Bernoulli Factory Algorithms

This version of the document is dated 2020-08-14.

Peter Occil

1 Introduction

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

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

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

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. You are welcome to suggest additional Bernoulli factory algorithms, especially—

- **specific continued fraction expansions**,
- **series expansions for the power series algorithms below, and**
- **algorithms that simulate probability mass functions or probability density functions, with or without a normalizing constant.**

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. For example, a Bernoulli factory algorithm can take a coin that returns heads with probability λ and produce a coin that returns heads with probability exp(−λ).

A *factory function* is a function that relates the old probability to the new one. Its domain is [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, in a given interval in [0, 1]—

- is continuous everywhere, and
- either returns a constant value in [0, 1] everywhere, or returns a value in [0, 1] at each of the points 0 and 1 and a value in (0, 1) at each other point.

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

If a function's graph touches 0 or 1 somewhere in (0, 1), papers have suggested dealing with this by modifying the function so it no longer touches 0 or 1 there (for example, $f = 2*\lambda$ might become $f = \min(2 * \lambda, 1 - \epsilon)$ where ϵ is in (0, 1/2) (Keane and O'Brien 1994)[2], (Huber 2014, introduction)[3]), or by somehow ensuring that λ does not come close to the point where the graph touches 0 or 1 (Nacu and Peres 2005, theorem 1)[4].

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

4 Algorithms

In the following algorithms:

- λ is the unknown probability of heads of the input coin.
- The **SampleGeometricBag** and **URandLess** algorithms are described in my article on **partially-sampled random numbers (PSRNs)**.
- The `ZeroOrOne` method should be implemented as shown in my article on **random sampling methods**.
- The instruction to "generate a uniform random number" can be implemented by creating an empty **uniform PSRN** (most accurate) or by generating `RNDEXCRANGE(0, 1)` or `RNDINT(1000)` (less accurate).
- Where an algorithm says "if a is less than b", where a and b are uniform random numbers, it means to run the **URandLess** algorithm on the two PSRNs, or do a less-than operation on a and b, as appropriate.
- 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 rounding error.

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

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

4.1 Algorithms for Functions of λ

4.1.1 Certain Power Series

Mendo (2019)[5] gave a Bernoulli factory algorithm for certain functions that can be rewritten as a series of the form—

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

where $c[i] >= 0$ are the coefficients of the series and sum to 1. 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)[5], 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".

(Łatuszyński et al. 2009/2011)[6] 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 non-increasing 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 probability given 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. Create an empty uniform PSRN.
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. Run the **URandLessThanFraction algorithm** on the PSRN and *l*. If the algorithm returns 1, return 1.
6. Run the **URandLessThanFraction algorithm** on the PSRN and *u*. If the algorithm returns 0, return 0.
7. 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(λ) and cos(λ).)

4.1.2 exp(−λ)

The algorithm in (Flajolet et al., 2010)[1] calls for generating a Poisson(λ) random number and returning 1 if that number is 0, or 0 otherwise. The Poisson generator in turn involves generating a geometric(λ) random number *G*[7], then *G* uniform random numbers, then returning *G* only if all *G* uniform numbers are sorted.[8] The algorithm follows.

1. Flip the input coin until the coin returns 0. Then set *G* to the number of times the coin returns 1 this way.
2. If *G* is 0, return 1.
3. Generate a uniform random number *w*, and set *i* to 1.
4. While *i* is less than *G*:
    1. Generate a uniform random number *U*.
    2. If *w* is less than *U*, break out of this loop and go to step 1.
    3. Add 1 to *i*, and set *w* to *U*.
5. Return 0. (*G* is now a Poisson(λ) random number, but is other than 0.)

This algorithm, however, runs very slowly as λ approaches 1.

Here is an alternative version of the algorithm above, which doesn't generate a geometric random number at the outset.

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

In turn, this algorithm likewise converges very slowly as λ approaches 1.

On the other hand, this third algorithm is converges quickly everywhere in (0, 1). (In other words, the algorithm is *uniformly fast*, meaning the average running time is bounded from above for all choices of λ and other parameters (Devroye 1986, esp. p. 717)[9].) This algorithm is adapted from the general martingale algorithm (in "Certain Power Series", above), and makes use of the fact that exp(−λ) 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. Create an empty uniform PSRN.
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 $l + w$. Otherwise, set *l* to $u - w$.
5. Run the **URandLessThanFraction algorithm** on the PSRN and *l*. If the algorithm returns 1, return 1.
6. Run the **URandLessThanFraction algorithm** on the PSRN and *u*. If the algorithm returns 0, return 0.
7. Add 1 to *n* and go to step 3.

4.1.3 exp(λ)*(1−λ)

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

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

4.1.4 exp(λ * *c* − * *c*)

Used in (Dughmi et al. 2017)[10] to apply an exponential weight (here, *c*) to an input coin.

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

4.1.5 exp(−λ − *c*)

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

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

4.1.6 1/(1+λ)

One algorithm is the general martingale algorithm, since when λ is in [0, 1], this function is an alternating series of the form `1 - x + x^2 - x^3 + ...`, whose coefficients are 1, 1, 1, 1, ... However, this algorithm converges slowly when λ is very close to 1.

A second algorithm is the so-called "even-parity" construction of (Flajolet et al., 2010)[1]. However, this algorithm too converges slowly when λ is very close to 1.

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.

A third algorithm is a special case of the two-coin Bernoulli factory of (Gonçalves et al., 2017)[11] and is uniformly fast, unlike the previous two algorithms. It will be called the **two-coin special case** in this document.

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

4.1.7 log(1+λ)

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

1. Create an empty uniform PSRN.
2. Flip the input coin. If it returns 0, flip the coin again and return the result.
3. Call the **SampleGeometricBag** algorithm with the PSRN. If it returns 0, flip the input coin and return the result.
4. Flip the input coin. If it returns 0, return 0.
5. Call the **SampleGeometricBag** algorithm with the PSRN. If it returns 0, return 0. Otherwise, go to step 2.

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

1. Create an empty uniform PSRN.
2. With probability 1/2, flip the input coin and return the result.
3. Call **SampleGeometricBag** on the PSRN, then flip the input coin. If the call and the flip both return 1, return 0. Otherwise, go to step 2.

4.1.8 1 − log(1+λ)

Invert the result of the algorithm for log(1+λ) (make it 1 if it's 0 and vice versa).[12]

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

This is the general two-coin algorithm of (Gonçalves et al., 2017)[11] and (Vats et al. 2020)[13]. It takes two input coins that each output heads (1) with probability λ or μ, respectively. It also takes a parameter β in the interval [0, 1], which is a so-called "portkey" or early rejection parameter (when β = 1, the formula simplifies to $c * \lambda / (c * \lambda + d * \mu)$).

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

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

This algorithm, also known as the **logistic Bernoulli factory** (Huber 2016)[14], (Morina et al., 2019)[15], 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 coin returns 1, return 1. Otherwise, go to step 1.

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

4.1.11 λ + μ

(Nacu and Peres 2005, proposition 14(iii))[4]. This algorithm takes two input coins that simulate λ or μ, respectively, and a parameter ϵ, which must be greater than 0 and chosen such that $\lambda + \mu < 1 - \epsilon$.

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

4.1.12 λ − μ

(Nacu and Peres 2005, proposition 14(iii-iv))[4]. This algorithm takes two input coins that simulate λ or μ, respectively, and a parameter ϵ, which must be greater than 0 and chosen such that $\lambda - \mu > \epsilon$ (and should be chosen such that ϵ is slightly less than λ − μ).

1. Create a ν input coin that does the following: "With probability 1/2, flip the λ input coin and return **1 minus the result**. Otherwise, flip the

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

4.1.13 1/(*c* + λ)

Works only if $c > 0$.

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

4.1.14 1 − λ

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

4.1.15 ν * λ + (1 − ν) * μ

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

4.1.16 λ + μ − (λ * μ)

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

4.1.17 (λ + μ) / 2

(Nacu and Peres 2005, proposition 14(iii))[4]; (Flajolet et al., 2010)[1]: With probability 1/2, flip the λ input coin and return the result. Otherwise, flip the μ input coin and return the result.

4.1.18 arctan(λ) /λ

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

1. Generate an empty uniform PSRN.
2. Call **SampleGeometricBag** twice on the PSRN, and flip the input coin twice. If any of these calls or flips returns 0, return 1.
3. Call **SampleGeometricBag** twice on the PSRN, and flip the input coin twice. If any of these calls or flips returns 0, return 0. Otherwise, go to step 2.

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

1. Create an empty uniform PSRN.
2. With probability 1/2, return 1.
3. Call **SampleGeometricBag** twice on the PSRN, and flip the input coin twice. If all of these calls and flips return 1, return 0. Otherwise, go to step 2.

4.1.19 arctan(λ)

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

4.1.20 cos(λ)

This algorithm adapts the general martingale algorithm for this function's series expansion. In fact, this is a special case of Algorithm 3 of (Łatuszyński et al. 2009/2011)[6] (which is more general than Proposition 3.4, the general martingale algorithm). The series expansion for cos(λ) is $1 - \lambda^2/(2!) + \lambda^4/(4!) - \ldots$, which is an alternating series except the exponent is increased by 2 (rather than 1) with each term. The coefficients are thus 1, 1/(2!), 1/(4!), ... A *lower truncation* of the series is a truncation of that series that ends with a minus term, and the corresponding *upper truncation* is the same truncation but without the last minus term. This series expansion meets the requirements of Algorithm 3 because—

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

The algorithm to simulate cos(λ) follows.

1. Set u to 1, set w to 1, set l to 0, set n to 1, and set fac to 2.
2. Create an empty uniform PSRN.
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 $l + w / fac$. Otherwise, set l to $u - w / fac$. (Here we divide by the factorial of 2-times-n.)
5. Run the **URandLessThanFraction algorithm** on the PSRN and l. If the algorithm returns 1, return 1.
6. Run the **URandLessThanFraction algorithm** on the PSRN and u. If the algorithm returns 0, return 0.
7. Add 1 to n, then multiply fac by $(n * 2 - 1) * (n * 2)$, then go to step 3.

4.1.21 sin(λ)

This algorithm is likewise a special case of Algorithm 3 of (Łatuszyński et al. 2009/2011)[6]. sin(λ) can be rewritten as $\lambda * (1 - \lambda^2/(3!) + \lambda^4/(5!) - \ldots)$, which includes an alternating series where the exponent is increased by 2 (rather than 1) with each term. The coefficients are thus 1, 1/(3!), 1/(5!), ... This series expansion meets the requirements of Algorithm 3 for the same reasons as the cos(λ) series does.

The algorithm to simulate sin(λ) 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. Create an empty uniform PSRN.
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 $l + w / fac$. Otherwise, set l to $u - w / fac$.
6. Run the **URandLessThanFraction algorithm** on the PSRN and l. If the algorithm returns 1, return 1.
7. Run the **URandLessThanFraction algorithm** on the PSRN and u. If the algorithm returns 0, return 0.
8. Add 1 to n, then multiply fac by (n * 2) * (n * 2 + 1), then go to step 3.

4.1.22 $\lambda^{x/y}$

In the algorithm below, the case where x/y is in (0, 1) is due to recent work by Mendo (2019)[5]. 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. Return 0 with probability $x/(y*i)$.
7. Add 1 to i and go to step 5.

Note: When x/y is less than 1, the minimum number of coin flips needed, on average, by this algorithm will grow without bound as λ approaches 0. In fact, no fast Bernoulli factory algorithm can avoid this unbounded growth without additional information on λ (Mendo 2019)[5]. See also the appendix, which also shows an alternative way to implement this and other Bernoulli factory algorithms using PSRNs, which exploits knowledge of λ but is not the focus of this article since it involves arithmetic.

