Bernoulli Factory Algorithms

This version of the document is dated 2021-04-02.

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

Given a coin with unknown probability of heads of $lambda$, sample the probability *f*($lambda$). In other words, turn a coin biased one way ($lambda$) into a coin biased another way (*f*($lambda$)). This is the *Bernoulli factory problem*.

And this page catalogs algorithms to solve this problem for a wide variety of functions, algorithms known as *Bernoulli factories*.

Many of these algorithms were suggested in (Flajolet et al., 2010)[(1)], but without step-by-step instructions in many cases. This page provides these instructions to help programmers implement the Bernoulli factories they describe. The Python module ***bernoulli.py*** includes implementations of several Bernoulli factories.

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

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

Supplemental notes are found in: **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.

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

2 Contents

3 About Bernoulli Factories

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

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 [0, 1], and returns a probability in [0, 1]. There are certain requirements for factory functions. As shown by Keane and O'Brien (1994)[2], a function f($lambda$) can serve as a factory function if and only if—

- f 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 bounded from below by 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.
floor($lambda$/2)*3+1/4	[0, 1]	No; discontinuous.
2*$lambda$	[0, 1] or \0, 1/2)	No; not polynomially bounded since its graph touches 1 somewhere in the interval (0, 1) on its domain.[3].
1−2*$lambda$	[0, 1] or [0, 1/2)	No; not polynomially bounded since its graph touches 0 somewhere in the interval (0, 1) on its domain.
2*$lambda$	[0, 1/2−ϵ]	Yes; continuous and polynomially bounded on domain (Keane and O'Brien 1994)[2].
min(2 * $lambda$, 1 − ϵ)	[0, 1]	Yes; continuous and polynomially bounded on domain (Huber 2014, introduction)[4].
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/$lambda$) + ϵ otherwise	[0, 1]	Yes; continuous and bounded away from 0 and 1.

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

4 Algorithms

In the following algorithms:

- $lambda$ is the unknown probability of heads of the input coin.
- choose(n, k) = n!/(k! * (n − k)!) is a binomial coefficient. It can be calculated, for example, by calculating i/(n−i+1) for each integer i in [n−k+1, n], then multiplying the results (Manolopoulos 2002)[5]. Note that for all 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.
- The instruction to "generate a uniform(0, 1) random number" can be implemented—
  - by creating a **uniform partially-sampled random number (PSRN)** with a positive sign, an integer part of 0, and an empty fractional part (most accurate), or
  - by generating `RNDRANGEMaxExc(0, 1)` or `RNDINT(1000) / 1000`, as described in "**Randomization and Sampling Methods**" (less accurate).
- The instruction to "generate an exponential random number" can be implemented—
  - by creating an empty **exponential PSRN** (most accurate), or
  - by getting the result of the **ExpRand** or **ExpRand2** algorithm (described in my article on PSRNs) with a rate of 1, or
  - by generating `-ln(1/RNDRANGEMinExc(0, 1))`, as described in "**Randomization and Sampling Methods**" (less accurate).
- The instruction to "choose [integers] with probability proportional to [*weights*]" can be implemented—
  - by taking the result of **WeightedChoice**(**NormalizeRatios**(*weights*))), where **WeightedChoice** and **NormalizeRatios** are given in "**Randomization and Sampling Methods**".
- To **sample from a random number *u*** means to generate a number that is 1 with probability u and 0 otherwise.
  - If the number is a uniform PSRN, call the **SampleGeometricBag** algorithm with the PSRN and take the result of that call (which will be 0 or 1) (most accurate). (**SampleGeometricBag** is described in my **article on PSRNs**.)
  - Otherwise, this can be implemented by generating another uniform(0, 1) random number v and generating 1 if v is less than u or 0 otherwise (less accurate).
- Where an algorithm says "if a is less than b", where a and b are random numbers, it means to run the **RandLess** algorithm on the two numbers (if they are both PSRNs), or do a less-than operation on a and b, as appropriate. (**RandLess** is described in my **article on PSRNs**.)
- Where an algorithm says "if a is less than (or equal to) b", where a and b are random numbers, it means to run the **RandLess** algorithm on the two numbers (if they are both PSRNs), or do a less-than-or -equal operation on a and b, as appropriate.
- 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(0, 1) random number 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**. Note that if x is not a rational number, then rounding error will result.
- For best results, the algorithms should be implemented using exact rational arithmetic (such as `Fraction` in Python or `Rational` in Ruby). Floating-point arithmetic is discouraged because it can introduce errors due to fixed-precision calculations, such as rounding and cancellations.

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

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

Bernoulli factory algorithms that sample the probability *f*($lambda$) act as unbiased estimators of *f*($lambda$). See the **appendix** for details.

4.1 Algorithms for General Functions of $lambda$

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

4.1.1 Certain Polynomials

Any polynomial can be written in *Bernstein form* as $\sum_{i = 0, ..., n}$ choose(n, i) * $lambda$^{i} * (1 − $lambda$)$^{n - i}$ * a[i], where n is the polynomial's *degree* and a[i] are its n plus one coefficients.

But the only polynomials that admit a Bernoulli factory are those whose coefficients are all in the interval [0, 1] (once the polynomials are written in Bernstein form), and these polynomials are the only functions that can be simulated with a fixed number of coin flips (Goyal and Sigman 2012**(6)**; Qian et al. 2011)**(7)**; see also Wästlund 1999, section 4**(8)**). 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.
2. With probability a[j], return 1. Otherwise, return 0.

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(0, 1) random number, 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.

   Notes:

   1. Each a[i] acts as a control point for a 1-dimensional **Bézier curve**, where $lambda$ is the relative position on that curve, the curve begins at a[0], and the curve ends at a[n]. For example, given control points 0.2, 0.3, and

0.6, the curve is at 0.2 when $lambda$ = 0, and 0.6 when $lambda$ = 1. (The curve, however, is not at 0.3 when $lambda$ = 1/2; in general, Bézier curves do not cross their control points other than the first and the last.)

2. The problem of simulating polynomials in Bernstein form is related to *stochastic logic*, which involves simulating probabilities that arise out of Boolean functions (functions that use only AND, OR, NOT, and XOR operations) that take a fixed number of bits as input, where each bit has a separate probability of being 1 rather than 0, and output a single bit (for further discussion see (Qian et al. 2011)**(7)**, Qian and Riedel 2008**(9)**).
3. These algorithms can serve as an approximate way to simulate any factory function *f* (or even any function that maps the interval [0, 1] to [0, 1], even if it's not continuous). In this case, *a*[*j*] is calculated as *f*(*j*/*n*), so that the resulting polynomial closely approximates the function; the higher *n* is, the better this approximation. In fact, if *f* is continuous, it's possible to choose *n* high enough to achieve a given maximum error. For more information, see my **Supplemental Notes on Bernoulli Factories**.

Examples:

1. Take the following parabolic function discussed in (Thomas and Blanchet 2012)**(10)**: (1−4*($lambda$−1/2)^2)*c, where c is in the interval (0, 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. By rewriting the polynomial in Bernstein form (such as via the matrix method by Ray and Nataraj (2012) **(11)**), we get coefficients (0, 2*c, 0). Thus, for this polynomial, *a*[0] is 0, *a*[1] is 2*c, and *a*[2] is 0. Thus, if c is in the interval (0, 1/2], we can simulate this function as follows: "Flip the input coin twice. If exactly one of the flips returns 1, return a number that is 1 with probability 2*c and 0 otherwise. Otherwise, return 0." For other values of c, the algorithm requires 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.**(12)**

2. There are a number of approximate methods to simulate $lambda$*c, where c > 1 and $lambda$ lies in \0, 1/c). ("Approximate" because this function touches 1 at 1/c, so it can't be a factory function.) Since the methods use only up to *n* flips, the approximation will be a polynomial of degree *n* (*n* is greater than 0; the greater *n* is, the better the approximation).

    - Henderson and Glynn (2003, Remark 4)[**(13)** approximates the function $lambda$*2 using a polynomial where *a*[*j*] = min((*j*/*n*)*2, 1−1/*n*). If *g*($lambda$) is that polynomial, then the error is no greater than 1−*g*(1/2). *g* can be computed with the SymPy computer algebra library as follows: `from sympy.stats import *; g=2*E( Min(sum(Bernoulli(("B%d" % (i)),z) for i in range(n))/n,(S(1)-S(1)/n)/2)).`
    - I found the following approximation for $lambda$*c**(14)**: "(1.) Set j to 0 and i to 0; (2.) If i ≥ n, return 0; (3.) Flip the input coin, and if it returns 1, add 1 to j; (4.) (Estimate the probability and return 1 if it 'went over'.) If (j/(i+1)) ≥ 1/c, return 1; (5.) Add 1 to i and go to step 2." Here, $lambda$*c is approximated by a polynomial where *a*[*j*] = min((*j*/*n*)*c, 1). If *g*($lambda$) is that polynomial, then the error is no greater than 1−*g*(1/c).

- The previous approximation generalizes the one given in section 6 of Nacu and Peres (2005)[15], which approximates $lambda$*2.

Multiple coins. Niazadeh et al. (2020)[16] describes monomials (involving one or more coins) of the form $\prod_{i = 1, ..., n}$ $lambda$[i]$^{a[i]}$ * (1−$lambda$[i])$^{b[i]}$, where there are n coins, $lambda$[i] is the probability of heads of coin i, and $a[i] \ge 0$ and $b[i] \ge 0$ 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 $P = \sum_{j = 1, ..., k} c[j]*M[j]($**$lambda$**$)$, where there are k monomials, $M[j](.)$ identifies monomial j, **$lambda$** identifies the coins' probabilities of heads, and $c[j] \ge 0$ is the weight for monomial j. (If there is only one coin, these polynomials are in Bernstein form if $c[j]$ is $\alpha[j]$*choose($k-1, j-1$) where $\alpha[j]$ is a coefficient in the interval [0, 1], and if $a[1] = j-1$ and $b[1] = k-j$ for each monomial j.)

