(n)$ is the inverse of the "Step 1b" column, and the $g(\lambda)$ in step 3 is simply λ.)

Target function $f(\lambda)$	**Step 1b in Example 7 reads "Set n to ..."**	**a_n**	**$w(n)$**	**Value of P**
cosh(λ)−1.	2*n + 2.	1/(n!).	$1/(2^{(n-2)/2+1})$.	$2^{n/2}/(n!)$.
exp(λ/4)/2.	n.	$1/(n!*2*4^n)$	$1/(2^{n+1})$.	$1/(2^n*(n!))$.
exp(λ)/4.	n.	1/(n!*4).	$1/(2^{n+1})$.	$2^{n-1}/(n!)$.
exp(λ)/6.	n.	1/(n!*6).	$1/(2^{n+1})$.	$2^n/(3*(n!))$.
exp(λ/2)/2.	n.	$1/(n!*2*2^n)$	$1/(2^{n+1})$.	1/(n!).
(exp(λ) −1)/2.	n + 1.	1/((n+1)!*4).	$1/(2^n)$.	$2^{n-1}/(n!)$.
sinh(λ)/2	2*n + 1.	1/(n!*2).	$1/(2^{(n-1)/2+1})$.	$2^{(n-1)/2}/(n!)$.
cosh(λ)/2	2*n.	1/(n!*2).	$1/(2^{n/2+1})$.	$2^{n/2}/(n!)$.

> **Note:** sinh(λ) is the hyperbolic sine function.

Examples 9: The table below shows generalized power series shifted downward and shows the algorithm changes needed to simulate the modified function. In the table, D is a rational number such that $0 \le D \le \varphi(0)$, where $\varphi(.)$ is the original function.

Original function ($\varphi(\lambda)$)	**Target function $f(\lambda)$**	**Step 1b in Example 7 reads "Set n to ..."**	**Value of P**
exp(λ)/4.	$\varphi(\lambda) - D$.	n.	(1/4−D)*2 or ($\varphi(0)$−D)*2 if n = 0; $2^{n-1}/(n!)$ otherwise.
exp(λ)/6.	$\varphi(\lambda) - D$.	n.	(1/6−D)*2 if n = 0; $2^n/(3*(n!))$ otherwise.
exp(λ/2)/2.	$\varphi(\lambda) - D$.	n.	(1/2−D)*2 if n = 0; $1/(n!)$ otherwise.
cosh(λ)/4.	$\varphi(\lambda) - D$.	2*n.	(1/4−D)*2 if n = 0; $2^{n/2}/(2*(n!))$ otherwise.

Example 10: Let $f = \exp(\lambda)/3$. Then this function is a generalized power series, with nonnegative coefficients, which can be tucked under probabilities of the form $w(n) = \left(\frac{2}{3}(1-\frac{2}{3})^n\right)$.

- Step 1 of **Algorithm 2** can read: "(1a.) Set n to 0. (1b.) With probability 2/3, go to substep 1c. Otherwise, add 1 to n and repeat this substep. (1c.) Set n to n."
- In step 2, P is $a_n/w(n) = \frac{1}{3\cdot n!} / \left(\frac{2}{3}(1-\frac{2}{3})^n\right) = \frac{(3/2)^{n + 1}}{n!}$ for each allowed n.
- In step 3, $g(\lambda)$ is simply λ.

Example 11: Let $f(\lambda)=\exp(\lambda)\cdot (1-\lambda)$. Run Mendo's algorithm for series of type 1, with $a_i = \frac{i-1}{i!}$ and $CS = 1$.

4.2.4 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 builds randomized bounds on $f(\lambda)$ based on the outcomes of the coin flips. These randomized bounds come from two sequences of polynomials:

- One sequence of polynomials converges from above to f, the other from below.
- For each sequence, the polynomials must have increasing degree.
- The polynomials are written in *Bernstein form* (see "**Certain Polynomials**").
- For each n, the degree-n polynomials' coefficients must lie at or "inside" those of the previous upper polynomial and the previous lower one (once the polynomials are elevated to degree n).

The following algorithm can be used to simulate factory functions via polynomials. In the algorithm:

- **fbelow**(n, k) is a lower bound of the k^{th} coefficient for a degree-n polynomial in Bernstein form that comes close to f from below, where $0 \le k \le n$. For example, this can be $f(k/n)$ minus a constant that depends on n. (See note 1 below.)
- **fabove**(n, k) is an upper bound of the k^{th} coefficient for a degree-n polynomial in Bernstein form that comes close to f from above. For example, this can be $f(k/n)$ plus a constant that depends on n. (See note 1.)

The algorithm implements the reverse-time martingale framework (Algorithm 4) in Łatuszyński et al. (2009/2011)[38] and the degree-doubling suggestion in Algorithm I of Flegal and Herbei (2012)[39], although an error in Algorithm I is noted below. The first algorithm follows.

1. Generate a uniform random variate between 0 and 1, 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 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) with 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. (The remaining steps are now done repeatedly until the algorithm finishes by returning a value.) 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. Calculate ℓ and u as follows:
    1. Define **FB**(a, b) as follows: Let c be choose(a, b). (Optionally, multiply c by 2^a; see note 3.) Calculate **fbelow**(a, b) as lower and upper bounds LB and UB that are accurate enough that floor(LB*c) = floor(UB*c), then return floor(LB*c)/c.
    2. Define **FA**(a, b) as follows: Let c be choose(a, b). (Optionally, multiply c by 2^a; see note 3.) Calculate **fabove**(a, b) as lower and upper bounds LB and UB that are accurate enough that ceil(LB*c) = ceil(UB*c), then return ceil(LB*c)/c.
    3. Set ℓ to **FB**(*degree*, *ones*) and set u to **FA**(*degree*, *ones*).
7. (This step and the next find the means of the previous ℓ and of 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*:
    1. Let *nh* be choose(*degree*, *ones*), and let k be min(*lastdegree*, *ones*).
    2. Set ℓs to $\sum_{j=0}^k$ **FB**(*lastdegree*,j)*choose(*degree*−*lastdegree*, *ones*−j)*choose(*lastdegree*,j)/*nh*.
    3. Set us to $\sum_{j=0}^k$ **FA**(*lastdegree*,j)*choose(*degree*−*lastdegree*, *ones*−j)*choose(*lastdegree*,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.) 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 5.

Another algorithm, given in Thomas and Blanchet (2012)[40], was based on the one from Nacu and Peres (2005)[41]. That algorithm is not given here, however.

> **Notes:**
>
> 1. The efficiency of this algorithm depends on many things, including how "smooth" f is (Holtz et al. 2011)[42] and how easy it is to calculate the appropriate values for **fbelow** and **fabove**. The best way to implement **fbelow** and **fabove** for a given function f will require a deep mathematical analysis of that function. For more information, see my **Supplemental Notes on Bernoulli Factories**.
> 2. In some cases, a single pair of polynomial sequences may not converge quickly to the desired function f, especially when f is not "smooth" enough. An intriguing suggestion from Thomas and Blanchet (2012)[43] 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.
> 3. Normally, the algorithm works only if $0 < \lambda < 1$. If λ can be 0 or 1 (meaning the input coin is allowed to return 1 every time or 0 every time), then based on a suggestion in Holtz et al. (2011)[44], the c in **FA** and **FB** can be multiplied by 2^a (as shown in step 6) to ensure correctness for every value of λ.

4.3 Algorithms for General Irrational Constants

This section shows general-purpose algorithms to generate heads with a probability equal to an *irrational number* (a number that isn't a ratio of two integers), when that number is known by its digit or series expansion, continued fraction, or continued logarithm.

But 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 probability of heads are available, then a ***randomness extraction*** method should be used to turn them into fair coins.

4.3.1 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 $0 \le$ `p` < 1. (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)[45], (Devroye 1986, p. 769)[46]), `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 uniform random integer in the interval [0, `BASE`) (if `BASE` is 2, then `v` is simply an unbiased random bit). Calculate `pa`, which is an approximation to `p` such that abs(`p`−`pa`) $\le$ `BASE`$^{-\texttt{k}}$. 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.[47] If `pk - 2 >= u`, return 1. If neither is the case, add 1 to `k` and go to step 2.

4.3.2 Continued Fractions

A *simple continued fraction* is a way to write a real number between 0 and 1. A simple continued fraction has the form— $0 + 1 / (a[1] + 1 / (a[2] + 1 / (a[3] + ...))),$ where 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")[48], I developed the following algorithm.

Algorithm 1. The following algorithm simulates a probability expressed as a simple continued fraction. This algorithm works only if each $a[i]$'s absolute value is 1 or greater and $a[1]$ is greater than 0, 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. Do the following process repeatedly until this run of the algorithm returns a value:
    1. With probability $kp/(1+kp)$, return a number that is 1 with probability $1/kp$ and 0 otherwise.
    2. Do a separate run of the currently running algorithm, but with $pos = pos + 1$. If the separate run returns s, return 0.

Algorithm 2.

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.

The following algorithm works only if each ratio $b[i]/a[i]$ has an absolute value of 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. Do the following process repeatedly until this run of the algorithm returns a value:
    1. With probability $kp/(1+kp)$, return a number that is 1 with probability $1/kp$ and 0 otherwise.
    2. Do a separate run of the currently running algorithm, but with $pos = pos + 1$ and $r = r$. If the separate run returns s, return 0.

Algorithm 3. This algorithm works only if each ratio $b[i]/a[i]$ 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. Do the following process repeatedly until this run of the algorithm returns a value:
   1. With probability $a[pos]/(1 + a[pos])$, return a number that is 1 with probability $b[pos]/a[pos]$ and 0 otherwise.
   2. Do a separate run of the currently running algorithm, but with $pos = pos + 1$. If the separate run returns 1, return 0.

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 so-called "even-parity construction"[49] (or the two-coin algorithm) 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 algorithm—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.

4.3.3 Continued Logarithms

The *continued logarithm* (Gosper 1978)[50], (Borwein et al., 2016)[51] of a number greater than 0 and less than 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. Do the following process repeatedly until this run of the algorithm returns a value:
    1. Generate an unbiased random bit. If that bit is 1 (which happens with probability 1/2), return a number that is 1 with probability 1/($2^{c[pos]}$) and 0 otherwise.
    2. Do a separate run of the currently running algorithm, but with pos = pos + 1. If the separate run returns 1, return 0.

For a correctness proof, see the appendix.

4.3.4 Certain Algebraic Numbers

A method to sample a probability equal to a polynomial's root appears in a French-language article by Penaud and Roques (2002)[52]. The following is an implementation of that method, using the discussion in the paper's section 1 and Algorithm 2, and incorporates a correction to Algorithm 2. The algorithm takes a polynomial as follows:

- It has the form $P(x) = a[0]*x^0 + a[1]*x^1 + ... + a[n]*x^n$, where $a[i]$, the *coefficients*, are all rational numbers, and $0 \le x \le 1$.
- It equals 0 (has a *root*) at exactly one point, and that point is greater than 0 and less than 1.

And the algorithm returns 1 with probability equal to the root, and 0 otherwise. The root R is known as an *algebraic number* because it satisfies the polynomial equation $P(R) = 0$. The algorithm follows.

1. Set r to 0 and d to 2.
2. Do the following process repeatedly, until this algorithm returns a value:
    1. Generate an unbiased random bit, call it z.
    2. Set t to $(r*2+1)/d$.
    3. If $P(0) > 0$:
        1. If z is 1 and $P(t)$ is less than 0, return 0.
        2. If z is 0 and $P(t)$ is greater than 0, return 1.
    4. If $P(0) < 0$:
        1. If z is 1 and $P(t)$ is greater than 0, return 0.
        2. If z is 0 and $P(t)$ is less than 0, return 1.
    5. Set r to $r*2+z$, then multiply d by 2.

> **Example** (Penaud and Roques 2002)[53]: Let $P(x) = 1 - x - x^2$. When $0 \le x \le 1$, this is a polynomial whose only root 1 is 2/(1+sqrt(5)), that is, 1 divided by the golden ratio or $1/\varphi$ or about 0.618, and $P(0) > 0$. Then given P, the algorithm above samples the probability $1/\varphi$ exactly.

4.3.5 Certain Converging Series

A general-purpose algorithm was given by Mendo (2020/2021)[54] that can simulate any probability, as long as—

- the probability is greater than 0 and less than 1,
- the probability can be written as a (possibly infinite) sum of rational numbers greater than 0, that is, as $p = a[0] + a[1] + ...$, and

- a sequence of rational numbers *err*[0], *err*[1], ... is available that is nowhere increasing and approaches 0 (*converges* to 0), where *err*[*n*] is not less than $p - (a[0] + ... + a[n])$.

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 $lamunq+\epsilon \le lam + 1/(2^k)$, 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 $lamunq+\epsilon \le bound$, set s to 0. Otherwise, if $lamunq > bound$, set s to 2. Otherwise, set s to 1.
9. Generate an unbiased random bit. If that bit is 1 (which happens 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, sums to the *base-2 logarithm* of the probability rather than that probability, 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 nowhere decreasing 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 variate E with rate ln(2). This can be done, for example, by using the **exponential distribution with rate ln(*x*)** algorithm given in "**Partially-Sampled Random Numbers**". (This step takes advantage of the exponential distribution's *memoryless property*: given that an exponential random variate E is greater than *intinf*, E minus *intinf* has the same distribution.)
4. Set n to 0.
5. Do the following process repeatedly until the algorithm returns a

value:

1. Set *inf* to max(0, *a*[*n*]), then set *sup* to min(0, *inf*+*err*[*n*]).
2. 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.
3. Set *n* to 1.

The case when the sequence *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 nowhere decreasing sequence, and all the *err*[*i*] must be 0 or greater.

1. Generate an exponential random variate *E* with rate 1. This can be done, for example, by using the **ExpRand** or **ExpRand2** algorithm given in "**Partially-Sampled Random Numbers**".
2. Set *n* to 0.
3. Do the following process repeatedly until the algorithm returns a value:
   1. Set *inf* to max(0, *a*[*n*]), then set *sup* to min(0, *inf*+*err*[*n*]).
   2. 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.
   3. Set *n* to 1.

> **Notes:**
>
> 1. Mendo (2020/2021)[55] as well as Carvalho and Moreira (2022)[56] discuss how to find error bounds on "cutting off" a series that work for many infinite series. This can be helpful in finding the appropriate sequences *a* and *err* needed for the first algorithm in this section.
> 2. If a number is known as a simple continued fraction whose partial denominators are integers, Citterio and Pavani (2016)[57] show how to calculate lower and upper bounds for that number. The bounds will be rational numbers whose numerator has at most a given number of digits.
>
> **Examples**:
>
> - Let $f(\lambda) = \cosh(1)-1$, namely, the hyperbolic cosine, minus 1, of 1. This function can be rewritten as a sum required by the first algorithm in this section, namely *f*'s *Taylor series* at 0. Then this algorithm can be used with *a*[*i*] = 1/(((*i*+1)*2)!) and *err*[*i*] = 2/((((*i*+1)*2)+1)!). [58]
> - Logarithms can form the basis of efficient algorithms to

> simulate the probability z = choose(n, k)/2^n when n can be very large (for example, as large as 2^{30}), without relying on floating-point arithmetic. In this example, the trivial algorithm for choose(n, k), a 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)[59]. Instead of calculating binomial coefficients directly, a series can be calculated that sums 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. See also an appendix in (Bringmann et al. 2014)[60].

4.4 Other General Algorithms

4.4.1 Convex Combinations

Assume there is one or more input coins $h_i(\lambda)$ that return 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)[61]. (Alternatively, the algorithm can be seen as returning heads with probability $\mathbf{E}[h_X(\lambda)]$, that is, the expected value, or "long-run average", 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 greater than 1 and less than 6, or it could be a Poisson random variate. (Specifically, the number X is generated with probability $g(X)$. If every $g(i)$ is a rational number, the following **algorithm** can generate X: "(1) Set X to 0 and d to 1. (2) With probability $g(X)/d$, return X; otherwise subtract $g(X)$ from d, add 1 to X, and repeat this step.")

2. Flip the coin represented by X and return the result.

> **Notes:**
>
> 1. **Building convex combinations.** Assume there is a function of the form $f(\lambda)$ = $w_0(\lambda)+w_1(\lambda)+...$, where $w_0, w_1, ...$ are continuous functions. Let $g(n)$ be the probability that a randomly chosen number X is n, such that $g(0) + g(1) + ... = 1$. Then by **generating X and flipping a coin with probability of heads of $w_X(\lambda)/g(X)$**, we can simulate the probability $f(\lambda)$ as the convex combination— $$f(\lambda)=g(0) \frac{w_0(\lambda)}{g(0)} + g(1) \frac{w_1(\lambda)}{g(1)} + ...$$ (where a term is omitted if division by 0 occurs), but this works only if the following conditions are met for each integer $n\ge 0$:
>
>   - $1 \ge g(n) \ge w_n(\lambda) \ge 0$, wherever $0 \le \lambda \le 1$.
>   - If $g(n)>0$, the function $w_n(\lambda)/g(n)$ admits a Bernoulli factory; see the section "About Bernoulli Factories".
>
>   See also Mendo (2019)[62].
>
> 2. **Constants writable as a sum of nonnegative numbers.** A special case of note 1. Let g be as in note 1, and let c be a constant written as— $$c=a_0+a_1+a_2+...,$$ where—
>
>   - a_n are each 0 or greater and sum to 1 or less, and
>   - $1 \ge g(n) \ge a_n \ge 0$ for each integer $n\ge 0$.
>
>   Then by **generating X and flipping a coin with probability of heads of $a_X/g(X)$**, we can simulate the probability c as the convex combination— $$f(\lambda)=g(0) \frac{a_0}{g(0)} + g(1) \frac{a_1}{g(1)} + ...,$$ where a term is omitted if division by 0 occurs.
>
> **Examples:**
>
> 1. Generate X, a Poisson random variate with mean μ, 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 probabilities and the coin for 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. (Wästlund 1999)[63]: Generate a Poisson random variate X with mean 1, 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 probabilities, namely $1/(i!*\exp(1))$, and $h_i()$ being $(1-\lambda)^i$.

3. Generate X, a Poisson random variate with mean μ, run the **ExpMinus** algorithm 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. Multivariate Bernoulli factory (Huber 2016)[64] of the form $R = C_0*\lambda_0 + C_1*\lambda_1 + ... + C_{m-1}*\lambda_{m-1}$, where C_i are known constants greater than 0, $\epsilon > 0$, and $R \le 1 - \epsilon$: Choose an integer in $[0, m)$ uniformly at random, call it i, then run a linear Bernoulli factory for $(m*C_i)*\lambda_i$. This differs from Huber's suggestion of "thinning" a random process driven by multiple input coins.

5. **Probability generating function** (PGF) (Dughmi et al. 2021)[65]. Generates heads with probability $\mathbf{E}[\lambda^X]$, that is, the expected value ("long-run average") of λ^X. $\mathbf{E}[\lambda^X]$ is the PGF for the distribution of X. The algorithm follows: (1) Generate a random integer X in some way; (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.

6. Assume X is the number of unbiased random bits that show 0 before the first 1 is generated. Then $g(n) = 1/(2^{n+1})$.

