More Algorithms for Arbitrary-Precision Sampling

This version of the document is dated 2023-03-10.

Peter Occil

Abstract: This page contains additional algorithms for arbitrary-precision sampling of distributions, Bernoulli factory algorithms (biased-coin to biased-coin algorithms), and algorithms to produce heads with an irrational probability. They supplement my pages on Bernoulli factory algorithms and partially-sampled random numbers.

2020 Mathematics Subject Classification: 68W20, 60-08, 60-04.

1 Introduction

This page contains additional algorithms for arbitrary-precision sampling of distributions, Bernoulli factory algorithms (biased-coin to biased-coin algorithms), and algorithms to produce heads with an irrational probability. These samplers are designed to not rely on floating-point arithmetic.

The samplers on this page may depend on algorithms given in the following pages:

- **Partially-Sampled Random Numbers for Accurate Sampling of Continuous Distributions**
- **Bernoulli Factory Algorithms**

Additional Bernoulli factory algorithms and irrational probability samplers are included here rather than in "**Bernoulli Factory Algorithms**" because that article is quite long as it is.

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.

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

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

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

Comments on other aspects of this document are welcome.

2 Contents

3 Bernoulli Factories

As a reminder, the *Bernoulli factory problem* is: We're given a coin that shows heads with an unknown probability, *λ*, and the goal is to use that coin (and possibly also a fair coin) to build a "new" coin that shows heads with a probability that depends on *λ*, call it *f*(*λ*). *f* is a Bernoulli factory function (or factory function) if this problem can be solved for that function.

This section contains additional algorithms to solve the Bernoulli factory problem for certain kinds of functions. Such algorithms could be placed in "**Bernoulli Factory Algorithms**", but, since that article is quite long as it is, they are included here instead.

In the methods below, *λ* is the unknown probability of heads of the coin involved in the Bernoulli factory problem.

3.1 Certain Piecewise Linear Functions

Let *f*(*λ*) be a function of the form min(*λ***mult*, 1−*ε*). This is a *piecewise linear function*, a function made up of two linear pieces (in this case, the pieces are a rising linear part and a constant part).

This section describes how to calculate the Bernstein coefficients for polynomials that converge from above and below to *f*, based on Thomas and Blanchet (2012)[1]. These polynomials can then be used to show heads with probability $f(\lambda)$ using the algorithms given in "**General Factory Functions**".

In this section, **fbelow(*n*, *k*)** and **fabove(*n*, *k*)** are the k^{th} coefficients (with k starting at 0) of the lower and upper polynomials, respectively, in Bernstein form of degree n.

The code at the end of this section uses the computer algebra library SymPy to calculate a list of parameters for a sequence of polynomials converging from above. The method to do so is called `calc_linear_func(eps, mult, count)`, where `eps` is ε, `mult` = *mult*, and `count` is the number of polynomials to generate. Each item returned by `calc_linear_func` is a list of two items: the degree of the polynomial, and a *Y parameter*. The procedure to calculate the required polynomials is then logically as follows (as written, it runs very slowly, though):

1. Set i to 1.
2. Run `calc_linear_func(eps, mult, i)` and get the degree and *Y parameter* for the last listed item, call them n and y, respectively.
3. Set x to $-((y-(1-\varepsilon))/\varepsilon)^5/mult + y/mult$. (This exact formula doesn't appear in the Thomas and Blanchet paper; rather it comes from the **supplemental source code** uploaded by A. C. Thomas at my request.)
4. For degree n, **fbelow(*n*, *k*)** is $\min((k/n)*mult, 1-\varepsilon)$, and **fabove(*n*, *k*)** is $\min((k/n)*y/x,y)$. (**fbelow** matches f because f is *concave* on the interval [0, 1], which roughly means that its rate of growth there never goes up.)
5. Add 1 to i and go to step 2.

It would be interesting to find general formulas to find the appropriate polynomials (degrees and *Y parameters*) given only the values for *mult* and ε, rather than find them "the hard way" via `calc_linear_func`. For this procedure, the degrees and *Y parameters* can be upper bounds, as long as the sequence of degrees is strictly increasing and the sequence of Y parameters is nowhere increasing.

> **Note:** In Nacu and Peres (2005)[2], the following polynomial sequences were suggested to simulate $\min(2\lambda, 1-2\varepsilon)$, provided $\varepsilon \lt 1/8$, where *n* is a power

> of 2. However, with these sequences, an extraordinary number of input coin flips is required to simulate this function each time.
>
> - **fbelow(*n*, *k*)** = $\min(2(k/n), 1-2\varepsilon)$.
> - **fabove(*n*, *k*)** = $\min(2(k/n), 1-2\varepsilon)+$ $ \frac{2\times\max(0, k/n+3\varepsilon - 1/2)}{\varepsilon(2-\sqrt{2})} \sqrt{2/n}+$ $\frac{72\times\max(0,k/n-1/9)}{1-\exp(-2\times\varepsilon^2)} \exp(-2n\times\varepsilon^2)$.