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

4.1.2 Certain Rational Functions

According to (Mossel and Peres 2005)[17], a function can be simulated by a finite-state machine (equivalently, a "probabilistic regular grammar" (Smith and Johnson 2007)[18], (Icard 2019)[19]) if and only if the function can be written as a rational function (ratio of polynomials) with rational coefficients, that takes in an input $lambda$ in some subset of (0, 1) and outputs a number in the interval (0, 1).

The following algorithm is suggested from the Mossel and Peres paper and from (Thomas and Blanchet 2012)[10]. It assumes the rational function is of the form $D($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.

The algorithm follows.

1. Flip the input coin n times, and let j be the number of times the coin returned 1 this way.
2. Choose 0, 1, or 2 with probability proportional to these weights: $[e[j] - d[j], d[j],$ choose(n, j) $- e[j]]$. 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, \$lambda\$ is the relative position on that curve, the curve begins at $\delta[0]$ or $\eta[0]$, and the curve ends at $\delta[n]$ or $\eta[n]$. See also the next section.

2. This algorithm could be modified to avoid additional randomness besides the input coin flips by packing the coin flips into an n-bit word and looking up whether that word is "passing", "failing", or neither, among all n-bit words with j ones, but this is not so trivial to do (especially because in general, a lookup table first has to be built in a setup step, which can be impractical unless 2^n is relatively small). Moreover, this approach works only if $d[i]$ and $e[i]$ are integers (or if $d[i]$ is replaced with floor($d[i]$) and $e[i]$ with ceil($e[i]$) (Nacu and Peres 2005)**(15)**, but this, of course, suffers from rounding error when done in this algorithm). See also (Thomas and Blanchet 2012)**(10)**.

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.

"Dice Enterprise" special case. The following algorithm implements a special case of the "Dice Enterprise" method of Morina et al. (2019)**(20)**. 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 \$lambda\$ 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(0, 1) random number and append it to the end of *ulist*.
4. (Monotonic coupling from the past (Morina et al., 2019)**(20)**, (Propp and Wilson 1996)**(21)**.) 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 $P_1(\$lambda\$) / (P_0(\$lambda\$) + P_1(\$lambda\$))$. After **preparing this function** (and noting that the maximum degree is $n = 4$), we get 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, we could include a third polynomial, 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 we would run the algorithm using R', and if it returns 2 [meaning that the constant polynomial was chosen], we would try again until the algorithm no longer returns 2.)

4.1.3 Certain Algebraic Functions

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

According to (Mossel and Peres 2005)**(17)**, a factory function can be simulated by a pushdown automaton only if that function can be a solution of a polynomial system with rational coefficients.**(22)**

The following algorithm simulates the following algebraic function:

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

where—

- W(k) is a number in the interval [0, β^k] and is the number of k-letter words that can be produced by the stochastic grammar in question,
- β is the alphabet size, or the number of "letters" in the alphabet (e.g., 2 for the cases discussed in the Flajolet paper), and is an integer 2 or greater,
- the *ordinary generating function* OGF(x) = W(0) + W(1) * x + W(2) * x^2 + W(3) * x^3 + ..., and
- the second formula incorporates a correction to Theorem 3.2 of the paper**(23)**.

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

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

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

Examples:

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

3b. Add 1 to i. Then, if $i < g$ and $d > 0$, go to step 3a.; 3c. Return 1 if d is 0 and i is g, or 0 otherwise."

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

    - choose(g, g/t) * $(\beta-1)^{g-g/t}$ (the number of g-letter words with exactly g/t A's, for an alphabet size of β) if g is divisible by t[(24)], and
    - 0 otherwise,

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

3. If W(g) has the form—

    choose($g * \alpha$, g) * $(\beta-1)^{g*\alpha-g}$ / $\beta^{g*\alpha-g}$,

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

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

    The ordinary generating function for this modified algorithm is thus—

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

4. The probability involved in example 2 likewise involves hypergeometric functions:

    - (1 − $lambda$) * ${}_{t-1}F_{t-2}(1/t, 2/t, ..., (t-1)/t; 1/(t-1), ..., (t-2)/(t-1); \$lambda\$^{t}*t^{t}*(\beta-1)^{t-1}/((t-1)^{t-1} * \beta^{t}))$.

4.1.4 Certain Power Series

Mendo (2019)[(25)] gave a Bernoulli factory algorithm for certain functions that can be rewritten as a series of the form—

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

where $c[i] \geq 0$ are the coefficients of the series and sum to 1. (According to Mendo, this

implies that the series is differentiable — its graph has no "sharp corners" — and takes on a value that approaches 0 or 1 as $lambda$ approaches 0 or 1, respectively). The algorithm follows:

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

As pointed out in Mendo (2019)**(25)**, variants of this algorithm work for power series of the form—

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

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

(Łatuszyński et al. 2009/2011)**(26)** gave an algorithm that works for a wide class of series and other constructs that converge to the desired probability from above and from below.

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

d[0] − *d[1]* * $lambda$ + *d[2]* * $lambda$^{2} − ...,

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

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

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

If the alternating series has the form—

d[0] − *d[1]* * $lambda$^{2} + *d[2]* * $lambda$^{4} − ...,

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

(Nacu and Peres 2005, proposition 16)[15]. This algorithm simulates a function of the form—

$$d[0] + d[1] * \$\text{lambda}\$ + d[2] * \$\text{lambda}\$^2 - \ldots,$$

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

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

4.1.5 General Factory Functions

A coin with unknown probability of heads of $lambda$ 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 sequence, 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). This is also called the *consistency requirement*.

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

- **fbelow**(n, k) is a lower bound of the k^{th} coefficient for a degree-n polynomial in Bernstein form that approximates f from below, where k is in the interval [0, n]. For example, this can be $f(k/n)$ minus a constant that depends on n. (See note 3 below.)
- **fabove**(n, k) is an upper bound of the k^{th} coefficient for a degree-n polynomial in Bernstein form that approximates f from above. For example, this can be $f(k/n)$ plus a constant that depends on n. (See note 3.)

The first algorithm implements the reverse-time martingale framework (Algorithm 4) in Łatuszyński et al. (2009/2011)[26] and the degree-doubling suggestion in Algorithm I of Flegal and Herbei (2012)[27], although an error in Algorithm I is noted below. The first algorithm follows.

1. Generate a uniform(0, 1) random number, call it *ret*.
2. Set ℓ and ℓt to 0. Set u and ut to 1. Set *lastdegree* to 0, and set *ones* to 0.
3. Set *degree* so that the first pair of polynomials has degree equal to *degree* and has coefficients all lying in [0, 1]. For example, this can be done as follows: Let **fbound**(n) be the minimum value for **fbelow**(n, k) and the maximum value for **fabove**(n,k) for any k in the interval [0, n]; then set *degree* to 1; then while **fbound**(*degree*) returns an upper or lower bound that is less than 0 or greater than

1, multiply *degree* by 2; then go to the next step.
4. Set *startdegree* to *degree*.
5. (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*). 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*).
    2. Define **FA**(*a*, *b*) as follows: Let *c* be choose(*a*, *b*). 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*).
    3. Let *c* be choose(*degree*, *ones*). Set ℓ to (**FB**(*degree*, *ones*))/*c* and set *u* to (**FA**(*degree*, *ones*))/*c*.
7. (This step and the next find the expected values of the previous ℓ and *u* given the current coin flips.) If *degree* equals *startdegree*, set ℓs to 0 and *us* to 1. (Algorithm I of Flegal and Herbei 2012 doesn't take this into account.)
8. If *degree* is greater than *startdegree*: Let *nh* be choose(*degree*, *ones*), and let *od* be *degree*/2. Set ℓs to $\sum_{j=0,\ldots,ones}$ **FB**(*od*,*j*)*choose(*degree*−*od*, *ones*−*j*)/*nh*, and set *us* to $\sum_{j=0,\ldots,ones}$ **FA**(*od*,*j*)*choose(*degree*−*od*, *ones*−*j*)/*nh*.
9. Let *m* be ($ut-\ell t$)/($us-\ell s$). Set ℓt to $\ell t+(\ell-\ell s)*m$, and set *ut* to $ut-(us-u)*m$.
10. If *ret* is less than (or equal to) ℓt, return 1. If *ret* is less than *ut*, go to the next step. If neither is the case, return 0. (If *ret* is a uniform PSRN, these comparisons should be done via the **URandLessThanReal algorithm**, which is described in my **article on PSRNs**.)
11. (Find the next pair of polynomials and restart the loop.) Increase *degree* so that the next pair of polynomials has degree equal to a higher value of *degree* and gets closer to the target function (for example, multiply *degree* by 2). Then, go to step 5.

The second algorithm was given in Thomas and Blanchet (2012)**(10)**; it assumes the same sequences of polynomials are available as in the previous algorithm. An algorithm equivalent to that algorithm is given below.