7. **Poisson to Bernoulli.** Suppose there is a stream of independent Poisson random variates with unknown mean p. Also suppose there is a continuous function $f(p)$

> satisfying $0\le f(p)\le 1$ whenever $p\ge 0$. Then consider the following simple algorithm, which takes an integer $n\gt 0$:
>
> 1. Take n variates from the stream and sum them. Call the sum X. (The result is then a Poisson random variate with mean $n\cdot p$.)
> 2. With probability $f(X/n)$, return 1. Otherwise, return 0.
>
> Then this algorithm outputs 1 with probability equal to $\phi(p)$, where $\phi(p)$ is the *Szász operator* (or *Szász–Mirakyan operator*) of f of degree n (e.g., Szász (1950)[66]). Indeed, the Szász operator can be written as a convex combination with $g(i)$ equal to the probability of getting i in step 1 and with h_X equal to $f(X/n)$. The algorithm is the same as in Goyal and Sigman (2012)[67], except coin flips with heads probability λ are replaced with Poisson variates of mean p.

The previous algorithm can be generalized further, so that an input coin that simulates the probability λ helps generate the random integer in step 1. Now, the overall algorithm returns 1 with probability — $$\sum_{k\ge 0} g(k,\lambda) h_k(\lambda).$$

This algorithm, called **Algorithm CC** in this document, follows.

1. Choose an integer 0 or greater at random, with help of the input coin for λ, so that k is chosen with probability $g(k,\lambda)$. Call the chosen integer X.
2. Flip the coin represented by X and return the result.

> **Notes:**
>
> 1. Step 1 of this algorithm is incomplete, since it doesn't explain how to generate X exactly. That depends on the probability $g(k,\lambda)$.
> 2. If we define S to be a set of integers 0 or greater, and replace step 2 with "If X is in the set S, return 1. Otherwise, return 0", then the algorithm returns 1 with probability $\sum_{k\text{ in }S} g(k,\lambda)$ (because $h_k(\lambda)$ is either 1 if k is in S, or 0 otherwise). Then the so-called "even-parity" construction[68] is a special case of this algorithm, if S is the even positive integers and zero and if the example below is used.

> **Example:** Step 1 can read "Flip the input coin for λ repeatedly until it returns 0. Set X to the number of times the coin returned 1 this way." Then step 1 generates X with probability $\lambda^X (1-\lambda)$.[69]

4.4.2 Bernoulli Race and Generalizations

The Bernoulli factory approach, which simulates a coin with unknown heads probability, leads to an algorithm to roll an n-face die where the chance of each face is unknown. Here is one such die-rolling algorithm (Schmon et al. 2019)[70]. It generalizes the so-called Bernoulli Race (see note 1 below) and returns i with probability— $$\phi_i = \frac{g(i)\cdot h_i(\pmb\mu)}{\sum_{k=0}^r g(k)\cdot h_k(\pmb\mu)},$$

where:

- r is an integer greater than 0. There are r+1 values this algorithm can choose from.
- $g(i)$ takes an integer i and returns a number 0 or greater. This serves as a *weight* for the "coin" labeled i; the higher the weight, the greater the probability the "coin" will be "flipped".
- $h_i(\boldsymbol{\mu})$ takes in a number i and the probabilities of heads of one or more input coins, and returns a number that is 0 or greater and 1 or less. This represents the "coin" for one of the r+1 choices.

The algorithm follows.

1. Generate a random integer i in some way, so that i is generated with probability proportional to the following weights: [$g(0)$, $g(1)$, ..., $g(r)$].
2. Run a Bernoulli factory algorithm for $h_i(\boldsymbol{\mu})$. If the run returns 0 (i is rejected), go to step 1.
3. i is accepted, so return i.

> **Notes:**
>
> 1. The *Bernoulli Race* (Dughmi et al. 2021)[71] is a special case of this algorithm with $g(k) = 1$ for every k. Say there are 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.

2. If we define S to be the integers [0, r] or a subset of them and replace step 3 with "If i is in the set S, return 1. Otherwise, return 0.", the algorithm returns 1 with probability $\sum_{k\text{ in }S} \phi_k$, and 0 otherwise. In that case, the modified algorithm has the so-called "die-coin algorithm" of Agrawal et al. (2021, Appendix D)[72] as a special case with —
   $g(k) = c^k * d^{r-k}$,
   $h_k(\lambda, \mu) = \lambda^k * \mu^{r-k}$ (for the following algorithm: flip the λ coin k times and the μ coin $r-k$ times; return 1 if all flips return 1, or 0 otherwise), and
   S is the set of integers that are 1 or greater and r or less,
   where $c \geq 0$, $d \geq 0$, and λ and μ are the probabilities of heads of two input coins. In that paper, c, d, λ, and μ correspond to c_y, c_x, p_y, and p_x, respectively.
3. Although not noted in the Schmon paper, the r in the algorithm can be infinity (see also Wästlund 1999, Theorem 2.7[73]). In that case, Step 1 is changed to say "Choose an integer 0 or greater at random with probability $g(k)$ for integer k. Call the chosen integer i." As an example, step 1 can sample from a Poisson distribution, which can take on any integer 0 or greater.

The previous algorithm can be generalized further, so that an input coin that simulates the probability λ helps generate the random integer in step 1. Now, the overall algorithm generates an integer X with probability— $$\frac{g(X,\lambda) h_X(\pmb \mu)}{\sum_{k\ge 0} g(k,\lambda) h_k(\pmb \mu)}.$$

In addition, the set of integers to choose from can be infinite. This algorithm, called **Algorithm BR** in this document, follows.

1. Choose an integer 0 or greater at random, with help of the input coin for λ, so that k is chosen with probability proportional to $g(k,\lambda)$. Call the chosen integer X. (If the integer must be less than or equal to an integer r, then the integer will have probability proportional to the following weights: [$g(0, \lambda)$, $g(1, \lambda)$, ..., $g(r, \lambda)$].)
2. Run a Bernoulli factory algorithm for $h_X(\boldsymbol{\mu})$. If the run returns 0 (i is rejected), go to step 1.
3. X is accepted, so return X.

> **Notes:**
>
> 1. Step 1 of this algorithm is incomplete, since it doesn't explain how to generate X exactly. That depends on the weights $g(k,\lambda)$.
> 2. The probability that s many values of X are rejected by this algorithm is $p(1 − p)^s$, where— $$p=\frac{\sum_{k\ge 0} g(k,\lambda) h_k(\pmb \mu)}{\sum_{k\ge 0} g(k,\lambda)}.$$
>
> **Example:** Step 1 can read "Flip the input coin for λ repeatedly until it returns 0. Set X to the number of times the coin returned 1 this way." Then step 1 generates X with probability $g(X,\lambda)=\lambda^X (1-\lambda)$.[74]

4.4.3 Flajolet's Probability Simulation Schemes

Flajolet et al. (2010)[75] described two schemes for probability simulation, inspired by restricted models of computing.

Certain algebraic functions. Flajolet et al. (2010)[76] showed a sampling method modeled on *pushdown automata* (state machines with a stack) that are given flips of a coin with unknown heads probability λ.[77] These flips form a *bitstring*, and each pushdown automaton accepts only 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". If a pushdown automaton terminates, it accepts a bitstring with probability $f(\lambda)$, where f must be an *algebraic function over rationals* (a function that can be a solution of a nonzero polynomial equation whose coefficients are rational numbers) (Mossel and Peres 2005)[78].

Specifically, the method simulates the following function (not necessarily algebraic): $$f(\lambda) = \sum_{k\ge 0} g(k,\lambda) h_k(\lambda),$$ where the paper uses $g(k, \lambda) = \lambda^k (1-\lambda)$ and $h_k(\lambda) = W(k)/\beta^k$, so that— $$f(\lambda) = (1-\lambda) OGF(\lambda/\beta),$$ where:

- $W(k)$ returns a number in the interval [0, β^k]. If $W(k)$ is an integer for every k, then $W(k)$ is the number of k-letter words that can be produced by the stochastic grammar in question.

- $\beta \ge 2$ is an integer. This is the alphabet size, or the number of "letters" in the alphabet. This is 2 for the cases discussed in the Flajolet paper (binary stochastic grammars), but it can be greater than 2 for more general stochastic grammars.
- $OGF(x) = W(0) + W(1) x + W(2) x^2 + W(3) x^3 + ...$ is an *ordinary generating function*. This is a *power series* whose *coefficients* are $W(i)$ (for example, $W(2)$ is coefficient 2).

The method uses **Algorithm CC**, where step 1 is done as follows: "Flip the input coin repeatedly until it returns 0. Set X to the number of times the coin returned 1 this way."[79] Optionally, step 2 can be done as described in Flajolet et al., (2010)[80]: generate an X-letter word uniformly at random and "parse" that word using a stochastic grammar to determine whether that word can be produced by that grammar.

> **Note:** The *radius of convergence* of OGF is the greatest number ρ such that OGF is defined at every point less than ρ away from the origin (0, 0). In this algorithm, the radius of convergence is in the interval $[1/\beta, 1]$ (Flajolet 1987)[81]. For example, the OGF involved in the square root construction given in the examples below has radius of convergence 1/2.
>
> **Examples:**
>
> 1. The following is an example from the Flajolet et al. paper. An X-letter binary word can be "parsed" as follows to determine whether that word encodes a ternary tree: "2. If X is 0, return 0. Otherwise, set i to 1 and d to 1.; 2a. 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.; 2b. Add 1 to i. Then, if $i < X$ and $d > 0$, go to step 3a.; 2c. Return 1 if d is 0 and i is X, or 0 otherwise."
>
> 2. $h_X(\lambda)$ can have the form— $${X\choose X/t}\cdot(1-\mathtt{Coin}(\lambda))^{X-X/t}\cdot(\mathtt{Coin}(\lambda))^{X/t},$$ if X is divisible by t, and 0 otherwise, where $\mathtt{Coin}(\lambda)$ is a Bernoulli factory function, and $t \ge 2$ is an integer. One special case is when $\mathtt{Coin}(\lambda) = 1/\beta$, where $\beta \ge 2$ is the alphabet size and is an integer. In that case, $W(X)$ is **the number of X-letter words with exactly X/t A's, for an alphabet size of β**, is equal to $h_X(\lambda) \beta^X$, and also has the following

form: $${X\choose X/t} (\beta-1)^{X-X/t} = {X\choose X/t} (1-1/\beta)^{X-X/t} (1/\beta)^{X/t} \beta^X,$$ if X is divisible by t, and 0 otherwise. (Here, $\beta-1^{X-X/t}$ is the number of ($X-X$/t)-letter words with only letters other than A.) Then step 2 of the algorithm can be done as follows: "2. If X is not divisible by t, return 0. Otherwise, run a Bernoulli factory algorithm for $\texttt{Coin}(\lambda)$, X times, and set y to the number of runs that return 1 this way (for example, if $\texttt{Coin}(\lambda) = 1/\beta$, generate X uniform random integers in the interval [0, β), then set y to the number of zeros generated this way), then return 1 if y equals X/t, or 0 otherwise." If $\beta = 2$, then this reproduces another example from the Flajolet paper, namely, lattice paths with upward steps of size $t-1$ and downward steps of size 1.
Although not required, $\texttt{Coin}(\lambda)$ can be a rational function (a ratio of two polynomials) whose coefficients are rational numbers; if so, f will be an *algebraic function* and can be simulated by a *pushdown automaton*.
An alternative algorithm is: "Set d to 0, then do the following process repeatedly until this run of the algorithm returns a value: (a) Flip the input coin. If it returns 1, go to substep (b). Otherwise, return either 1 if d is 0, or 0 otherwise. (b) Run a Bernoulli factory algorithm for $\texttt{Coin}(\lambda)$. If the run returns 1, add ($t-1$) to d. Otherwise, subtract 1 from d."

3. $h_X(\lambda)$ can have the form— $${X\alpha\choose X}\cdot(1-\mathtt{Coin}(\lambda))^{X\cdot \alpha-X}\cdot(\mathtt{Coin}(\lambda))^{X},$$ where $\texttt{Coin}(\lambda)$ is a Bernoulli factory function (as in example 2), and $\alpha \geq 1$ is an integer. One special case is when $\texttt{Coin}(\lambda) = 1/\beta$, where $\beta \geq 2$ is the alphabet size and is an integer. In that case, $W(X)$ is equal to $h_X(\lambda) \beta^X$ and also has the following form:$${X\alpha\choose X}\cdot(\beta-1)^{X\cdot \alpha-X} (1/\beta)^{X\cdot \alpha-X} = {X\alpha\choose X}\cdot(1-1/\beta)^{X\cdot \alpha-X}.$$ Then step 2 of the algorithm can be done as follows: "2. Run a Bernoulli factory algorithm for $\texttt{Coin}(\lambda)$, $X * \alpha$ times, and set y to the number of runs that return 1 this way, then return 1 if y equals X, or 0 otherwise." If $\alpha = 2$ and $\beta = 2$ (or $\texttt{Coin}(\lambda) = 1/2$), then this expresses the *square-root construction* sqrt($1 - \lambda$), mentioned in the Flajolet et al. paper. If α is 1, the modified algorithm simulates the following probability: $(\lambda-1)/(\lambda*\texttt{Coin}(\lambda)-1)$. If

α=2, the probability is $(1-\lambda)/\sqrt{1+4\lambda\mathtt{Coin}(\lambda)(\mathtt{Coin}(\lambda)-1)}$.
Interestingly, I have found that if α is 2 or greater, the probability simplifies to involve a hypergeometric function. Specifically, if $\mathtt{Coin}(\lambda) = 1/\beta$, the probability becomes— $$f(\lambda)=(1-\lambda)\times_{\alpha-1} F_{\alpha-2}\left(\frac{1}{\alpha},\frac{2}{\alpha},...,\frac{\alpha-1}{\alpha}; \frac{1}{\alpha-1},\frac{2}{\alpha-1},...,\frac{\alpha-2}{\alpha-1}; \lambda\frac{\alpha^\alpha}{(\alpha-1)^{\alpha-1}2^{\alpha}}\right),$$ if $\beta = 2$, or more generally— $$f(\lambda)=(1-\lambda)\times_{\alpha-1} F_{\alpha-2} \left(\frac{1}{\alpha},\frac{2}{\alpha},...,\frac{\alpha-1}{\alpha}; \frac{1}{\alpha-1},\frac{2}{\alpha-1},...,\frac{\alpha-2}{\alpha-1}; \lambda\frac{\alpha^\alpha(\beta-1)^{\alpha-1}}{(\alpha-1)^{\alpha-1}\beta^{\alpha}}\right).$$

The ordinary generating function for this modified algorithm is thus— $$OGF(z) = 1\times_{\alpha-1} F_{\alpha-2}\left(\frac{1}{\alpha},\frac{2}{\alpha},...,\frac{\alpha-1}{\alpha}; \frac{1}{\alpha-1},\frac{2}{\alpha-1},...,\frac{\alpha-2}{\alpha-1}; z\frac{\alpha^\alpha (\beta-1)^{\alpha-1}}{(\alpha-1)^{\alpha-1}\beta^{\alpha-1}}\right).$$

4. The probability involved in example 2 likewise involves hypergeometric functions: $$f(\lambda)=(1-\lambda)\times_{t-1} F_{t-2}\left(\frac{1}{t},\frac{2}{t},...,\frac{t-1}{t}; \frac{1}{t-1},\frac{2}{t-1},...,\frac{t-2}{t-1}; \lambda^t \frac{t^t (\beta-1)^{t-1}}{(t-1)^{t-1}\beta^t}\right).$$

5. If $W(X)$ is **the number of X-letter words with a two-letter alphabet that meet some condition**, where the chance of the letter "heads" is $\mathtt{Coin}(\lambda)$, and $\mathtt{Coin}(\lambda)$ is a Bernoulli factory function (as in example 2), then $h_X(\lambda)$ can be written as— $$\sum_{m=0}^X V(X,m) (\mathtt{Coin}(\lambda))^m (1-\mathtt{Coin}(\lambda))^{X-m},$$ where $V(X,m)$ satisfies $0\le V(X,m)\le {X\choose m}$ and is the number of X-letter words that have m heads and meet that condition, so that

> — $$W(X) = h_X(\lambda) (1/\mathtt{Coin}(\lambda))^X = \sum_{m=0}^X V(X,m) \left(\frac{1-\mathtt{Coin}(\lambda)}{\mathtt{Coin}(\lambda)}\right)^{X-m}.$$

The von Neumann schema. Flajolet et al. (2010)[82], section 2, describes what it calls the *von Neumann schema*, which produces random integers based on a coin with unknown heads probability. To describe the schema, the following definition is needed:

- A *permutation class* is a rule that describes how a sequence of numbers must be ordered. The ordering of the numbers is called a *permutation*. Two examples of permutation classes cover permutations sorted in descending order, and permutations whose highest number appears first. When checking whether a sequence follows a permutation class, only less-than and greater-than comparisons between two numbers are allowed.

Now, given a permutation class and an input coin, the von Neumann schema generates a random integer $n\ge 0$, with probability equal to— $$w_n(\lambda) = \frac{g(n,\lambda) h_n(\lambda)}{\sum_{k\ge 0} g(k,\lambda) h_k(\lambda)},$$ where the schema uses $g(k, \lambda) = \lambda^k (1-\lambda)$ and $h_k(\lambda) = \frac{V(k)}{k!}$, so that— $$w_n(\lambda)=\frac{(1-\lambda) \lambda^n V(n)/(n!)}{(1-\lambda) EGF(\lambda)} = \frac{\lambda^n V(n)/(n!)}{EGF(\lambda)},$$ where:

- $V(n)$ returns a number in the interval [0, n!]. If $V(n)$ is an integer for every n, this is the number of permutations of size n that belong in the permutation class.
- $EGF(\lambda) = \sum_{k\ge 0} \lambda^k \frac{V(k)}{k!}$ is an *exponential generating function*, which completely determines a permutation class.
- The probability that r many values of X are rejected by the von Neumann schema (for the choices of g and h above) is $p(1 − p)^r$, where $p=(1-\lambda) EGF(\lambda)$.

The von Neumann schema uses **Algorithm BR**, where in step 1, the von Neumann schema as given in the Flajolet paper does the following: "Flip the input coin repeatedly until it returns 0. Set X to the number of times the coin returned 1 this way."[83] Optionally, step 2 can be implemented as described in Flajolet et al., (2010)[84]: generate

X uniform random variates between 0 and 1, then determine whether those numbers satisfy the given permutation class, or generate as many of those numbers as necessary to make this determination.

> **Note:** The von Neumann schema can sample from any *power series distribution* (such as Poisson, negative binomial, and logarithmic series), given a suitable exponential generating function. However, the number of input coin flips required by the schema grows without bound as λ approaches 1.
>
> **Examples:**
>
> 1. Examples of permutation classes include the following (using the notation in "Analytic Combinatorics" (Flajolet and Sedgewick 2009)[85]):
>
>   - Single-cycle permutations, or permutations whose highest number appears first (EGF(λ) = Cyc(λ) = $\ln(1/(1-\lambda))$; V(n) = $((n-1)!)$ [or 0 if n is 0)]).
>   - Sorted permutations, or permutations whose numbers are sorted in descending order (EGF(λ) = Set(λ) = $\exp(\lambda)$; V(n) = 1).
>   - All permutations (EGF(λ) = Seq(λ) = $1/(1-\lambda)$; V(n) = $n!$),
>   - Alternating permutations of even size (EGF(λ) = $1/\cos(\lambda)$ = $\sec(\lambda)$; V(n) = W(n/2) if n is even[86] and 0 otherwise, where the W(m) starting at m = 0 is **A000364** in the *On-Line Encyclopedia of Integer Sequences*).
>   - Alternating permutations of odd size (EGF(λ) = $\tan(\lambda)$; V(n) = W((n+1)/2) if n is odd[87] and 0 otherwise, where the W(m) starting at m = 1 is **A000182**).
>
> 2. Using the class of *sorted permutations*, we can generate a Poisson random variate with mean λ via the von Neumann schema, where λ is the probability of heads of the input coin. This would lead to an algorithm for $\exp(-\lambda)$ — outputting 1 if a Poisson random variate with mean λ is 0, or 0 otherwise — but for the reason given in the note, this algorithm gets slower as λ approaches 1. Also, if $c > 0$ is a real number, adding a Poisson random variate with mean floor(c) to one with mean $c-$floor(c) generates a Poisson random variate with mean c.