SymPy code for piecewise linear functions:

```
def bernstein_n(func, x, n, pt=None):
  # Bernstein operator.
  # Create a polynomial that approximates func, which in turn uses
  # the symbol x.  The polynomial's degree is n and is evaluated
  # at the point pt (or at x if not given).
  if pt==None: pt=x
  ret=0
  v=[binomial(n,j) for j in range(n//2+1)]
  for i in range(0, n+1):
    oldret=ret
    bino=v[i] if i<len(v) else v[n-i]
    ret+=func.subs(x,S(i)/n)*bino*pt**i*(1-pt)**(n-i)
    if pt!=x and ret==oldret and ret>0: break
  return ret

def inflec(y,eps=S(2)/10,mult=2):
  # Calculate the inflection point (x) given y, eps, and mult.
  # The formula is not found in the paper by Thomas and
  # Blanchet 2012, but in
  # the supplemental source code uploaded by
  # A.C. Thomas.
  po=5 # Degree of y-to-x polynomial curve
  eps=S(eps)
  mult=S(mult)
  x=-((y-(1-eps))/eps)**po/mult + y/mult
  return x

def xfunc(y,sym,eps=S(2)/10,mult=2):
  # Calculate Bernstein "control polygon" given y,
  # eps, and mult.
  return Min(sym*y/inflec(y,eps,mult),y)
```

```
def calc_linear_func(eps=S(5)/10, mult=1, count=10):
   # Calculates the degrees and Y parameters
   # of a sequence of polynomials that converge
   # from above to min(x*mult, 1-eps).
   # eps must be greater than 0 and less than 1.
   # Default is 10 polynomials.
   polys=[]
   eps=S(eps)
   mult=S(mult)
   count=S(count)
   bs=20
   ypt=1-(eps/4)
   x=symbols('x')
   tfunc=Min(x*mult,1-eps)
   tfn=tfunc.subs(x,(1-eps)/mult).n()
   xpt=xfunc(ypt,x,eps=eps,mult=mult)
   bits=5
   i=0
   lastbxn = 1
   diffs=[]
   while i<count:
     bx=bernstein_n(xpt,x,bits,(1-eps)/mult)
     bxn=bx.n()
     if bxn > tfn and bxn < lastbxn:
       # Dominates target function
       #if oldbx!=None:
       #   diffs.append(bx)
       #   diffs.append(oldbx-bx)
       #oldbx=bx
       oldxpt=xpt
       lastbxn = bxn
       polys.append([bits,ypt])
       print("    [%d,%s]," % (bits,ypt))
       # Find y2 such that y2 < ypt and
       # bernstein_n(oldxpt,x,bits,inflec(y2, ...)) >= y2,
       # so that next Bernstein expansion will go
       # underneath the previous one
       while True:
         ypt-=(ypt-(1-eps))/4
         xpt=inflec(ypt,eps=eps,mult=mult).n()
         bxs=bernstein_n(oldxpt,x,bits,xpt).n()
         if bxs>=ypt.n():
            break
       xpt=xfunc(ypt,x,eps=eps,mult=mult)
       bits+=20
       i+=1
     else:
       bits=int(bits*200/100)
   return polys

calc_linear_func(count=8)
```

3.2 Pushdown Automata for Square-Root-Like Functions

In this section, ${n \choose m}$ = choose(n, m) is a binomial coefficient.

The following algorithm extends the square-root construction of Flajolet et al. (2010)[3], takes an input coin with probability of heads λ (where $0 \le \lambda < 1$), and returns 1 with probability—

$$f(\lambda)=\frac{1-\lambda}{\sqrt{1+4\lambda\mathtt{Coin}(\lambda)(\mathtt{Coin}(\lambda)-1)}} = (1-\lambda)\sum_{n\ge 0} \lambda^n (\mathtt{Coin}(\lambda))^n (1-\mathtt{Coin}(\lambda))^n {2n \choose n}$$ $$= (1-\lambda)\sum_{n\ge 0} (\lambda \mathtt{Coin}(\lambda) (1-\mathtt{Coin}(\lambda)))^n {2n \choose n}$$ $$= \sum_{n\ge 0} (1-\lambda) \lambda^n h_n(\lambda) = \sum_{n\ge 0} g(n, \lambda) h_n(\lambda),$$

and 0 otherwise, where:

- `Coin`(λ) is a Bernoulli factory function. If `Coin` is a rational function (a ratio of two polynomials) whose coefficients are rational numbers, then *f* is an *algebraic function* (a function that can be a solution of a nonzero polynomial equation) and can be simulated by a *pushdown automaton*, or a state machine with a stack (see the algorithm below and the note that follows it). But this algorithm will still work even if `Coin` is not a rational function. In the original square-root construction, `Coin`(λ) = 1/2.
- $g(n, \lambda) = (1-\lambda) \lambda^n$; this is the probability of running the `Coin` Bernoulli factory $2 \times n$ times.
- $h_n(\lambda) = (\mathtt{Coin}(\lambda))^n (1-\mathtt{Coin}

(\lambda))^n {2n \choose n}$; this is the probability of getting as many ones as zeros from the `Coin` Bernoulli factory.

Equivalently— $$f(\lambda)=(1-\lambda) OGF(\lambda \mathtt{Coin}(\lambda) (1-\mathtt{Coin}(\lambda))),$$ where $OGF(x) = \sum_{n\ge 0} x^n {2n \choose n}$ is the algorithm's ordinary generating function (also known as counting generating function).

The algorithm follows.

1. Set *d* to 0.
2. Do the following process repeatedly until this run of the algorithm returns a value:
    1. Flip the input coin. If it returns 1, go to the next substep. Otherwise, return either 1 if *d* is 0, or 0 otherwise.
    2. Run a Bernoulli factory algorithm for `Coin`(λ). If the run returns 1, add 1 to *d*. Otherwise, subtract 1 from *d*.
    3. Repeat the previous substep.