1. Set *ones* to 0, and set *lastdegree* to 0.
2. Set *degree* so that the first pair of polynomials has degree equal to *degree* and has coefficients all lying in [0, 1]. For example, this can be done as follows: Let **fbound**(*n*) be the minimum value for **fbelow**(*n*, *k*) and the maximum value for **fabove**(*n*,*k*) for any *k* in the interval [0, *n*]; then set *degree* to 1; then while **fbound**(*degree*) returns an upper or lower bound that is less than 0 or greater than 1, multiply *degree* by 2; then go to the next step.
3. Set *startdegree* to *degree*.
4. (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*.
5. Set *c* to choose(*degree*, *ones*). Optionally, multiply *c* by 2^{degree} (see note 3 below).
6. Find *acount* and *bcount* as follows:
    1. Calculate **fbelow**(*degree*, *ones*) as lower and upper bounds *LB* and *UB* that are accurate enough that floor(*LB***c*) = floor(*UB***c*). Then set *a*[*degree*,*ones*] and *acount* to floor(*LB***c*).
    2. Calculate 1−**fabove**(*degree*, *ones*) as lower and upper bounds *LB* and *UB* that are accurate enough that floor(*LB***c*) = floor(*UB***c*). Then set *b*[*degree*,*ones*] and *bcount* to floor(*LB***c*).
    3. Subtract (*acount* + *bcount*) from *c*.
7. If *degree* is greater than *startdegree*, then:
    1. Let *diff* be *degree*−*lastdegree*, let *u* be max(0, *ones*−*lastdegree*), and let *v* be

min(*ones*, *diff*). (The following substeps remove outcomes from *acount* and *bcount* that would have terminated the algorithm earlier. The procedure differs from step (f) of section 3 of the paper, which appears to be incorrect, and the procedure was derived from the **supplemental source code** uploaded by A. C. Thomas at my request.)

2. Set g to choose(*lastdegree*, *ones*$-u$). Set h to 1. If c was multiplied as in step 5, multiply h by $2^{lastdegree}$ (see note 3 below).
3. For each integer k in the interval [u, v]:
    1. Set d to choose(*diff*, k). Let ω be *ones*$-k$.
    2. If not already calculated: Calculate **fbelow**(*lastdegree*, ω) as lower and upper bounds LB and UB that are accurate enough that floor(LB*g*h) = floor(UB*g*h). Then set a[*lastdegree*, ω] to floor(LB*g*h).
    3. If not already calculated: Calculate 1$-$**fabove**(*lastdegree*, ω) as lower and upper bounds LB and UB that are accurate enough that floor(LB*g*h) = floor(UB*g*h). Then set b[*lastdegree*, ω] to floor(LB*g*h).
    4. Subtract (a[*lastdegree*, ω]*d) from *acount*.
    5. Subtract (b[*lastdegree*, ω]*d) from *bcount*.
    6. Multiply g by ω, then divide g by (*lastdegree*$+1-\omega$). (Sets g to choose(*lastdegree*, (*ones*$-k$)-1).)
8. Choose 0, 1, or 2 with probability proportional to the following weights: [*acount*, *bcount*, c].
9. If the number chosen by the previous step is 0, return 1. If the number chosen by that step is 1, return 0.
10. (Find the next pair of polynomials and restart the loop.) Set *lastdegree* to *degree*, then increase *degree* so that the next pair of polynomials has degree equal to a higher value of *degree* and gets closer to the target function (for example, multiply *degree* by 2). Then, go to step 4.

Notes:

1. The efficiency of these two algorithms depends on many things, including how "smooth" f is 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)**(10)** is to use multiple pairs of polynomial sequences that converge to f, where each pair is optimized for particular ranges of $lambda$: first flip the input coin several times to get a rough estimate of $lambda$, then choose the pair that's optimized for the estimated $lambda$, and run either algorithm in this section on that pair.
3. The second algorithm, as presented in Thomas and Blanchet (2012)**(10)**, was based on the one from Nacu and Peres (2005)**(15)**. In both papers, the algorithm works only if $lambda$ is in the interval (0, 1). If $lambda$ 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)**(28)**, the c in step 5 can be multiplied by 2^{degree} and the h in step 7, substep 2, multiplied by $2^{lastdegree}$ to ensure correctness for all values of $lambda$.

4.2 Algorithms for General Irrational Constants

This section shows general-purpose algorithms 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 bias are available, then a ***randomness extraction*** method should be used to turn them into fair coins.

4.2.1 Digit Expansions

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

In the algorithm (see also (Brassard et al., 2019)[**(29)**, (Devroye 1986, p. 769)**(30)**), `BASE` is the digit base, such as 2 for binary or 10 for decimal.

1. Set `u` to 0 and `k` to 1.
2. Set `u` to `(u * BASE) + v`, where `v` is a random integer in the interval [0, `BASE`) (such as `RNDINTEXC(BASE)`, or simply an unbiased random bit if `BASE` is 2). Calculate `pa`, which is an approximation to `p` such that abs(`p`−`pa`) ≤ $\texttt{BASE}^{-\texttt{k}}$. 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. If `pk - 2 >= u`, return 1. If neither is the case, add 1 to `k` and go to step 2.

4.2.2 Continued Fractions

The following algorithm simulates a probability expressed as a simple continued fraction of the following form: 0 + 1 / (*a*[1] + 1 / (*a*[2] + 1 / (*a*[3] + ...))). The *a*[*i*] are the *partial denominators*, none of which may have an absolute value less than 1. Inspired by (Flajolet et al., 2010, "Finite graphs (Markov chains) and rational functions")**(1)**, I developed the following algorithm.

Algorithm 1. This algorithm works only if each *a*[*i*]'s absolute value is 1 or greater and *a*[1] is 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.

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

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

1. Set *r* to 1 / (*r* * *b*[*pos*]), then set *k* to *a*[*pos*] * *r*. (*k* is the partial denominator for the equivalent simple continued fraction.)
2. If the partial numerator/denominator pair at *pos* is the last, return a number that is 1 with probability 1/abs(*k*) and 0 otherwise.
3. Set *kp* to 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 "even parity construction" (or the two-coin special case) as well as the "1 − *x*" construction can be used to develop rational number simulators that are as big as their continued fraction expansions, as suggested in the cited part of the Flajolet paper. However, it only works if the size of the continued fraction expansion (here, *size*) is known in advance.
    1. Set *i* to *size*.

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

4.2.3 Continued Logarithms

The *continued logarithm* (Gosper 1978)[31], (Borwein et al., 2016)[32] of a number in (0, 1) has the following continued fraction form: 0 + (1 / $2^{c[1]}$) / (1 + (1 / $2^{c[2]}$) / (1 + ...)), where *c*[*i*] are the coefficients of the continued logarithm and all 0 or greater. I have come up with the following algorithm that simulates a probability expressed as a continued logarithm expansion.

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

1. If the coefficient at *pos* is the last, return a number that is 1 with probability 1/($2^{c[pos]}$) and 0 otherwise.
2. 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.

4.2.4 Certain Converging Series

A general-purpose algorithm was given by Mendo (2020)[33] that can simulate any probability in (0, 1), as long as it can be rewritten as a converging series—

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

The algorithm follows.

1. Set ϵ to 1, then set *n*, *lamunq*, *lam*, *s*, and *k* to 0 each.
2. Add 1 to *k*, then add *s*/(2^k) to *lam*.
3. If *lamunq*+ϵ ≤ *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, is instead a sequence that converges to the *base-2 logarithm* of a probability in (0, 1), the following algorithm I developed simulates that probability. For simplicity's sake, even though logarithms for such probabilities are negative, all the $a[i]$ must be 0 or greater (and thus are the negated values of the already negative logarithm approximations) and must form a nondecreasing sequence, and all the $err[i]$ must be 0 or greater.

1. Set $intinf$ to floor(max(0, abs(a[0]))). (This is the absolute integer part of the first term in the series, or 0, whichever is greater.)
2. If $intinf$ is greater than 0, generate unbiased random bits until a zero bit or $intinf$ bits were generated this way. If a zero was generated this way, return 0.
3. Generate an exponential random number E with rate ln(2). This can be done, for example, by using the algorithm given in "**More Algorithms for Arbitrary-Precision Sampling**". (We take advantage of the exponential distribution's *memoryless property*: given that an exponential random number E is greater than $intinf$, E minus $intinf$ has the same distribution.)
4. Set n to 0.
5. 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 a converges to a *natural logarithm* rather than a base-2 logarithm is trivial by comparison. Again for this algorithm, all the $a[i]$ must be 0 or greater and form a nondecreasing sequence, and all the $err[i]$ must be 0 or greater.

1. Generate an exponential random number E (with rate 1).
2. Set n to 0.
3. 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.

Examples:

- Let f($lambda$) = cosh(1)−1. The first algorithm in this section can simulate this constant if step 6 is modified to read: "Let m be ((n+1)*2), and let α be 1/(m!) (a term of the Taylor series). Add α to $lamunq$ and set ϵ to 2/((m+1)!) (the error term).".**(34)**
- Logarithms can form the basis of efficient algorithms to simulate the probability z = choose(n, k)/2^n when n can be very large (e.g., as large as 2^{30}), without relying on floating-point arithmetic. In this example, the trivial algorithm for choose(n, k), the binomial coefficient, will generally require a growing amount of storage that depends on n and k. On the

other hand, any constant can be simulated using up to two unbiased random bits on average, and even slightly less than that for the constants at hand here (Kozen 2014)[35]. Instead of calculating the binomial coefficient directly, a series can be calculated that converges to that coefficient's logarithm, such as ln(choose(n, k)), which is economical in space even for large n and k. Then the algorithm above can be used with that series to simulate the probability z. A similar approach has been implemented (see **interval.py** and **betadist.py**). See also an appendix in (Bringmann et al. 2014)[36].

4.3 Other General Algorithms

4.3.1 Convex Combinations

Assume we have one or more input coins h_i($lambda$) that return heads with a probability that depends on $lambda$. (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)[8]. (Alternatively, the algorithm can be seen as returning heads with probability **E**[h_X($lambda$)], that is, the expected or average value of h_X where X is the number that identifies the randomly chosen coin.)

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

Notes:

1. **Building convex combinations.** Assume we have a function of the form f($lambda$) = $\sum_{n=0,1,...} w_n$($lambda$), where w_n are continuous functions whose maximum values in the domain [0, 1] sum to 1 or less. Let $g(n)$ be the probability that a random number X is n. 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($lambda$) = $\sum_{n=0,1,...} g(n)$ * (w_n($lambda$) / $g(n)$),

   but this works only if the following two conditions are met for each integer $n \geq 0$:

   - $g(n) \geq w_n$($lambda$) ≥ 0 for all $lambda$ in the interval [0, 1] (which roughly means that w_n is bounded from above or "dominated" by $g(n)$).
   - The function w_n($lambda$)/$g(n)$ admits a Bernoulli factory (which it won't if it touches 0 or 1 inside the interval (0, 1), but isn't constant, for example).