3. The algorithm for $\exp(-\lambda)$, described in example 2, is as follows:

    1. Flip the input coin repeatedly until it returns 0. Set X to the number of times the coin returned 1 this way.
    2. With probability $1/((X)!)$, X is accepted so return a number that is 1 if X is 0 and 0 otherwise. Otherwise, go to the previous step.

4. For the class of *alternating permutations of even size* (see example 1), step 2 in **Algorithm BR** can be implemented as follows (Flajolet et al. 2010, sec. 2.2)[88]:

    - (2a.) (Limited to even-sized permutations.) If X is odd[89], reject X (and go to step 1).
    - (2b.) Generate a uniform random variate between 0 and 1, call it U, then set i to 1.
    - (2c.) While i is less than X:
        - Generate a uniform random variate between 0 and 1, call it V.
        - If i is odd[90] and V is less than U, or if i is even[91] and U is less than V, reject X (and go to step 1).
        - Add 1 to i, then set U to V.

5. For the class of *alternating permutations of odd size* (see example 1), step 2 in **Algorithm BR** can be implemented as in example 4, except 2a reads: "(2a.) (Limited to odd-sized permutations.) If X is even[92], reject X (and go to step 1)." (Flajolet et al. 2010, sec. 2.2)[93].

6. By computing— $$\frac{\sum_{k\ge 0} g(2k+1,\lambda) h_{2k+1}(\lambda)}{\sum_{k\ge 0} g(k,\lambda) h_k(\lambda)}$$ (which is the probability of getting an odd-numbered output), and using the class of sorted permutations ($h_i(\lambda)=1/(i!)$), it is found that the von Neumann schema's output is odd with probability $\exp(-\lambda)\times \sinh(\lambda)$, where $sinh$ is the hyperbolic sine function.

7. The X generated in step 1 can follow any distribution of integers 0 or greater, not just the distribution used by the von Neumann schema (because **Algorithm BR** is more general than the von Neumann schema). (In that case, the function

> $g(k, \lambda)$ will be the probability of getting k under the new distribution.) For example, if X is a Poisson random variate with mean $z^2/4$, where $z > 0$, and if 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.

Examples for the von Neumann schema. Examples contained in Theorem 2.3 of Flajolet et al. (2010)[94]. In the table:

- λ is the unknown heads probability of a coin.
- μ is another coin that flips the λ coin and returns 1 minus the result (thus simulating $1 - \lambda$).

Function	Values Allowed	Algorithm
$\exp(-\lambda)$	$0 \le \lambda < 1$	Uses von Neumann schema algorithm (VNS) with **sorted** permutations, and the λ coin. Return 1 if VNS returns 0, and 0 otherwise.
$\exp(\lambda - 1) = \exp(-(1 - \lambda))$	$0 < \lambda \le 1$	Uses VNS with sorted permutations, and the μ coin. Return 1 if VNS returns 0, and 0 otherwise.
$(1-\lambda)*\exp(\lambda)$	$0 \le \lambda < 1$	Uses VNS with sorted permutations, and the λ coin. Return 1 if VNS finishes in one iteration, and 0 otherwise.
$\lambda*\exp(1-\lambda)$	$0 < \lambda \le 1$	Uses VNS with sorted permutations, and the μ coin. Return 1 if VNS finishes in one iteration, and 0 otherwise.
$\lambda/\ln(1/(1-\lambda))$	$0 \le \lambda < 1$	Uses VNS with **single-cycle** permutations, and the λ coin. Return 1 if VNS returns **1**, and 0 otherwise.
$(1-\lambda)/\ln(1/\lambda)$	$0 < \lambda \le 1$	Uses VNS with single-cycle permutations, and the μ coin. Return 1 if VNS returns **1**, and 0 otherwise.
$(1-\lambda)*\ln(1/(1-\lambda))$	$0 \le \lambda < 1$	Uses VNS with single-cycle permutations, and the λ coin. Return 1 if VNS finishes in one iteration, and

		0 otherwise.
$\lambda*\ln(1/\lambda)$	$0 < \lambda \leq 1$	Uses VNS with single-cycle permutations, and the μ coin. Return 1 if VNS finishes in one iteration, and 0 otherwise.
$\cos(\lambda)$	$0 \leq \lambda < 1$	Uses VNS with **alternating even-sized** permutations, and the λ coin. Return 1 if VNS returns **0**, and 0 otherwise.
$(1-\lambda)/\cos(\lambda) = (1-\lambda)*\sec(\lambda)$	$0 \leq \lambda < 1$	Uses VNS with alternating even-sized permutations, and the λ coin. Return 1 if VNS finishes in one iteration, and 0 otherwise.
$\lambda/\tan(\lambda)$	$0 \leq \lambda < 1$	Uses VNS with **alternating odd-sized** permutations, and the λ coin. Return 1 if VNS returns **1**, and 0 otherwise.
$(1-\lambda)*\tan(\lambda)$	$0 \leq \lambda < 1$	Uses VNS with alternating odd-sized permutations, and the λ coin. Return 1 if VNS finishes in one iteration, and 0 otherwise.

Recap. As can be seen—

- the scheme for algebraic functions uses **Algorithm CC** with $g(k, \lambda) = \lambda^k (1-\lambda)$ and $h_k(\lambda) = W(k)/\beta^k$, and
- the *von Neumann schema* uses **Algorithm BR** with $g(k, \lambda) = \lambda^k (1-\lambda)$ and $h_k(\lambda) = V(k)/(k!)$,

and both schemes implement step 1 of the algorithm in the same way. However, different choices for g and h will lead to modified schemes that could lead to Bernoulli factory algorithms for new functions.

4.4.4 Integrals

Roughly speaking, the *integral* of $f(x)$ on an interval $[a, b]$ is the "area under the graph" of that function when the function is restricted to that interval. If f is continuous there, this is the value that $\frac{1}

{n} (f(a+(b-a)(1-\frac{1}{2})/n)+f(a+(b-a)(2-\frac{1}{2})/n)+...+f(a+(b-a)(n-\frac{1}{2})/n))$ approaches as n gets larger and larger.

Algorithm 1. (Flajolet et al., 2010)[95] showed how to turn an algorithm that simulates $f(\lambda)$ into an algorithm that simulates the probability—

- $\frac{1}{\lambda} \int_0^\lambda f(u),du$ ($\frac{1}{\lambda}$ times the integral of $f(u)$ on $[0, \lambda]$), or equivalently,
- $\int_0^1 f(\lambda u),du$ (the integral of $f(\lambda u)$ on $[0, 1]$),

namely the following algorithm:

1. Generate u, a uniform random variate between 0 and 1, call it u.
2. Create an input coin that does the following: "Flip the original input coin, then **sample from the number u**. Return 1 if both the call and the flip return 1, and return 0 otherwise."
3. Run the original Bernoulli factory algorithm, using the input coin described in step 2 rather than the original input coin. Return the result of that run.

Algorithm 2. A special case of Algorithm 1 is the integral $\int_0^1 f(u),du$, when the original input coin always returns 1:

1. Generate a uniform random variate between 0 and 1, call it u.
2. Create an input coin that does the following: "**Sample from the number u** and return the result."
3. Run the original Bernoulli factory algorithm, using the input coin described in step 2 rather than the original input coin. Return the result of that run.

Algorithm 3. I have found that it's possible to simulate the following integral, namely— $$\int_a^b f(\lambda u),du,$$ where $0\le a\lt b\le 1$, using the following algorithm:

1. Generate u, a uniform random variate between 0 and 1. 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.)
2. Create an input coin that does the following: "**Sample from the number u** and return the result."
3. Run the original Bernoulli factory algorithm, using the input coin described in step 2. If the run returns 0, return 0. Otherwise,

generate a uniform random variate between 0 and 1 ν and return a number that is 0 if ν is less than a or is greater than b, or 1 otherwise.

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

4.5 Algorithms for Specific Functions of λ

This section and the next one describe algorithms for specific functions, especially when they have a more convenient simulation than the general-purpose algorithms given earlier. They can be grouped as follows:

- Functions involving the exponential function $\exp(x)$.
- Rational functions of several variables.
- Addition, subtraction, and division.
- Powers and roots.
- Linear Bernoulli factories.
- Transcendental functions.
- Other factory functions.

4.5.1 ExpMinus ($\exp(-z)$)

In this document, the **ExpMinus** algorithm is a Bernoulli factory taking a parameter z. The parameter z is 0 or greater and can be written in any of the following ways:

1. As a rational number, namely x/y where $x \geq 0$ and $y > 0$ are integers.
2. As an integer and fractional part, namely $m + \nu$ where $m \geq 0$ is an integer and ν ($0 \leq \nu \leq 1$) is the probability of heads of a coin. (Specifically, the "coin" must implement a Bernoulli factory algorithm that returns 1 [or outputs heads] with probability equal to the fractional part ν.[96])
3. As a finite sum of positive numbers, each of which can be written in either of the preceding ways. For example, if $z = \pi$, it can be written as a sum of four numbers, each of which is $(\pi / 4)$, that is, $m = 0$ and $\nu = (\pi / 4)$. (This case makes use of the identity $\exp(-(b+c)) = \exp(-b) * \exp(-c)$. Here, $\pi/4$ has a not-so-trivial Bernoulli factory algorithm described in this article.)

The **ExpMinus** algorithm is as follows. To flip a coin with probability of heads of $\exp(-z)$:

- In case 1, use the following algorithm (Canonne et al. 2020)[97]:

  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 $x = x - \text{floor}(x/y) * y$ 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.

  Or the following algorithm:

  - If x is 0, return 1. Otherwise, generate N, a Poisson random variate with mean x/y (see "**Poisson Distribution**" for one way to do this), and return a number that is 1 if N is 0, or 0 otherwise.

- In case 2, use case 2 of the **algorithm for $\exp(-(\lambda * z))$** with parameter z, where λ represents a coin that always returns 1.

- In case 3, rewrite the z parameter as a sum of positive numbers. For each number, run either case 1 or case 2 (depending on how the number is written) of the **ExpMinus** algorithm with that number as the parameter. If any of these runs returns 0, return 0; otherwise, return 1. (See also (Canonne et al. 2020)[98].)

> **Examples:** The **ExpMinus** algorithm with the following parameters can be implemented as follows:
>
> - Parameter π: Run the **algorithm for $\exp(-(\lambda * z))$**, four times, with parameter $z = 0 + \nu$, where ν is a Bernoulli factory for $(\pi/4)$, and λ represents a coin that always returns 1. If any of these runs returns 0, return 0; otherwise, return 1.
> - Parameter 3: Run case 1 of the algorithm where x=3 and y=1.
> - Parameter 7/5: Run case 1 of the algorithm where x=7 and y=5.

> **Note:** $\exp(-z) = \exp(1-z)/\exp(1) = 1/\exp(z) = 1-(\exp(z)-1)/\exp(z)$.

4.5.2 LogisticExp (1 − expit($z/2^{prec}$))

This is the probability that the binary digit at *prec* (the $prec^{\text{th}}$ binary digit after the point, where *prec* is greater than 0) is set for an exponential random variate with rate z. In this document, the **LogisticExp** algorithm is a Bernoulli factory taking the following parameters in this order:

1. z is 0 or greater, and written as a rational number (case 1), as an integer and fractional part (case 2), or as a sum of positive numbers (case 3), as described in the **"ExpMinus" section**.
2. *prec* is an integer 0 or greater.

The **LogisticExp** algorithm is as follows. To flip a coin with probability of heads of $1/(1+\exp(z/2^{prec})) = 1 - \text{expit}(\lambda/2^{prec})$:

- Run the **algorithm for expit(λ*z)** where $z = z$, and where λ represents a coin that returns a number that is 1 with probability $1/(2^{prec})$ or 0 otherwise. Return 1 minus the result of that run (leading to **1 − expit(λ*z)**).

4.5.3 exp(−(λ * z))

In the following algorithm:

- z is 0 or greater, and written as a rational number (case 1), as an integer and fractional part (case 2), or as a sum of positive numbers (case 3), as described in the **"ExpMinus" section**.
- λ is the probability of heads of an input coin, with $0 \le \lambda \le 1$.

The algorithm follows.

- In case 1 ($z = x/y$) (see also algorithm for $\exp(-((1-\lambda)^1 * c))$ in "Other Factory Functions"):
  1. Special case: If x is 0, return 1.
  2. Generate N, a Poisson random variate with mean x/y. (See "**Poisson Distribution**" for one way to do this.)
  3. Flip the λ input coin until a flip returns 1 or the coin is flipped N times, whichever comes first. Return 0 if N is greater than 0 and any of the flips, including the last, returns 1. Otherwise,

return 1. (The flips transform a Poisson variate with mean x/y to one with mean $\lambda*x/y$; see (Devroye 1986, p. 487)[99].)

- In case 2 ($z = m + \nu$):
  1. Set j to 0, then while $j < m+1$:
     1. Generate N, a Poisson random variate with mean 1.
     2. If $j = m$, flip the ν input coin N times and set N to the number of flips that return 1 this way. (This transforms a Poisson variate with mean 1 to one with mean ν; see (Devroye 1986, p. 487)[100].)
     3. Flip the λ input coin until a flip returns 1 or the coin is flipped N times, whichever comes first. Return 0 if N is greater than 0 and any of the flips, including the last, returns 1.
     4. Add 1 to j.
  2. Return 1.
- In case 3, rewrite the z parameter as a sum of positive numbers. For each number, run either case 1 or case 2 (depending on how the number is written) of this algorithm with that number as the parameter. If any of these runs returns 0, return 0; otherwise, return 1.

> **Notes:**
>
> 1. The following is a proof of case 2 of this algorithm. First, suppose $\lambda = 1$. Each iteration of the loop in the algorithm returns 0 if a Poisson random variate with mean t (see second substep of step 1) is other than 0, where t is ν in the last iteration, or 1 otherwise. Since the Poisson variate is 0 with probability $\exp(-t)$, the iteration will terminate the algorithm with probability $1-\exp(-t)$ and "succeed" with probability $\exp(-t)$. If all the iterations "succeed", the algorithm will return 1, which will happen with probability $\exp(-\nu) \cdot (\exp(-1))^m = \exp(-(m+\nu))$. Now suppose $0 \le \lambda < 1$. Then (due to the third substep of step 1) the Poisson variate just mentioned has mean $t\lambda$ rather than t, so that each iteration succeeds with probability $1-\exp(-t\lambda)$ and the final algorithm returns 1 with probability $\exp(-\nu\lambda) \cdot (\exp(-\lambda))^m = \exp(-(m+\nu)\lambda)$.
> 2. When z is a rational number with $0 \le z \le 1$, this function can be rewritten as a power series expansion. In that case, one

way to simulate the function is to run the **general martingale algorithm** (see "**Certain Power Series**"), with $g(\lambda) = \lambda$, and with parameter $d_0 = 1$ and coefficients $a_i = \frac{(-1)^i z^i}{i!}$, and return the result of that algorithm.

3. When z is a rational number 0 or greater, this function can be simulated as follows: Let m be floor(c). Call the algorithm in note 2 m times with $z = 1$. If any of these calls returns 0, return 0. Otherwise, if z is an integer (that is, if floor(z) = z), return 1. Otherwise, call the algorithm in note 2 once, with $z = z - \text{floor}(z)$. Return the result of this call.
4. When $m = 0$ and $\mu = 1$, this function, in case 2, becomes $\exp(-\lambda)$ and can be rewritten as a power series expansion. In that case, one way to simulate the function is to use the **general martingale algorithm** (see "**Certain Power Series**"), with $g(\lambda)=\lambda$, and with $d_0 = 1$ and coefficients $a_i = (-1)^i/(i!)$.[101]

4.5.4 exp(−exp(*m* + *λ*))

In the following algorithm, m is an integer 0 or greater.

1. Generate n, a Poisson random variate with mean 3^{m+1}. (See "**Poisson Distribution**" for one way to do this.)
2. If n is greater than 0, do the following n times or until this algorithm returns a value:
   - Run the algorithm for **exp(*λ*)/3** (see "Certain Power Series"), m times, with λ being a coin that always returns 0. Then run the algorithm for **exp(*λ*)/3** once, with λ being the input coin. If all these runs return 1, return 0.
3. Return 1.

> **Note:** The following is a proof this algorithm is valid. Rewrite $\exp(m+\lambda)$ = $3^{m+1}\cdot\left(\frac{\exp(1)}{3}\right)^m\cdot\frac{\exp(\lambda)}{3}$. Step 1 generates a Poisson variate with mean 3^{m+1}. This variate is then thinned to a Poisson variate with mean $\exp(m+\lambda)$ in step 2, returning early if the new variate would be greater than 0 (because a Poisson variate with mean $\exp(m+\lambda)$ is 0 with probability $\exp(-\exp(m+\lambda))$).

4.5.5 exp(−(*m* + *λ*)^*k*^)

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

1. If k is 0, run the **ExpMinus** algorithm with parameter 1, and return the result.
2. If k is 1, run the **ExpMinus** algorithm with parameter $m + \lambda$, and return the result.
3. (Expand $(m + \lambda)^k$ to a polynomial in λ in rest of algorithm. First the λ^0 term.) Run the **ExpMinus** algorithm with parameter m^k. If the algorithm returns 0, return 0.
4. (Now the λ^k term.) Run the **ExpMinus** algorithm with parameter $0 + \mu$, where μ represents an input coin that does: "Flip the λ input coin k times and return either 1 if all the flips return 1, or 0 otherwise". If the algorithm returns 0, return 0.
5. (Now the other terms.) If m is 0, return 1.
6. Set i to 1, then while $i < k$:
    1. Set w to choose(k, i) * m^{k-i}.
    2. (Now the λ^i term.) Run the **ExpMinus** algorithm, w times, with parameter $0 + \mu$, where μ represents an input coin that does: "Flip the λ input coin i times and return either 1 if all the flips return 1, or 0 otherwise". If any of these calls returns 0, return 0.
    3. Add 1 to i.
7. Return 1.

4.5.6 exp(*λ*)*(1−*λ*)

(Flajolet et al., 2010)[102]:

1. Set k and w each to 0.
2. Flip the input coin. If it returns 0, return 1.
3. Generate u, a uniform random variate between 0 and 1.
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.

4.5.7 (1 − exp(−(*m* + *λ*))) / (*m* + *λ*)

In this algorithm, $m + \lambda$ must be greater than 0.

1. If $m = 0$, run the **general martingale algorithm** (see "**Certain Power Series**"), with $g(\lambda)=\lambda$, and with $d_0 = 1$

and coefficients $a_i = \frac{(-1)^i}{(i+1)!}$, and return the result of that algorithm.
2. (m>0.) Run the **ExpMinus** algorithm with parameter $z = m + \lambda$. If it returns 1, return 0.
3. Run the algorithm for $\boldsymbol{d/(c+\lambda)}$ with d=1 and c=m, and return the result of that algorithm.

4.5.8 expit(z) or 1−1/(1+exp(z)) or exp(z)/(1+exp(z)) or 1/(1+exp(−z))

expit(z), also known as the *logistic function*, is the probability that a random variate from the logistic distribution is z or less.

z is a number (positive or not) whose absolute value (abs(z)) is written in one of the ways described in the **"ExpMinus" section**.