> **Note:** A *pushdown automaton* is a state machine that keeps a stack of symbols. In this document, the input for this automaton is a stream of flips of a coin that shows heads with probability λ, and the output is 0 or 1 depending on which state the automaton ends up in when it empties the stack (Mossel and Peres 2005)[4]. That paper shows that a pushdown automaton, as defined here, can simulate only *algebraic functions*, that is, functions that can be a solution of a nonzero polynomial equation. The **appendix** defines these machines in more detail and has proofs on which algebraic functions are possible with pushdown automata.
>
> As a pushdown automaton, this algorithm (except the "Repeat the previous substep" part) can be expressed as follows. Let the stack have the single symbol EMPTY, and start at the state POS-S1. Based on the current state, the last coin flip (HEADS or TAILS), and the symbol on the top of the stack, set the new state and replace the top stack symbol with zero, one, or two symbols. These *transition rules* can be written as follows:
>
> - (POS-S1, HEADS, *topsymbol*) → (POS-S2, {*topsymbol*}) (set state to POS-S2, keep *topsymbol* on the stack).
> - (NEG-S1, HEADS, *topsymbol*) → (NEG-S2, {*topsymbol*}).
> - (POS-S1, TAILS, EMPTY) → (ONE, {}) (set state to ONE, pop the top symbol from the stack).
> - (NEG-S1, TAILS, EMPTY) → (ONE, {}).

> - (POS-S1, TAILS, X) → (ZERO, {}).
> - (NEG-S1, TAILS, X) → (ZERO, {}).
> - (ZERO, *flip*, *topsymbol*) → (ZERO, {}).
> - (POS-S2, *flip*, *topsymbol*) → Add enough transition rules to the automaton to simulate $g(\lambda)$ by a finite-state machine (only possible if g is rational with rational coefficients (Mossel and Peres 2005)[5]). Transition to POS-S2-ZERO if the machine outputs 0, or POS-S2-ONE if the machine outputs 1.
> - (NEG-S2, *flip*, *topsymbol*) → Same as before, but the transitioning states are NEG-S2-ZERO and NEG-S2-ONE, respectively.
> - (POS-S2-ONE, *flip*, *topsymbol*) → (POS-S1, {*topsymbol*, X}) (replace top stack symbol with *topsymbol*, then push X to the stack).
> - (POS-S2-ZERO, *flip*, EMPTY) → (NEG-S1, {EMPTY, X}).
> - (POS-S2-ZERO, *flip*, X) → (POS-S1, {}).
> - (NEG-S2-ZERO, *flip*, *topsymbol*) → (NEG-S1, {*topsymbol*, X}).
> - (NEG-S2-ONE, *flip*, EMPTY) → (POS-S1, {EMPTY, X}).
> - (NEG-S2-ONE, *flip*, X) → (NEG-S1, {}).
>
> The machine stops when it removes EMPTY from the stack, and the result is either ZERO (0) or ONE (1).

For the following algorithm, which extends the end of Note 1 of the Flajolet paper, the probability is— $$f(\lambda)=(1-\lambda) \sum_{n\ge 0} \lambda^{Hn} \mathtt{Coin}(\lambda)^n (1-\mathtt{Coin}(\lambda))^{Hn-n} {Hn \choose n},$$ where $H \ge 2$ is an integer; and `Coin` has the same meaning as earlier.

1. Set d to 0.
2. Do the following process repeatedly until this run of the algorithm returns a value:
    1. Flip the input coin. If it returns 1, go to the next substep. Otherwise, return either 1 if d is 0, or 0 otherwise.
    2. Run a Bernoulli factory algorithm for `Coin`(λ). If the run returns 1, add $(H-1)$ to d. Otherwise, subtract 1 from d.

The following algorithm simulates the probability— $$ f(\lambda) = (1-\lambda) \sum_{n\ge 0} \lambda^n \left(\sum_{m\ge 0} W(n,m) \mathtt{Coin}(\lambda)^m (1-\mathtt{Coin}(\lambda))^{n-m} {n \choose m}\right)$$ $$= (1-\lambda) \sum_{n\ge 0} \lambda^n \left(\sum_{m\ge 0} V(n,m) \mathtt{Coin}(\lambda)^m (1-\mathtt{Coin}(\lambda))^{n-m}\right),$$ where `Coin` has the same meaning as earlier; $W(n, m)$ is 1 if $m*H$ equals $(n-m)*T$, or 0 otherwise; and $H\ge 1$ and $T\ge 1$ are integers. (In the first formula, the sum in parentheses is a polynomial in Bernstein form, in the variable `Coin`(λ) and with only zeros and ones as coefficients. Because of the λ^n, the polynomial gets smaller as n gets larger. $V(n, m)$ is the number of n-letter words that have m heads *and* describe a walk that ends at the beginning.)

```
1. Set $d$ to 0.
2. Do the following process repeatedly until this run of the algorithm returns a value:
    1. Flip the input coin. If it returns 1, go to the next substep. Otherwise, return either 1 if $d$ is 0, or 0 otherwise.
    2. Run a Bernoulli factory algorithm for Coin($\lambda$). If the run returns 1 ("heads"), add $H$ to $d$. Otherwise ("tails"), subtract $T$ from $d$.
```