See also Mendo (2019)[25].

2. **Constants with non-negative series expansions.** A special case of note 1. Let g be as in note 1. Assume we have a constant with the following series expansion: $c = \sum_{n=0,1,...} a_n$, where a_n are each 0 or greater and sum to 1 or less. 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—

   $c = \sum_{n=0,1,...} g(n) * (a_n / g(n))$,

   but only if $g(n) \geq a_n \geq 0$ for each integer $n \geq 0$.

Examples:

1. Generate a Poisson(μ) random number X, then flip the input coin. With probability $1/(1+X)$, return the result of the coin flip; otherwise, return 0. This corresponds to $g(i)$ being the Poisson(μ) probabilities and 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. Generate a Poisson(μ) random number X and return 1 if X is 0, or 0 otherwise. This is a Bernoulli factory for $\exp(-\mu)$ mentioned earlier, and corresponds to $g(i)$ being the Poisson(μ) probabilities and the coin for h_i returning 1 if i is 0, and 0 otherwise.
3. Generate a Poisson(μ) random number X, run the **algorithm for exp(−z)** with $z = X$, and return the result. The probability of returning 1 this way is $\mathbf{E}[\exp(-X)]$, or $\exp(\mu*\exp(-1)-\mu)$. The following Python code uses the computer algebra library SymPy to find this probability: `from sympy.stats import *; E(exp(-Poisson('P', x))).simplify()`.
4. *Bernoulli Race* (Dughmi et al. 2017)[37]: Say we have n coins, then choose one of them uniformly at random and flip that coin. If the flip returns 1, return X; otherwise, repeat this algorithm. This algorithm chooses a random coin based on its probability of heads. Each iteration corresponds to $g(i)$ being $1/n$ and $h_i()$ being the probability for the corresponding coin i.
5. (Wästlund 1999)[8]: Generate a Poisson(1) random number X, then flip the input coin X times. Return 0 if any of the flips returns 1, or 1 otherwise. This is a Bernoulli factory for $\exp(-\$lambda\$)$, and corresponds to $g(i)$ being the Poisson(1) probabilities, namely $1/(i!*\exp(1))$, and $h_i()$ being $(1-\$lambda\$)^i$.
6. Multivariate Bernoulli factory (Huber 2016)[38] 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, and $R \leq 1 - \epsilon$ for any $\epsilon > 0$: 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 Poisson process driven by multiple input coins.
7. **Probability generating function** (PGF) (Dughmi et al. 2017)[37]. Generates heads with probability $\mathbf{E}[\$lambda\$^X]$, that is, the expected or average value of $\$lambda\$^X$. $\mathbf{E}[\$lambda\$^X]$ 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.

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

4.3.2 Integrals

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

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

This can be done by modifying the algorithm as follows:

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

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

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

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

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

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

4.4 Algorithms for Specific Functions of $lambda$

This section describes algorithms for specific functions, especially when they have a more convenient simulation than the general-purpose algorithms given earlier.

4.4.1 exp(−$lambda$)

This algorithm converges quickly everywhere in (0, 1). (In other words, the algorithm is *uniformly fast*, meaning the average running time is finite for all choices of $lambda$ and other parameters (Devroye 1986, esp. p. 717)**(30)**.**(39)**) This algorithm is adapted from the general martingale algorithm (in "Certain Power Series", above), and makes use of the fact that exp(−$lambda$) can be rewritten as $1 - \$lambda\$ + \$lambda\$^2/2 - \$lambda\$^3/6 + \$lambda\$^4/24 - ...$, which is an alternating series whose coefficients are 1, 1, 1/(2!), 1/(3!), 1/(4!), ...

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

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

4.4.2 exp(−($lambda$^k * c))

In the algorithms in this section, *k* is an integer 0 or greater, and *c* is a real number.

The first algorithm works when ***c* is 0 or greater**.

1. Special case: If *c* is 0, return 1. If *k* is 0, run the **algorithm for exp(−*z*)** (given later in this page) with $z = c$, and return the result.
2. Generate a Poisson(*c*) random integer, call it *N*. (See the appendix on the von Neumann schema for information on generating this integer exactly.)
3. Set *i* to 0, then while $i < N$:
    1. Flip the input coin until the flip returns 0 or the coin is flipped *k* times, whichever comes first. Return 0 if all of the coin flips (including the last) return 1.
    2. Add 1 to *i*.
4. Return 1.

The second algorithm applies the general martingale algorithm, but works only when ***c* is a rational number in the interval [0, 1]**. The target function is represented as a series $1 - \lambda^{k}*c + \lambda^{2*k}*c/2! - \lambda^{3*k}*x/3!$, ..., and the coefficients are 1, *c*, *c*/(2!), *c*/(3!), ...

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

The third algorithm builds on the second algorithm and works when ***c* is a rational number 0 or greater**.

1. Let *m* be floor(*c*). Call the second algorithm *m* times with $k = k$ and $c = 1$. If any of these calls returns 0, return 0.
2. If *c* is an integer, return 1.
3. Call the second algorithm once, with $k = k$ and $c = c - \text{floor}(c)$. Return the result of this call.

Example: **exp(−((1−$lambda$)^{1} * *c*))** ((Dughmi et al. 2017)[37]; applies an exponential weight—here, *c*— to an input coin): "(1) If *c* is 0, return 1; (2) Generate a Poisson(*c*) random integer, call it *N*; (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."

4.4.3 exp(−($lambda$ + *m*)^{k})

In the following algorithm, *m* and *k* are both integers 0 or greater unless noted otherwise.

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

4.4.4 exp($lambda$)*(1−$lambda$)

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

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

4.4.5 1/(2^{k} + $lambda$) or exp(−(*k* + $lambda$)*ln(2))

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

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

4.4.6 1/(2$^{m*(k}$ + $lambda$)) or 1/((2^{m})*(*k* + $lambda$)) or exp(−(*k* + $lambda$)*ln(2^{m}))

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

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

4.4.7 1/(1+$lambda$)

This algorithm is a special case of the two-coin Bernoulli factory of (Gonçalves et al., 2017)[40] and is uniformly fast. It will be called the **two-coin special case** in this document.[41]

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

4.4.8 $lambda$/(1+$lambda$)

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

4.4.9 *c* * $lambda$ * *β* / (*β* * (*c* * $lambda$ + *d* * *μ*) − (*β* − 1) * (*c* + *d*))

This is the general **two-coin algorithm** of (Gonçalves et al., 2017)[40] and (Vats et al. 2020)[42]. It takes two input coins that each output heads (1) with probability $lambda$ or μ, respectively. It also takes a parameter β in the interval [0, 1], which is a so-called "portkey" or early rejection parameter (when $\beta = 1$, the formula simplifies to c * $lambda$ / (c * $lambda$ + d * μ)). In Vats et al. (2020)[42], β, c, d, $lambda$ 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 $lambda$ input coin. Otherwise, flip the μ input coin. If the $lambda$ 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.

4.4.10 *c* * $lambda$ / (*c* * $lambda$ + *d*) or (*c*/*d*) * $lambda$ / (1 + (*c*/*d*) * $lambda$))

This algorithm, also known as the **logistic Bernoulli factory** (Huber 2016)[38], (Morina et al., 2019)[20], is a special case of the two-coin algorithm above, but this time uses only one input coin.

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

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

4.4.11 (*d* + $lambda$) / *c*

This algorithm currently works only if d and c are 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.

4.4.12 *d* / (*c* + $lambda$)

In this algorithm, c must be 1 or greater and d must be in the interval [0, c]. See also the algorithms for continued fractions. (For example, when $d = 1$, this algorithm can simulate a probability of the form $1 / z$, where z is greater than 0 and made up of an integer part (c) and a fractional part ($lambda$) 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.

Example: **1 / (*c* + $lambda$)**: Run the algorithm for d / (c + $lambda$) with $d = 1$.

4.4.13 (*d* + *μ*) / (*c* + $lambda$)

Combines the algorithms in the previous two sections. This algorithm currently works only if d and c are 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 $lambda$ input coin. If the flip returns 1, return 0. Otherwise, go to step 1.

4.4.14 (*d* + *μ*) / ((*d* + *μ*) + (*c* + $lambda$))

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

1. Run the sub-algorithm given later, using the μ input coin and with $a = d+c$ and $b = 1+d+c$. If it returns 1:
    1. If c is 0, return 1.
    2. Run the sub-algorithm using the μ input coin and with $a = d$ and $b = d + c$. If it returns 1, return 1. Otherwise, return 0.
2. Flip the $lambda$ input coin. If the flip returns 1, return 0. Otherwise, go to step 1.

The following sub-algorithm simulates $(a+\mu) / (b+\mu)$.

1. With probability $b / (1+b)$, do the following:
    1. Generate an integer in [0, b) uniformly at random, call it i.
    2. If $i < a$, return 1. If $i = a$, flip the input coin and return the result. If neither is the case, return 0.

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

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

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

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

4.4.16 1 / (1 + (c/d)*$lambda$)

This algorithm is a special case of the two-coin algorithm. In this algorithm, c/d must be 0 or greater.

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

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

4.4.17 $lambda$ + μ

(Nacu and Peres 2005, proposition 14(iii))[15]. This algorithm takes two input coins that simulate $lambda$ or μ, respectively, and a parameter ϵ, which must be greater than 0 and chosen such that $lambda$ + μ < 1 − ϵ.