- If z is known to be 0 or greater:
  1. Create an R coin that runs the **ExpMinus** algorithm with parameter z.
  2. Run the algorithm for $\boldsymbol{d/(c+\lambda)}$ with d=1, c=1, and with λ being the R coin, and return the result of that run.
- If z is known to be 0 or less:
  1. Create a R coin that runs the **ExpMinus** algorithm with parameter abs(z).
  2. Run the algorithm for $\boldsymbol{d/(c+\lambda)}$ with d=1, c=1, and with λ being the R coin, and return **1 minus the result** of that run.

> **Note**:
>
> 1. This algorithm can be used to simulate **expit(λ* z)**, where λ is the probability of heads of an input coin, with $0 \leq \lambda \leq 1$, except it runs the **algorithm for exp(−(λ * z))** instead of the **ExpMinus** algorithm.
> 2. expit(z) = (tanh(z/2)+1)/2. `tanh` is the hyperbolic tangent function.

4.5.9 expit(z)*2 − 1 or tanh(z/2)

In this algorithm, z is 0 or greater and is written in one of the ways described in the **"ExpMinus" section**. `tanh` is the hyperbolic tangent function.

- Do the following process repeatedly, until this algorithm returns a

value:

1. Run the **ExpMinus** algorithm with parameter z. Let r be the result of that run.
2. Generate an unbiased random bit. If that bit is 1 (which happens with probability 1/2), return $1-r$. Otherwise, if r is 1, return 0.

> **Note:** Follows from observing that $\tanh(z/2) = (d + (1 - \mu)) / (c + \mu)$, where $\mu = \exp(-z)$, $d = 0$, and $c = 1$. (See **algorithm for $(d + \mu) / (c + \lambda)$**.)

4.5.10 λ*exp(z) / (λ*exp(z) + (1 − λ)) or λ*exp(z) / (1 + λ*(exp(z) − 1))

In this algorithm:

- z is an "exponential shift" (Peres et al. 2021)[103] or "exponential twist" (Sadowsky and Bucklew 1990)[104]. Its absolute value is written in one of the ways described in the **"ExpMinus" section**.
- λ is a coin that shows heads with probability equal to the probability to be shifted.

The algorithm follows:

- Do the following process repeatedly, until this algorithm returns a value:
  1. Flip the λ input coin. Let *flip* be the result of that flip.
  2. Run the algorithm for **expit(z)** with z=z. If the run returns 1 and if *flip* is 1, return 1. If the run returns 0 and if *flip* is 0, return 0.

> **Note:** This is also a special case of the two-coin algorithm, where β=1, c=exp(z), d=1, $\lambda = \lambda$, and $\mu = 1 - \lambda$.

4.5.11 (1 + exp(z − w)) / (1 + exp(z))

In this algorithm, z is a number (positive or not), and w is 0 or greater, and their absolute values are each written in one of the ways described in the **"ExpMinus" section**".

- If z is known to be 0 or greater:
  - Run the **ExpMinus** algorithm with parameter w, then run the **expit(z)** algorithm with parameter z. If the **ExpMinus** run

returns 1 and the **expit** run returns 0, return 0. Otherwise, return 1.

- If z is known to be 0 or less:
  - Run the **ExpMinus** algorithm with parameter w, then run the **expit(z)** algorithm with parameter abs(z). If both runs return 0, return 0. Otherwise, return 1.

> **Notes:**
>
> 1. (1 + exp(z−1)) / (1 + exp(z)) = $1-\frac{1 - e^{-1}}{e^{-z} + 1}$. (1 + exp(1−1)) / (1 + exp(1)) = 2 / (1 + exp(2)) = (1 + exp(0)) / (1 + exp(1)).
> 2. For the similar function **(1 + exp(z)) / (1 + exp(z+1))**, use this algorithm with w = 1, except add 1 to z (if z is written as an integer and fractional part, add 1 to the integer part; if written as a sum of numbers, append 1 to those numbers).

4.5.12 1/(2^m*(k + λ)) or exp(−(k + λ)*ln(2^m))

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

1. (Factor function in two parts. First, simulate $1/(2^{mk})$.) 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. (Rest of algorithm simulates $1/(2^{m\lambda})$.) Create an input coin μ that does the following: "Flip the input coin, then run the **algorithm for ln(1+y/z)** (given later) with y/z = 1/1. If both the call and the flip return 1, return 1. Otherwise, return 0." (Simulates $\ln(2) \lambda$.)
3. Run the **ExpMinus** algorithm, with parameter 0 + μ (using the μ input coin), m times. If any of the runs returns 0, return 0. Otherwise, return 1.

4.5.13 1/(2^(x/y)*(λ)) or exp(−(λ)*ln(2^x/y))

Based on the previous algorithm. In this algorithm, $x \geq 0$ and $y > 0$ are integers.

1. Special case: If x is 0, return 1.

2. Let c = ceil(x/y). Create an input coin μ that does the following: "Flip the input coin, then run the **algorithm for ln(1+y/z)** (given later) with y/z = 1/1. If both the call and the flip return 1, return a number that is 1 with probability x/(y*c) and 0 otherwise. Otherwise, return 0." (Simulates $\ln(2) \frac{xy}{c} \lambda$.)
3. Run the **ExpMinus** algorithm, with parameter 0 + μ (using the μ input coin), c times. If any of the runs returns 0, return 0. Otherwise, return 1.

4.5.14 Two-Coin Algorithm ($c * \lambda * \beta$ / (β * ($c * \lambda + d * \mu$) − (β − 1) * (c + d)))

This is the general **two-coin algorithm** of (Gonçalves et al., 2017)[105] and (Vats et al. 2022)[106]. It takes two input coins that each output heads (1) with probability λ or μ, respectively. It also takes parameters c and d, each 0 or greater, and β in the interval [0, 1], which is a so-called "portkey" or early rejection parameter (when β = 1, the formula simplifies to $c * \lambda$ / ($c * \lambda + d * \mu$)). In Vats et al. (2022)[107], β, c, d, λ and μ correspond to β, c_y, c_x, p_y, and p_x, respectively, in the "portkey" algorithm, or to β, $\tilde{c}_x$, $\tilde{c}_y$, $\tilde{p}_x$, and $\tilde{p}_y$, respectively, in the "flipped portkey" algorithm.

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.

4.5.15 $c * \lambda$ / ($c * \lambda + d$) or (c/d) * λ / (1 + (c/d) * λ))

This algorithm, also known as the **logistic Bernoulli factory** (Huber 2016)[108], (Morina et al., 2022)[109], 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:** Huber (2016) specifies this Bernoulli factory in terms of a Poisson point process, which seems to require much more randomness on average.

4.5.16 (*d* + *λ*) / *c*

In this algorithm, d and c must be integers, and $0 \le d < c$.

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, return 0.

4.5.17 *d* / (*c* + *λ*)

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

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.

> **Note**: A quick proof this algorithm works: Let x be the desired probability. Then—
> $x = (c / (1 + c)) * (d/c) +$
> $(1-c / (1 + c)) * (\lambda*0 + (1-\lambda)*x)$,
> and solving for x leads to $x=d/(c+\lambda)$.

4.5.18 (*d* + *μ*) / (*c* + *λ*)

Combines the algorithms in the previous two sections.

In this algorithm, c and d must be integers, and $0 \le d < c$.

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, return 0.
2. Flip the λ input coin. If the flip returns 1, return 0. Otherwise, go to step 1.

4.5.19 $(d + \mu) / ((d + \mu) + (c + \lambda))$

In this algorithm, c and d are integers 0 or greater, and λ and μ are the probabilities of heads of two different input coins. In the intended use of this algorithm, λ and μ are backed by the fractional parts of two uniform partially-sampled random numbers (PSRNs), and c and d are their integer parts, respectively.

1. Let $D = d$ and $C = c$. Run the algorithm for $(d + \mu) / (c + \lambda)$ with λ and μ both being the μ input coin, with $d = D+C$, and with $c = 1+D + C$. If the run returns 1:
    1. If c is 0, return 1.
    2. Run the algorithm for $(d + \mu) / (c + \lambda)$ with λ and μ both being the μ input coin, with $d = D$, and with $c = D + C$. If the run returns 1, return 1. Otherwise, return 0.
2. Flip the λ input coin. If the flip returns 1, return 0. Otherwise, go to step 1.

4.5.20 $d^k / (c + \lambda)^k$, or $(d / (c + \lambda))^k$

In this algorithm, c and d must be rational numbers, $c \geq 1$, and $0 \leq d \leq c$, 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.

4.5.21 $1/(1+\lambda)$

This algorithm is a special case of the two-coin algorithm of (Gonçalves et al., 2017)[110] and has bounded expected running time for all λ parameters.[111]

1. Generate an unbiased random bit. If that bit is 1 (which happens with probability 1/2), return 1.
2. Flip the input coin. If it returns 1, return 0. Otherwise, go to step

1.

> **Note:** In this special case of the two-coin algorithm, β=1, c=1, d=1, old λ equals 1, and μ equals new λ.

4.5.22 1/(2 − λ)

1. Generate an unbiased random bit. If that bit is 1 (which happens with probability 1/2), return 1.
2. Flip the input coin. **If it returns 0**, return 0. Otherwise, go to step 1.

> **Note:** Can be derived from the previous algorithm by observing that $1/(2 - \lambda) = 1/(1 + (1 - \lambda))$.

4.5.23 1/(1+(m+λ)^2)

This is a rational function (ratio of two polynomials) with variable λ, and this rational function admits the following algorithm. In this algorithm, m must be an integer 0 or greater, and λ is the unknown heads probability of a coin.

1. Let d be the three-item list [1, 2, 1] (for numerator 1). Let e be the three-item list [1+m^2, 2*(1+m^2+m), 1+m^2+2*m+1] (for denominator). Find the highest number in e, then divide each item in d and in e by that number (using rational arithmetic).
2. Run the first algorithm for **rational functions** in "Bernoulli Factory Algorithms", with n = 2, and with d and e given above.

4.5.24 1 / (1 + (x/y)*λ)

Another special case of the two-coin algorithm. In this algorithm, x/y must be 0 or greater.

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

> **Note:** In this special case of the two-coin algorithm, β=1, c=1, d=x/y, old λ equals 1, and μ equals new λ.

> **Example**: ***μ* / (1 + (*x*/*y*)**λ)** (takes two input coins that simulate λ or μ, respectively): Run the **algorithm for 1 / (1 + (*x*/*y*)**λ)** using the λ input coin. If it returns 0, return 0. Otherwise, flip the μ input coin and return the result.

4.5.25 $\lambda^{x/y}$

In the algorithm below, the case where $0 < x/y < 1$ is due to Mendo (2019)[112]. 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) (equivalent to x - y*floor(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. Do the following process repeatedly, until this algorithm returns a value:
    1. Flip the input coin; if it returns 1, return 1.
    2. With probability $x/(y*i)$, return 0. (Note: $x/(y*i) = (x/y) * (1/i)$.)
    3. Add 1 to i.

> **Notes:**
>
> 1. When x/y is less than 1, the expected number of flips grows 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)[113].
> 2. Another algorithm is discussed in the online community **Cross Validated**.

4.5.26 sqrt(λ)

Special case of the previous algorithm with $\mu = 1/2$.

- Set i to 1. Then do the following process repeatedly, until this algorithm returns a value:
  1. Flip the input coin. If it returns 1, return 1.
  2. With probability 1/(i*2), return 0.
  3. Add 1 to i and go to step 1.

4.5.27 arctan(λ) /λ

arctan(λ) is the inverse tangent of λ.

Based on the algorithm from Flajolet et al. (2010)[114], but uses the two-coin algorithm (which has bounded expected running time for every λ parameter) rather than the even-parity construction (which does not).[115][116]

- Do the following process repeatedly, until this algorithm returns a value:
  1. Generate an unbiased random bit. If that bit is 1 (which happens with probability 1/2), return 1.
  2. Generate u, a uniform random variate between 0 and 1, 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.

4.5.28 arctan(λ) /π

- Do the following process repeatedly, until this algorithm returns a value:
  1. Run the **algorithm for 1/π**. If the run returns 0, return 0.
  2. Generate an unbiased random bit. If that bit is 1 (which happens with probability 1/2), flip the input coin and return the result.
  3. Generate u, a uniform random variate between 0 and 1, if it wasn't generated yet.
  4. **Sample from the number *u*** twice, and flip the input coin twice. If all of these calls and flips return 1, return 0.

4.5.29 arctan(λ)

(Flajolet et al., 2010)[117]: 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.

4.5.30 cos(λ)

This function can be rewritten as a power series expansion. To simulate it, use the **general martingale algorithm** (see "**Certain Power Series**"), with $g(\lambda)=\lambda$, and with $d_0 = 1$ and coefficients $a_i = (-1)^{i/2} / (i!)$ if i is even[118] and 0 otherwise.

4.5.31 sin(λ*sqrt(*c*)) / (λ*sqrt(*c*))

This function can be rewritten as a power series expansion. To simulate it, use the **general martingale algorithm** (see "**Certain Power Series**"), with $g(\lambda)=\lambda$, and with $d_0 = 1$ and coefficients $a_i = \frac{ (-1)^{i/2} c^{i/2}}{(i+1)!}$ if i is even[119] and 0 otherwise. In this algorithm, *c* must be a rational number in the interval (0, 6].

4.5.32 sin(λ)

Equals the previous function times λ, with $c = 1$.

- Flip the input coin. If it returns 0, return 0. Otherwise, run the algorithm for **sin(λ*sqrt(*c*)) / (λ*sqrt(*c*))** with $c = 1$, then return the result.

4.5.33 ln(1+λ)

Based on the algorithm from Flajolet et al. (2010)[120], but uses the two-coin algorithm (which has bounded expected running time for every λ parameter) rather than the even-parity construction (which does not).[121][122]

- Do the following process repeatedly, until this algorithm returns a value:
  1. Generate an unbiased random bit. If that bit is 1 (which happens with probability 1/2), flip the input coin and return the result.
  2. Generate u, a uniform random variate between 0 and 1, 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.

4.5.34 ln(c+λ)/(c+λ)

In this algorithm:

- c is a rational number and is 1 or greater.
- λ is the unknown heads probability of a coin.

The algorithm follows.

1. Run the algorithm for **d / (c + λ)**, with d=1 and c=c, repeatedly, until the run returns 1, then set g to the number of runs that returned 0 this way.
2. If g is 0, return 0. Otherwise, return a number that is 1 with probability 1/g or 0 otherwise.

> **Note:** This algorithm is based on the **von Neumann schema** with the single-cycle permutation class. In this case, given a coin that shows heads with probability z, the schema will terminate in one iteration with probability $(1-z)*\ln(1/(1-z))$. (In step 2 of the algorithm, returning 0 means that the von Neumann schema would require another iteration.) Thus, if the coin shows heads with probability $1 - z$, the one-iteration probability is $z*\ln(1/z)$, so if the coin shows heads with probability $1 - 1/(m+z)$, the one-iteration probability is $(1/(m+z))*\ln(1/(1/(m+z))) = \ln(m+z)/(m+z)$.

4.5.35 arcsin(λ) + sqrt(1 − λ^2) − 1

(Flajolet et al., 2010)[123]. arcsin(λ) is the inverse sine of λ. The algorithm given here uses the two-coin algorithm rather than the even-parity construction[124].

1. Generate u, a uniform random variate between 0 and 1.
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. Generate an unbiased random bit. If that bit is 1 (which happens

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.

4.5.36 tanh(*z*)

`tanh` is the hyperbolic tangent function. In this algorithm, z is 0 or greater and is written in one of the ways described in the **"ExpMinus" section**.[125]

- Do the following process repeatedly, until this algorithm returns a value:
  1. Run the **ExpMinus** algorithm, with parameter z, twice. Let r be a number that is 1 if both runs returned 1, or 0 otherwise.
  2. Generate an unbiased random bit. If that bit is 1 (which happens with probability 1/2), return $1-r$. Otherwise, if r is 1, return 0.

> **Note:** Follows from observing that $\tanh(z) = (d + (1 - \mu)) / (c + \mu)$, where $\mu = (\exp(-z))^2$, $d = 0$, and $c = 1$.

4.5.37 Expressions Involving Polylogarithms

The following algorithm simulates the expression $\mathrm{Li}_r(\lambda) * (1 / \lambda - 1)$, where $\mathrm{Li}_r(.)$ is a polylogarithm of order r, and r is an integer 1 or greater. However, even with a relatively small r such as 6, the expression quickly approaches a straight line.

If λ is 1/2, this expression simplifies to $\mathrm{Li}_r(1/2)$. See also (Flajolet et al., 2010)[126]. See also "**Convex Combinations**" (the case of 1/2 works by decomposing the series forming the polylogarithmic constant into $g(i) = (1/2)^i$, which sums to 1, and $h_i() = 1/i^r$, where $i \geq 1$).

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.

4.5.38 min(*λ*, 1/2) and min(*λ*, 1−*λ*)

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

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

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

1. (Random walk.) Generate unbiased random bits until more zeros than ones are generated this way for the first time. Then set m to ($n-1$)/2+1, where n is the number of bits generated this way.
2. (Build a degree-m*2 polynomial equivalent to $(4*\lambda*(1-\lambda))^m/2$.) Let z be (4^m/2)/choose(m*2,m). Define a polynomial of degree m*2 whose (m*2)+1 Bernstein coefficients are all zero except the m^{th} coefficient (starting at 0), whose value is z. Elevate the degree of this polynomial enough times so that all its coefficients are 1 or less (degree elevation increases the polynomial's degree without changing its shape or position; see the derivation at the end of this section). Let d be the new polynomial's degree.
3. (Simulate the polynomial, whose degree is d (Goyal and Sigman 2012)[127].) Flip the input coin d times and set h to the number of ones generated this way. Let a be the h^{th} Bernstein coefficient (starting at 0) of the new polynomial. With probability a, return 1. Otherwise, return 0.

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

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

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

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

Example: Let f be $\lambda/2$ if $\lambda\le 1/2$, and $1/2-\lambda/2$ otherwise. Then use the algorithm for 0 at 0; $\nu/2$ at 1/2; and $\nu*\mu$ at 1, where ν is a coin that returns 1 with probability 1/2 and 0 otherwise, and μ is a coin that always returns 0.

Note: The following explains how the algorithm is derived. The function min(λ, 1/2) can be rewritten as $A + B$ where—

- A = (1/2) * λ, and
- B = (1/2) * min(λ, 1−λ)
  = (1/2) * ((1−sqrt(1−4*λ*(1−λ)))/2)
  = (1/2) * $\sum_{k\ge 1} h_k(\lambda)$,

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

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

(see also Wästlund (1999)[128]; Dale et al. (2015)[129]). The right-hand side of h, which is the polynomial built in step 3 of the algorithm for min(λ, 1−λ), is a polynomial of degree k*2 with Bernstein coefficients—

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

Unfortunately, z can be greater than 1, so that the polynomial can't be simulated, as is, using the Bernoulli factory algorithm for **polynomials in Bernstein form**. Fortunately, the polynomial's degree can be elevated to bring the Bernstein coefficients to 1 or less (for degree elevation and other algorithms, see Tsai and Farouki (2001)[130]). Moreover, due to the special form of the Bernstein coefficients in this case, the degree elevation process can be greatly simplified. Given an even degree d as well as z (as defined above), the degree elevation is as follows:

1. Set r to floor(d/3) + 1. (This starting value is because when this routine finishes, r/d appears to converge to 1/3 as d gets large, for the polynomial in question.) Let c be choose(d,d/2).
2. Create a list of d+r+1 Bernstein coefficients, all zeros.
3. For each integer i satisfying $0 \le i \le d$+r:

    - If $\max(0, i-r) \le d/2$ and if $d/2 \le \min(d, i)$, set the i^{th} Bernstein coefficient (starting at 0) to $z*c*\text{choose}(r,i-d/2)* / \text{choose}(d+r, i)$.
4. If all the Bernstein coefficients are 1 or less, return them. Otherwise, add $d/2$ to r and go to step 2.