4.1.23 λ^{μ}

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

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

4.1.24 sqrt(λ)

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

4.1.25 arcsin(λ) + sqrt(1 − λ^2) − 1

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

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

4.1.26 arcsin(λ) / 2

The Flajolet paper doesn't explain in detail how arcsin(λ)/2 arises out of arcsin(λ) + sqrt(1 − λ^2) − 1 via Bernoulli factory constructions, but here is an algorithm.[16] Note, however, that the number of input coin flips is expected to grow without bound as λ approaches 1.

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

4.1.27 λ * μ

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

4.1.28 λ * x/y (linear Bernoulli factories)

In general, this function will touch 0 or 1 somewhere in [0, 1], when $x/y > 0$. 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)[3]. It uses three parameters: x, y, and ϵ, such that $x/y > 0$ and ϵ is greater than 0. When x/y is greater than 1, the ϵ parameter has to be chosen such that $\lambda * x/y < 1 - \epsilon$, in order to bound the function away from 0 and 1. As a result, some knowledge of λ has to be available to the algorithm. (In fact, as simulation results show, the choice of ϵ is crucial to this algorithm's performance; for best results, ϵ should be chosen such that $\lambda * x/y$ is slightly less than $1 - \epsilon$.) The algorithm as described below also includes certain special cases, not mentioned in Huber, to make it more general.

1. Special cases: If x is 0, return 0. Otherwise, if x equals y, flip the input coin and return the result. Otherwise, if x is less than y, then: (a) With probability x/y, flip the input coin and return the result; otherwise (b) return 0.
2. Set c to x/y, and set k to 23 / (5 * ϵ).

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

The second algorithm is called the **2016 algorithm** (Huber 2016)**(14)** 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 λ * x/y <= 1 − ϵ.

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 = β * 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/β, return 1.
7. Run this algorithm recursively, with x/y = β * x/y and ϵ = 1 − β * (1 − ϵ). If it returns 0, return 0.
8. The **high-power logistic Bernoulli factory** is what Huber calls this step. Set s to 1, then while s is greater than 0 and less than or equal to m minus 2:
    1. Run the **logistic Bernoulli factory** algorithm with c/d = β * 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 λ * 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 λ * x / y <= m < 1/2.

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

4.1.29 (λ * x/y)^i

(Huber 2019)**(17)**. This algorithm, called the **2019 algorithm** in this document, uses four parameters: x, y, i, and ϵ, such that x/y > 0, i >= 0 is an integer, and ϵ is greater than 0. When x/y is greater than 1, the ϵ parameter has to be chosen such that λ * x/y < 1 − ϵ. 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 (μ$^{i/1}$)** (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 / ϵ:
    1. Set β to (1 − ϵ / 2) / (1 − ϵ).
    2. Run the **algorithm for (1/β)^i** (described later). If it returns 0, return 0.
    3. Multiply c by β, then divide ϵ by 2.
7. Run the **logistic Bernoulli factory** with c/d = c, then set z to the result. Set i to i + 1 − z * 2, then go to step 5.

4.1.30 ϵ / λ

(Lee et al. 2014)**(18)** This algorithm, in addition to the input coin, takes a parameter ϵ, which must be greater than 0 and be chosen such that ϵ is less than λ.

1. If β to max(ϵ, 1/2) and set γ to 1 − (1 − β) / (1 − (β / 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 − ϵ), i = 1 (for the 2019 algorithm), and ϵ = γ. If the result is 0, return 0. Otherwise, go to step 3. (Note that running the algorithm this way simulates the probability (λ − ϵ)/(1 − ϵ) or 1 − (1 − λ)/(1 − ϵ)).

4.1.31 Certain Rational Functions

According to (Mossel and Peres 2005)**(19)**, a function can be simulated by a finite-state machine (equivalently, a "probabilistic regular grammar" (Smith and Johnson 2007)**(20)**, (Icard 2019)**(21)**) if and only if the function can be written as a rational function with rational coefficients, that takes in an input λ in some subset of (0, 1) and outputs a number in the interval (0, 1).

The following algorithm is suggested from the Mossel and Peres paper and from (Thomas and Blanchet 2012)**(22)**. It assumes the rational function is of the form D(λ)/E(λ), where—

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

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

The algorithm follows.

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

   Notes:

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

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

   2. This algorithm could be modified to avoid additional randomness besides the input coin flips by packing the coin flips into an n-bit word and looking up whether that word is "passing", "failing", or neither, among all n-bit words with j ones, but this is not so trivial to do (especially because in general, a lookup table first has to be built in a setup step, which can be impractical unless 2^n is relatively small). Moreover, $d[i]$ and $e[i]$ would have to be limited to integers.

4.1.32 Bernstein Polynomials

A *Bernstein polynomial* is a polynomial of the form $\Sigma_{i = 0, ..., n}$ choose(n, i) * λ^i * $(1 - \lambda)^{n - i}$ * $a[i]$, where n is the polynomial's degree and $a[i]$ are the control points for the polynomial's corresponding Bézier curve. According to (Goyal and Sigman 2012)[23], a function can be simulated with a fixed number of input coin flips if and only if it's a Bernstein polynomial whose control points are all in the interval [0, 1] (see also (Wästlund 1999, section 4)[24]). They also give an algorithm for simulating these polynomials, which is given below.

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

   Note: Each $a[i]$ acts as a control point for a 1-dimensional **Bézier curve**, where λ is the relative position on that curve, the curve begins at $a[0]$, and the curve ends at $a[n]$. For example, given control points 0.2, 0.3, and 0.6, the curve is at 0.2 when $\lambda = 0$, and 0.6 when $\lambda = 1$.

4.2 Algorithms for Irrational Constants

The following algorithms generate heads with a probability equal to an irrational number. (On the other hand, probabilities that are *rational* constants are trivial to simulate. If fair coins are available, the `ZeroOrOne` method should be used. If coins with unknown bias are available, then a *randomness extraction* method such as the von Neumann algorithm should be used to turn them into fair coins. Randomness extraction is outside the scope of this document, however.)

4.2.1 Digit Expansions

Probabilities can be expressed as a digit expansion (of the form `0.dddddd...`). The following assumes that the probability is in 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)[25], (Devroye 1986, p. 769)[9]), `BASE` is the digit base, such as 2 for binary or 10 for decimal.

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

4.2.2 Continued Fractions

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

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

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

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

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

1. Set r to 1 / (r * b[*pos*]), then set k to a[*pos*] * r. (k is the partial denominator for the equivalent simple continued fraction.)
2. If the partial numerator/denominator pair at *pos* is the last, return a number that is 1 with probability 1/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. With probability $kp/(1+kp)$, return a number that is 1 with probability $1/kp$ and 0 otherwise.
6. Run this algorithm recursively, but with $pos = pos + 1$ and $r = r$. If the result is s, return 0. Otherwise, go to step 5.

Algorithm 3. This algorithm works only if each $b[i]/a[i]$ is 1 or less and if each $b[i]$ and each $a[i]$ is greater than 0, but otherwise, each $b[i]$ and

each $a[i]$ may be a non-integer. The algorithm begins with pos equal to 1. Then the following steps are taken.

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

See the appendix for a correctness proof of Algorithm 3.

Notes:

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

  1. Set i to $size$.
  2. Create an input coin that does the following: "Return a number that is 1 with probability $1/a[size]$ or 0 otherwise".
  3. While i is 1 or greater:
     1. Set k to $a[i]$.
     2. Create an input coin that takes the previous input coin and k and does the following: "(a) With probability $k/(1+k)$, return a number that is 1 with probability $1/k$ and 0 otherwise; (b) Flip the previous input coin. If the result is 1, return 0. Otherwise, go to step (a)". (The probability $k/(1+k)$ is related to $\lambda/(1+\lambda) = 1 - 1/(1+\lambda)$, 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)[26], (Borwein et al., 2016)[27] of a number in (0, 1) has the following continued fraction form: $0 + (1 / 2^{c[1]}) / (1 + (1 / 2^{c[2]}) / (1 + ...))$, where $c[i]$ are the coefficients of the continued logarithm and all 0 or greater. I have come up with the following algorithm that simulates a probability expressed as a continued logarithm expansion:

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

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

For a correctness proof, see the appendix.

4.2.4 1 / φ

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

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

4.2.5 sqrt(2) − 1

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

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

4.2.6 1/sqrt(2)

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

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

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

4.2.7 arctan(x/y) * y/x

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

1. Create an empty uniform PSRN.
2. Generate a number that is 1 with probability $x * x/(y * y)$, or 0 otherwise. If the number is 0, return 1.
3. Call **SampleGeometricBag** twice on the PSRN. 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. Call **SampleGeometricBag** twice on the PSRN. 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. Create an empty uniform PSRN.
2. With probability 1/2, return 1.
3. With probability $x * x/(y * y)$, call **SampleGeometricBag** twice on the PSRN. If both of these calls return 1, return 0.
4. Go to step 2.

4.2.8 π / 12

Two algorithms:

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

4.2.9 π / 4

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

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

4.2.10 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, 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.

4.2.11 $(a/b)^{x/y}$

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)[(5)]. This algorithm works only if—

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

The algorithm follows.

1. If x/y is less than 0, swap a and b, and remove the sign from x/y. If a/b is now no longer in the interval [0, 1], return an error.
2. If x/y is equal to 1, return 1 with probability a/b and 0 otherwise.
3. If x is 0, return 1. Otherwise, if a is 0, return 0. Otherwise, if a equals b, return 1.
4. If x/y is greater than 1:
    1. Set $ipart$ to floor(x/y) and $fpart$ to `rem(x, y)`.
    2. If $fpart$ is greater than 0, subtract 1 from $ipart$, then call this algorithm recursively with x = floor($fpart$/2) and $y = y$, then call this algorithm, again recursively, with $x = fpart -$ floor($fpart$/2) and $y = y$. Return 0 if either call returns 0. (This is done rather than the more obvious approach in order to avoid calling this algorithm with fractional parts very close to 0, because the algorithm runs much more slowly than for fractional parts closer to 1.)
    3. If $ipart$ is 1 or greater, generate a random number that is 1 with probability a^{ipart}/b^{ipart} or 0 otherwise. (Or generate $ipart$ many random numbers that are each 1 with probability a/b or 0 otherwise, then multiply them all into one number.) If that number is 0, return 0.
    4. Return 1.
5. Set i to 1.
6. With probability a/b, return 1.
7. Otherwise, with probability $x/(y*i)$, return 0.
8. Add 1 to i and go to step 6.

4.2.12 exp(−x/y)

This algorithm takes integers $x >= 0$ and $y > 0$ and outputs 1 with probability `exp(-x/y)` or 0 otherwise. It originates from (Canonne et al. 2020)[(28)].

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

4.2.13 exp(−z)

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

More specifically:

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

The algorithm is then as follows:

- For each component $LC[i]$, call the **algorithm for exp(− $LI[i]$/1)**, and call the **general martingale algorithm** adapted for **exp(−λ)** using the input coin that simulates $LF[i]$. If any of these calls returns 0, return 0; otherwise, return 1. (See also (Canonne et al. 2020)[28].)