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 $lambda$ input coin and return the result. Otherwise, flip the μ input coin and return the result."
2. Call the **2014 algorithm**, the **2016 algorithm**, or the **2019 algorithm**, described later, using the ν input coin, x/y = 2/1, i = 1 (for the 2019 algorithm), and ϵ = ϵ, and return the result.

4.4.18 $lambda$ − μ

(Nacu and Peres 2005, proposition 14(iii-iv))[15]. This algorithm takes two input coins that simulate $lambda$ or μ, respectively, and a parameter ϵ, which must be greater than 0 and chosen such that $lambda$ − μ > ϵ (and should be chosen such that ϵ is slightly less than $lambda$ − μ).

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 $lambda$ input coin and return **1 minus the result**. Otherwise, flip the μ input coin and return the result."
2. Call the **2014 algorithm**, the **2016 algorithm**, or the **2019 algorithm**, described later, using the ν input coin, x/y = 2/1, i = 1 (for the 2019 algorithm), and ϵ = ϵ, and return 1 minus the result.

4.4.19 1 − $lambda$

(Flajolet et al., 2010)[1]: Flip the $lambda$ input coin and return 0 if the result is 1, or 1 otherwise.

4.4.20 ν * $lambda$ + (1 − ν) * μ

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

4.4.21 $lambda$ + μ − ($lambda$ * μ)

(Flajolet et al., 2010)[1]: Flip the $lambda$ input coin and the μ input coin. Return 1 if either flip returns 1, and 0 otherwise.

4.4.22 ($lambda$ + μ) / 2

(Nacu and Peres 2005, proposition 14(iii))[15]; (Flajolet et al., 2010)[1]: Generate an unbiased random bit. If that bit is 1 (which happens with probability 1/2), flip the $lambda$ input coin and return the result. Otherwise, flip the μ input coin and return the result.

4.4.23 $lambda$$^{x/y}$

In the algorithm below, the case where x/y is in (0, 1) is due to Mendo (2019)[25]. The algorithm works only when x/y is 0 or greater.

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

    Note: When x/y is less than 1, the minimum number of coin flips needed, on average, by this algorithm will grow without bound as $lambda$ approaches 0. In fact, no fast Bernoulli factory algorithm can avoid this unbounded growth without additional information on $lambda$ (Mendo 2019)[25].

4.4.24 $lambda$$^{\mu}$

This algorithm is based on the previous one, but changed to accept a second input coin

(which outputs heads with probability μ) rather than a fixed value for the exponent. To the best of my knowledge, I am not aware of any article or paper by others that presents this particular Bernoulli factory.

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

4.4.25 sqrt($lambda$)

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

4.4.26 $lambda$ * μ

(Flajolet et al., 2010)**(1)**: Flip the $lambda$ input coin and the μ input coin. Return 1 if both flips return 1, and 0 otherwise.

4.4.27 $lambda$ * x/y (linear Bernoulli factories)

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

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

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

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

The second algorithm is called the **2016 algorithm** (Huber 2016)**(38)** and uses the same parameters x, y, and ϵ, and its description uses the same special cases. The difference here is that it involves a so-called "logistic Bernoulli factory", which is replaced in this

document with a different one that simulates the same function. When x/y is greater than 1, the ϵ parameter has to be chosen such that $lambda$ * $x/y \le 1 - \epsilon$.

1. The same special cases as for the 2014 algorithm apply.
2. Set m to ceil(1 + 9 / (2 * ϵ)).
3. Set β to 1 + 1 / (m − 1).
4. **Algorithm A** is what Huber calls this step. Set s to 1, then while s is greater than 0 and less than m:
    1. Run the **logistic Bernoulli factory** algorithm with $c/d = \beta * x/y$.
    2. Set s to $s - z * 2 + 1$, where z is the result of the logistic Bernoulli factory.
5. If s is other than 0, return 0.
6. With probability $1/\beta$, return 1.
7. Do a separate run of the currently running algorithm, with $x/y = \beta * x/y$ and $\epsilon = 1 - \beta * (1 - \epsilon)$. If the separate run returns 0, return 0.
8. The **high-power logistic Bernoulli factory** is what Huber calls this step. Set s to 1, then while s is greater than 0 and less than or equal to m minus 2:
    1. Run the **logistic Bernoulli factory** algorithm with $c/d = \beta * x/y$.
    2. Set s to $s + z * 2 - 1$, where z is the result of the logistic Bernoulli factory.
9. If s is equal to m minus 1, return 1.
10. Subtract 1 from m and go to step 7.

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

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

    Note: For approximate methods to simulate $lambda$*(x/y), see the examples in "**Certain Polynomials**".

4.4.28 ($lambda$ * x/y)^i

(Huber 2019)[43]. This algorithm, called the **2019 algorithm** in this document, uses four parameters: x, y, i, and ϵ, such that $x/y > 0$, $i \ge 0$ is an integer, and ϵ is greater than 0. When x/y is greater than 1, the ϵ parameter has to be chosen such that $lambda$ * $x/y < 1 - \epsilon$. It also has special cases not mentioned in Huber 2019.

1. Special cases: If i is 0, return 1. If x is 0, return 0. Otherwise, if x equals y and i equals 1, flip the input coin and return the result.
2. Special case: If x is less than y and $i = 1$, then: (a) With probability x/y, flip the input coin and return the result; otherwise (b) return 0.
3. Special case: If x is less than y, then create a secondary coin μ that does the following: "(a) With probability x/y, flip the input coin and return the result; otherwise (b) return 0", then run the **algorithm for ($\mu^{i/1}$)** (described earlier) using this secondary coin.
4. Set t to 355/100 and c to x/y.
5. If i is 0, return 1.
6. While $i = t / \epsilon$:
    1. Set β to $(1 - \epsilon / 2) / (1 - \epsilon)$.
    2. Run the **algorithm for (1/β)^i** (the algorithm labeled x^y and given in the

irrational constants section). If it returns 0, return 0.
    3. Multiply c by β, then divide ϵ by 2.
7. Run the **logistic Bernoulli factory** with $c/d = c$, then set z to the result. Set i to $i + 1 - z * 2$, then go to step 5.

4.4.29 ϵ / $lambda$

(Lee et al. 2014)[44] This algorithm, in addition to the input coin, takes a parameter ϵ, which must be greater than 0 and be chosen such that ϵ is less than $lambda$.

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

4.4.30 arctan($lambda$) /$lambda$

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

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

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

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

4.4.31 arctan($lambda$)

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

4.4.32 cos($lambda$)

This algorithm adapts the general martingale algorithm for this function's series expansion. In fact, this is a special case of Algorithm 3 of (Łatuszyński et al. 2009/2011)[26] (which is more general than Proposition 3.4, the general martingale algorithm). The series expansion for cos($lambda$) is 1 $-$ $lambda$^{2}/(2!) + $lambda$^{4}/(4!) $-$ ..., which is an alternating series except the exponent is increased by 2 (rather than 1) with each term. The coefficients are thus 1, 1/(2!), 1/(4!), ... A *lower truncation* of the series is a truncation of that series that ends with a minus term, and the corresponding *upper truncation* is the same truncation but without the last minus term. This series expansion meets the requirements of Algorithm 3 because—

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

The algorithm to simulate cos($lambda$) follows.

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

4.4.33 sin($lambda$)

This algorithm is likewise a special case of Algorithm 3 of (Łatuszyński et al. 2009/2011)[26]. sin($lambda$) can be rewritten as $lambda$ * (1 − $lambda$^{2}/(3!) + $lambda$^{4}/(5!) − ...), which includes an alternating series where the exponent is increased by 2 (rather than 1) with each term. The coefficients are thus 1, 1/(3!), 1/(5!), ... This series expansion meets the requirements of Algorithm 3 for the same reasons as the cos($lambda$) series does.

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

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

4.4.34 (1−$lambda$)/cos($lambda$)

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

1. Flip the input coin until the flip returns 0. Then set G to the number of times the flip returns 1 this way.
2. If G is **odd**, return 0.

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

4.4.35 (1−$lambda$) * tan($lambda$)

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

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

4.4.36 ln(1+$lambda$)

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

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

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

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

4.4.37 1 − ln(1+$lambda$)

Return 1 minus the result of the algorithm for ln(1+$lambda$).[45]

4.4.38 arcsin($lambda$) + sqrt(1 − $lambda$^{2}) − 1

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

1. Generate a uniform(0, 1) random number u.

2. Create a secondary coin μ that does the following: "**Sample from the number u** twice, and flip the input coin twice. If all of these calls and flips return 1, return 0. Otherwise, return 1."
3. Call the **algorithm for $\mu^{1/2}$** using the secondary coin μ. If it returns 0, return 0.
4. 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.

4.4.39 arcsin($lambda$) / 2

The Flajolet paper doesn't explain in detail how arcsin($lambda$)/2 arises out of arcsin($lambda$) + sqrt(1 − $lambda$^{2}) − 1 via Bernoulli factory constructions, but here is an algorithm.**(46)** However, the number of input coin flips is expected to grow without bound as $lambda$ approaches 1.

1. Generate an unbiased random bit. If that bit is 1 (which happens with probability 1/2), run the **algorithm for arcsin($lambda$) + sqrt(1 − $lambda$^{2}) − 1** and return the result.
2. Create a secondary coin μ that does the following: "Flip the input coin twice. If both flips return 1, return 0. Otherwise, return 1." (The coin simulates 1 − $lambda$^{2}.)
3. Call the **algorithm for $\mu^{1/2}$** using the secondary coin μ. If it returns 0, return 1; otherwise, return 0. (This step effectively cancels out the sqrt(1 − $lambda$^{2}) − 1 part and divides by 2.)

4.4.40 Expressions Involving Polylogarithms

The following algorithm simulates the expression Li_r($lambda$) * (1 / $lambda$ − 1), where 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 $lambda$ is 1/2, this expression simplifies to $\mathrm{Li}_r(1/2)$. See also (Flajolet et al., 2010)**(1)**. 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 $h_i() = i^r$, 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 Algorithms for Specific Constants

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

4.5.1 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, 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.