4 Irrational Probabilities

4.1 Ratio of Lower Gamma Functions ($\gamma(m, x)/\gamma(m, 1)$).

```
1. Set $ret$ to the result of **kthsmallest** with the two parameters $m$ and $m$. (Thus, $ret$ is distributed as $u^{1/m}$ where $u$ is a uniform random variate greater than 0 and less than 1; although **kthsmallest** accepts only integers, this formula works for every $m$ greater than 0.)
2. Set $k$ to 1, then set $u$ to point to the same value as $ret$.
3. Generate a uniform(0, 1) random variate $v$.
4. If $v$ is less than $u$: Set $u$ to $v$, then add 1 to $k$, then go to step 3.
5. If $k$ is odd[6], return a number that is 1 if $ret$ is less than $x$ and 0 otherwise. (If $ret$ is implemented as a uniform partially-sampled random number (PSRN), this comparison should be done via **URandLessThanReal**.) If $k$ is even[7], go to step 1.
```

Derivation: See Formula 1 in the section "**Probabilities Arising from Certain Permutations**", where:

- `ECDF(x)` is the probability that a uniform random variate greater than 0 and less than 1 is x or less, namely x if x is in [0, 1], 0 if x is less than 0, and 1 otherwise.

- `DPDF(x)` is the probability density function for the maximum of *m* uniform random variates in [0, 1], namely $m*x^{m-1}$ if *x* is in [0, 1], and 0 otherwise.

4.2 4/(3**π*)

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

1. Generate two PSRNs in the form of a uniformly chosen point inside a 2-dimensional quarter hypersphere (that is, a quarter of a "filled circle"; see "**Uniform Distribution Inside N-Dimensional Shapes**" in the article "More Algorithms for Arbitrary-Precision Sampling", as well as the examples there).
2. Let *x* be one of those PSRNs. Run **SampleGeometricBag** on that PSRN and return the result (which will be either 0 or 1).

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

4.3 (1 + exp(*k*)) / (1 + exp(*k* + 1))

This algorithm simulates this probability by computing lower and upper bounds of exp(1), which improve as more and more digits are calculated. These bounds are calculated through an algorithm by Citterio and Pavani (2016)[8]. Note the use of the methodology in Łatuszyński et al. (2009/2011, algorithm 2)[9] in this algorithm. In this algorithm, *k* must be an integer 0 or greater.

1. If *k* is 0, run the **algorithm for 2 / (1 + exp(2))** and return the result. If *k* is 1, run the **algorithm for (1 + exp(1)) / (1 + exp(2))** and return the result.
2. Generate a uniform(0, 1) random variate, call it *ret*.
3. If *k* is 3 or greater, return 0 if *ret* is greater than 38/100, or 1 if *ret*

is less than 36/100. (This is an early return step. If *ret* is implemented as a uniform PSRN, these comparisons should be done via the **URandLessThanReal algorithm**, which is described in my **article on PSRNs**.)

4. Set d to 2.
5. Calculate a lower and upper bound of exp(1) (LB and UB, respectively) in the form of rational numbers whose numerator has at most d digits, using the Citterio and Pavani algorithm. For details, see later.
6. Set rl to $(1+LB^k)$ / $(1+UB^{k+1})$, and set ru to $(1+UB^k)$ / $(1+LB^{k+1})$; both these numbers should be calculated using rational arithmetic.
7. If *ret* is greater than ru, return 0. If *ret* is less than rl, return 1. (If *ret* is implemented as a uniform PSRN, these comparisons should be done via **URandLessThanReal**.)
8. Add 1 to d and go to step 5.

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

Define the following operations:

- **Setup:** Set p to the list `[0, 1]`, set q to the list `[1, 0]`, set a to the list `[0, 0, 2]` (two zeros, followed by the integer part for exp(1)), set v to 0, and set av to 0.
- **Ensure *n*:** While v is less than or equal to n:
  1. (Ensure partial denominator v, starting from 0, is available.) If v + 2 is greater than or equal to the size of a, append 1, av, and 1, in that order, to the list a, then add 2 to av.
  2. (Calculate convergent v, starting from 0.) Append $a[n+2]$ * $p[n+1]+p[n]$ to the list p, and append $a[n+2]$ * $q[n+1]+q[n]$ to the list q. (Positions in lists start at 0. For example, $p[0]$ means the first item in p; $p[1]$ means the second; and so on.)
  3. Add 1 to v.
- **Get the numerator for convergent *n*:** Ensure n, then return $p[n+2]$.
- **Get convergent *n*:** Ensure n, then return $p[n+2]/q[n+2]$.
- **Get semiconvergent *n* given *d*:**
  1. Ensure n, then set m to floor$(((10^d)-1-p[n+1])/p[n+2])$.
  2. Return $(p[n+2]$ * m $+p[n+1])$ / $(q[n+2]$ * m $+q[n+1])$.

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

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

5 Sampling Distributions Using Incomplete Information

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

Algorithm 1. Suppose there is an oracle that produces independent random variates on a closed interval [a, b], and these numbers have an unknown mean of μ. The goal is now to produce nonnegative random variates whose expected value ("long-run average") is $f(\mu)$. Unless f is constant, this is possible if and only if—

- f is continuous on the closed interval, and
- $f(\mu)$ is greater than or equal to ε*min($(\mu - a)^n$, $(b - \mu)^n$) for some integer n and some ε greater than 0 (loosely speaking, f is nonnegative and neither touches 0 in the interior of the interval nor moves away from 0 more slowly than a polynomial)

(Jacob and Thiery 2015)[11]. (Here, a and b are both rational numbers and may be less than 0.)

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

1. Create a λ input coin that does the following: "Take a number from the oracle, call it x. With probability $(x-a)/(b-a)$ (see note below), return 1. Otherwise, return 0."
2. Run a Bernoulli factory algorithm for $g(\lambda)$, using the λ input coin. Then return K times the result.