4.2.14 $(a/b)^z$

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

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

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

4.2.15 1 / 1 + exp(x / (y * 2^{prec})) (LogisticExp)

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

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

4.2.16 1 / 1 + exp(z / 2^{prec})) (LogisticExp)

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

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

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

4.3 General Algorithms

4.3.1 Convex Combinations

Assume we have one or more input coins $h_i(\lambda)$ that returns heads with a probability that depends on λ. The following algorithm chooses one of these coins at random then flips that coin. Specifically, the algorithm simulates the following function: $g(0) * h_0(\lambda) + g(1) * h_1(\lambda) + \ldots$, where $g(i)$ is the probability that coin i will be chosen, and h_i is the function simulated by coin i. See (Wästlund 1999, Theorem 2.7)[24]. (Alternatively, the algorithm can be seen as simulating $\mathbf{E}[h_X(\lambda)]$, that is, the expected or average value of h_X where X is the number that identifies the randomly chosen coin.)

1. Generate a random integer X in some way. For example, it could be a uniform random integer in [1, 6], or it could be a Poisson random number.
2. Flip the coin represented by X and return the result.

Examples:

1. As one example, generate a Poisson random number X, then flip the input coin. With probability $1/(1+X)$, return the result of the coin flip; otherwise, return 0.
2. *Bernoulli Race* (Dughmi et al. 2017)[10]: If 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.

4.3.2 Simulating the Probability Generating Function

The following algorithm is a special case of the convex combination method. It generates heads with probability $\mathbf{E}[\lambda^X]$, that is, the expected or average value of λ^X. $\mathbf{E}[\lambda^X]$ is the *probability generating function*, also known as *factorial moment generating function*, for the distribution of X (Dughmi et al. 2017)[10].

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

4.3.3 URandLessThanFraction

The following helper algorithm is used by some of the algorithms on this page. It returns 1 if a PSRN turns out to be less than a fraction, *frac*, which is a number in the interval [0, 1].

1. If *frac* is 0 or 1, return 0 or 1, respectively. (The case of 1 is a degenerate case since the PSRN could, at least in theory, represent an infinite sequence of ones, making it equal to 1.)
2. Set *pt* to 1/*base*, and set i to 0. (*base* is the base, or radix, of the PSRN's digits, such as 2 for binary or 10 for decimal.)
3. Set *d1* to the digit at the i^{th} position (starting from 0) of the uniform PSRN. If there is no digit there, put a digit chosen uniformly at random at that position and set *d1* to that digit.

4. Set *d2* to floor(*frac* / *pt*). (For example, in base 2, set *d2* to 0 if *frac* is less than *pt*, or 1 otherwise.)
5. If *d1* is less than *d2*, return 1. If *d1* is greater than *d2*, return 0.
6. If *frac* >= *pt*, subtract *pt* from *frac*.
7. Divide *pt* by *base*, add 1 to *i*, and go to step 3.

5 Correctness and Performance Charts

The following charts show the correctness of many of the algorithms on this page and show their performance in terms of the number of bits they use on average. For each algorithm, and for each of 100 λ values evenly spaced from 0.0001 to 0.9999:

- 500 runs of the algorithm were done. Then...
- The number of bits used by the runs were averaged, as were the return values of the runs (since the return value is either 0 or 1, the mean return value will be in the interval [0, 1]). The number of bits used included the number of bits used to produce each coin flip, assuming the coin flip procedure for λ was generated using the `Bernoulli#coin()` method in *bernoulli.py*, which produces that probability in an optimal or near-optimal way.

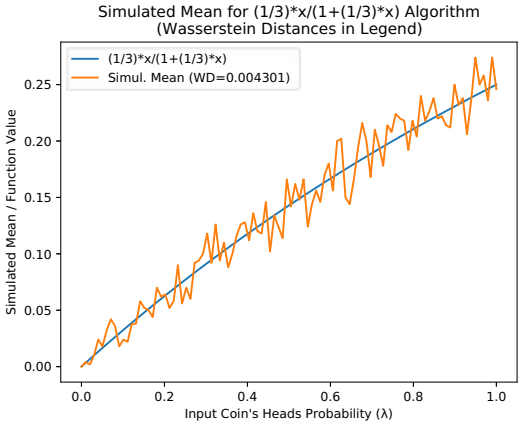
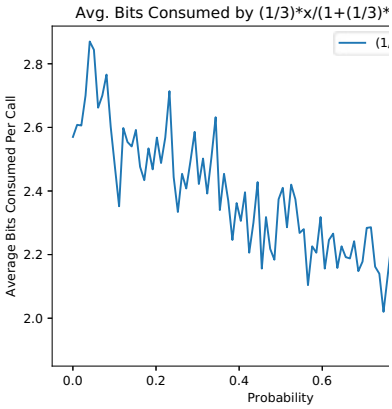
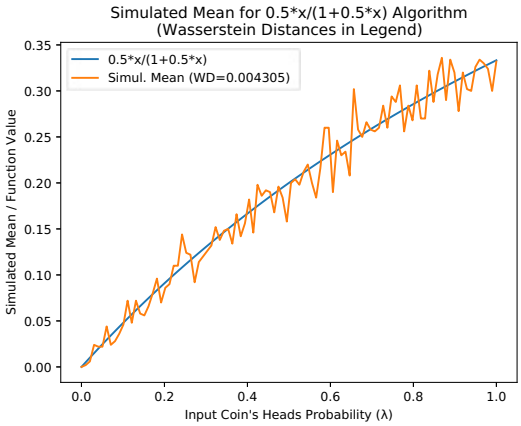
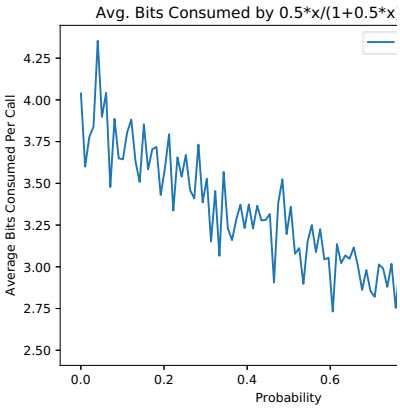
For each algorithm, if a single run was detected to use more than 5000 bits for a given λ, the entire data point for that λ was suppressed in the charts below.

In addition, for each algorithm, a chart appears showing the minimum number of input coin flips that any fast Bernoulli factory algorithm will need on average to simulate the given function, based on work by Mendo (2019)[5]. Note that some functions require a growing number of coin flips as λ approaches 0 or 1. Note that for the 2014, 2016, and 2019 algorithms—

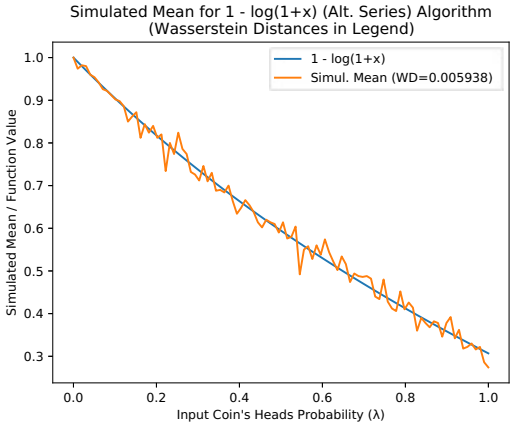
- an ϵ of $1 - (x + c) * 1.001$ was used (or 0.0001 if ϵ would be greater than 1), and
- an ϵ of $(x - c) * 0.9995$ for the subtraction variants.

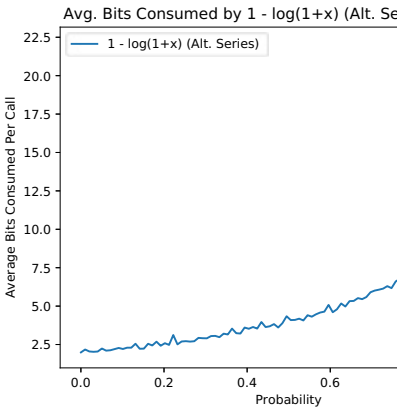
Points with invalid ϵ values were suppressed. For the low-mean algorithm, an *m* of max(0.49999, *x***c**1.02) was used unless noted otherwise.

5.1 The Charts

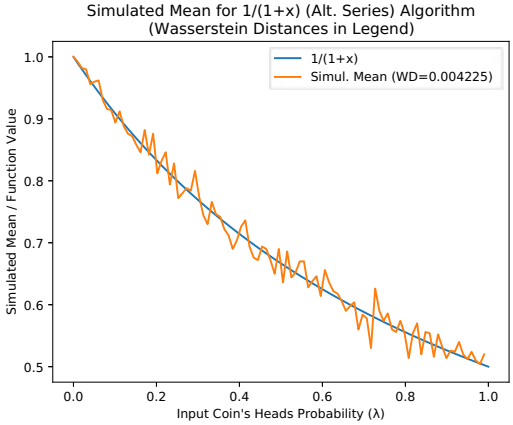
Algorithm	Simulated Mean	Average Bits Consumed
(1/3)*x/(1+(1/3)*x)		
0.5*x/(1+0.5*x)		

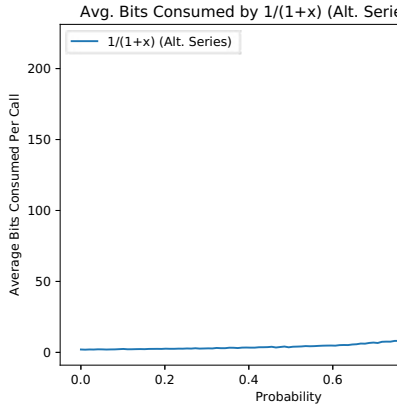
1 - log(1+x) (Alt. Series)



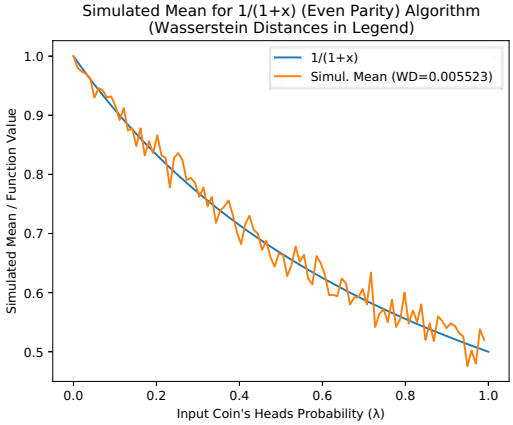


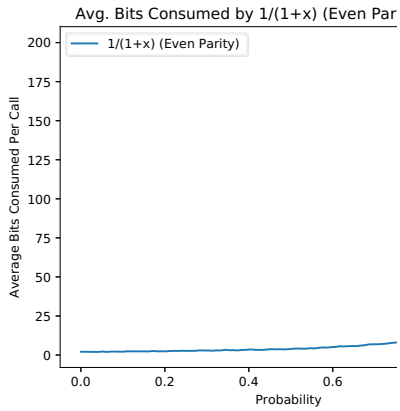
1/(1+x) (Alt. Series)



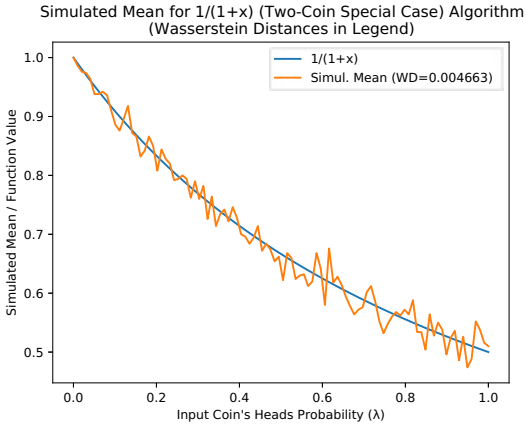


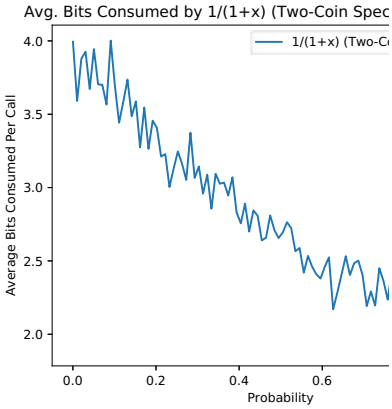
1/(1+x) (Even Parity)