4.5.2 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.

4.5.3 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.

4.5.4 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.

4.5.5 arctan(*x*/*y*) * *y*/*x*

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

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

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

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

4.5.6 π / 12

Two algorithms:

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

4.5.7 π / 4

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

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

4.5.8 1 / π

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

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

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

4.5.9 $(a/b)^{x/y}$

In the algorithm below, a, b, x, and y are integers, and the case where x/y is in (0, 1) is due to recent work by Mendo (2019)[(25)]. This algorithm works only if—

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

The algorithm follows.

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

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

4.5.10 exp(−*x*/*y*)

This algorithm takes integers $x \geq 0$ and $y > 0$ and outputs 1 with probability `exp(-x/y)` or 0 otherwise. It originates from (Canonne et al. 2020)**(47)**.

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

4.5.11 exp(−*z*)

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

More specifically:

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

The algorithm is then as follows:

- For each component $LC[i]$, call the **algorithm for exp(− *LI*[*i*]/1)**, and call the **general martingale algorithm** adapted for **exp(−$lambda$)** using the input coin that simulates $LF[i]$. If any of these calls returns 0, return 0; otherwise, return 1. (See also (Canonne et al. 2020)**(47)**.)

4.5.12 $(a/b)^z$

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

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

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

4.5.13 1 / (1 + exp(x / (y * 2^{prec})) (LogisticExp)

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

1. Generate an unbiased random bit. If that bit is 1 (which happens with probability 1/2), return 1.
2. Call the **algorithm for exp(− x/(y * 2^{prec}))**. If the call returns 1, return 1. Otherwise, go to step 1.

4.5.14 1 / (1 + exp(z / 2^{prec})) (LogisticExp)

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

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

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

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

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

1. Generate three uniform(0,1) random numbers.
2. 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 numbers.)

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(0,1) random numbers.
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.

4.5.16 erf(x)/erf(1)

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

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

In fact, this algorithm takes advantage of a theorem related to the Forsythe method of random sampling (Forsythe 1972)[48]. 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.

4.5.17 2 / (1 + exp(2)) or (1 + exp(0)) / (1 + exp(1))

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

1. Generate an **exponential** random number ex, then set k to 1.
2. Set u to point to the same value as ex.
3. Generate a **uniform(0,1)** random number v.

4. Set *stop* to 1 if *u* is less than *v*, and 0 otherwise.
5. If *stop* is 1 and *k* **is even**, return a number that is 0 if *ex* is **less than 1**, and 1 otherwise. Otherwise, if *stop* is 1, go to step 1.
6. Set *u* to *v*, then add 1 to *k*, then go to step 3.

4.5.18 (1 + exp(1)) / (1 + exp(2))

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

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

4.5.19 (1 + exp(*k*)) / (1 + exp(*k* + 1))

This algorithm simulates this probability by computing lower and upper bounds of exp(1), which improve as more and more digits are calculated. These bounds are calculated by an algorithm by Citterio and Pavani (2016)[49]. Note the use of the methodology in (Łatuszyński et al. 2009/2011, algorithm 2)[26] in this algorithm. In this algorithm, *k* must be an integer 0 or greater.

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

4.5.20 Euler's Constant *γ*

The following algorithm to simulate Euler's constant *γ* is due to Mendo (2020)[33]. This solves an open question given in (Flajolet et al., 2010)[1]. The series used was given by Sondow (2005)[50]. An algorithm for *γ* appears here even though it is not yet known whether this constant is irrational.

1. Set *ϵ* to 1, then set *n*, *lamunq*, *lam*, *s*, *k*, and *prev* to 0 each.

2. Add 1 to k, then add $s/(2^k)$ to lam.
3. If $lamunq+\epsilon \le lam + 1/(2^k)$, 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 series for γ to $lamunq$, and sets ϵ to an upper bound on the error that results if the series is truncated after summing this and the previous terms.) If n is 0, add 1/2 to $lamunq$ and set ϵ to 1/2. Otherwise, add $B(n)/(2*n*(2*n+1)*(2*n+2))$ to $lamunq$ and set ϵ to $\min(prev, (2+B(n)+(1/n))/(16*n*n))$, where $B(n)$ is the minimum number of bits needed to store n (or the smallest $b \ge 1$ 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.

4.5.21 exp(−*x*/*y*) * *z*/*t*

This algorithm is again based on an algorithm due to Mendo (2020)[33]. In this algorithm, x, y, z, and t are integers greater than 0, except x and/or z may be 0, and must be such that $\exp(-x/y) * z/t$ is in the interval [0, 1].

1. If z is 0, return 0. If x is 0, return a number that is 1 with probability z/t and 0 otherwise.
2. Set ϵ to 1, then set n, $lamunq$, lam, s, and k to 0 each.
3. Add 1 to k, then add $s/(2^k)$ to lam.
4. If $lamunq+\epsilon \le lam + 1/(2^k)$, 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 alternating series, multiplied by z/t, to $lamunq$, and sets ϵ to an upper bound on how close the current sum is to the desired probability.) Let m be $n*2$. Set ϵ to $z*x^m/(t*(m!)*y^m)$. If m is 0, add $z*(y-x)/(t*y)$ to $lamunq$. Otherwise, add $z*x^m*(m*y-x+y) / (t*y^{m+1}*((m+1)!))$ to $lamunq$.
8. Add 1 to n and go to step 4.
9. Let $bound$ be $lam+1/(2^k)$. If $lamunq+\epsilon \le bound$, 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.

4.5.22 ln(2)

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

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

4.5.23 ln(1+*y*/*z*)

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

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

5 Requests and Open Questions

1. Let a permutation class (such as numbers in descending order) and two continuous probability distributions D and E be given. Consider the following algorithm: Generate a sequence of independent random numbers (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. In this case:

    1. What is the probability that n is returned?
    2. What is the probability that n is odd or even or belongs to a certain class of numbers?
    3. What is the distribution function (CDF) of the first generated number given that n is odd, or that n is even?

    Obviously, these answers depend on the specific permutation class and/or distributions D and E. Thus, answers that work only for particular classes and/or distributions are welcome. See also my Stack Exchange question **Probabilities arising from permutations**.

2. To apply some of the general algorithms for Bernoulli factories, I request expressions of mathematical functions that can be expressed in any of the following ways:

    - Series expansions for continuous functions that equal 0 or 1 at the points 0 and 1.
    - A series expansion with non-negative terms that can be "tucked" under a discrete probability mass function.
    - Series expansions for alternating power series whose coefficients are all in the interval [0, 1] and form a nonincreasing sequence.
    - Series expansions with non-negative coefficients and for which bounds on the truncation error are available.
    - Upper and lower bound approximations that converge to a given constant. These upper and lower bounds must be nonincreasing or nondecreasing, respectively.
    - Sequences of approximating functions (such as rational functions) that converge from above and below to a given function. These sequences must be nonincreasing or nondecreasing, respectively (but the approximating functions themselves need not be).

- Simple **continued fractions** that express useful constants.
- A way to compute two sequences of polynomials written in Bernstein form that converge from above and below to a factory function as follows: (a) Each sequence's polynomials must have coefficients lying in [0, 1], and be of increasing degree; (b) 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). For a formal statement of these polynomials, see my **question on Mathematics Stack Exchange**.

  The **supplemental notes** include formulas for computing these polynomials for the vast majority of functions likely to occur in practice, but not all of them. (a) Are there Bernoulli Factory functions used in practice that don't have a polynomial approximation scheme given in the the supplemental notes? (b) Are there specific functions (especially those in practical use) for which there are practical and faster formulas for building polynomials that converge to those functions in a manner needed for the Bernoulli factory problem (besides those I list in this article or the supplemental notes)?

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

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

4. Part of the reverse-time martingale algorithm of Łatuszyński et al. (2009/2011)**(26)** (see "**General Factory Functions**") to simulate a factory function f($lambda$) is as follows. For each n starting with 1:

   1. Flip the input coin, and compute the n^{th} upper and lower bounds of f given the number of heads so far, call them L and U.
   2. Compute the $(n-1)^{\text{th}}$ upper and lower bounds of f given the number of heads so far, call them L' and U'. (These bounds must be the same regardless of the outcomes of future coin flips, and the interval $[L', U']$ must equal or entirely contain the interval $[L, U]$.)

   These parts of the algorithm appear to work for any two sequences of functions (not just polynomials) that converge to f, where L or L' and U or U' are their lower and upper bound approximations. The section on general factory functions shows how this algorithm can be implemented for polynomials. But how do these steps work when the approximating functions (the functions that converge to f) are rational functions with integer coefficients? Rational functions with rational coefficients? Arbitrary approximating functions?

5. According to (Mossel and Peres 2005)**(17)**, a pushdown automaton can take a coin with unknown probability of heads of $lambda$ and turn it into a coin with probability of heads of f($lambda$) only if f is a factory function and can be a solution of a polynomial system with rational coefficients. (See "**Certain Algebraic Functions**".) Are there any results showing whether the converse is true; namely, can a pushdown automaton simulate *any* f of this kind? Note that this question is not quite the same as the question of which algebraic functions can be simulated by a context-free grammar (either in general or restricted to those of a certain ambiguity and/or alphabet size), and is not quite the same as the question of which *probability generating functions* can be simulated by context-free grammars or pushdown

automata, although answers to those questions would be nice. (See also Icard 2019[19]. Answering this question might involve ideas from analytic combinatorics; e.g., see the recent works of Cyril Banderier and colleagues.)