> **Note:** The check "With probability $(x-a)/(b-a)$" is exact if the oracle produces only rational numbers. If the oracle can produce irrational numbers (such as numbers that follow a beta distribution or another non-discrete distribution), then the code for the oracle should use uniform **partially-sampled random numbers (PSRNs)**. In that case, the check can be implemented as follows. Let x be a uniform PSRN representing a number generated by the oracle. Set y to **RandUniformFromReal**($b-a$), then the check succeeds if **URandLess**(y, **UniformAddRational**(x, $-a$)) returns 1, and fails otherwise.
>
> **Example:** Suppose an oracle produces random variates in the interval [3, 13] with unknown mean μ, and the goal is to use the oracle to produce nonnegative random variates with mean $f(\mu) = -319/100 + \mu*103/50 - \mu^2*11/100$, which is a polynomial with Bernstein coefficients [2, 9, 5] in the given interval. Then since 8 is greater than the maximum of f in that interval, $g(\lambda)$ is a degree-2 polynomial in the interval [0, 1] that has Bernstein coefficients [2/8, 9/8, 5/8]. g can't be simulated as is, though, but increasing g's degree to 3 leads to the Bernstein coefficients [1/4, 5/6, 23/24, 5/8], which are all less than 1 so that the following algorithm can be used (see "**Certain Polynomials**"):
>
> 1. Set *heads* to 0.
> 2. Generate three random variates from the oracle (which must produce random variates in the interval [3, 13]). For each number x: With probability $(x-3)/(10-3)$, add 1 to *heads*.
> 3. Depending on *heads*, return 8 (that is, 1 times the upper bound) with the given probability, or 0 otherwise: *heads*=0 → probability 1/4; 1 → 5/6; 2 → 23/24; 3 → 5/8.

Algorithm 2. This algorithm takes an oracle and produces nonnegative random variates whose expected value ("long-run average") is the mean of $f(X)$, where X is a number produced by the oracle. The algorithm appears in the appendix, however, because it requires applying an arbitrary function (here, f) to a potentially irrational number.

Algorithm 3. For this algorithm, see the appendix.

Algorithm 4. Say there is an oracle in the form of a fair die. The number of faces of the die, *n*, is at least 2 but otherwise unknown. Each face shows a different integer 0 or greater and less than *n*. The question arises: Which probability distributions based on the number of faces can be sampled with this oracle? This question was studied in the French-language dissertation of R. Duvignau (2015, section 5.2)[12], and the following are four of these distributions.

Bernoulli 1/n. It's trivial to generate a Bernoulli variate that is 1 with probability 1/*n* and 0 otherwise: just take a number from the oracle and return either 1 if that number is 0, or 0 otherwise. Alternatively, take two numbers from the oracle and return either 1 if both are the same, or 0 otherwise (Duvignau 2015, p. 153)[13].

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

Binomial with parameters n and 1/n. Using the oracle, the following algorithm generates a binomial variate of this kind (Duvignau 2015, Algorithm 20)[14]:

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

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

1. Take items from the oracle until *k* different items were taken this way. Let *U* be a list of these *k* items, in the order in which they were first taken.
2. Create an empty list *L*.
3. For each integer *i* in [0, *k*):
    1. Create an empty list *M*.

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

> **Note:** Duvignau proved a result (Theorem 5.2) that answers the question: Which probability distributions based on the unknown n can be sampled with the oracle?[15] The result applies to a family of (discrete) distributions with the same unknown parameter n, starting with either 1 or a greater integer. Let Supp(m) be the set of values taken on by the distribution with parameter equal to m. Then that family can be sampled using the oracle if and only if:

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

6 Sum of Uniform Random Variates

This page presents new algorithms to sample the sum of uniform(0, 1) random variates and the ratio of two uniform(0, 1) random variates, with the help of **partially-sampled random numbers** (PSRNs), with arbitrary precision and without relying on floating-point arithmetic. See the page on PSRNs for more information on some of the algorithms made use of here, including **SampleGeometricBag** and **FillGeometricBag**.

The algorithms on this page work no matter what base the digits of the partially-sampled number are stored in (such as base 2 for decimal or base 10 for binary), unless noted otherwise.

6.1 About the Uniform Sum Distribution

The sum of *n* uniform(0, 1) random variates has the following probability density function (PDF) (see **MathWorld**):

$$f(x)=\left(\sum_{k=0}^n (-1)^k {n\choose k} (x-k)^{n-1} \text{sign}(x-k)\right)/(2(n-1)!),$$

where ${n\choose k}$ is a *binomial coefficient*, or the number of ways to choose *k* out of *n* labeled items, and sign(*x*) is 1 if *x* is greater than 0, or 0 if *x* is 0, or −1 is less than 0.[16][17][18]

This is a polynomial of degree $n - 1$. For *n* uniform numbers, the distribution can take on values that are 0 or greater and *n* or less.

The samplers given below for the uniform sum logically work as follows:

1. The distribution is divided into pieces that are each 1 unit long (thus, for example, if *n* is 4, there will be four pieces).

2. An integer in [0, *n*) is chosen uniformly at random, call it *i*, then the piece identified by *i* is chosen. There are **many algorithms to choose an integer** this way, but an algorithm that is "optimal" in terms of the number of bits it uses, as well as unbiased, should be chosen.