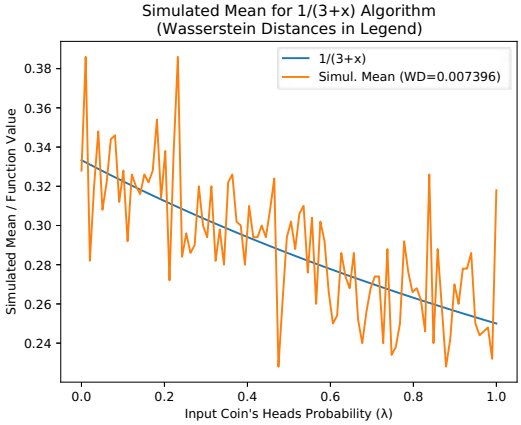


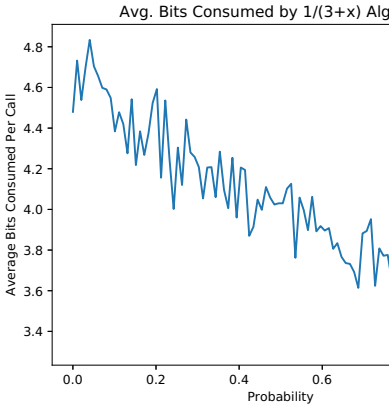


1/(1+x) (Two-Coin Special Case)

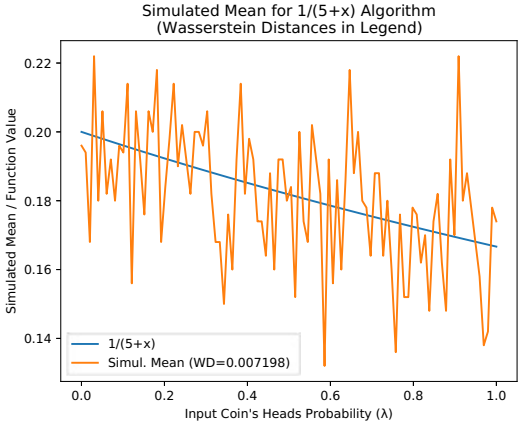

Simulated Mean for 1/(1+x) (Two-Coin Special Case) Algorithm
(Wasserstein Distances in Legend)
1/(1+x)
Simul. Mean (WD=0.004663)
Simulated Mean / Function Value
Input Coin's Heads Probability (λ)

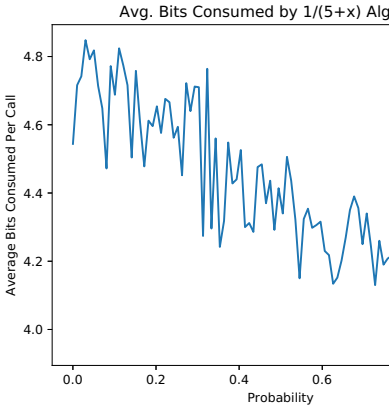

Avg. Bits Consumed by 1/(1+x) (Two-Coin Spec
1/(1+x) (Two-C
Average Bits Consumed Per Call
Probability


Simulated Mean for 1/(3+x) Algorithm
(Wasserstein Distances in Legend)
1/(3+x)
Simul. Mean (WD=0.007396)
Simulated Mean / Function Value
Input Coin's Heads Probability (λ)


Avg. Bits Consumed by 1/(3+x) Alg
Average Bits Consumed Per Call
Probability

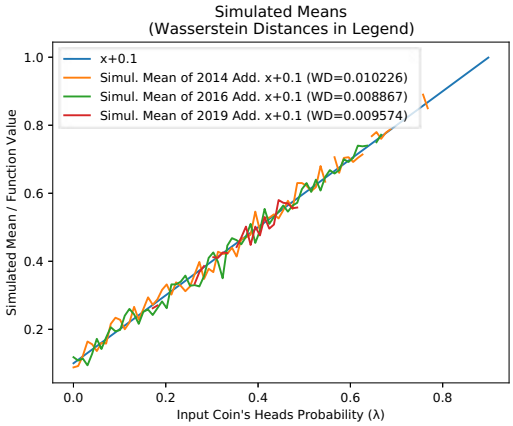
1/(3+x)

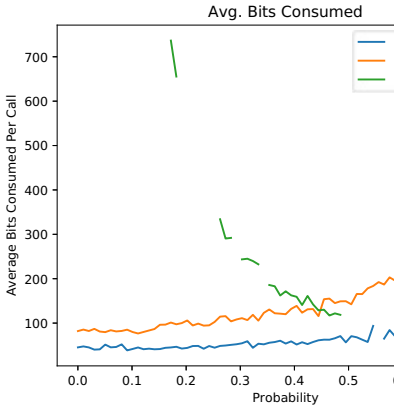

Simulated Mean for 1/(5+x) Algorithm
(Wasserstein Distances in Legend)
1/(5+x)
Simul. Mean (WD=0.007198)
Simulated Mean / Function Value
Input Coin's Heads Probability (λ)


Avg. Bits Consumed by 1/(5+x) Alg
Average Bits Consumed Per Call
Probability

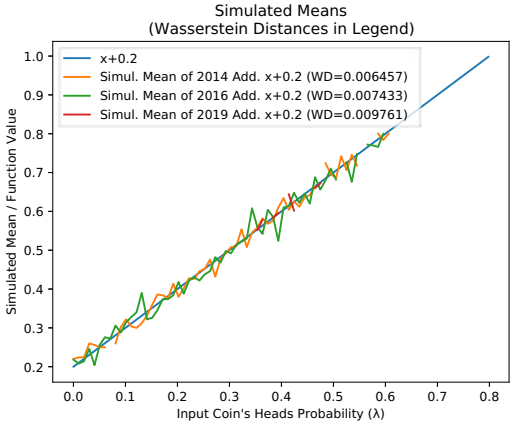
1/(5+x)