6 Correctness and Performance Charts

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

7 Acknowledgments

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

8 Notes

- [1] Flajolet, P., Pelletier, M., Soria, M., "**On Buffon machines and numbers**", arXiv:0906.5560 [math.PR], 2010.
- [2] Keane, M. S., and O'Brien, G. L., "A Bernoulli factory", *ACM Transactions on Modeling and Computer Simulation* 4(2), 1994.
- [3] There is an analogue to the Bernoulli factory problem called the *quantum Bernoulli factory*, with the same goal of simulating functions of unknown probabilities, but this time with algorithms that employ quantum-mechanical operations (unlike *classical* algorithms that employ no such operations). However, quantum-mechanical programming is far from being accessible to most programmers at the same level as classical programming, and will likely remain so for the foreseeable future. For this reason, the *quantum Bernoulli factory* is outside the scope of this document, but it should be noted that more factory functions can be "constructed" using quantum-mechanical operations than by classical algorithms. For example, a factory function defined in [0, 1] has to meet the requirements proved by Keane and O'Brien except it can touch 0 and/or 1 at a finite number of points in the domain (Dale, H., Jennings, D. and Rudolph, T., 2015, "Provable quantum advantage in randomness processing", *Nature communications* 6(1), pp. 1-4).
- [4] Huber, M., "**Nearly optimal Bernoulli factories for linear functions**", arXiv:1308.1562v2 [math.PR], 2014.
- [5] 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.**
- [6] Goyal, V. and Sigman, K., 2012. On simulating a class of Bernstein polynomials. ACM Transactions on Modeling and Computer Simulation (TOMACS), 22(2), pp.1-5.
- [7] Weikang Qian, Marc D. Riedel, Ivo Rosenberg, "Uniform approximation and Bernstein polynomials with coefficients in the unit interval", *European Journal of Combinatorics* 32(3), 2011, **https://doi.org/10.1016/j.ejc.2010.11.004** **http://www.sciencedirect.com/science/article/pii/S0195669810001666**
- [8] Wästlund, J., "**Functions arising by coin flipping**", 1999.
- [9] 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.
- [10] Thomas, A.C., Blanchet, J., "**A Practical Implementation of the Bernoulli Factory**", arXiv:1106.2508v3 [stat.AP], 2012.
- [11] S. Ray, P.S.V. Nataraj, "A Matrix Method for Efficient Computation of Bernstein Coefficients", Reliable Computing 17(1), 2012.
- [12] 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).

- [13] Henderson, S.G., Glynn, P.W., "Nonexistence of a class of variate generation schemes", *Operations Research Letters* 31 (2003).
- [14] For this approximation, if n were infinity, the method would return 1 with probability 1 and so would not approximate $lambda$*c, of course.
- [15] Nacu, Şerban, and Yuval Peres. "**Fast simulation of new coins from old**", The Annals of Applied Probability 15, no. 1A (2005): 93-115.
- [16] Niazadeh, R., Leme, R.P., Schneider, J., "**Combinatorial Bernoulli Factories: Matchings, Flows, and Polytopes**", arXiv:2011.03865v1 [cs.DS], Nov. 7, 2020.
- [17] Mossel, Elchanan, and Yuval Peres. New coins from old: computing with unknown bias. Combinatorica, 25(6), pp.707-724.
- [18] Smith, N. A. and Johnson, M. (2007). Weighted and probabilistic context-free grammars are equally expressive. Computational Linguistics, 33(4):477–491.
- [19] Icard, Thomas F., "Calibrating generative models: The probabilistic Chomsky–Schützenberger hierarchy." Journal of Mathematical Psychology 95 (2020): 102308.
- [20] Morina, G., Łatuszyński, K., et al., "**From the Bernoulli Factory to a Dice Enterprise via Perfect Sampling of Markov Chains**", arXiv:1912.09229 [math.PR], 2019/2020.
- [21] Propp, J.G., Wilson, D.B., "Exact sampling with coupled Markov chains and applications to statistical mechanics", 1996.
- [22] A *pushdown automaton*, as used here, is defined in Mossel and Peres 2005 and is described as a machine that maintains a stack of symbols and transitions from one state to another based on the current state, the symbol at the top of the stack, and the outcome of a biased coin flip. With each state transition, the machine adds symbols to the stack or removes symbols from it. When the stack is empty, the machine halts, and the result is 0 or 1 depending on the machine's state at that time.
- [23] The probability given in Theorem 3.2 of the Flajolet paper, namely just "$\sum_{k = 0, 1, 2, ...} (W(k) * (\lambda/2)^k)$", appears to be incorrect in conjunction with Figure 4 of that paper.
- [24] Here, "choose(g, g/t)" means that out of g letters, g/t of them must be A's, and "$(\beta-1)^{g-g/t}$" is the number of words that have $g-g/t$ letters other than A, given that the remaining letters were A's.
- [25] 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.
- [26] 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.
- [27] Flegal, J.M., Herbei, R., "Exact sampling from intractible probability distributions via a Bernoulli factory", *Electronic Journal of Statistics* 6, 10-37, 2012.
- [28] Holtz, O., Nazarov, F., Peres, Y., "New Coins from Old, Smoothly", *Constructive Approximation* 33 (2011).
- [29] Brassard, G., Devroye, L., Gravel, C., "Remote Sampling with Applications to General Entanglement Simulation", *Entropy* 2019(21)(92), **https://doi.org/10.3390/e21010092** .
- [30] Devroye, L., ***Non-Uniform Random Variate Generation***, 1986.
- [31] Bill Gosper, "Continued Fraction Arithmetic", 1978.
- [32] Borwein, J. et al. “Continued Logarithms and Associated Continued Fractions.” *Experimental Mathematics* 26 (2017): 412 - 429.
- [33] Mendo, L., "**Simulating a coin with irrational bias using rational arithmetic**", arXiv:2010.14901 [math.PR], 2020.
- [34] The error term, which follows from *Taylor's theorem*, has a numerator of 2 because 2 is higher than the maximum value at the point 1 (in cosh(1)) that f's slope, slope-of-slope, etc. functions can achieve.
- [35] Kozen, D., **"Optimal Coin Flipping"**, 2014.
- [36] 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.
- [37] Shaddin Dughmi, Jason D. Hartline, Robert Kleinberg, and Rad Niazadeh. 2017. Bernoulli Factories and Black-Box Reductions in Mechanism Design. In *Proceedings of 49th Annual ACM SIGACT Symposium on the Theory of Computing*, Montreal, Canada, June 2017 (STOC’17).
- [38] Huber, M., "**Optimal linear Bernoulli factories for small mean problems**", arXiv:1507.00843v2 [math.PR], 2016.
- [39] Another algorithm for exp(−$lambda$) involves the von Neumann schema described in the

appendix, but unfortunately, it converges slowly as $lambda$ approaches 1.

- (40) Gonçalves, F. B., Łatuszyński, K. G., Roberts, G. O. (2017). Exact Monte Carlo likelihood-based inference for jump-diffusion processes.
- (41) There are two other algorithms for this function, but they both converge very slowly when $lambda$ is very close to 1. One is the general martingale algorithm, since when $lambda$ is in [0, 1], this function is an alternating series of the form `1 - x + x^2 - x^3 + ...`, whose coefficients are 1, 1, 1, 1, ... The other is the so-called "even-parity" construction from Flajolet et al. 2010: "(1) Flip the input coin. If it returns 0, return 1. (2) Flip the input coin. If it returns 0, return 0. Otherwise, go to step 1."
- (42) Vats, D., Gonçalves, F. B., Łatuszyński, K. G., Roberts, G. O. "**Efficient Bernoulli factory MCMC for intractable likelihoods**", arXiv:2004.07471 [stat.CO], 2020.
- (43) Huber, M., "**Designing perfect simulation algorithms using local correctness**", arXiv:1907.06748v1 [cs.DS], 2019.
- (44) Lee, A., Doucet, A. and Łatuszyński, K., 2014. "**Perfect simulation using atomic regeneration with application to Sequential Monte Carlo**", arXiv:1407.5770v1 [stat.CO].
- (45) Another algorithm for this function uses the general martingale algorithm, but uses more bits on average as $lambda$ approaches 1. Here, the alternating series is `1 - x + x^2/2 - x^3/3 + ...`, whose coefficients are 1, 1, 1/2, 1/3, ...
- (46) One of the only implementations I could find of this, if not the only, was a **Haskell implementation**.
- (47) Canonne, C., Kamath, G., Steinke, T., "**The Discrete Gaussian for Differential Privacy**", arXiv:2004.00010 [cs.DS], 2020.
- (48) Forsythe, G.E., "Von Neumann's Comparison Method for Random Sampling from the Normal and Other Distributions", *Mathematics of Computation* 26(120), October 1972.
- (49) Citterio, M., Pavani, R., "A Fast Computation of the Best *k*-Digit Rational Approximation to a Real Number", *Mediterranean Journal of Mathematics* 13 (2016).
- (50) Sondow, Jonathan. “New Vacca-Type Rational Series for Euler's Constant and Its 'Alternating' Analog ln 4/*π*.”, 2005.
- (51) von Neumann, J., "Various techniques used in connection with random digits", 1951.
- (52) Pae, S., "Random number generation using a biased source", dissertation, University of Illinois at Urbana-Champaign, 2005.
- (53) Peres, Y., "Iterating von Neumann's procedure for extracting random bits", Annals of Statistics 1992,20,1, p. 590-597.
- (54) Estimating $lambda$ as *λ′*, then finding *f*(*λ′*), is not necessarily an unbiased estimator of *f*($lambda$), even if *λ′* is an unbiased estimator. Indeed, even though standard deviation equals the square root of variance, taking the square root of the bias-corrected sample variance does not lead to an unbiased estimator of the standard deviation.
- (55) Glynn, P.W., "Exact simulation vs exact estimation", *Proceedings of the 2016 Winter Simulation Conference*, 2016.
- (56) Flajolet, P., Sedgewick, R., *Analytic Combinatorics*, Cambridge University Press, 2009.
- (57) Monahan, J.. "Extensions of von Neumann’s method for generating random variables." Mathematics of Computation 33 (1979): 1065-1069.
- (58) 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.