4.6 Algorithms for Specific Functions of λ (Probability-Sensitive)

This section describes algorithms for specific functions that require knowing certain information on the probability of input coins.

4.6.1 $\lambda + \mu$

(Nacu and Peres 2005, proposition 14(iii))[131]. This algorithm takes two input coins that simulate λ or μ, respectively, and a parameter ϵ such that $0 < \epsilon \le 1 - \lambda - \mu$.

1. Create a ν input coin that does the following: "Generate an unbiased random bit. If that bit is 1 (which happens with probability 1/2), flip the λ input coin and return the result. Otherwise, flip the μ input coin and return the result."
2. Run a **linear Bernoulli factory** using the ν input coin, $x/y = 2/1$, and $\epsilon = \epsilon$, and return the result.

4.6.2 $\lambda - \mu$

(Nacu and Peres 2005, proposition 14(iii-iv))[132]. This algorithm takes two input coins that simulate λ or μ, respectively, and a parameter ϵ such that $0 < \epsilon \le \lambda - \mu$ (the greater ϵ is, the more efficient).

1. Create a ν input coin that does the following: "Generate an unbiased random bit. If that bit is 1 (which happens 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. Run a **linear Bernoulli factory** using the ν input coin, $x/y = 2/1$, and $\epsilon = \epsilon$, and return 1 minus the result.

4.6.3 ϵ / λ

(Lee et al. 2014)[133]. In the following algorithm:

- λ is the probability of heads of an input coin.
- ϵ is a rational number that satisfies $0 < \epsilon \leq \lambda \leq 1$.

The algorithm follows.

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 a **linear Bernoulli factory** with the μ input coin, $x/y = 1 / (1 - \epsilon)$, and $\epsilon = \gamma$. If the result is 0, return 0. Otherwise, go to step 3. (Running the linear Bernoulli factory this way simulates the probability $(\lambda - \epsilon)/(1 - \epsilon)$ or $1 - (1 - \lambda)/(1 - \epsilon)$).

4.6.4 μ / λ

(Morina 2021)[134]. In this division algorithm:

- μ is the probability of heads of an input coin and represents the dividend.
- λ is the probability of heads of another input coin, represents the divisor, and satisfies $0 \leq \mu < \lambda \leq 1$.
- ϵ is a rational number that satisfies $0 < \epsilon \leq \lambda - \mu$. ϵ can be a positive rational number that equals a lower bound for λ minus an upper bound for μ.

The algorithm follows.

- Do the following process repeatedly, until this algorithm returns a value:
  1. Generate an unbiased random bit (either 0 or 1 with equal probability).
  2. If the bit generated in step 1 is 1, flip the μ input coin. If it returns 1, return 1.
  3. If the bit generated in step 1 is 0, run the **algorithm for $\lambda - \mu$** with $\epsilon = \epsilon$. If it returns 1, return 0.

4.6.5 $\lambda * x/y$

In general, this function will touch 0 or 1 at some point greater than 0 and less than 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 in this document comes from Huber (2014)[135]. It uses three parameters:

- x and y are integers such that $x/y > 0$ and $y!=0$.
- ϵ is a rational number greater than 0 and less than 1. If x/y is greater than 1, ϵ must be such that $0 < \epsilon \leq 1 - \lambda * x/y$, in order to bound the function away from 0 and 1. The greater ϵ is, the more efficient.

As a result, some knowledge of λ has to be available to the algorithm. 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 do the following: "With probability x/y, flip the input coin and return the result; otherwise return 0."
2. Set c to x/y, and set k to $23 / (5 * \epsilon)$.
3. If ϵ is greater than 644/1000, set ϵ to 644/1000.
4. Set i to 1.
5. While i is not 0:
    1. Flip the input coin. If it returns 0, then generate numbers that are each 1 with probability $(c - 1) / c$ and 0 otherwise, until 1 is generated this way, then add 1 to i for each number generated this way (including the last).
    2. Subtract 1 from i.
    3. If i is k or greater:
        1. Generate i numbers that are each 1 with probability $2 / (\epsilon + 2)$ or 0 otherwise. If any of those numbers is 0, return 0.
        2. Multiply c by $2 / (\epsilon + 2)$, then divide ϵ by 2, then multiply k by 2.
6. (i is 0.) Return 1.

Huber (2016)[136] presented a second algorithm using the same three parameters, but it's omitted here because it appears to perform worse than the algorithm given above and the **algorithm for $(\lambda * x/y)^i$**

below (see also Morina 2021[137]).

Huber (2016) also included a third algorithm that simulates $\lambda * x / y$. The algorithm works only if $\lambda * x / y$ is known to be less than 1/2. This third algorithm takes three parameters:

- x and y are integers such that $x/y > 0$ and $y!=0$.
- m is a rational number such that $\lambda * x / y \leq m < 1/2$.

The algorithm follows.

1. The same special cases as for the first algorithm in this section 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 a **linear Bernoulli factory** with $x/y = (x/y) / (2 * m)$ and $\epsilon = 1 - m$.

> **Note:** For approximate methods to simulate $\lambda*(x/y)$, see the page "**Supplemental Notes for Bernoulli Factory Algorithms**".

4.6.6 $(\lambda * x/y)^i$

(Huber 2019)[138]. This algorithm uses four parameters:

- x and y are integers such that $x/y > 0$ and $y!=0$.
- i is an integer 0 or greater.
- ϵ is a rational number such that $0 < \epsilon < 1$. If x/y is greater than 1, ϵ must be such that $0 < \epsilon \leq 1 - \lambda * x/y$.

The algorithm 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 do the following: "With probability x/y, flip the input coin and return the result; otherwise return 0."
3. Special case: If x is less than y, then create a secondary coin that does the following: "With probability x/y, flip the input coin and return the result; otherwise return 0", then flip the secondary coin i times until a flip returns 0, whichever comes first, then return a number that is 1 if all the flips, including the last, return 1, or 0 otherwise.

4. Set t to 355/100 and c to x/y.
5. While i is not 0:
    1. While $i > t / \epsilon$:
        1. Set β to $(1 - \epsilon / 2) / (1 - \epsilon)$.
        2. Run the **algorithm for $(a/b)^z$** (given in the irrational constants section) with a=1, b=β, and $z = i$. If the run returns 0, return 0.
        3. Multiply c by β, then divide ϵ by 2.
    2. Run the **logistic Bernoulli factory** with $c/d = c$, then set z to the result. Set i to $i + 1 - z * 2$.
6. (i is 0.) Return 1.

4.6.7 Linear Bernoulli Factories

In this document, a **linear Bernoulli factory** refers to one of the following:

- The first algorithm for $\boldsymbol{\lambda * x/y}$ with the stated parameters x, y, and ϵ.
- The **algorithm for $(\lambda * x/y)^i$** with the stated parameters x, y, and ϵ, and with $i = 1$ (see previous section).

4.6.8 λ^μ

This algorithm is based on the **algorithm for $\lambda^{x/y}$**, but changed to accept a second input coin (which outputs heads with probability μ) rather than a fixed value for the exponent. For this algorithm, λ and μ may not both be 0.

- Set i to 1. Then do the following process repeatedly, until this algorithm returns a value:
    1. Flip the input coin that simulates the base, λ; if it returns 1, return 1.
    2. Flip the input coin that simulates the exponent, μ; if it returns 1, return 0 with probability $1/i$.
    3. Add 1 to i.

4.6.9 $(1-\lambda)/\cos(\lambda)$

(Flajolet et al., 2010)[139]. Uses an average number of flips that grows without bound as λ goes to 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 u, a uniform random variate between 0 and 1, then set i to 1.
4. While i is less than G:
    1. Generate a uniform random variate between 0 and 1 V.
    2. If i is odd[140] and V is less than U, return 0.
    3. If i is even[141] and U is less than V, return 0.
    4. Add 1 to i, then set U to V.
5. Return 1.

4.6.10 (1−λ) * tan(λ)

(Flajolet et al., 2010)[142]. Uses an average number of flips that grows without bound as λ goes to 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 u, a uniform random variate between 0 and 1, then set i to 1.
4. While i is less than G:
    1. Generate a uniform random variate between 0 and 1 V.
    2. If i is odd[143] and V is less than U, return 0.
    3. If i is even[144] and U is less than V, return 0.
    4. Add 1 to i, then set U to V.
5. Return 1.

4.6.11 ln((*c* + *d* + λ)/*c*)

In this algorithm, d and c are integers, $0 < c$, and $c > d \geq 0$, and $(c + d + \lambda)/c \leq \exp(1)$.

- Do the following process repeatedly, until this algorithm returns a value:
    1. Generate an unbiased random bit. If that bit is 1 (which happens with probability 1/2), run the **algorithm for (*d* + λ) / *c*** with $d = d$ and $c = c$, and return the result.
    2. Generate u, a uniform random variate between 0 and 1, if u wasn't generated yet.
    3. **Sample from the number *u***, then run the **algorithm for (*d***

+ λ) / c with $d = d$ and $c = c$. If both calls return 1, return 0.

4.6.12 arcsin(λ) / 2

The Flajolet paper doesn't explain in detail how arcsin(λ)/2 arises out of arcsin(λ) + sqrt(1 − λ^2) − 1 via Bernoulli factory constructions, but here is an algorithm.[145] However, the number of input coin flips is expected to grow without bound as λ approaches 1.

1. Generate an unbiased random bit. If that bit is 1 (which happens with probability 1/2), run the **algorithm for arcsin(λ) + sqrt(1 − λ^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 − λ^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 − λ^2) − 1 part and divides by 2.)

4.7 Other Factory Functions

Algorithms in bold are given in this page.

To simulate:	Follow this algorithm:
1/sqrt(π)	Create λ coin for algorithm **1/π**. Run algorithm for **sqrt(λ)**.
1/sqrt(h+λ)	(λ is unknown heads probability of a coin; h≥1 is a rational number.) Create μ coin for algorithm **d/(c+λ)** with $c=h$ and $d=1$. Run algorithm for **sqrt(λ)** with λ being the μ coin.
1 / (c + λ)	(λ is unknown heads probability of a coin; c≥1 is a rational number.) Run algorithm for **d / (c + λ)** with $d = 1$.
1 / (1 + λ^2)	(Slope function of arctan(λ). λ is unknown heads probability of a coin.) Create μ coin that flips λ coin twice and returns either 1 if both flips return 1, or 0 otherwise.

	Run algorithm for ***d* / (*c* + *λ*)** with d=1, c=1, and λ being the μ coin.
$1 / (c + \exp(-z))$	($z \geq 0$ is written as described in **"ExpMinus" section**; $c \geq 1$ is a rational number.) Create μ coin for **ExpMinus** algorithm with parameter z. Run algorithm for ***d* / (*c* + *λ*)** with d=1, c=c, and λ being the μ coin.
$1/(2^{k + \lambda})$ or $\exp(-(k + \lambda)*\ln(2))$	(λ is unknown heads probability of a coin. $k \geq 0$ is an integer.) Run algorithm **1/(2^*m**(*k* + *λ*))** with k=k and m=1.
$1-\exp(-z) = (\exp(z)-1) * \exp(-z) = (\exp(z)-1) / \exp(z)$	($z \geq 0$ is written as described in **"ExpMinus" section**.) Run **ExpMinus** algorithm with parameter z, and return 1 minus the result.
$\exp(-((1-\lambda)^1 * c))$	((Dughmi et al. 2021)[146]; applies an exponential weight—here, c—to an input coin. λ is unknown heads probability of a coin.) (1) If c is 0, return 1. (2) Generate N, a Poisson random variate with mean c. (3) Flip the input coin until the flip returns 0 or the coin is flipped N times, whichever comes first, then return a number that is 1 if N is 0 or all of the coin flips (including the last) return 1, or 0 otherwise.
$\exp(\lambda^2) - \lambda*\exp(\lambda^2)$	(λ is unknown heads probability of a coin.) Run **general martingale algorithm** with $g(\lambda)=\lambda$, $d_0=1$, and $a_i=\frac{(-1)^i}{(\text{floor}(i/2))!}$.
$1 - 1 / (1+(\mu*\lambda/(1 - \mu)) = (\mu*\lambda/(1 - \mu) / (1+(\mu*\lambda/(1 - \mu))$	(Special case of **logistic Bernoulli factory**; $0 \leq \lambda \leq 1$, $0 \leq \mu < 1$, and both are unknown heads probabilities of two coins.) (1) Flip the μ coin. If it returns 0, return 0. (Coin samples probability $\mu/(\mu + (1 - \mu)) = \mu$.) (2) Flip the λ coin. If it returns 1, return 1. Otherwise, go to step 1.
	(λ is unknown heads probability of a coin.)

$\lambda/(1+\lambda)$	Run algorithm for **$1/(1+\lambda)$**, then return 1 minus the result.
$c * \lambda / (c * \lambda + (d+\mu)) = (c/(d+\mu)) * \lambda / (1 + (c/(d+\mu)) * \lambda))$	(c≥0 is an integer; d≥0 is an integer; $0 \le \lambda \le 1$, $0 \le \mu < 1$, and both are unknown heads probabilities of two coins.) (1) If c is 0, return 0. (2) Let $D = d$ and $C = c$, then run the algorithm for **$(d + \mu) / (c + \lambda)$** with λ and μ both being the μ input coin, with $d = D$, and with $c = D + C$. If the run returns 1, return 0. (3) Flip the λ input coin. If the flip returns 1, return 1. Otherwise, go to step 2.
$(d+\mu) / (c * \lambda + (d+\mu))$	(c, d, λ, and μ are as in the previous algorithm.) Run the previous algorithm and return 1 minus the result.
$\exp(z)/(1+\exp(z))^2$	(Equals expit(z)*(1−expit(z)). z is described in "expit(z)" section.) Run the algorithm for **expit(z)** twice, with m=0. If the first run returns 1 and the second returns 0, return 1. Otherwise, return 0.
$\nu * 1 + (1 - \nu) * \mu = \nu + \mu - (\nu*\mu)$	(*Logical OR*. Flajolet et al., 2010[147]. Special case of $\nu * \lambda + (1 - \nu) * \mu$ with $\lambda = 1$. ν and μ are unknown heads probabilities of two coins.) Flip the ν input coin and the μ input coin. Return 1 if either flip returns 1, and 0 otherwise.
$1 - \nu$	(*Complement*. Flajolet et al., 2010[148]. Special case of $\nu * \lambda + (1 - \nu) * \mu$ with $\lambda = 0$ and $\mu = 1$. ν is unknown heads probability of a coin.) Flip the ν input coin and return 1 minus the result.
$\nu * \lambda$	(*Logical AND* or *Product*. Flajolet et al., 2010[149]. Special case of $\nu * \lambda + (1 - \nu) * \mu$ with $\mu = 0$. ν and λ are unknown heads probabilities of two coins.) Flip the ν input coin and the λ input coin. Return 1 if both flips return 1, and 0 otherwise.
	(*Mean*. Nacu and Peres 2005, proposition 14(iii)[150]; Flajolet et al., 2010[151]. Special case

$(\lambda + \mu)/2 = (1/2)*\lambda + (1/2)*\mu$	of $\nu * \lambda + (1 - \nu) * \mu$ with $\nu = 1/2$. λ and μ are unknown heads probabilities of two coins.) Generate an unbiased random bit. If that bit is 1 (which happens with probability 1/2), flip the λ input coin and return the result. Otherwise, flip the μ input coin and return the result.
$(1+\lambda)/2 = (1/2) + (1/2)*\lambda$	(λ is unknown heads probability of a coin.) Generate an unbiased random bit. If that bit is 1, return 1. Otherwise, flip the input coin and return the result.
$(1-\lambda)/2$	(λ is unknown heads probability of a coin.) Generate an unbiased random bit. If that bit is 1, return 0. Otherwise, flip the input coin and return 1 minus the result.
$1 - \ln(1+\lambda)$	(λ is unknown heads probability of a coin.) Run algorithm for **$\ln(1+\lambda)$**, then return 1 minus the result.[152]
$\sin(\sqrt{\lambda}*\sqrt{c}) / (\sqrt{\lambda}*\sqrt{c})$	(c is a rational number; $0 < c \leq 6$. λ is unknown heads probability of a coin.) Run **general martingale algorithm** with $g(\lambda)=\lambda$, and with $d_0 = 1$ and coefficients $a_i = \frac{ (-1)^{i} c^{i}} {(i+i+1)!}$.
$3 - \exp(1)$	Run the algorithm for **$\exp(1) - 2$**, then return 1 minus the result.
$1/(\exp(1)-1)$	Run the algorithm for **$1/(\exp(1)+c-2)$** with $c = 1$.
r/π	(r is a rational number; $0 \leq r \leq 3$.) Create λ coin for algorithm **$\pi - 3$**. Create μ coin that does: "With probability $r - \text{floor}(r)$, return 1; otherwise return 0." If $r=0$, return 0. If $r=3$, run algorithm for **$d / (c + \lambda)$** with $d=n$ and $c=3$. If $0 < r < 3$, run algorithm for **$(d + \mu) / (c + \lambda)$** with $d=\text{floor}(r)$ and $c=3$.
$\exp(1)/\pi$	Create μ coin for algorithm **$\exp(1) - 2$**. Create λ coin for algorithm **$\pi - 3$**. Run algorithm for **$(d + \mu) / (c + \lambda)$** with $d=2$ and $c=3$.

exp(1)/4	Generate unbiased random bits (each bit is 0 or 1 with equal probability) until a zero is generated this way, then set n to the number of ones generated this way. Set n to 2*n + 2. With probability $2^{n-1}/(n!)$, return 1. Otherwise return 0.
r*λ − r + r*exp(−λ)	(r is a rational number greater than 0, but not greater than 2. λ is the unknown heads probability of a coin.) Run the **general martingale algorithm** with $g(\lambda) = \lambda$, and with $d_0 = r/2$ and coefficients $a_i = \frac{r}{(i)!} (-1)^i$ if $i\ge 2$ and $a_i=0$ otherwise.
r*exp(−1) = r/exp(1)	(r is a rational number; 0≤r≤2.) If r=0, return 0. If r=2, un algorithm for ***d* / (*c* + *λ*)** with d=n and c=2. If 0<r<2, run algorithm for ***c**λ − *c* + *c**exp(−*λ*)** with r=r and λ being a coin that always returns 1.
λ/(2−λ) = (λ/2)/(1−(λ/2))	(λ is the unknown heads probability of a coin.) (1) Flip λ coin; return 0 if it returns 0. (2) Run algorithm for **1/(2−*λ*)**.
(1−λ)/(1+λ)	(λ is the unknown heads probability of a coin.) (1) Flip λ coin; return 0 if it returns 1. (2) Run algorithm for ***d* / (*c* + *λ*)** with d=1 and c=1.

4.8 Algorithms for Specific Constants

This section shows algorithms to simulate a probability equal to a specific kind of irrational number.

4.8.1 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 (about 0.618), whose continued fraction's partial denominators are 1, 1, 1, 1, ...

1. Generate an unbiased random bit. If that bit is 1 (which happens with probability 1/2), return 1.
2. Do a separate run of the currently running algorithm. If the separate run returns 1, return 0. Otherwise, go to step 1.