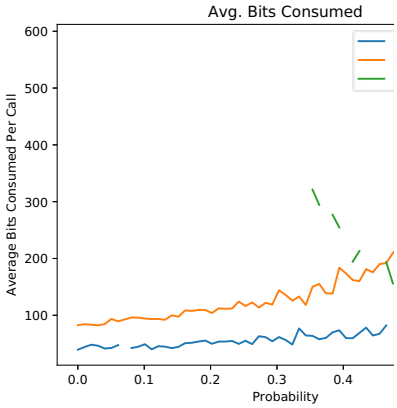
2014 Add. x+0.1



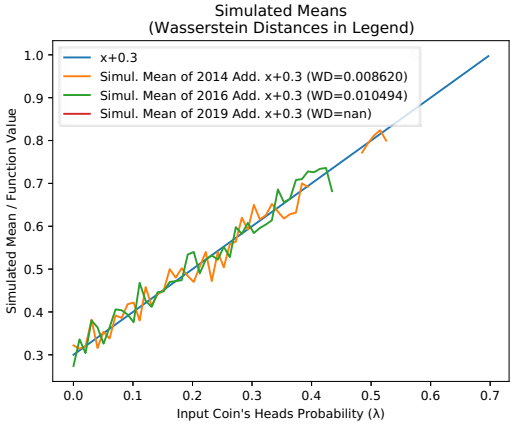


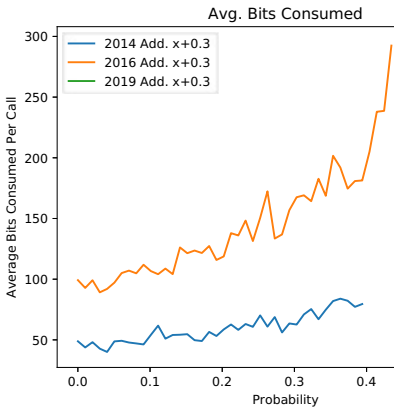
2014 Add. x+0.2



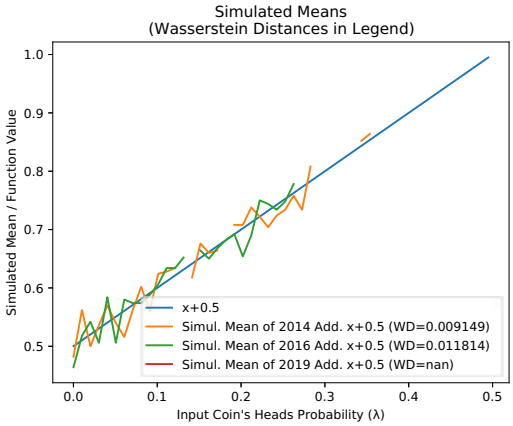


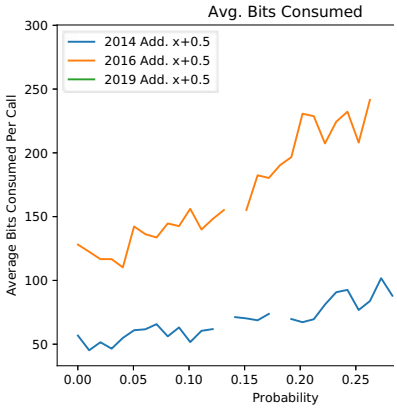
2014 Add. x+0.3



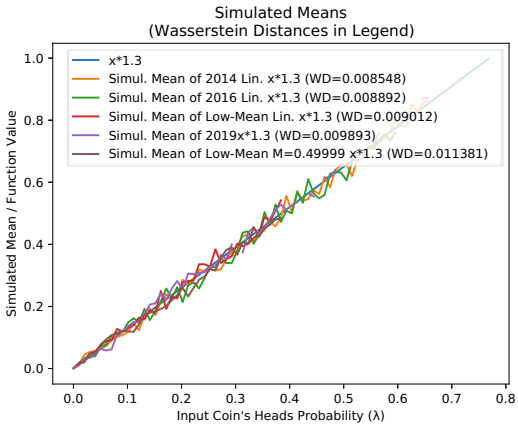


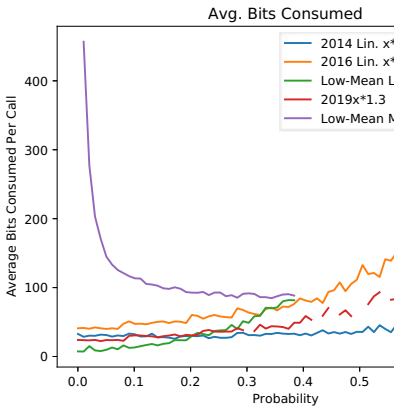
2014 Add. x+0.5



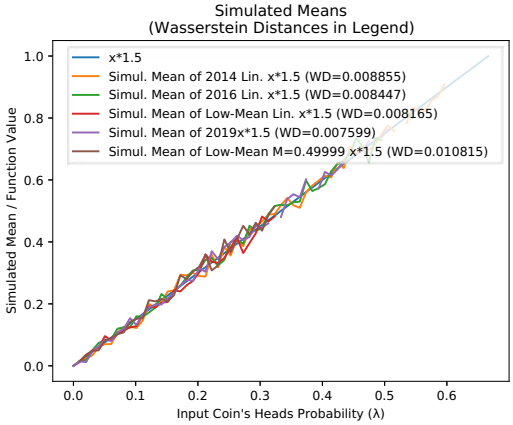


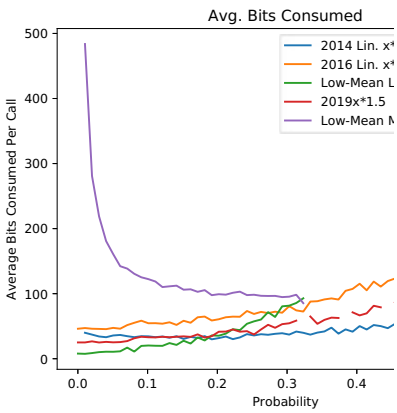
2014 Lin. x*1.3



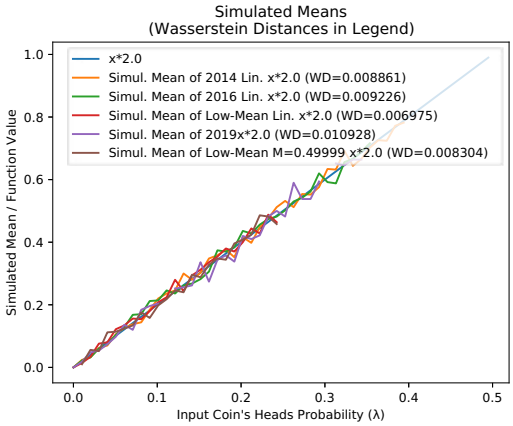


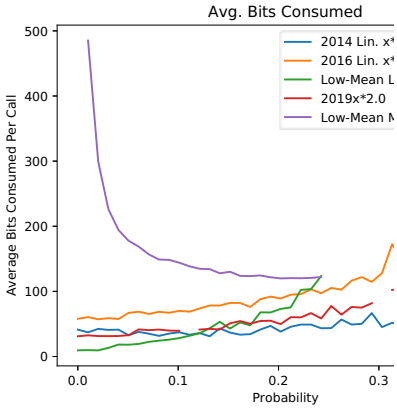
2014 Lin. x*1.5



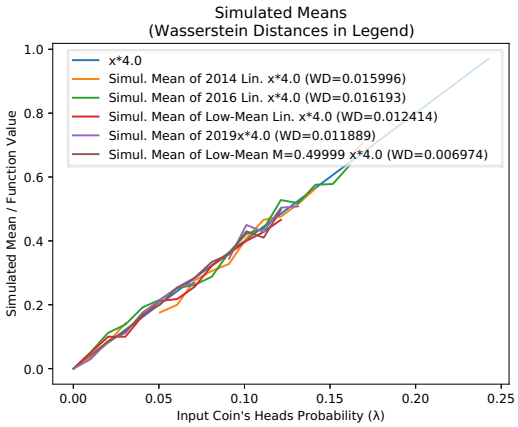


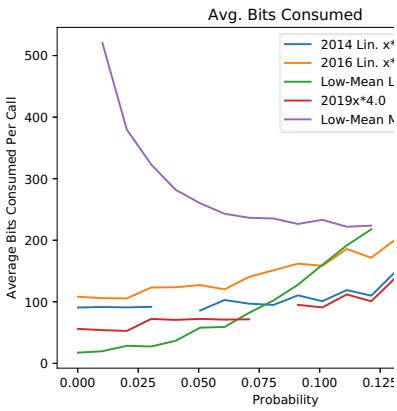
2014 Lin. x*2.0



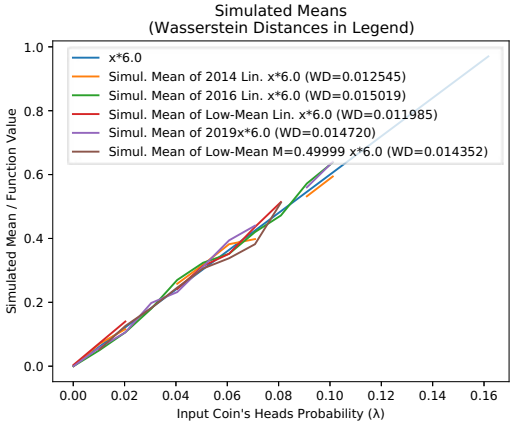


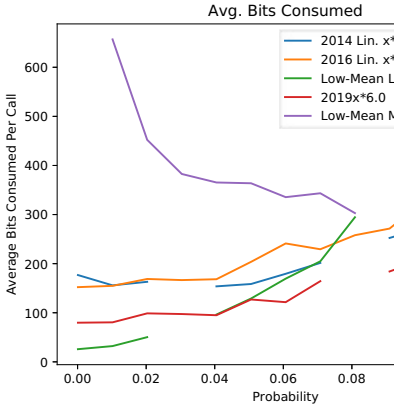
2014 Lin. x*4.0



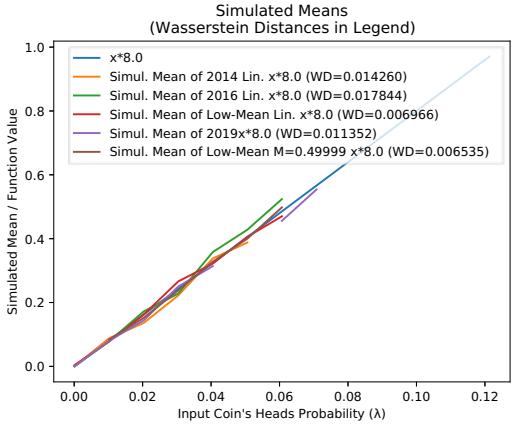


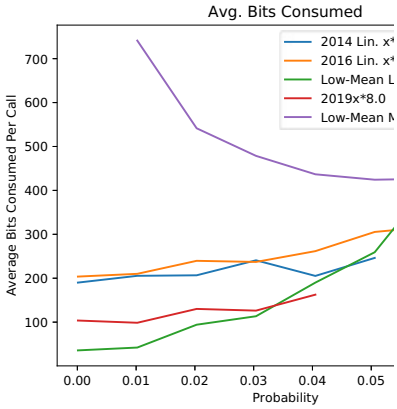
2014 Lin. x*6.0



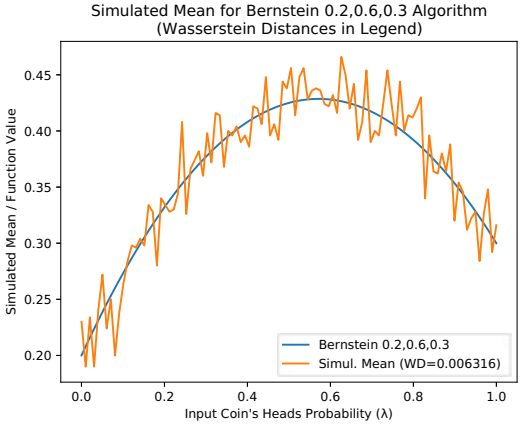


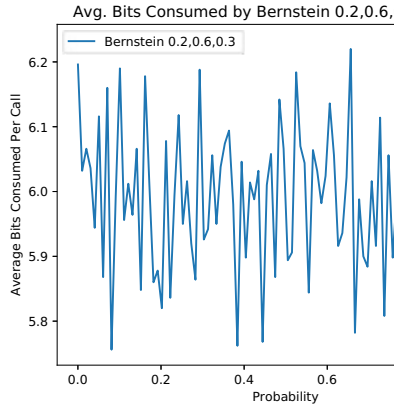
2014 Lin. x*8.0



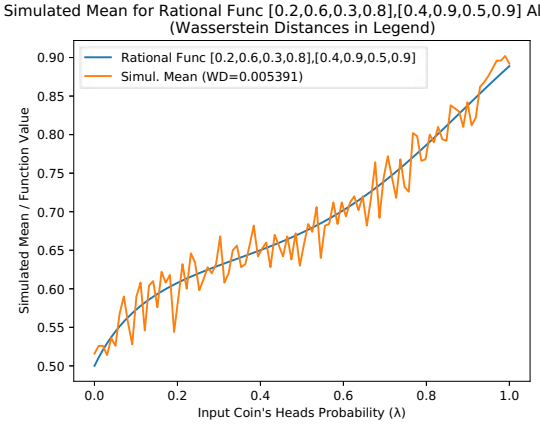


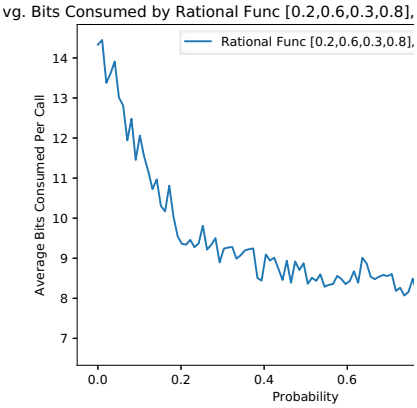
Bernstein 0.2,0.6,0.3



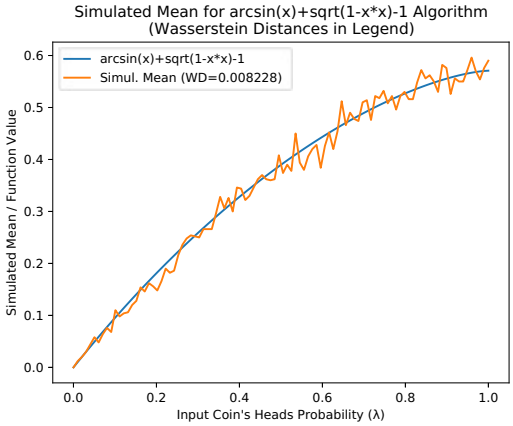


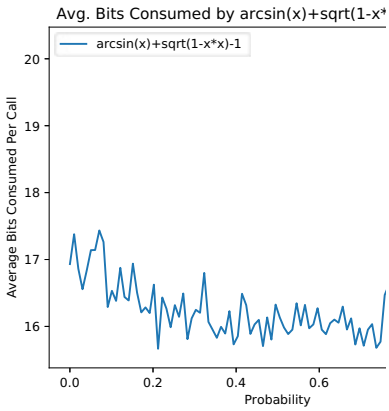
Rational Func [0.2,0.6,0.3,0.8], [0.4,0.9,0.5,0.9]