3. The PDF at [*i*, *i* + 1] is simulated. This is done by shifting the PDF so the desired piece of the PDF is at [0, 1] rather than its usual place. More specifically, the PDF is now as follows: $$f(x)=\left(\sum_{k=0}^n (-1)^k {n\choose k} ((x+i)-k)^{n-1} \text{sign}((x+i)-k)\right)/(2(n-1)!),$$ where *x* is a real number in [0, 1]. Since *f*′ is a polynomial, it can be rewritten in Bernstein form, so that it has *Bernstein coefficients*, which are equivalent to control points describing the shape of the curve drawn out by *f*′. (The Bernstein coefficients are the backbone of the well-known Bézier curve.) A polynomial can be written in Bernstein form as— $$\sum_{k=0}^m {m\choose k} x^k (1-x)^{m-k} a[k],$$ where *a*[*k*] are the control points and *m* is the polynomial's degree (here,

$n - 1$). In this case, there will be n control points, which together trace out a 1-dimensional Bézier curve. For example, given control points 0.2, 0.3, and 0.6, the curve is at 0.2 when $x = 0$, and 0.6 when $x = 1$. (Note that the curve is not at 0.3 when $x = 1/2$; in general, Bézier curves do not cross their control points other than the first and the last.)

   Moreover, this polynomial can be simulated because its Bernstein coefficients all lie in [0, 1] (Goyal and Sigman 2012)[19].

4. The sampler creates a "coin" made up of a uniform partially-sampled random number (PSRN) whose contents are built up on demand using an algorithm called **SampleGeometricBag**. It flips this "coin" $n - 1$ times and counts the number of times the coin returned 1 this way, call it j. (The "coin" will return 1 with probability equal to the to-be-determined uniform random variate.)
5. Based on j, the sampler accepts the PSRN with probability equal to the control point $a[j]$. (See (Goyal and Sigman 2012)[20].)
6. If the PSRN is accepted, the sampler optionally fills it up with uniform random digits, then sets the PSRN's integer part to i, then the sampler returns the finished PSRN. If the PSRN is not accepted, the sampler starts over from step 2.

6.2 Finding Parameters

Using the uniform sum sampler for an arbitrary n requires finding the Bernstein control points for each of the n pieces of the uniform sum PDF. This can be found, for example, with the Python code below, which uses the SymPy computer algebra library. In the code:

- `unifsum(x,n,v)` calculates the PDF of the sum of `n` uniform random variates when the variable `x` is shifted by `v` units.
- `find_control_points` returns the control points for each piece of the PDF for the sum of `n` uniform random variates, starting with piece 0.
- `find_areas` returns the relative areas for each piece of that PDF. This can be useful to implement a variant of the sampler above, as detailed later in this section.

```
def unifsum(x,n,v):
    # Builds up the PDF at x (with offset v)
    # of the sum of n uniform random variates
    ret=0
    x=x+v # v is an offset
    for k in range(n+1):
           s=(-1)**k*binomial(n,k)*(x-k)**(n-1)
           # Equivalent to k>x+v since x is limited
           # to [0, 1]
           if k>v: ret-=s
           else: ret+=s
    return ret/(2*factorial(n-1))

def find_areas(n):
   x=symbols('x', real=True)
   areas=[integrate(unifsum(x,n,i),(x,0,1)) for i in range(n)]
   g=prod([v.q for v in areas])
   areas=[int(v*g) for v in areas]
   g=gcd(areas)
   areas=[v/int(g) for v in areas]
   return areas

def find_control_points(n, scale_pieces=False):
 x=symbols('x', real=True)
 controls=[]
 for i in range(n):
  # Find the "usual" coefficients of the uniform
  # sum polynomial at offset i.
  poly=Poly(unifsum(x, n, i))
  coeffs=[poly.coeff_monomial(x**i) for i in range(n)]
  # Build coefficient vector
  coeffs=Matrix(coeffs)
  # Build power-to-Bernstein basis matrix
  mat=[[0 for _ in range(n)] for _ in range(n)]
  for j in range(n):
    for k in range(n):
       if k==0 or j==n-1:
         mat[j][k]=1
       elif k<=j:
         mat[j][k]=binomial(j, j-k) / binomial(n-1, k)
       else:
         mat[j][k]=0
  mat=Matrix(mat)
  # Get the Bernstein control points
  mv = mat*coeffs
  mvc = [Rational(mv[i]) for i in range(n)]
  maxcoeff = max(mvc)
  # If requested, scale up control points to raise acceptance rate
  if scale_pieces:
     mvc = [v/maxcoeff for v in mvc]
  mv = [[v.p, v.q] for v in mvc]
  controls.append(mv)
 return controls
```

The basis matrix is found, for example, as Equation 42 of (Ray and Nataraj 2012)[21].

For example, if *n* = 4 (so a sum of four uniform random variates is desired), the following control points are used for each piece of the PDF:

Piece	Control Points
0	0, 0, 0, 1/6
1	1/6, 1/3, 2/3, 2/3
2	2/3, 2/3, 1/3, 1/6
3	1/6, 0, 0, 0

For more efficient results, all these control points could be scaled so that the highest control point is equal to 1. This doesn't affect the algorithm's correctness because scaling a Bézier curve's control points scales the curve accordingly, as is well known. In the example above, after multiplying by 3/2 (the reciprocal of the highest control point, which is 2/3), the table would now look like this:

Piece	Control Points
0	0, 0, 0, 1/4
1	1/4, 1/2, 1, 1
2	1, 1, 1/2, 1/4
3	1/4, 0, 0, 0

Notice the following:

- All these control points are rational numbers, and the sampler may have to determine whether an event is true with probability equal to a control point. For rational numbers like these, it is possible to

determine this exactly (using only random bits), using the **ZeroOrOne** method given in my **article on randomization and sampling methods**.
- The first and last piece of the PDF have a predictable set of control points. Namely the control points are as follows:
    - Piece 0: 0, 0, ..., 0, 1/((n − 1)!), where (n − 1)! = 1*2*3*...*(n−1).
    - Piece n − 1: 1/((n − 1)!), 0, 0, ..., 0.