4.8.2 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. Do a separate run of the currently running algorithm. If the separate run returns 1, return 0. Otherwise, go to step 1.

4.8.3 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. Do a separate run of the currently running algorithm, but with *pos* = *pos* + 1. If the separate run returns 1, return 0. Otherwise, go to step 1.

4.8.4 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. Do a separate run of the currently running algorithm, but with $k = k + 4$. If the separate run returns 1, return 0. Otherwise, go to step 1.

4.8.5 arctan(*x*/*y*) * *y*/*x*

(Flajolet et al., 2010)[153]:

1. Generate u, a uniform random variate between 0 and 1.
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[154] is equivalent to the two-coin algorithm, which has bounded expected running time for all λ parameters, the algorithm above can be modified as follows:

1. Generate an unbiased random bit. If that bit is 1 (which happens with probability 1/2), return 1.
2. Generate u, a uniform random variate between 0 and 1, 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.

4.8.6 *π* / 12

Two algorithms:

- First algorithm: Use the algorithm for **arcsin(λ) / 2**, but 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 an algorithm for **π / 4** and return the result.

4.8.7 *π* / 4

Three algorithms:

- First algorithm (Flajolet et al., 2010)[155]: Generate a random integer n satisfying $0 \le n \le 5$, call it n. If n is less than 3, return the result of the **algorithm for arctan(*x*/*y*) * *y*/*x*** with x=1 and y=2. Otherwise, if n is 3, return 0. Otherwise, return the result of the **algorithm for arctan(*x*/*y*) * *y*/*x*** with x=1 and y=3.
- Second algorithm (since arctan(1) = π / 4): Run the second **algorithm for arctan(*x*/*y*) * *y*/*x*** with x=1 and y=1.
- Third algorithm: See the appendix.

A fourth algorithm to sample $\pi/4$ is based on the section "**Uniform Distribution Inside N-Dimensional Shapes**", especially its Note 5, in "More Algorithms for Arbitrary-Precision Sampling". In effect, it samples a 2-dimensional point with coordinates between 0 and 1 and determines if that point is within 1 unit of the origin (0, 0), which will happen with probability $\pi/4$.

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

4.8.8 $\pi/4 - 1/2$ or $(\pi - 2)/4$

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

1. Set S to 2. Then set *c1* and *c2* to 0.
2. Do the following process repeatedly, until the algorithm returns a value:
    1. Set *c1* to 2**c1* plus an unbiased random bit (either 0 or 1 with equal probability). Then, set *c2* to 2**c2* plus an unbiased random bit.
    2. Set *diamond* to *MAYBE* and *disk* to *MAYBE*.

3. If ((*c1*+1) + (*c2*+1)) < *S*, set *diamond* to *YES*.
4. If ((*c1*) + (*c2*)) > *S*, set *diamond* to *NO*.
5. If ((*c1*+1)^2 + (*c2*+1)^2) < *S*^{2}, set *disk* to *YES*.
6. If ((*c1*)^2 + (*c2*)^2) > *S*^{2}, set *disk* to *NO*.
7. If *disk* is *YES* and *diamond* is *NO*, return 1. Otherwise, if *diamond* is *YES* or *disk* is *NO*, return 0.
8. Multiply *S* by 2.

4.8.9 (π − 3)/4

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

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

4.8.10 π − 3

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

1. Set *S* to 2. Then set *c1* and *c2* to 0.
2. Do the following process repeatedly, until the algorithm aborts it or returns a value:

1. Set S to 32. Then set $c1$ to a uniform random integer in the half-open interval [0, S) and $c2$ to another uniform random integer in [0, S).
2. (Return 1 if in retained boxes.) If $c1$ is 0 and $c2$ is 0, or if $c1$ is 0 and $c2$ is 1, return 1.
3. (Check if outside removed boxes.) If $((c1+1)^2 + (c2+1)^2) >= 1024$, abort this process and go to step 3. (Otherwise, $c1$ and $c2$ are rejected and this process continues.)

3. Set S to 64.
4. (Sample the modified quarter disk.) Do the following process repeatedly, until the algorithm returns a value:
   1. Set $c1$ to $2*c1$ plus an unbiased random bit (either 0 or 1 with equal probability). Then, set $c2$ to $2*c2$ plus an unbiased random bit.
   2. If $((c1+1)^2 + (c2+1)^2) < S^2$, return 1. (Point is inside the quarter disk, whose area is $\pi/4$.)
   3. If $((c1)^2 + (c2)^2) > S^2$, return 0. (Point is outside the quarter disk.)
   4. Multiply S by 2.

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

4.8.11 4/(3*π)

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

1. Set S to 2. Then set $c1$ and $c2$ to 0.
2. Do the following process repeatedly, until the algorithm returns a value:
   1. Set $c1$ to $2*c1$ plus an unbiased random bit (either 0 or 1 with equal probability). Then, set $c2$ to $2*c2$ plus an unbiased random bit.
   2. If $((c1+1)^2 + (c2+1)^2) < S^2$, do the following. (Point is inside the quarter disk, whose area is $\pi/4$. Now $c1$, the point's x coordinate, is treated as a uniform random variate between $c1/S$ and $(c1+1)/S$, and the following substeps return 1 with

probability equal to that variate.)

   1. Generate z, a uniform random integer in the interval [0, S). If z is less than $c1$, return 1. If z is greater than $c1$, return 0.
   2. Generate two unbiased random bits (each either 0 or 1 with equal probability). If the bits are different, return the first bit. Otherwise, repeat this substep.
3. If $((c1)^2 + (c2)^2) > S^2$, abort these substeps and go to step 1 ("Set S..."). (Point is outside the quarter disk.)
4. Multiply S by 2.

> **Note:** The mean value 4/(3*π) can be derived as follows. The relative probability that x is "close" to z, where $0\le z \le 1$, is $p(z)$ = sqrt(1 − z*z). Now find the integral of z*$p(z)$/c (where c=π/4 is the integral of $p(z)$ on the closed unit interval); see "[**Integrals**". The result is the mean value 4/(3*π). The following code in the Python programming language prints this mean value using the SymPy computer algebra library: `p=sqrt(1-z*z); c=integrate(p,(z,0,1)); print(integrate(z*p/c,(z,0,1)));`.

4.8.12 1 / π

(Flajolet et al., 2010)[156]:

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. (In fact, 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.

4.8.13 (a/b)^z

In the algorithm below:

- $a \geq 0$ is an integer.
- $b > 0$ is an integer.
- z is a number (positive or not), and its absolute value (abs(z)) is written as a rational number (case 1), as an integer and fractional part (case 2), or as a sum of positive numbers (case 3), as described in the **"ExpMinus" section**.
- If z is known to be 0 or greater then it must be that $0 \leq a/b \leq 1$, or
- If z is known to be less than 0, then it must be that $a/b \geq 1$.

The algorithm follows.

- In case 1 ($z = x/y$):
  1. If z is known to be less than 0, swap a and b, and remove the sign from z. If a/b is now less than 0 or greater than 1, return an error.
  2. If x equals y, 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) (equivalent to x - y*floor(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 at random a number that is 1 with probability a^{ipart}/b^{ipart} or 0 otherwise. (Or generate, at random, *ipart* many 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. (Note on steps 5 to 8: This case where $0 < x/y < 1$ is due to recent work by Mendo (2019)[157].) 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.
- In case 2 (abs(z) = $m + \nu$; here, $0 < \nu \leq 1$ unless a/b is not zero):
  1. If z is known to be less than 0, swap a and b, and remove the

sign from z. If a/b is now less than 0 or greater than 1, return an error.
2. If a is 0 and m is not 0, return 0. If a equals b, return 1.
3. If m is 1 or greater, generate at random a number that is 1 with probability a^m/b^m or 0 otherwise. (Or generate, at random, m many 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. (Note on steps 4 to 7: This case where $0 < z < 1$ is due to recent work by Mendo (2019)[158].) Set i to 1.
5. With probability a/b, return 1.
6. Flip the ν input coin. If it returns 0, return 0 with probability $1/i$.
7. Add 1 to i and go to step 6.

- In case 3:
1. If z is known to be less than 0, swap a and b, and remove the sign from z. If a/b is now less than 0 or greater than 1, return an error.
2. If a is 0, return 0 (z will be positive here). If a equals b, return 1.
3. Rewrite the z parameter's absolute value as a sum of positive numbers. For each number, run either case 1 or case 2 (depending on how the number is written) of this algorithm with that number as the parameter. If any of these runs returns 0, return 0; otherwise, return 1.

4.8.14 1/(exp(1) + c − 2)

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

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

- Do the following process repeatedly until this run of the algorithm returns a value:
1. If *pos* is divisible by 3 (that is, if rem(*pos*, 3) equals 0): Let k be (*pos*/3)*2. With probability $k/(1+k)$, return a number that is 1 with probability $1/k$ and 0 otherwise.
2. If *pos* is 1: With probability $c/(1+c)$, return a number that is 1 with probability $1/c$ and 0 otherwise.

3. If *pos* is greater than 1 and not divisible by 3: Generate an unbiased random bit. If that bit is 1 (which happens with probability 1/2), return 1.
4. Do a separate run of the currently running algorithm, but with *pos* = *pos* + 1. If the separate run returns 1, return 0.

4.8.15 exp(1) − 2

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

4.8.16 $\zeta(3) * 3 / 4$ and Other Zeta-Related Constants

(Flajolet et al., 2010)[159]. It can be seen as a triple integral of the function $1/(1 + a * b * c)$, where a, b, and c are uniform random variates between 0 and 1. This algorithm is given below, but using the two-coin algorithm instead of the even-parity construction[160]. Here, $\zeta(x)$ is the Riemann zeta function.

1. Generate three uniform random variates between 0 and 1.
2. Generate an unbiased random bit. If that bit is 1 (which happens 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 variates.)

> **Note:** The triple integral in section 5 of the paper is $\zeta(3) * 3 / 4$, not $\zeta(3) * 7 / 8$.

This can be extended to cover any constant of the form $\zeta(k) * (1 - 2^{-(k-1)})$ where $k \geq 2$ 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 random variates between 0 and 1.
2. Generate an unbiased random bit. If that bit is 1 (which happens 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.

4.8.17 erf(*x*)/erf(1)

In the following algorithm, *x* is a real number that is 0 or greater and 1 or less.

1. Generate a uniform random variate between 0 and 1, 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, *v* is created, which is the maximum of two uniform [0, 1] random variates.) Generate two uniform random variates between 0 and 1, 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[161], 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)[162]. 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.

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

In this algorithm, *m* must be greater than 0, and *x* is a real number that is 0 or greater and 1 or less.

1. Set *ret* to a number distributed as the maximum of *m* uniform random variates between 0 and 1. (See note 1 below.)
2. Set *k* to 1, then set *u* to point to the same value as *ret*.
3. Generate a uniform random variate between 0 and 1, call it *v*.
4. If *v* is less than *u*: Set *u* to *v*, then add 1 to *k*, then go to step 3.
5. If *k* is odd[163], return a number that is 1 if *ret* is less than *x* and 0

otherwise. If k is even[164], go to step 1. (If *ret* is implemented as a uniform partially-sampled random number, this comparison should be done via the **URandLessThanReal algorithm**, which is described in my **article on PSRNs**.)

> **Notes:**
>
> 1. In step 1 of the algorithm above, *ret* is distributed as u, where $u^{1/m}$ where u is a uniform random variate between 0 and 1.(Devroye 1986, p. 431)[165] (This formula works for every m greater than 0, not just integers.) Alternatively, *ret* can be generated using the **kthsmallest** algorithm with the two parameters m and m (see "**Partially-Sampled Random Numbers"**), but then m must be an integer. Alternatively, *ret* can be generated as follows, but then m must be an integer:
>
>   1. Generate x and y, two uniform random variates between 0 and 1.
>   2. Do the following m times. If x is less than y, set x to point to y; either way, set y to a new uniform random variate between 0 and 1.
>   3. Set *ret* to point to x.
>
> 2. Derivation: See Formula 1 in the section "**Probabilities Arising from Certain Permutations**", where:
>
>   - `ECDF(x)` is the probability that a uniform random variate between 0 and 1 is x or less, namely x if x is greater than 0 and less than 1; 0 if x is 0 or less; and 1 otherwise.
>   - `DPDF(x)` is the probability density function for the maximum of m uniform random variates between 0 and 1, namely $m*x^{m-1}$ if x is greater than 0 and less than 1, and 0 otherwise.

4.9.1 Euler–Mascheroni constant γ

The following algorithm to simulate the Euler–Mascheroni constant γ (about 0.5772) is due to Mendo (2020/2021)[166]. This solves an open question given in (Flajolet et al., 2010)[167]. An algorithm for the Euler–Mascheroni constant appears here even though it is not yet known

whether this constant is irrational. Sondow (2005)[168] described how the Euler–Mascheroni constant can be rewritten as an infinite sum, which is the form used in this algorithm.

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 $lamunq+\epsilon \le lam + 1/(2^k)$, 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 infinite sum for γ to $lamunq$, and sets ϵ to an upper bound on the error that results if the infinite sum is "cut off" 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 integer $b \ge 1$ 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 $lamunq+\epsilon \le bound$, set s to 0. Otherwise, if $lamunq > bound$, set s to 2. Otherwise, set s to 1.
9. Generate an unbiased random bit. If that bit is 1 (which happens 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.

> **Note:** The following is another algorithm for this constant. As **I learned**, the fractional part of $1/U$, where U is a uniform random variate between 0 and 1, has a mean equal to 1 minus the Euler–Mascheroni constant γ, about 0.5772.[169] This leads to the following algorithm to sample a probability equal to γ:
>
> 1. Generate a random variate of the form $1/U - \text{floor}(1/U)$, where U is a uniform random variate between 0 and 1. This can be done by generating a uniform PSRN for **the reciprocal of a uniform random variate**, then setting that PSRN's integer part to 0. Call the variate (or PSRN) f.
> 2. **Sample from the number *f*** (for example, call **SampleGeometricBag** on f if f is implemented as a uniform PSRN). Return 0 if the run returns 1, or 1 otherwise.

4.9.2 exp(−*x*/*y*) * *z*/*t*

This algorithm is again based on an algorithm due to Mendo (2020/2021)[170]. The algorithm takes integers $x \ge 0$, $y > 0$, $z \ge 0$, and $t > 0$, such that $0 \le \exp(-x/y) * z/t \le 1$.

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 $lamunq+\epsilon \le lam + 1/(2^k)$, 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 well-known infinite sum, 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 $lamunq+\epsilon \le bound$, set s to 0. Otherwise, if $lamunq > bound$, set s to 2. Otherwise, set s to 1.
10. Generate an unbiased random bit. If that bit is 1 (which happens 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.

4.9.3 Certain Numbers Based on the Golden Ratio

The following algorithm given by Fishman and Miller (2013)[171] finds the continued fraction expansion of certain numbers described as—

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

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

First, define the following operations:

- **Get the previous and next Fibonacci-based number given k, m, and ℓ**:
  1. If k is 0 or less, return an error.
  2. Set $g0$ to 0, $g1$ to 1, x to 0, and y to 0.

3. Do the following k times: Set y to $m * g1 + \ell * g0$, then set x to $g0$, then set $g0$ to $g1$, then set $g1$ to y.
4. Return x and y, in that order.

- **Get the partial denominator given *pos*, *k*, *m*, and ℓ** (this partial denominator is part of the continued fraction expansion found by Fishman and Miller):
  1. **Get the previous and next Fibonacci-based number given *k*, *m*, and ℓ**, call them p and n, respectively.
  2. If ℓ is 1 and k is odd[172], return $p + n$.
  3. If ℓ is -1 and pos is 0, return $n - p - 1$.
  4. If ℓ is 1 and pos is 0, return $(n + p) - 1$.
  5. If ℓ is -1 and pos is even[173], return $n - p - 2$. (The paper had an error here; the correction given here was verified by Miller via personal communication.)
  6. If ℓ is 1 and pos is even[174], return $(n + p) - 2$.
  7. Return 1.

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

1. **Get the partial denominator given *pos*, *k*, *m*, and ℓ**, call it kp.
2. Do the following process repeatedly, until this run of the algorithm returns a value:
   1. With probability $kp/(1 + kp)$, return a number that is 1 with probability $1/kp$ and 0 otherwise.
   2. Do a separate run of the currently running algorithm, but with $pos = pos + 1$. If the separate run returns 1, return 0.

4.9.4 ln(1+y/z)

See also the algorithm given earlier for $\ln(1+\lambda)$. In this algorithm, y/z is a rational number that is 0 or greater and 1 or less. (Thus, the special case ln(2) results when $y/z = 1/1$.)

1. If y is 0, return 0.
2. Do the following process repeatedly, until this algorithm returns a value:
   1. Generate an unbiased random bit. If that bit is 1 (which happens with probability 1/2), return a number that is 1 with

probability y/z and 0 otherwise.

2. Generate u, a uniform random variate between 0 and 1, if u wasn't generated yet.
3. **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.

4.9.5 $\ln(\pi)/\pi$

Special case of the algorithm for $\ln(c+\lambda)/(c+\lambda)$.

1. Run the algorithm for $1/\pi$ repeatedly, until the run returns 1, then set g to the number of runs that returned 0 this way.
2. If g is 0, return 0. Otherwise, return a number that is 1 with probability $1/g$ or 0 otherwise.

5 Requests and Open Questions

See my page "**Open Questions on the Bernoulli Factory Problem**" for open questions, answers to which will greatly improve my articles on Bernoulli factories. These questions include:

- **Polynomials that approach a factory function "fast"**.
- **New coins from old, smoothly**.
- **Tossing Heads According to a Concave Function**.
- **Simulable and strongly simulable functions**.
- **Multiple-Output Bernoulli Factories**.
- **From coin flips to algebraic functions via pushdown automata**.

Other questions:

- **Probabilities arising from permutations**.
- 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**".

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

8 Notes

9 Appendix

9.1 Using the Input Coin Alone for Randomness

A function $f(\lambda)$ is *strongly simulable* (Keane and O'Brien 1994)[175] if there is a Bernoulli factory algorithm for that function that uses *only* the input coin as its source of randomness.

If a Bernoulli factory algorithm uses a fair coin, it can often generate flips of the fair coin using the input coin instead, with the help of ***randomness extraction*** techniques.

> **Example:** If a Bernoulli factory algorithm would generate an unbiased random bit, instead it could flip the input coin twice until the flip returns 0 then 1 or 1 then 0 this way, then take the result as 0 or 1, respectively (von Neumann 1951)[176]. But this trick works only if the input coin's probability of heads is neither 0 nor 1.

When Keane and O'Brien (1994)[177] introduced Bernoulli factories, they showed already that $f(\lambda)$ is strongly simulable whenever it admits a Bernoulli factory and its domain includes neither 0 nor 1 (so the input coin doesn't show heads every time or tails every time) — just use the von Neumann trick as in the example above. But does f remain strongly simulable if its domain includes 0 and/or 1? That's a complexer question; see the **supplemental notes**.

9.2 The Entropy Bound

There is a lower bound on the average number of coin flips needed to turn a coin with one probability of heads (λ) into a coin with another ($\tau = f(\lambda)$). It's called the *entropy bound* (see, for example, (Pae 2005)[178], (Peres 1992)[179]) 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, an algorithm whose only randomness comes from the input coin will require more coin flips to simulate that constant, the more strongly that coin leans towards heads or tails. But this formula works only for such algorithms, even if f isn't a constant.