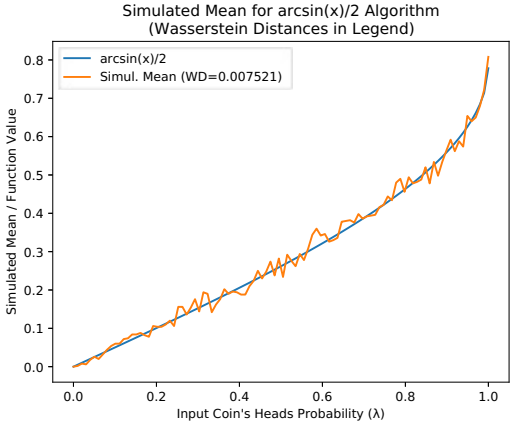


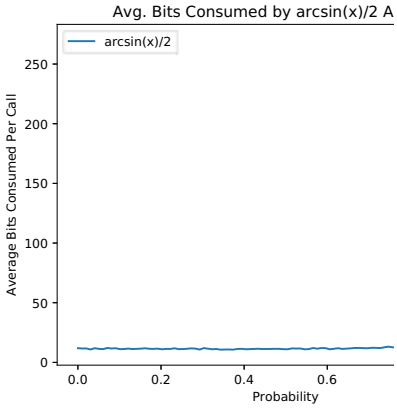
arcsin(x)+sqrt(1-x*x)-1



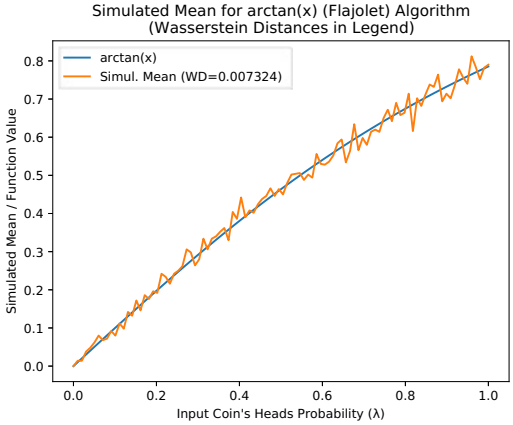


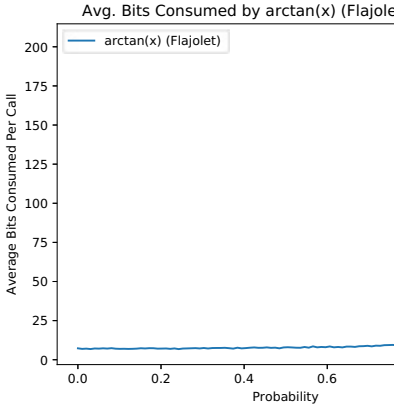
arcsin(x)/2



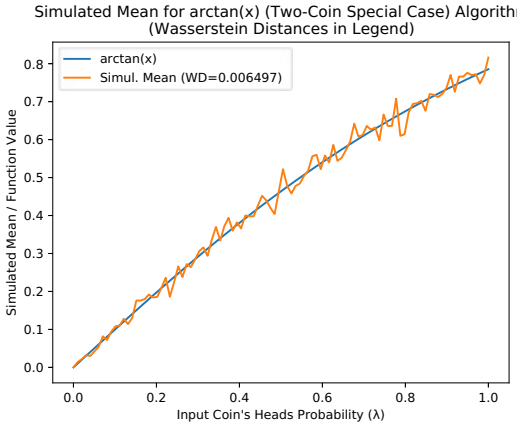


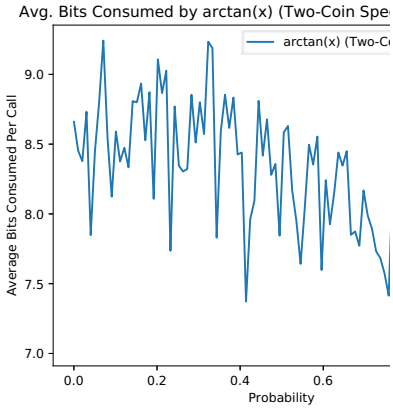
arctan(x)
(Flajolet)



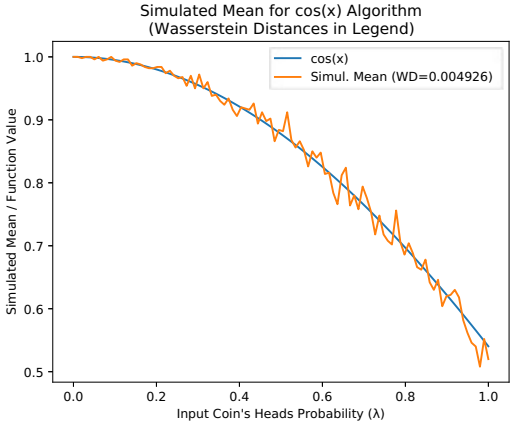


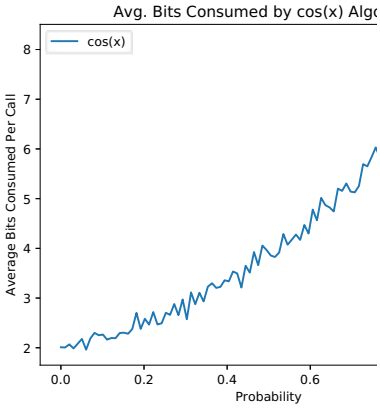
arctan(x) (Two-Coin Special Case)



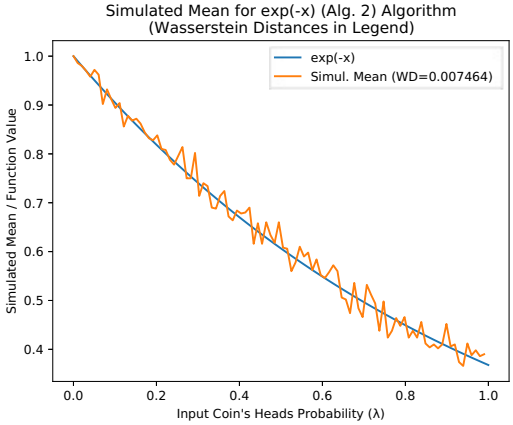


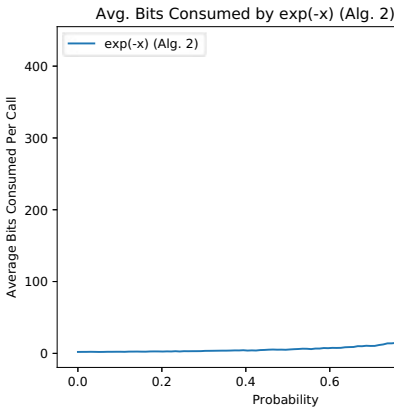
cos(x)



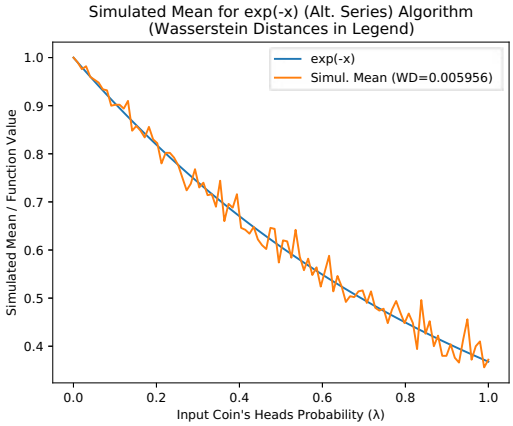


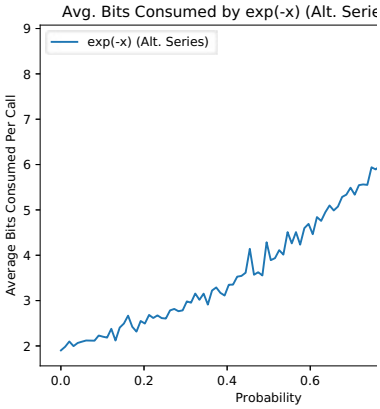
exp(-x) (Alg. 2)



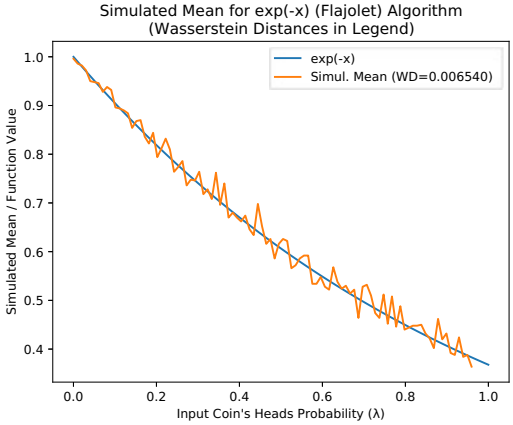


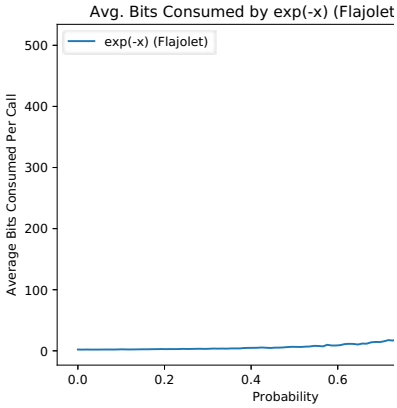
exp(-x) (Alt. Series)



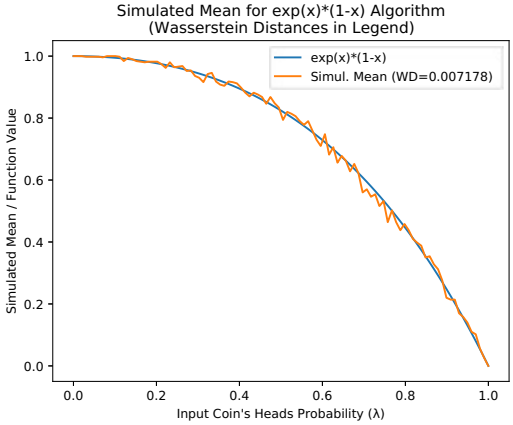


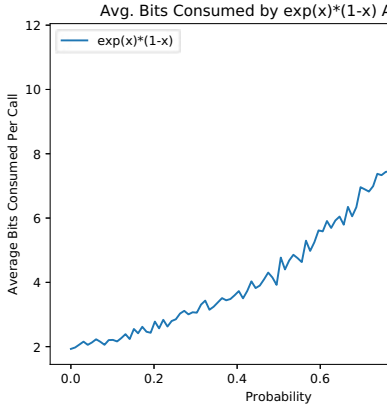
exp(-x) (Flajolet)



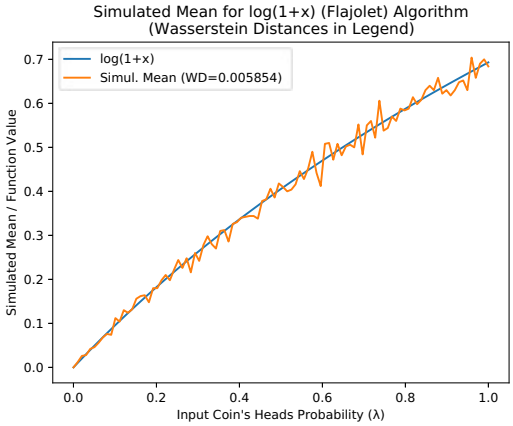


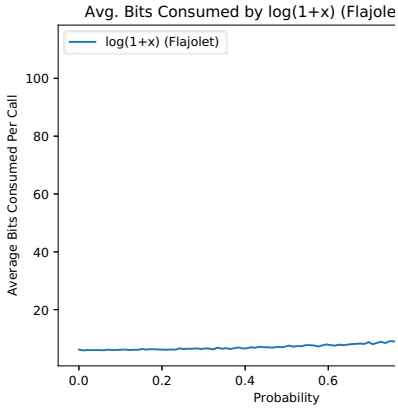
exp(x)*(1-x)