If the areas of the PDF's pieces are known in advance (and SymPy makes them easy to find as the `find_areas` method shows), then the sampler could be modified as follows, since each piece is now chosen with probability proportional to the chance that a random variate there will be sampled:

- Step 2 is changed to read: "An integer in [0, n) is chosen with probability proportional to the corresponding piece's area, call the integer i, then the piece identified by i is chosen. There are many **algorithms to choose an integer** this way."

- The last sentence in step 6 is changed to read: "If the PSRN is not accepted, the sampler starts over from step 3." With this, the same piece is sampled again.

- The following are additional modifications that should be done to the sampler. However, not applying them does not affect the sampler's correctness.

    - The control points should be scaled so that the highest control point of *each* piece is equal to 1. See the table below for an example.
    - If piece 0 is being sampled and the PSRN's digits are binary (base 2), the "coin" described in step 4 uses a modified version of **SampleGeometricBag** in which a 1 (rather than any other digit) is sampled from the PSRN when it reads from or writes to that PSRN. Moreover, the PSRN is always accepted regardless of the result of the "coin" flip.
    - If piece n − 1 is being sampled and the PSRN's digits are binary (base 2), the "coin" uses a modified version of **SampleGeometricBag** in which a 0 (rather than any other digit) is sampled, and the PSRN is always accepted.

Piece	Control Points

0	0, 0, 0, 1
1	1/4, 1/2, 1, 1
2	1, 1, 1/2, 1/4
3	1, 0, 0, 0

6.3 Sum of Two Uniform Random Variates

The following algorithm samples the sum of two uniform random variates.

1. Create a positive-sign zero-integer-part uniform PSRN (partially-sampled random number), call it *ret*.
2. Generate an unbiased random bit (that is, either 0 or 1, chosen with equal probability).
3. Remove all digits from *ret*. (This algorithm works for digits of any base, including base 10 for decimal, or base 2 for binary.)
4. Call the **SampleGeometricBag** algorithm on *ret*, then generate an unbiased random bit.
5. If the bit generated in step 2 is 1 and the result of **SampleGeometricBag** is 1, optionally fill *ret* with uniform random digits as necessary to give its fractional part the desired number of digits (similarly to **FillGeometricBag**), then return *ret*.
6. If the bit generated in step 2 is 0 and the result of **SampleGeometricBag** is 0, optionally fill *ret* as in step 5, then set *ret*'s integer part to 1, then return *ret*.
7. Go to step 3.

For base 2, the following algorithm also works, using certain "tricks" described in the next section.

1. Generate an unbiased random bit (that is, either 0 or 1, chosen with equal probability), call it *d*.
2. Generate unbiased random bits until 0 is generated this way. Set *g* to the number of one-bits generated by this step.
3. Create a positive-sign zero-integer-part uniform PSRN (partially-sampled random number), call it *ret*. Then, set the digit at position *g* of the PSRN's fractional part to *d* (positions start at 0 in the PSRN).
4. Optionally, fill *ret* with uniform random digits as necessary to give its fractional part the desired number of digits (similarly to **FillGeometricBag**). Then set *ret*'s integer part to $(1 - d)$, then

```
    return ret
```

And here is Python code that implements this algorithm. It uses floating-point arithmetic only at the end, to convert the result to a convenient form, and that it relies on methods from *randomgen.py* and *bernoulli.py*.

```
def sum_of_uniform(bern, precision=53):
    """ Exact simulation of the sum of two uniform
          random variates. """
    bag=[]
    rb=bern.randbit()
    while True:
       bag.clear()
       if rb==1:
          # 0 to 1
          if bern.geometric_bag(bag)==1:
             return bern.fill_geometric_bag(bag, precision)
       else:
          # 1 to 2
          if bern.geometric_bag(bag)==0:
             return 1.0 + bern.fill_geometric_bag(bag, precision)

def sum_of_uniform_base2(bern, precision=53):
    """ Exact simulation of the sum of two uniform
          random variates (base 2). """
    if bern.randbit()==1:
      g=0
      while bern.randbit()==0:
          g+=1
      bag=[None for i in range(g+1)]
      bag[g]=1
      return bern.fill_geometric_bag(bag)
    else:
      g=0
      while bern.randbit()==0:
          g+=1
      bag=[None for i in range(g+1)]
      bag[g]=0
      return bern.fill_geometric_bag(bag) + 1.0
```

6.4 Sum of Three Uniform Random Variates

The following algorithm samples the sum of three uniform random variates.