For certain values of λ, Kozen (2014)[180] showed a tighter lower bound of this kind, but in general, this bound is not so easy to describe 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 some integer n.

9.3 Bernoulli Factories and Unbiased Estimation

If an algorithm—

- takes flips of a coin with an unknown probability of heads (λ), and
- produces heads with a probability that depends on λ ($f(\lambda)$) and tails otherwise,

the algorithm acts as an *unbiased estimator* of $f(\lambda)$ that produces estimates in [0, 1] with probability 1 (Łatuszyński et al. 2009/2011)[181]. (And an estimator like this is possible only if f is a factory function; see Łatuszyński.) Because the algorithm is *unbiased*, its expected value (or mean or "long-run average") is $f(\lambda)$. Since the algorithm is unbiased and outputs only 0 or 1, this leads to the following: With probability 1, given an infinite sequence of independent outputs of the algorithm, the average of the first n outputs approaches $f(\lambda)$ as n gets *large* (as a result of the *law of large numbers*).

On the other hand—

- estimating λ as λ' (for example, by averaging multiple flips of a λ-coin), then
- calculating $f(\lambda')$,

is not necessarily an unbiased estimator of $f(\lambda)$, even if λ' is an unbiased estimator.

This page focuses on *unbiased* estimators because "exact sampling" depends on being unbiased. See also (Mossel and Peres 2005, section 4)[182].

> **Note:** Bias and variance are the two sources of error in a randomized estimation algorithm. An unbiased estimator has no bias, but is not without error. In the case at hand here, the variance of a Bernoulli factory for $f(\lambda)$ equals $f(\lambda) * (1-f(\lambda))$ and can go as high as 1/4. ("Variance reduction" methods are outside the scope of this document.) An estimation algorithm's *mean squared error* equals variance plus square of bias.

9.4 Correctness Proof for the Continued Logarithm Simulation Algorithm

Theorem. *If the algorithm given in "Continued Logarithms" terminates with probability 1, it 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)[183]. Using Huber's theorem requires proving two things:

- The algorithm finishes with probability 1 by assumption.
- Second, we show the algorithm is locally correct when the recursive call in the loop is replaced with a "black box" 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. Step 2 defines a loop that divides the probability space into three pieces: the first piece takes up one half, the second piece (in the second substep) 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 the first substep with probability p = (1 / $2^{c[i]}$) / 2 and fail either at the first substep of the loop with probability $f1$ = (1 − 1 / $2^{c[i]}$) / 2, or at the second substep 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. □

9.5 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 1 or greater, 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 finishes with probability 1 because with each recursion, the method does a recursive run with no greater probability than not; observe that $a[i]$ can never be more than 1, so that $a[i]/(1+a[i])$, that is, the probability of finishing the run in each iteration, is always 1/2 or greater.
- If the recursive call in the loop is replaced with a "black box" 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. Step 2 defines 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 (in the second substep) 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 the first substep with probability $p = (b[i] / a[pos]) * h$ and fail either at the first substep of the loop with probability $f1 = (1 - b[i] / a[pos]) * h$, or at the second substep 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. □

9.6 Proof of the General Martingale Algorithm

This proof of the **general martingale algorithm** is similar to the proof for certain alternating series with only nonzero coefficients, given in Łatuszyński et al. (2019/2011)[184], section 3.1. Suppose we repeatedly flip a coin that shows heads with probability $g(\lambda)$ and we get the following results: $X_1, X_2, ...$, where each result is either 1 if the coin shows heads or 0 otherwise. Then define two sequences U and L as follows:

- $U_0=d_0$ and $L_0=0$.
- For each $n>0$, U_n is $L_{n-1} + |a_n|\times X_1\times...\times X_n$ if $a_n > 0$, otherwise $U_{n-1} - |a_n|\times X_1\times...\times X_n$ if no nonzero coefficients follow a_n and $a_n < 0$, otherwise U_{n-1}.

- For each $n>0$, L_n is $U_{n-1} - |a_n|\times X_1\times...\times X_n$ if $a_n < 0$, otherwise $L_{n-1} + |a_n|\times X_1\times...\times X_n$ if no nonzero coefficients follow a_n and $a_n > 0$, otherwise L_{n-1}.

Then it's clear that with probability 1, for every $n\ge 1$—

- $L_n \le U_n$,
- U_n is 0 or greater and L_n is 1 or less, and
- $L_{n-1} \le L_n$ and $U_{n-1} \ge U_n$.

Moreover, if there are infinitely many nonzero coefficients, the *U* and *L* sequences have expected values ("long-run averages") converging to $f(\lambda)$ with probability 1; otherwise $f(\lambda)$ is a polynomial in $g(\lambda)$, and U_n and L_n have expected values that approach $f(\lambda)$ as n gets large. These conditions are required for the paper's Algorithm 3 (and thus the **general martingale algorithm**) to be valid.

9.7 Algorithm for sin(λ*π/2)

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

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

> **Notes:**

1. The following is a derivation of this algorithm. Write— $$f(\lambda) = \sin(\lambda \pi/2) = 1-g(1-\lambda),$$ where — $$g(\mu) = 1-\sin((1-\mu) \pi/2)$$ $$= \sum_{n\ge 0} \frac{(\mu\pi/2)^{4n+2}}{(4n+2)!} - \frac{(\mu\pi/2)^{4n+4}}{(4n+4)!}$$ $$= \sum_{n\ge 0} w_n(\mu) = \sum_{n\ge 0} w_n(1) \frac{w_n(\mu)}{w_n(1)}.$$ This is a **convex combination** of $w_n(1)$ and $\frac{w_n(\mu)}{w_n(1)}$ — to simulate $g(\mu)$, first an integer *n* is chosen with probability $w_n(1)$ and then a coin that shows heads with probability $\frac{w_n(\mu)}{w_n(1)}$ is flipped. Finally, to simulate $f(\lambda)$, the input coin is "inverted" ($\mu = 1-\lambda$), $g(\mu)$ is simulated using the "inverted" coin, and 1 minus the simulation result is returned.

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

2. $\sin(\lambda*\pi/2) = \cos((1-\lambda)*\pi/2)$.

9.8 Probabilities Arising from Certain Permutations

Certain interesting probability functions can arise from permutations.

Inspired by the von Neumann schema, the following algorithm can be described:

Let a *permutation class* (defined in **"Flajolet's Probability Simulation Schemes"**) and two distributions *D* and *E*, which are both continuous with probability density functions, be given. Consider

the following algorithm: Generate a sequence of independent random variates (where the first is distributed as D and the rest as E) until the sequence no longer follows the permutation class, then return n, which is how many numbers were generated this way minus 1.

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

| Permutation Class | Distributions D and E | The algorithm returns n with this probability: | The probability that n is ... | --- | --- | --- | --- | --- | | Numbers sorted in descending order | Arbitrary; $D = E$ | n / ((n + 1)!). | Odd is 1−exp(−1); even is exp(−1). See note 3. | | Numbers sorted in descending order | Each arbitrary | ($\int_{(-\infty,\infty)}$ DPDF(z) * ((ECDF(z))$^{n-1}$/((n−1)!) − (ECDF(z))^n/(n!)) dz), for every $n > 0$ (see also proof of Theorem 2.1 of (Devroye 1986, Chapter IV)[185]. DPDF and ECDF are defined later. | Odd is denominator of formula 1 below. | | Alternating numbers | Arbitrary; $D = E$ | (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). See note 3. | | Any | Arbitrary; $D = E$ | ($\int_{[0,\ 1]}$ 1 * (z^{n-1}*V(n)/((n−1)!) − z^n*V(n+1)/(n!)) dz), for every $n > 0$. $V(n)$ is the number of permutations of size n that belong in the permutation class. For this algorithm, $V(n)$ must be greater than 0 and less than or equal to n factorial; this algorithm won't work, for example, if there are 0 permutations of odd size. | Odd is 1 − 1 / EGF(1); even is 1/EGF(1).
Less than k is ($V(0)$ − $V(k)$/(k!)) / $V(0)$. See note 3. |

| Permutation Class | Distributions D and E | The probability that the first number in the sequence is x or less given that n is ... | --- | --- | --- | --- | | Numbers sorted in descending order | Each arbitrary | Odd is $\psi(x)$ = ($\int_{(-\infty,\ x)}$ exp(−ECDF(z)) * DPDF(z) dz) / ($\int_{(-\infty,\ \infty)}$ exp(−ECDF(z)) * DPDF(z) dz) (Formula 1; see Theorem 2.1(iii) of (Devroye 1986, Chapter IV)[186]; see also Forsythe 1972[187]). Here, DPDF is the probability density function (PDF) of D, and ECDF is the cumulative distribution function for E.
If x is a uniform random variate greater than 0 and less than 1, this probability becomes $\int_{[0,\ 1]}$ $\psi(z)$ dz. | | Numbers sorted in descending order | Each arbitrary | Even is ($\int_{(-\infty,\ x)}$ (1 − exp(−ECDF(z))) * DPDF(z) dz) / ($\int_{(-\infty,\ \infty)}$ (1 − exp(−ECDF(z))) * DPDF(z) dz) (Formula 2; see also Monahan 1979[188]). DPDF and ECDF are as above. | | Numbers sorted in descending order | Both uniform variates between

0 and 1 | Odd is ((1−exp(−x)))/(1−exp(−1)). Therefore, the first number in the sequence is distributed as exponential with rate 1 and "cut off" to be not less than 0 and not greater than 1 (von Neumann 1951)[189]. | | Numbers sorted in descending order | D is a uniform variate between 0 and 1; E is max. of two uniform variates between 0 and 1. | Odd is erf(x)/erf(1) (uses Formula 1, where DPDF(z) = 1 and ECDF(z) = z^2 for 0≤z≤1; see also **erf(x)/erf(1)**). |

| Permutation Class | Distributions D and E | The probability that the first number in the sequence is...| --- | --- | --- | --- | | Numbers sorted in descending order | D is an exponential variate with rate 1; E is a uniform variate between 0 and 1. | 1 or less given that n is even is 1 − 2 / (1 + exp(2)) = 1 − (1 + exp(0)) / (1 + exp(1)) = (exp(1)−1)/(exp(1)+1) (uses Formula 2, where DPDF(z) = exp(−z) and ECDF(z) = min(1,z) for z≥0). | | Numbers sorted in descending order | D is an exponential variate with rate 1; E is a uniform variate between 0 and 1. | 1/2 or less given that n is odd is 1 − (1 + exp(1)) / (1 + exp(2)) = (exp(2) − exp(1)) / (exp(2)+1) (uses Formula 1, where DPDF(z) = exp(−z) and ECDF(z) = min(1,z) for z≥0). |

> **Notes:**
>
> 1. All the functions possible for formulas 1 and 2 are nowhere decreasing functions. Both formulas express what are called *cumulative distribution functions*, namely F_D(x given that n is odd) or F_D(x given that n is even), respectively.
> 2. EGF(z) is the *exponential generating function* (EGF) for the kind of permutation involved in the algorithm. For example, the class of *alternating permutations* (permutations whose numbers alternate between low and high, that is, $X1 > X2 < X3 > ...$) uses the EGF tan(λ)+1/cos(λ). Other examples of EGFs were given in the section on the von Neumann schema.
> 3. The results that point to this note have the special case that both D and E are uniform random variates between 0 and 1. Indeed, if each variate x in the sequence is transformed with CDF(x), where CDF is D's cumulative distribution function, then with probability 1, x becomes a uniform random variate greater than 0 and less than 1, with the same numerical order as before. See also **this Stack Exchange question**.

9.9 Derivation of an Algorithm for π / 4

The following is a derivation of the Madhava–Gregory–Leibniz (MGL) generator for simulating the probability $\pi/4$ (Flajolet et al. 2010)[190]. It works as follows. Let S be a set of non-negative integers. Then:

1. Generate a uniform random variate between 0 and 1, call it U.
2. **Sample from the number U** repeatedly until the sampling "fails" (returns 0). Set k to the number of "successes". (Thus, this step generates k with probability $g(k,U) = (1-U) U^k$.)
3. If k is in S, return 1; otherwise, return 0.

This can be seen as running **Algorithm CC** with an input coin for a randomly generated probability (a uniform random variate between 0 and 1). Given that step 1 generates U, the probability this algorithm returns 1 is— $$\sum_{k\text{ in }S} g(k,U) = \sum_{k\text{ in }S} (1-U) U^k,$$ and the overall algorithm uses the "**integral method**", so that the overall algorithm returns 1 with probability— $$\int_0^1\sum_{k\text{ in }S} (1-U) U^k,dU,$$ which, in the case of the MGL generator (where S is the set of non-negative integers with a remainder of 0 or 1 after division by 4), equals $\int_0^1 \frac{1}{U^2+1},dU = \pi/4$.

The derivation below relies on the following fact: The probability satisfies— $$\int_0^1\sum_{k\text{ in }S} g(k,U),dU = \sum_{k\text{ in }S}\int_0^1 g(k,U),dU.$$ Swapping the integral and the sum is not always possible, but it is in this case because the conditions of so-called Tonelli's theorem are met: $g(k,U)$ is continuous and non-negative whenever k is in S and $0\le U\le 1$; and S and the interval $[0, 1]$ have natural sigma-finite measures.

Now to show how the MGL generator produces the probability $\pi/4$. Let $C(k)$ be the probability that this algorithm's step 2 generates a number k, namely— $$C(k)=\int_0^1 g(k,U),dU = \int_0^1 (1-U) U^k,dU = \frac{1}{k^2+3k+2}.$$ Then the MGL series for $\pi/4$ is formed by—

$$\pi/4 = (1/1-1/3)+(1/5-1/7)+...=2/3+2/35+2/99+...$$

$$=(C(0)+C(1))+(C(4)+C(5))+(C(8)+C(9))+...$$

$$=\sum_{k\ge 0} C(4k)+C(4k+1),$$

where the last sum takes $C(k)$ for each k in the set S given for the MGL generator.

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 variates equals n:
  - Two random variates 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 variates, but this terminology is avoided in this article because it has several conflicting meanings in academic works).
  - One Bernoulli random variate with mean 5/9.

  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 variates equals n, where each of the two variates 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 variates equals n (a *negative binomial distribution*), where each of the m variates expresses the number of failures before the first success and the chance of a success is p.
> 3. $p * f(z - 1) + (1 - p) * f(z)$ is the probability that the sum of

> two independent random variates — a Bernoulli variate with mean p as well as an integer that equals x with probability $f(x)$ — equals z.

9.11 Preparing Rational Functions

This section describes how to turn a single-variable rational function (ratio of polynomials) into an array of polynomials needed to apply the **"Dice Enterprise" special case** described in "**Certain Rational Functions**". In short, the steps to do so can be described as *separating*, *homogenizing*, and *augmenting*.

Separating. If a rational function's numerator (D) and denominator (E) are written—

- as a sum of terms of the form $z*\lambda^i*(1-\lambda)^j$, where z is a real number and $i \geq 0$ and $j \geq 0$ are integers (called *form 1* in this section),

then the function can be separated into two polynomials that sum to the denominator. (Here, $i+j$ is the term's *degree*, and the polynomial's degree is the highest degree among its terms.) To do this separation, subtract the numerator from the denominator to get a new polynomial (G) such that $G = E - D$ (or $D + G = E$). (Then D and G are the two polynomials that will be used.) Similarly, if we have multiple rational functions with a common denominator, namely ($D1/E$), ..., (DN/E), where $D1$, ..., DN and E are written in form 1, then they can be separated into $N + 1$ polynomials by subtracting the numerators from the denominator, so that $G = E - D1 - ... - DN$. (Then $D1$, ..., DN and G are the polynomials that will be used.) To use the polynomials in the algorithm, however, they need to be *homogenized*, then *augmented*, as described next.

> **Example:** We have the rational function $(4*\lambda^1*(1-\lambda)^2) / (7 - 5*\lambda^1*(1-\lambda)^2)$. Subtracting the numerator from the denominator leads to: $7 - 1*\lambda^1*(1-\lambda)^2$.

Homogenizing. The next step is to *homogenize* the polynomials so they have the same degree and a particular form. For this step, choose n to be an integer no less than the highest degree among the polynomials.

Suppose a polynomial—

- is 0 or greater for every λ 0 or greater, but not greater than 1,
- has degree n or less, and
- is written in form 1 as given above.

Then the polynomial can be turned into a *homogeneous polynomial* of degree n (all its terms have degree n) as follows.

- For each integer m in [0, n], the new homogeneous polynomial's coefficient at m is found as follows:
  1. Set r to 0.
  2. For each term (in the old polynomial) of the form $z*\lambda^i*(1-\lambda)^j$:
     - If $i \le m$, and $(n-m) \ge j$, and $i + j \le n$, add z*choose($n-(i+j)$, $(n-m)-j$) to r.
  3. Now, r is the new coefficient (corresponding to the term $r*\lambda^m*(1-\lambda)^{n-m}$).

If the polynomial is written in so-called "power form" as *c[0]* + *c[1]*$*\lambda$ + *c[2]*$*\lambda^2$ + ... + *c[n]*$*\lambda^n$, then the method is instead as follows:

- For each integer m in [0, n], the new homogeneous polynomial's coefficient at m is found as follows:
  1. Set r to 0.
  2. For each integer i in [0, m], if there is a coefficient *c[i]*, add *c[i]*$*$choose($n-i$, $n-m$) to r.
  3. Now, r is the new coefficient (corresponding to the term $r*\lambda^m*(1-\lambda)^{n-m}$).

> **Example:** Let the following polynomial be given: $3*\lambda^2 + 10*\lambda^1*(1-\lambda)^2$. This is a degree-3 polynomial, and we seek to turn it into a degree-5 homogeneous polynomial. The result becomes the sum of the terms—
>
> - $0 * \lambda^0*(1-\lambda)^5$;
> - 10*choose(2, 2) $* \lambda^1*(1-\lambda)^4 = 10* \lambda^1*(1-\lambda)^4$;
> - (3*choose(3, 3) + 10*choose(2, 1)) $* \lambda^2*(1-\lambda)^3 = 23* \lambda^2*(1-\lambda)^3$;
> - (3*choose(3, 2) + 10*choose(2, 0)) $* \lambda^3*(1-\lambda)^2 = 19* \lambda^3*(1-\lambda)^2$;
> - 3*choose(3, 1) $* \lambda^4*(1-\lambda)^1 = 9* \lambda^4*(1-\lambda)^1$; and
> - 3*choose(3, 0) $* \lambda^5*(1-\lambda)^0 = 3* \lambda^5*(1-\lambda)^0$,

> resulting in the coefficients (0, 10, 23, 19, 9, 3) for the new homogeneous polynomial.

Augmenting. If we have an array of homogeneous single-variable polynomials of the same degree, they are ready for use in the **Dice Enterprise special case** if—

- the polynomials have the same degree, namely n,
- their coefficients are all 0 or greater, and
- the sum of j^{th} coefficients is greater than 0, for each j starting at 0 and ending at n, except that the list of sums may begin and/or end with zeros.

If those conditions are not met, then each polynomial can be *augmented* as often as necessary to meet the conditions (Morina et al., 2022)[191]. For polynomials of the kind relevant here, augmenting a polynomial amounts to degree elevation similar to that of polynomials in Bernstein form (see also Tsai and Farouki 2001[192]). It is implemented as follows:

- Let n be the polynomial's old degree. For each k in [0, n+1], the new polynomial's coefficient at k is found as follows:
  - Let $c[j]$ be the old polynomial's j^{th} coefficient (starting at 0). Calculate $c[j]$ * choose(1, $k-j$) for each integer j satisfying $\max(0, k-1) \le j \le \min(n, k)$, then add them together. The sum is the new coefficient.