9 Appendix

9.1 Randomized vs. Non-Randomized Algorithms

A *non-randomized algorithm* is a simulation algorithm that uses nothing but the input coin as a source of randomness (in contrast to *randomized algorithms*, which do use other sources of randomness) (Mendo 2019)**(25)**. Instead of generating outside randomness, a

randomized algorithm can implement a ***randomness extraction*** procedure to generate that randomness using the input coins themselves. In this way, the algorithm becomes a *non-randomized algorithm*. For example, if an algorithm implements the **two-coin special case** by generating a random bit in step 1, it could replace generating that bit with flipping the input coin twice until the flip returns 0 then 1 or 1 then 0 this way, then taking the result as 0 or 1, respectively (von Neumann 1951)**(51)**. A non-randomized algorithm works only if the probability of heads of any of the input coins is neither 0 nor 1.

In fact, there is a lower bound on the average number of coin flips needed to turn a coin with one probability of heads of ($lambda$) into a coin with another ($\tau = f$($lambda$)). It's called the *entropy bound* (see, e.g., (Pae 2005)**(52)**, (Peres 1992)**(53)**) and is calculated as—

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

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

For certain values of $lambda$, Kozen (2014)**(35)** showed a tighter lower bound of this kind, but this bound is generally non-trivial and assumes $lambda$ is known. However, if $lambda$ is 1/2 (the input coin is unbiased), this bound is simple: at least 2 flips of the input coin are needed on average to simulate a known constant τ, except when τ is a multiple of $1/(2^n)$ for any integer n.

9.2 Simulating Probabilities vs. Estimating Probabilities

If an algorithm—

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

the algorithm acts as an *unbiased estimator* of f($lambda$) that produces estimates in [0, 1] almost surely (Łatuszyński et al. 2009/2011)**(26)**. As a result, the probability f($lambda$) can be simulated in theory by—

1. finding in some way an unbiased estimate of f($lambda$);**(54)**
2. generating a uniform random number in [0,1], call it u; and
3. returning 1 if u is less than v, or 0 otherwise.

In practice, however, this method is prone to numerous errors, and they include errors due to the use of fixed precision in steps 1 and 2, such as rounding and cancellations. For this reason and also because "exact sampling" is the focus of this page, this page does not cover algorithms that directly estimate $lambda$ or f($lambda$). See also (Mossel and Peres 2005, section 4.3)**(17)**.

Only *factory functions* can have unbiased estimation algorithms whose estimates lie in [0, 1] almost surely (Łatuszyński et al. 2009/2011)**(26)** For example, function A can't serve as a factory function, so no simulator for that function (and no unbiased estimator of the kind just given) is possible. This *is* possible for function B, however (Keane and O'Brien

1994)[2].

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

Glynn (2016)[55] distinguishes between—

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

where $g(X)$ is a random value that follows the desired distribution, based on random numbers X. Again, the focus of this page is "exact sampling" (*exact simulation*), not "exact estimation", but the input coin with probability of heads of $lambda$ can be any "exact estimator" of $lambda$ (as defined above) that outputs either 0 or 1.

9.3 Correctness Proof for the Continued Logarithm Simulation Algorithm

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

Proof. This proof of correctness takes advantage of Huber's "fundamental theorem of perfect simulation" (Huber 2019)[43]. Using Huber's theorem requires proving two things:

- First, we note that the algorithm clearly halts almost surely, since step 1 will stop the algorithm if it reaches the last coefficient, and step 2 always gives a chance that the algorithm will return a value, even if it's called recursively or the number of coefficients is infinite. Thus, the chance the algorithm has to be called recursively or with more iterations shrinks and shrinks as the algorithm does more recursions and iterations.
- Second, we show the algorithm is locally correct when the recursive call in the loop is replaced with an oracle that simulates the correct "continued sub-logarithm". If step 1 reaches the last coefficient, the algorithm obviously passes with the correct probability. Otherwise, we will be simulating the probability $(1 / 2^{c[i]}) / (1 + x)$, where x is the "continued sub-logarithm" and will be at most 1 by construction. 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. □

9.4 Correctness Proof for Continued Fraction Simulation Algorithm 3

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

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

- The algorithm halts almost surely for the same reason as the similar continued logarithm simulator.
- If the recursive call in the loop is replaced with an oracle that simulates the correct "sub-fraction", the algorithm is locally correct. If step 1 reaches the last element of the continued fraction, the algorithm obviously passes with the correct probability. Otherwise, we will be simulating the probability $b[i] / (a[i] + x)$, where x is the "continued sub-fraction" and will be at most 1 by assumption. 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. □

9.5 The von Neumann Schema

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

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

where—

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

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

Examples of permutation classes include—

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

using the notation in "Analytic Combinatorics" (Flajolet and Sedgewick 2009)**(56)**.

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

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

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

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

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

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

```
    p=(1-x)*f
    return p*(1-p)**r

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

> **Note:** The von Neumann schema can simulate any *power series distribution* (such as Poisson, negative binomial, geometric, and logarithmic series), given a suitable exponential generating function. However, because of step 2, the number of input coin flips required by the schema grows without bound as $lambda$ approaches 1.
>
> **Example:** Using the class of *sorted permutations*, we can generate a Poisson($lambda$) random number via the von Neumann schema, where $lambda$ is the probability of heads of the input coin. This would lead to an algorithm for exp(−$lambda$) — return 1 if a Poisson($lambda$) random number is 0, or 0 otherwise — but for the reason given in the note, this algorithm converges slowly as $lambda$ approaches 1. Also, if $c > 0$ is a real number, a Poisson(floor(c)) plus a Poisson(c−floor(c)) random number generates a Poisson(c) random number.

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

- (κ(n; $lambda$)*V(n)/(n!)) / p,

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

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

9.6 Probabilities Arising from Certain Permutations

Certain interesting probability functions can arise from permutations, such as permutations that are sorted or permutations whose highest number appears first. Inspired by the **von Neumann schema** given earlier in this appendix, we can describe the following algorithm:

Let a *permutation class* (such as numbers in descending order) and two continuous probability distributions D and E be given. Consider the following algorithm: Generate a sequence of independent random numbers (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 | Both uniform(0,1) | n / ((n + 1)!). | Odd is 1−exp(−1). Even is exp(−1). | | Numbers sorted in descending order | Each arbitrary | ($\int_{(-\infty,\infty)}$ DPDF(z) * (ECDF(z)$^{n-1}$/((n−1)!) − ECDF(z)^n/(n!)) dz), for all n > 0 (see also proof of Theorem 2.1 of (Devroye 1986, Chapter IV)**(30)**. DPDF and ECDF are defined later. | Odd is denominator of formula 1 below. | | Alternating numbers | Both uniform(0,1) | (a_n * (n + 1) − a_{n+1}) / (n + 1)!, where a_i is the integer at position i (starting at 0) of the sequence **A000111** in the *On-Line Encyclopedia of Integer Sequences*. | Odd is 1−cos(1)/(sin(1)+1); even is cos(1)/(sin(1)+1). | | Any | Both uniform(0,1) | ($\int_{[0,1]}$ 1 * (z^{n-1}*V(n)/((n−1)!) − z^n*V(n+1)/(n!)) dz), for all n > 0. $V(n)$ is the number of permutations of size n that meet the permutation class's requirements. For this algorithm, $V(n)$ must be in the interval (0, n!]; 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!)) / $V(0)$. |

| Permutation Class | Distributions D and E | The probability that the first number in the sequence is less than x 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)**(30)**; see also Forsythe 1972**(48)**). Here, DPDF is the probability density function (PDF) of D, and ECDF is the cumulative distribution function (CDF) of E. If x is uniform(0, 1), this probability becomes $\int_{[0,\,1]}$ $\psi(z)$ dz. | | Numbers sorted in descending order | 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**(57)**). DPDF and ECDF are as above. | | Numbers sorted in descending order | Both uniform(0,1) | Odd is ((1−exp(−x))−exp(1))/(1−exp(1)). Therefore, the first number in the sequence is distributed as exponential(1) and "truncated" to the interval [0, 1] (von Neumann 1951)**(51)**. | | Numbers sorted in descending order | D is uniform(0,1); E is max. of two uniform(0,1) | Odd is erf(x)/erf(1) (uses Formula 1, where DPDF(z) = 1 and ECDF(z) = z^2 for z in [0, 1]; see also **erf(x)/erf(1)**). |

Notes:

1. All the functions possible for formulas 1 and 2 are nondecreasing functions. Both formulas express the cumulative distribution function 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($lambda$)+1/cos($lambda$). Other examples of EGFs were given in the section on the von Neumann schema.

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

9.7 Sketch of Derivation of the Algorithm for 1 / π

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

The algorithm is an application of the **convex combination** technique. Namely, 1 / π can

be seen as a convex combination of two components:

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

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

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

  Notes:

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

9.8 Calculating Bounds for exp(1)

The following implements the parts of Citterio and Pavani's algorithm (2016)**(49)** needed to calculate lower and upper bounds for exp(1) in the form of rational numbers.

Define the following operations:

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

1. Ensure n, then set m to floor(((10^d)−1−p[n+1])/p[n+2]).
2. Return (p[n+2] * m +p[n+1]) / (q[n+2] * m +q[n+1]).

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

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

9.9 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≥0 and j≥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 D + G = E. 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. 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 all $lambda$ in the interval [0, 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 k 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 $m \geq i$, and $(n-m) \geq j$, and $i + j \leq n$, 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−$lambda$)^$n-m$).

Example: We have the following polynomial: 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., 2019)**(20)**. 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**(58)**). 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 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 j in the interval [max(0, k−1), 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