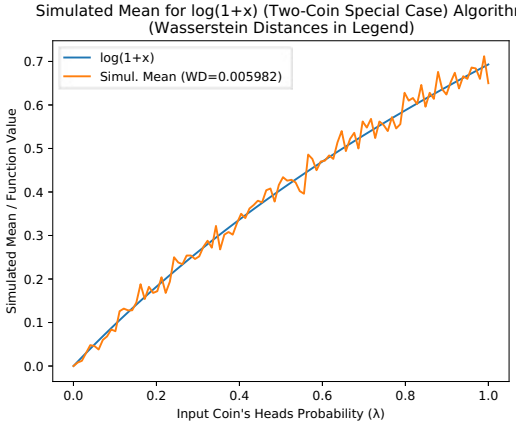


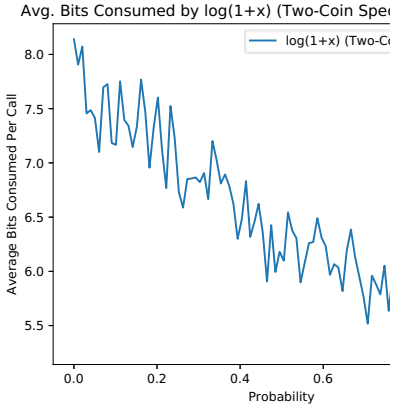
log(1+x) (Flajolet)



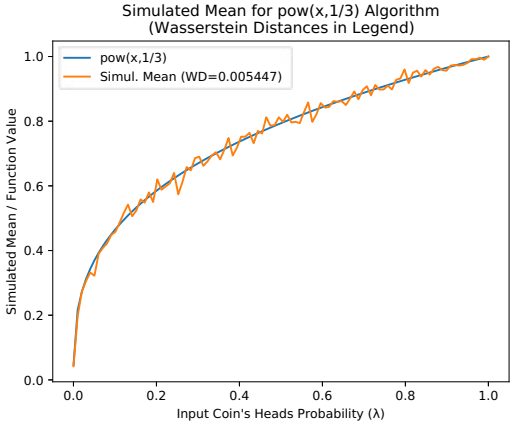


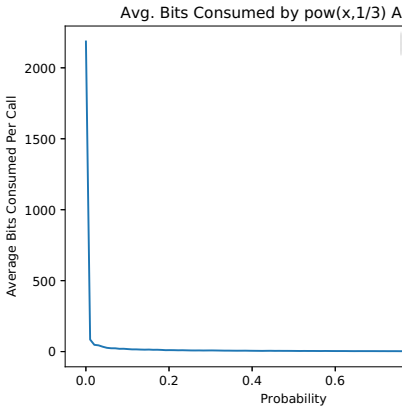
log(1+x) (Two-Coin Special Case)



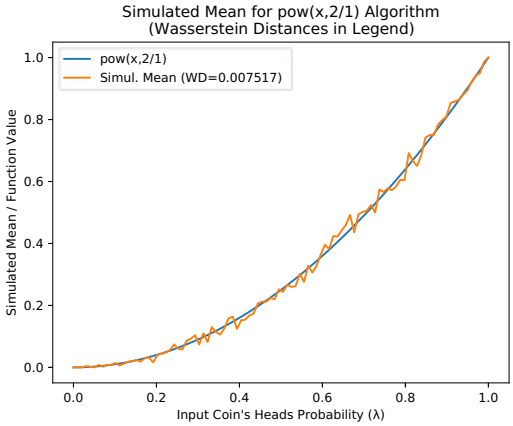


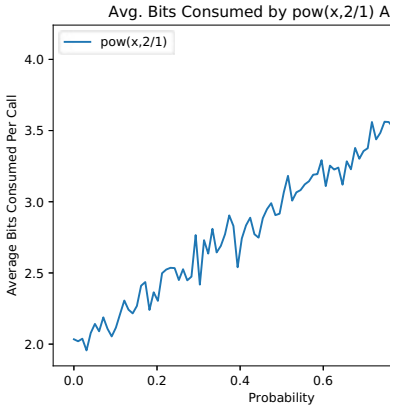
pow(x,1/3)



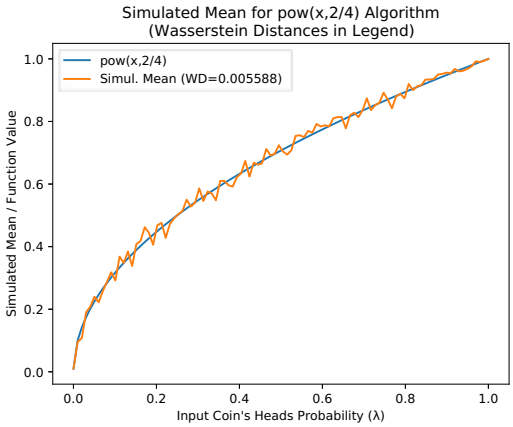


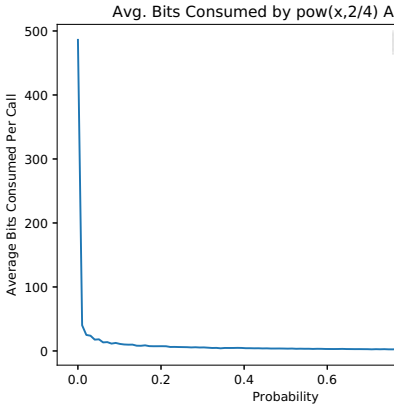
pow(x,2/1)



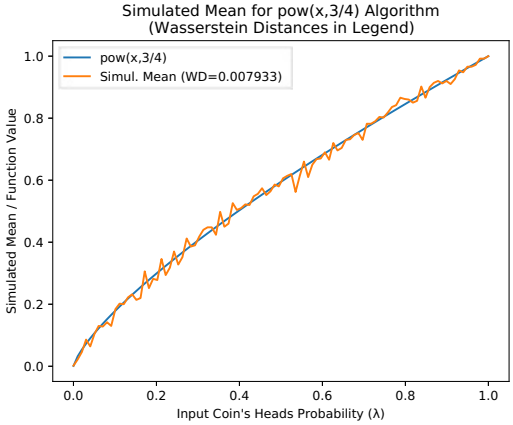


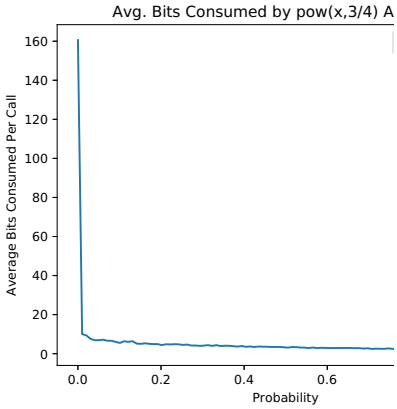
pow(x,2/4)



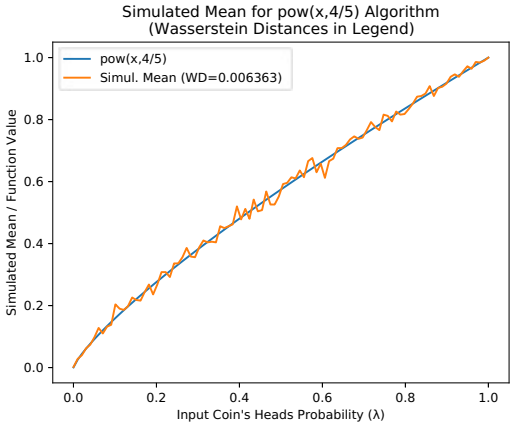


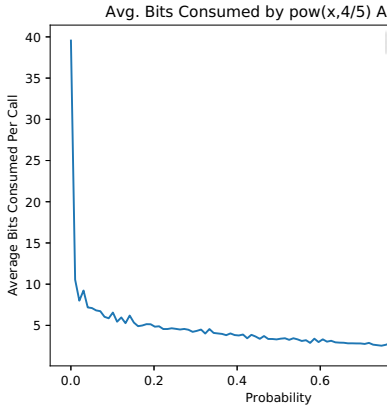
pow(x,3/4)



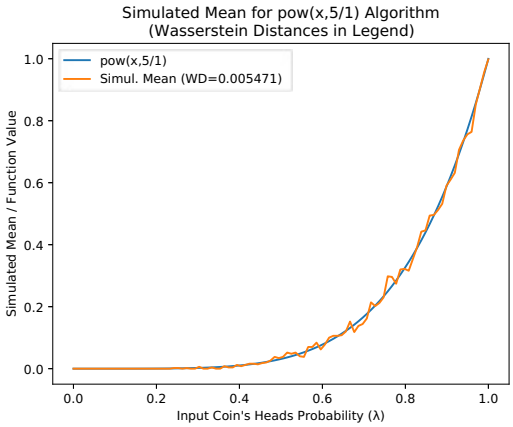


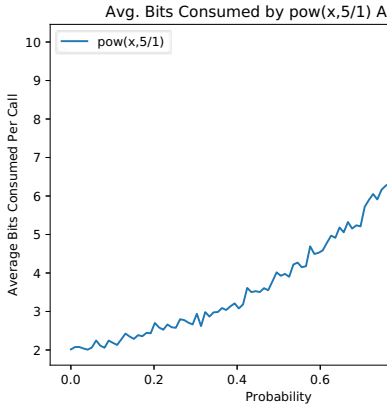
pow(x,4/5)



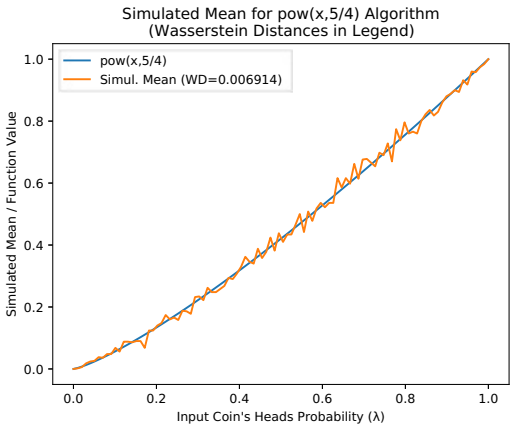


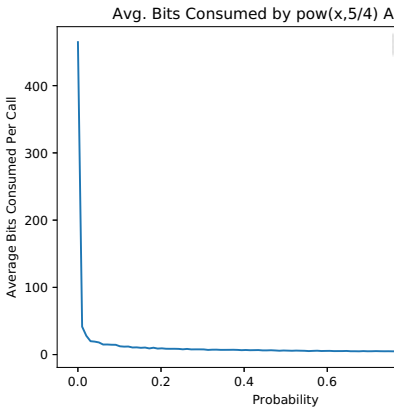
pow(x,5/1)



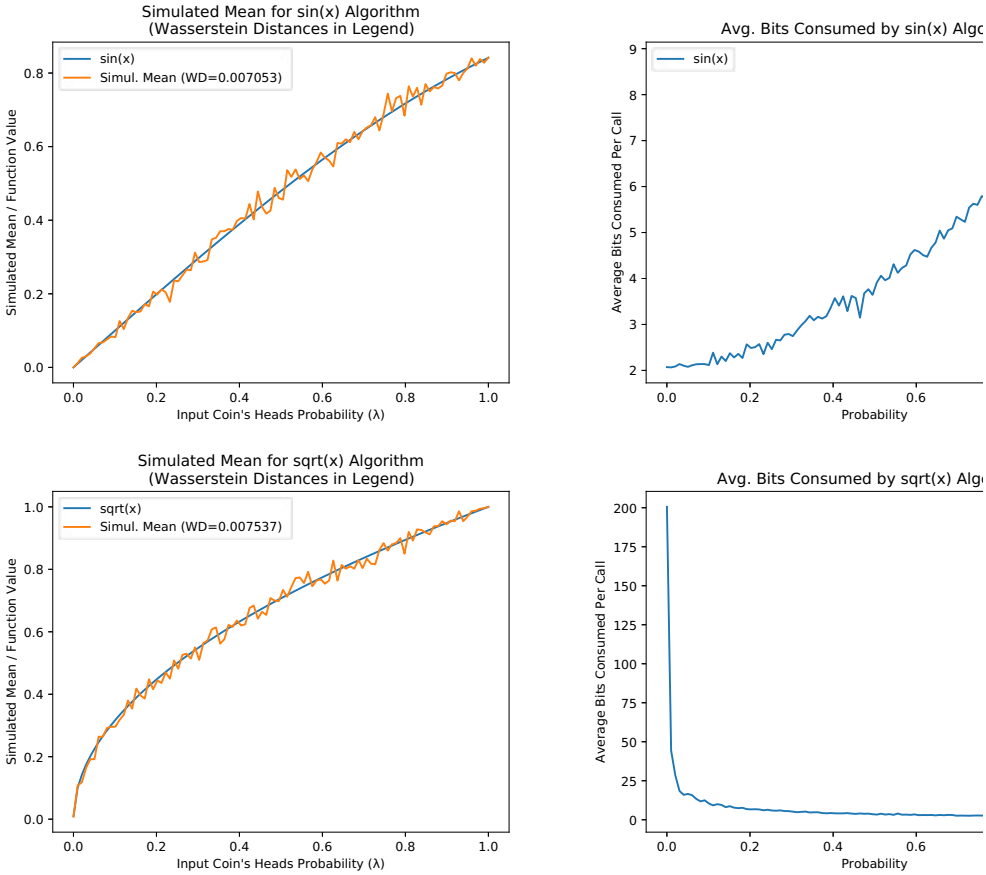


pow(x,5/4)





sin(x)



sqrt(x)

6 Notes

(1) Flajolet, P., Pelletier, M., Soria, M., "**On Buffon machines and numbers**", arXiv:0906.5560v2 [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) Huber, M., "**Nearly optimal Bernoulli factories for linear functions**", arXiv:1308.1562v2 [math.PR], 2014.