According to the Morina paper, it's enough to do n augmentations on each polynomial for the whole array to meet the conditions above (although fewer than n will often suffice).

> **Note**: For best results, the input polynomials' coefficients should be rational numbers. If they are not, then special methods are needed to ensure exact results, such as interval arithmetic that calculates lower and upper bounds.

10 License

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

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

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

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

5. 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 whose domain is [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 (Dale, H., Jennings, D. and Rudolph, T., 2015, "Provable quantum advantage in randomness processing", *Nature communications* 6(1), pp. 1-4).↩

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

7. Huber, M., "**Nearly optimal Bernoulli factories for linear functions**", arXiv:1308.1562v2 [math.PR], 2014.↩

8. 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**.↩

9. 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.↩

10. 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,↩

11. Wästlund, J., "**Functions arising by coin flipping**", 1999.↩

12. Then *j* is a *binomial* random variate expressing the number of successes in *n* trials that each succeed with probability λ.↩

13. 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,↩

14. 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.↩

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

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

17. 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).↩

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

19. Niazadeh, R., Paes Leme, R., Schneider, J., "**Combinatorial Bernoulli Factories: Matchings, Flows, and Polytopes**", in *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, pp. 833-846, June 2021; also at **https://arxiv.org/abs/2011.03865.pdf**.↩

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

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

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

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

24. Giulio Morina. Krzysztof Łatuszyński. Piotr Nayar. Alex Wendland. "From the Bernoulli factory to a dice enterprise via perfect sampling of Markov chains." Ann. Appl. Probab. 32 (1) 327 - 359, February 2022. **https://doi.org/10.1214/21-AAP1679**↩

25. Giulio Morina. Krzysztof Łatuszyński. Piotr Nayar. Alex Wendland. "From the Bernoulli factory to a dice enterprise via perfect sampling of Markov chains." Ann. Appl. Probab. 32 (1) 327 - 359, February 2022. **https://doi.org/10.1214/21-AAP1679**↩

26. Propp, J.G., Wilson, D.B., "Exact sampling with coupled Markov chains and applications to statistical mechanics", 1996.↩

27. 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.↩

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

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

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

31. Knuth, Donald E. and Andrew Chi-Chih Yao. "The complexity of nonuniform random number generation", in *Algorithms and Complexity: New Directions and Recent Results*, 1976.↩

32. 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.↩

33. 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.↩

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

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

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

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

38. 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.↩

39. Flegal, J.M., Herbei, R., "Exact sampling from intractible probability distributions via a Bernoulli factory", *Electronic Journal of Statistics* 6, 10-37, 2012.↩

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

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

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

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

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

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

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

47. Note that `u * BASE`$^{-k}$ is not just within `BASE`$^{-k}$ of its "true" result, but also not more than that result. Hence `pk + 1 <= u` rather than `pk + 2 <= u`.↩

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

49. The "even-parity" construction (Flajolet et al. 2010) is so called because it involves flipping the input coin repeatedly until it returns zero, then counting the number of ones. The final result is 1 if that number is even, or 0 otherwise. However, the number of flips needed by this method grows without bound as λ (the probability the input coin returns 1) approaches 1. See also the note for **Algorithm CC**.↩

50. Bill Gosper, "Continued Fraction Arithmetic", 1978.↩

51. Borwein, J. et al. “Continued Logarithms and Associated Continued Fractions.” *Experimental Mathematics* 26 (2017): 412 - 429.↩

52. Penaud, J.G., Roques, O., "Tirage à pile ou face de mots de Fibonacci", *Discrete Mathematics* 256, 2002.↩

53. Penaud, J.G., Roques, O., "Tirage à pile ou face de mots de Fibonacci", *Discrete Mathematics* 256, 2002.↩

54. Mendo, L., "**Simulating a coin with irrational bias using rational arithmetic**", arXiv:2010.14901 [math.PR], 2020/2021.↩

55. Mendo, L., "**Simulating a coin with irrational bias using rational arithmetic**", arXiv:2010.14901 [math.PR], 2020/2021.↩

56. Carvalho, Luiz Max, and Guido A. Moreira. "**Adaptive truncation of infinite sums: applications to Statistics**", arXiv:2202.06121 (2022).↩

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

58. The error term, which follows from the so-called Lagrange remainder for Taylor series, 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.↩

59. Kozen, D., **"Optimal Coin Flipping"**, 2014.↩

60. 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.↩

61. Wästlund, J., "**Functions arising by coin flipping**", 1999.↩

62. 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.↩

63. Wästlund, J., "**Functions arising by coin flipping**", 1999.↩

64. Huber, M., "**Optimal linear Bernoulli factories for small mean problems**", arXiv:1507.00843v2 [math.PR], 2016.↩

65. Dughmi, Shaddin, Jason Hartline, Robert D. Kleinberg, and Rad Niazadeh. "Bernoulli factories and black-box reductions in mechanism design." Journal of the ACM (JACM) 68, no. 2 (2021): 1-30.↩

66. Szász, O., "Generalization of S. Bernstein's Polynomials to the Infinite Interval", Journal of Research of the National Bureau of Standards 45 (1950).↩

67. 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.↩

68. The "even-parity" construction (Flajolet et al. 2010) is so called because it involves flipping the input coin repeatedly until it returns zero, then counting the number of ones. The final result is 1 if that number is even, or 0 otherwise. However, the number of

flips needed by this method grows without bound as λ (the probability the input coin returns 1) approaches 1. See also the note for **Algorithm CC**.↩

69. However, the number of flips needed by this method will then grow without bound as λ approaches 1. Also, this article avoids calling the value X produced this way a "geometric" random variate. Indeed, there is no single way to give the probabilities of a "geometric" random variate; different academic works define the variate differently.↩

70. Schmon, S.M., Doucet, A. and Deligiannidis, G., 2019, April. Bernoulli race particle filters. In The 22nd International Conference on Artificial Intelligence and Statistics (pp. 2350-2358).↩

71. Dughmi, Shaddin, Jason Hartline, Robert D. Kleinberg, and Rad Niazadeh. "Bernoulli factories and black-box reductions in mechanism design." Journal of the ACM (JACM) 68, no. 2 (2021): 1-30.↩

72. Agrawal, S., Vats, D., Łatuszyński, K. and Roberts, G.O., 2021. "**Optimal Scaling of MCMC Beyond Metropolis**", arXiv:2104.02020 [stat.CO], 2021.↩

73. Wästlund, J., "**Functions arising by coin flipping**", 1999.↩

74. However, the number of flips needed by this method will then grow without bound as λ approaches 1. Also, this article avoids calling the value X produced this way a "geometric" random variate. Indeed, there is no single way to give the probabilities of a "geometric" random variate; different academic works define the variate differently.↩

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

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

77. The **appendix to the supplemental notes** defines pushdown automata in more detail and has proofs on which algebraic functions are possible with these conceptual machines.↩

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

79. However, the number of flips needed by this method will then grow without bound as λ approaches 1. Also, this article avoids calling the value *X* produced this way a "geometric" random variate. Indeed, there is no single way to give the probabilities of a "geometric" random variate; different academic works define the variate differently.↩

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

81. Flajolet, Ph., "Analytic models and ambiguity of context-free languages", *Theoretical Computer Science* 49, pp. 283-309, 1987↩

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

83. However, the number of flips needed by this method will then grow without bound as λ approaches 1. Also, this article avoids calling the value *X* produced this way a "geometric" random variate. Indeed, there is no single way to give the probabilities of a "geometric" random variate; different academic works define the variate differently.↩

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

85. Flajolet, P. and Sedgewick, R., *Analytic Combinatorics*, 2009.↩

86. "*x* is even" means that *x* is an integer and divisible by 2. This is true if $x - 2*\text{floor}(x/2)$ equals 0, or if *x* is an integer and the least significant bit of abs(*x*) is 0.↩

87. "*x* is odd" means that *x* is an integer and not divisible by 2. This is true if $x - 2*\text{floor}(x/2)$ equals 1, or if *x* is an integer and the least significant bit of abs(*x*) is 1.↩

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

89. "*x* is odd" means that *x* is an integer and not divisible by 2. This is true if $x - 2*\text{floor}(x/2)$ equals 1, or if *x* is an integer and the least significant bit of abs(*x*) is 1.↩

90. "x is odd" means that x is an integer and not divisible by 2. This is true if $x - 2*\text{floor}(x/2)$ equals 1, or if x is an integer and the least significant bit of abs(x) is 1.↩

91. "x is even" means that x is an integer and divisible by 2. This is true if $x - 2*\text{floor}(x/2)$ equals 0, or if x is an integer and the least significant bit of abs(x) is 0.↩

92. "x is even" means that x is an integer and divisible by 2. This is true if $x - 2*\text{floor}(x/2)$ equals 0, or if x is an integer and the least significant bit of abs(x) is 0.↩

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

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

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

96. In fact, thanks to the "geometric bag" technique of Flajolet et al. (2010), the fractional part ν can even come from a uniform partially-sampled random number (PSRN).↩

97. Another algorithm for $\exp(-\lambda)$ involves the von Neumann schema, but unfortunately, it converges slowly as λ approaches 1.↩

98. Canonne, C., Kamath, G., Steinke, T., "**The Discrete Gaussian for Differential Privacy**", arXiv:2004.00010 [cs.DS], 2020.↩

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

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

101. Another algorithm for $\exp(-\lambda)$ involves the von Neumann schema, but unfortunately, it converges slowly as λ approaches 1.↩

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

103. Peres, N., Lee, A.R. and Keich, U., 2021. Exactly computing the tail of the Poisson-Binomial Distribution. ACM Transactions on Mathematical Software (TOMS), 47(4), pp.1-19.↩

104. Sadowsky, Bucklew, On large deviations theory and asymptotically efficient Monte Carlo↩

105. Gonçalves, F. B., Łatuszyński, K. G., Roberts, G. O. (2017). Exact Monte Carlo likelihood-based inference for jump-diffusion processes.↩

106. Vats, D., Gonçalves, F. B., Łatuszyński, K. G., Roberts, G. O., "Efficient Bernoulli factory Markov chain Monte Carlo for intractable posteriors", *Biometrika* 109(2), June 2022 (also in arXiv:2004.07471 [stat.CO]).↩

107. Vats, D., Gonçalves, F. B., Łatuszyński, K. G., Roberts, G. O., "Efficient Bernoulli factory Markov chain Monte Carlo for intractable posteriors", *Biometrika* 109(2), June 2022 (also in arXiv:2004.07471 [stat.CO]).↩

108. Huber, M., "**Optimal linear Bernoulli factories for small mean problems**", arXiv:1507.00843v2 [math.PR], 2016.↩

109. Giulio Morina. Krzysztof Łatuszyński. Piotr Nayar. Alex Wendland. "From the Bernoulli factory to a dice enterprise via perfect sampling of Markov chains." Ann. Appl. Probab. 32 (1) 327 - 359, February 2022. **https://doi.org/10.1214/21-AAP1679**↩

110. Gonçalves, F. B., Łatuszyński, K. G., Roberts, G. O. (2017). Exact Monte Carlo likelihood-based inference for jump-diffusion processes.↩

111. 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** (see "More Algorithms for Arbitrary-Precision Sampling") with $g(\lambda)=\lambda$, $d_0 = 1$, and $a_i=(-1)^i$. 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."↩

112. 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.↩

113. 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.↩

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

115. The "even-parity" construction (Flajolet et al. 2010) is so called because it involves flipping the input coin repeatedly until it returns zero, then counting the number of ones. The final result is 1 if that number is even, or 0 otherwise. However, the number of flips needed by this method grows without bound as λ (the probability the input coin returns 1) approaches 1. See also the note for **Algorithm CC**.↩

116. Peres, N., Lee, A.R. and Keich, U., 2021. Exactly computing the tail of the Poisson-Binomial Distribution. ACM Transactions on Mathematical Software (TOMS), 47(4), pp.1-19.↩

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

118. "x is even" means that x is an integer and divisible by 2. This is true if $x - 2*\text{floor}(x/2)$ equals 0, or if x is an integer and the least significant bit of abs(x) is 0.↩

119. "x is even" means that x is an integer and divisible by 2. This is true if $x - 2*\text{floor}(x/2)$ equals 0, or if x is an integer and the least significant bit of abs(x) is 0.↩

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

121. The "even-parity" construction (Flajolet et al. 2010) is so called because it involves flipping the input coin repeatedly until it returns zero, then counting the number of ones. The final result is 1 if that number is even, or 0 otherwise. However, the number of flips needed by this method grows without bound as λ (the probability the input coin returns 1) approaches 1. See also the note for **Algorithm CC**.↩

122. Sadowsky, Bucklew, On large deviations theory and asymptotically efficient Monte Carlo↩

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

124. The "even-parity" construction (Flajolet et al. 2010) is so called because it involves flipping the input coin repeatedly until it returns zero, then counting the number of ones. The final result is 1 if that number is even, or 0 otherwise. However, the number of flips needed by this method grows without bound as λ (the probability the input coin returns 1) approaches 1. See also the note for **Algorithm CC**.↩

125. There is another algorithm for tanh(z), based on Lambert's continued fraction for tanh(.), but it works only if $0 \le z \le 1$ and if z is the probability of heads of an input coin. The algorithm begins with k equal to 1. Then: (1) If k is 1, generate an unbiased random bit, then if that bit is 1, flip the input coin and return the result; (2) If k is greater than 1, then with probability $k/(1+k)$, flip the input coin twice, and if either or both flips returned 0, return 0, and if both flips returned 1, return a number that is 1 with probability $1/k$ and 0 otherwise; (3) Do a separate run of the currently running algorithm, but with $k = k + 2$. If the separate run returns 1, return 0; (4) Go to step 2.↩

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

127. 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.↩

128. Wästlund, J., "**Functions arising by coin flipping**", 1999.↩

129. Dale, H., Jennings, D. and Rudolph, T., 2015, "Provable quantum advantage in randomness processing", *Nature communications* 6(1), pp. 1-4.↩

130. Tsai, Yi-Feng, Farouki, R.T., "Algorithm 812: BPOLY: An Object-Oriented Library of Numerical Algorithms for Polynomials in Bernstein Form", *ACM Trans. Math. Softw.* 27(2), 2001.↩

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

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

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

134. Morina, Giulio (2021) Extending the Bernoulli Factory to a dice enterprise. PhD thesis, University of Warwick.↩

135. Huber, M., "**Nearly optimal Bernoulli factories for linear functions**", arXiv:1308.1562v2 [math.PR], 2014.↩

136. Huber, M., "**Optimal linear Bernoulli factories for small mean problems**", arXiv:1507.00843v2 [math.PR], 2016.↩

137. Morina, Giulio (2021) Extending the Bernoulli Factory to a dice enterprise. PhD thesis, University of Warwick.↩

138. Huber, M., "**Designing perfect simulation algorithms using local correctness**", arXiv:1907.06748v1 [cs.DS], 2019.↩

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

140. "x is odd" means that x is an integer and not divisible by 2. This is true if $x - 2*\text{floor}(x/2)$ equals 1, or if x is an integer and the least significant bit of abs(x) is 1.↩

141. "x is even" means that x is an integer and divisible by 2. This is true if $x - 2*\text{floor}(x/2)$ equals 0, or if x is an integer and the least significant bit of abs(x) is 0.↩

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

143. "x is odd" means that x is an integer and not divisible by 2. This is true if $x - 2*\text{floor}(x/2)$ equals 1, or if x is an integer and the least significant bit of abs(x) is 1.↩

144. "x is even" means that x is an integer and divisible by 2. This is true if $x - 2*\text{floor}(x/2)$ equals 0, or if x is an integer and the least significant bit of abs(x) is 0.↩

145. One of the only implementations I could find of this, if not the only, was a **Haskell implementation**.↩

146. Dughmi, Shaddin, Jason Hartline, Robert D. Kleinberg, and Rad Niazadeh. "Bernoulli factories and black-box reductions in mechanism design." Journal of the ACM (JACM) 68, no. 2 (2021): 1-30.↩

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

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

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

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

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

152. Another algorithm for this function uses the **general martingale algorithm** with $g(\lambda)=\lambda$, $d_0 = 1$ and $a_i= (-1)^{i+1}/i$ (except $a_0 = 0$), but uses more bits on average as λ approaches 1.↩

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

154. The "even-parity" construction (Flajolet et al. 2010) is so called because it involves flipping the input coin repeatedly until it returns zero, then counting the number of ones. The final result is 1 if that number is even, or 0 otherwise. However, the number of flips needed by this method grows without bound as λ (the probability the input coin returns 1) approaches 1. See also the note for **Algorithm CC**.↩

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

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

157. 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.↩

158. 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.↩

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

160. The "even-parity" construction (Flajolet et al. 2010) is so called because it involves flipping the input coin repeatedly until it returns zero, then counting the number of ones. The final result is 1 if that number is even, or 0 otherwise. However, the number of flips needed by this method grows without bound as λ (the probability the input coin returns 1) approaches 1. See also the note for **Algorithm CC**.↩

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

162. Forsythe, G.E., "Von Neumann's Comparison Method for Random Sampling from the Normal and Other Distributions", *Mathematics of Computation* 26(120), October 1972.↩

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

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

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

166. Mendo, L., "**Simulating a coin with irrational bias using rational arithmetic**", arXiv:2010.14901 [math.PR], 2020/2021.↩

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

168. Sondow, Jonathan. “New Vacca-Type Rational Series for Euler's Constant and Its 'Alternating' Analog ln 4/π.”, 2005.↩

169. It can also be said that the integral (see "**Integrals**") of x − floor(1/x), where x is greater than 0 but not greater than 1, equals 1 minus γ. See, for example, Havil, J., *Gamma: Exploring Euler's Constant*, 2003.↩

170. Mendo, L., "**Simulating a coin with irrational bias using rational arithmetic**", arXiv:2010.14901 [math.PR], 2020/2021.↩

171. Fishman, D., Miller, S.J., "Closed Form Continued Fraction Expansions of Special Quadratic Irrationals", ISRN Combinatorics Vol. 2013, Article ID 414623 (2013).↩

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

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

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

175. 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.↩

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

177. 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.↩

178. Pae, S., "Random number generation using a biased source", dissertation, University of Illinois at Urbana-Champaign, 2005.↩

179. Peres, Y., "Iterating von Neumann's procedure for extracting random bits", Annals of Statistics 1992,20,1, p. 590-597.↩

180. Kozen, D., **"Optimal Coin Flipping"**, 2014.↩

181. 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.↩

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

183. Huber, M., "**Designing perfect simulation algorithms using local correctness**", arXiv:1907.06748v1 [cs.DS], 2019.↩

184. 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.↩

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

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

187. Forsythe, G.E., "Von Neumann's Comparison Method for Random Sampling from the Normal and Other Distributions", *Mathematics of Computation* 26(120), October 1972.↩

188. Monahan, J.. "Extensions of von Neumann’s method for generating random variables." Mathematics of Computation 33 (1979): 1065-1069.↩

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

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

191. Giulio Morina. Krzysztof Łatuszyński. Piotr Nayar. Alex Wendland. "From the Bernoulli factory to a dice enterprise via perfect sampling of Markov chains." Ann. Appl. Probab. 32 (1) 327 - 359, February 2022. **https://doi.org/10.1214/21-AAP1679**↩

192. Tsai, Yi-Feng, Farouki, R.T., "Algorithm 812: BPOLY: An Object-Oriented Library of Numerical Algorithms for Polynomials in Bernstein Form", *ACM Trans. Math. Softw.* 27(2), 2001.↩