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

(5) 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.

(6) 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.

(7) As used here and in the Flajolet paper, a geometric random number is the number of successes before the first failure, where the success probability is λ.

(8) The Flajolet paper describes what it calls the *von Neumann schema*, which, given a permutation class and an input coin, generates a random non-negative integer n with probability equal to (λ^n * V(n) / n!) / EGF(λ), where EGF(λ) = $\Sigma_{k\,=\,0,\,1,\,\ldots}$ (λ^k * V(k) / k!), and V(n) is the number of *valid* permutations of size n. Here, EGF(λ) is the *exponential generating function*. Effectively, a geometric(λ) random number G (see previous note) is accepted with probability V(G)/G! (where G! is the number of *possible* permutations of size G, or 1 if G is 0), and rejected otherwise. The probability that r geometric random numbers are rejected this way is p*(1 − p)^r, where p = (1 − λ) * EGF(λ).

(9) Devroye, L., ***Non-Uniform Random Variate Generation***, 1986.

(10) 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).

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

(12) Another algorithm for this function uses the general martingale algorithm, but uses more bits on average as λ approaches 1. Here, the alternating series is `1 - x + x^2/2 - x^3/3 + ...`, whose coefficients are 1, 1, 1/2, 1/3, ...

(13) Vats, D., Gonçalves, F. B., Łatuszyński, K. G., Roberts, G. O. Efficient Bernoulli factory MCMC for intractable likelihoods, arXiv:2004.07471v1 [stat.CO], 2020.

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

(15) Morina, G., Łatuszyński, K., et al., "**From the Bernoulli Factory to a Dice Enterprise via Perfect Sampling of Markov Chains**", arXiv:1912.09229v1 [math.PR], 2019.

[16] One of the only implementations I could find of this, if not the only, was a **Haskell implementation**.

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

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

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

[20] Smith, N. A. and Johnson, M. (2007). Weighted and probabilistic context-free grammars are equally expressive. Computational Linguistics, 33(4):477–491.

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

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

[23] 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.

[24] Wästlund, J., "**Functions arising by coin flipping**", 1999.

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

[26] Bill Gosper, "Continued Fraction Arithmetic", 1978.

[27] Borwein, J.M., Calkin, N.J., et al., "Continued logarithms and associated continued fractions", 2016.

[28] Canonne, C., Kamath, G., Steinke, T., "**The Discrete Gaussian for Differential Privacy**", arXiv:2004.00010v2 [cs.DS], 2020.

[29] von Neumann, J., "Various techniques used in connection with random digits", 1951.

[30] Pae, S., "Random number generation using a biased source", dissertation, University of Illinois at Urbana-Champaign, 2015.

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

[32] Devroye, L., Gravel, C., "**Sampling with arbitrary precision**", arXiv:1502.02539v5 [cs.IT], 2015.

7 Appendix

7.1 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)**(5)**. Instead of generating outside randomness, a randomized algorithm can implement a *randomness extraction* procedure (such as the von Neumann algorithm) 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 coin returns 0 then 1 or 1 then 0 this way, then taking the result as 0 or 1, respectively (von Neumann 1951)**(29)**.

In fact, there is a well-known lower bound on the average number of coin flips needed to turn a coin with one bias (λ) into a coin with another bias (τ = *f*(λ)). It's called the *entropy bound* (see, e.g., (Pae 2005)**(30)**, (Peres 1992)**(31)**) and is calculated as—

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

For example, if *f*(λ) 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 bias is λ). Note that this formula only works if nothing but coin flips is allowed as randomness.

7.2 Simulating Probabilities vs. Estimating Probabilities

A Bernoulli factory or another algorithm that produces heads with a given probability is the same as building an unbiased estimator for that probability (Łatuszyński et al. 2009/2011)**(6)**. (In this note, an *unbiased probability estimator* is an unbiased estimator whose estimates are in [0, 1] almost surely.) As a result—

1. estimating the input coin's probability of heads (λ) in some way (such as by flipping the coin many times and averaging the results),
2. calculating $v = f(\lambda)$,
3. generating a uniform random number in [0,1], call it u, and
4. returning 1 if u is less than v, or 0 otherwise,

will simulate the probability *f*(λ) in theory. In practice, however, this method is prone to numerous errors, including errors in the estimate in step 1, and rounding and approximation errors in steps 2 and 3. For this reason and also because "exact sampling" is the focus of this page, this document does not cover algorithms that directly estimate λ (such as in step 1). As (Mossel and Peres 2005)**(19)** says: "The difficulty here is that [λ] is unknown. It is easy to estimate [λ], and therefore [*f*(λ)]. However, to get a coin with an exact bias [*f*(λ)] is harder", and that is what Bernoulli factory algorithms are designed to do.

As also shown in (Łatuszyński et al. 2009/2011)**(6)**, however, if *f*(λ) can't serve as a factory function, no unbiased probability estimator of that function is possible, since sampling it isn't possible. For example, function A can't serve as a factory function, so no simulator (or unbiased probability estimator) for that function is possible. This is possible for function B, however (Keane and O'Brien 1994)**(2)**.

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

7.3 Convergence of Bernoulli Factories

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

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

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

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

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

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

- The average number of coin flips needed by this algorithm will grow without bound as x approaches 0, and Mendo (2019)**(5)** showed that this is a lower bound; that is, no Bernoulli factory algorithm can do much better without knowing more information on x.
- $x^{y/z}$ has a slope that tends to a vertical slope near 0, so that the so-called ***Lipschitz condition*** is not met at 0. And (Nacu and Peres 2005, propositions 10 and 23)**(4)** showed that the Lipschitz condition is necessary for a Bernoulli factory to have an upper bound on the average running time.

Thus, a practical implementation of this algorithm may have to switch to an alternative implementation (such as the one described in the next section) when it detects that the geometric bag's first few digits are zeros.

7.4 Alternative Implementation of Bernoulli Factories

Say we have a Bernoulli factory algorithm that takes a coin with probability of heads of p and outputs 1 with probability $f(p)$. If this algorithm takes a geometric bag (a partially-sampled uniform random number or PSRN) as the input coin and flips that coin using **SampleGeometricBag**, the algorithm could instead be implemented as follows in order to return 1 with probability $f(U)$, where U is the number represented by the geometric bag (see also (Brassard et al., 2019)**(25)**, (Devroye 1986, p. 769)**(9)**, (Devroye and Gravel 2015)**(32)**:

1. Set v to 0 and k to 1.
2. Set v to $b * v + d$, where b is the base (or radix) of the geometric bag's digits, and d is a digit chosen uniformly at random.
3. Calculate an approximation of $f(U)$ as follows:
    1. Set n to the number of items (sampled and unsampled digits) in the geometric bag.
    2. Of the first n items in the geometric bag, sample each of the unsampled digits uniformly at random. Then let uk be the geometric bag's digit expansion up to the first n digits after the point.
    3. Calculate the lowest and highest values of f in the interval $[uk, uk + b^{-n}]$, call them $fmin$ and $fmax$. If $\text{abs}(fmin - fmax) <= 2 * b^{-k}$, calculate $(fmax + fmin) / 2$ as the approximation. Otherwise, add 1 to n and go to the previous substep.
4. Let pk be the approximation's digit expansion up to the k digits after the point. For example, if $f(U)$ is π and k is 2, pk is 314.
5. If $pk + 1 <= v$, return 0. If $pk - 2 >= v$, return 1. If neither is the case, add 1 to k and go to step 2.

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

7.5 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)**(17)**. 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.
- Second, we show the algorithm is locally correct when the recursive call in step 3 is replaced with an oracle that simulates the correct "continued sub-logarithm". If step 1 reaches the last coefficient, the algorithm obviously passes with the correct probability. Otherwise, we will be simulating the probability $(1 / 2^{c[i]}) / (1 + x)$, where x is the "continued sub-logarithm" and will be at most 1 by construction. Steps 2 and 3 define a loop that divides the probability space into three pieces: the first piece takes up one half, the second piece (step 3) takes up a portion of the other half (which here is equal to x/2), and the last piece is the "rejection piece" that reruns the loop. Since this loop changes no variables that affect later iterations, each iteration acts like an acceptance/rejection algorithm already proved to be a perfect simulator by Huber. The algorithm will pass at step 2 with probability $p = (1 / 2^{c[i]}) / 2$ and fail either at step 2 with probability $f1 = (1 - 1 / 2^{c[i]}) / 2$, or at step 3 with probability $f2 = x/2$ (all these probabilities are relative to the whole iteration). Finally, dividing the passes by the sum of passes and fails $(p / (p + f1 + f2))$ leads to $(1 / 2^{c[i]}) / (1 + x)$, which is the probability we wanted.

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

7.6 Correctness Proof for Continued Fraction Simulation Algorithm 3

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

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

- The algorithm halts almost surely for the same reason as the similar continued logarithm simulator.
- If the call in step 3 is replaced with an oracle that simulates the correct "sub-fraction", the algorithm is locally correct. If step 1 reaches the last element of the continued fraction, the algorithm obviously passes with the correct probability. Otherwise, we will be simulating the probability $b[i] / (a[i] + x)$, where x is the "continued sub-fraction" and will be at most 1 by assumption. Steps 2 and 3 define a loop that

divides the probability space into three pieces: the first piece takes up a part equal to $h = a[i]/(a[i] + 1)$, the second piece (step 3) takes up a portion of the remainder (which here is equal to $x * (1 - h)$), and the last piece is the "rejection piece". The algorithm will pass at step 2 with probability $p = (b[i] / a[pos]) * h$ and fail either at step 2 with probability $f1 = (1 - b[i] / a[pos]) * h$, or at step 3 with probability $f2 = x * (1 - h)$ (all these probabilities are relative to the whole iteration). Finally, dividing the passes by the sum of passes and fails leads to $b[i] / (a[i] + x)$, which is the probability we wanted, so that both of Huber's conditions are satisfied and we are done. □

8 License