1. Create a positive-sign zero-integer-part uniform PSRN, call it *ret*.
2. Choose an integer in [0, 6), uniformly at random. (With this, the left piece is chosen at a 1/6 chance, the right piece at 1/6, and the middle piece at 2/3, corresponding to the relative areas occupied by the three pieces.)
3. Remove all digits from *ret*.
4. If 0 was chosen by step 2, we will sample from the left piece of the function for the sum of three uniform random variates. This piece runs along the interval [0, 1) and is a polynomial with Bernstein coefficients of (0, 1, 1/2) (and is thus a Bézier curve with those control points). Due to the particular form of the control points, the piece can be sampled in one of the following ways:

    - Call the **SampleGeometricBag** algorithm twice on *ret*. If both of these calls return 1, then accept *ret* with probability 1/2. This is the most "naïve" approach.
    - Call the **SampleGeometricBag** algorithm twice on *ret*. If both of these calls return 1, then accept *ret*. This version of the step is still correct since it merely scales the polynomial so its upper bound is closer to 1, which is the top of the left piece, thus improving the acceptance rate of this step.
    - Base-2 only: Call a modified version of **SampleGeometricBag** twice on *ret*; in this modified algorithm, a 1 (rather than any other digit) is sampled from *ret* when that algorithm reads or writes a digit in *ret*. Then *ret* is accepted. This version will always accept *ret* on the first try, without rejection, and is still correct because *ret* would be accepted by this step only if **SampleGeometricBag** succeeds both times, which will happen only if that algorithm reads or writes out a 1 each time (because otherwise the control point is 0, meaning that *ret* is accepted with probability 0).

If *ret* was accepted, optionally fill *ret* with uniform random digits as necessary to give its fractional part the desired number of digits (similarly to **FillGeometricBag**), then return *ret*.

5. If 2, 3, 4, or 5 was chosen by step 2, we will sample from the middle piece of the PDF, which runs along the interval [1, 2) and is a polynomial with Bernstein coefficients (control points) of (1/2, 1, 1/2). Call the **SampleGeometricBag** algorithm twice on *ret*. If neither or both of these calls return 1, then accept *ret*. Otherwise, if one of these calls returns 1 and the other 0, then accept *ret* with probability 1/2. If *ret* was accepted, optionally fill *ret* as given in step 4, then set *ret*'s integer part to 1, then return *ret*.

6. If 1 was chosen by step 2, we will sample from the right piece of the PDF, which runs along the interval [2, 3) and is a polynomial with Bernstein coefficients (control points) of (1/2, 0, 0). Due to the particular form of the control points, the piece can be sampled in one of the following ways:

    - Call the **SampleGeometricBag** algorithm twice on *ret*. If both of these calls return 0, then accept *ret* with probability 1/2. This is the most "naïve" approach.
    - Call the **SampleGeometricBag** algorithm twice on *ret*. If both of these calls return 0, then accept *ret*. This version is correct for a similar reason as in step 4.
    - Base-2 only: Call a modified version of **SampleGeometricBag** twice on *ret*; in this modified algorithm, a 0 (rather than any other digit) is sampled from *ret* when that algorithm reads or writes a digit in *ret*. Then *ret* is accepted. This version is correct for a similar reason as in step 4.

    If *ret* was accepted, optionally fill *ret* as given in step 4, then set *ret*'s integer part to 2, then return *ret*.

7. Go to step 3.

And here is Python code that implements this algorithm.

```
def sum_of_uniform3(bern):
    """ Exact simulation of the sum of three uniform
          random variates. """
    r=6
    while r>=6:
       r=bern.randbit() + bern.randbit() * 2 + bern.randbit() * 4
    while True:
       # Choose a piece of the PDF uniformly (but
       # not by area).
       bag=[]
       if r==0:
          # Left piece
          if bern.geometric_bag(bag) == 1 and \
             bern.geometric_bag(bag) == 1:
              # Accepted
             return bern.fill_geometric_bag(bag)
       elif r<=4:
          # Middle piece
          ones=bern.geometric_bag(bag) + bern.geometric_bag(bag)
          if (ones == 0 or ones == 2) and bern.randbit() == 0:
              # Accepted
             return 1.0 + bern.fill_geometric_bag(bag)
          if ones == 1:
              # Accepted
             return 1.0 + bern.fill_geometric_bag(bag)
       else:
          # Right piece
          if bern.randbit() == 0 and \
             bern.geometric_bag(bag) == 0 and \
             bern.geometric_bag(bag) == 0:
              # Accepted
             return 2.0 + bern.fill_geometric_bag(bag)
```

7 Acknowledgments

Michael Shoemate gave comments on this article.

8 Notes

9 Appendix

9.1 Probability Transformations

The following algorithm takes a uniform partially-sampled random number (PSRN) as a "coin" and flips that "coin" using **SampleGeometricBag**. Given that "coin" and a function f as described below, the algorithm returns 1 with probability $f(U)$, where U is the number built up by the uniform PSRN (see also Brassard et al., (2019)[22], (Devroye 1986, p. 769)[23], (Devroye and Gravel 2020)[24]. In the algorithm:

- The uniform PSRN's sign must be positive and its integer part must be 0.
- For correctness, $f(U)$ must meet the following conditions:
    - If the algorithm will be run multiple times with the same PSRN, $f(U)$ must be the constant 0 or 1, or be continuous and polynomially bounded on the open interval (0, 1) (polynomially bounded means that both $f(U)$ and $1-f(U)$ are greater than or equal to $\min(U^n, (1-U)^n)$ for some integer n (Keane and O'Brien 1994)[25]).
    - Otherwise, $f(U)$ must map the interval [0, 1] to [0, 1] and be continuous everywhere or "almost everywhere".

  The first set of conditions is the same as those for the Bernoulli factory problem (see "**About Bernoulli Factories**) and ensure this algorithm is unbiased (see also Łatuszyński et al. (2009/2011)[26]).

The algorithm follows.

1. Set v to 0 and k to 1.
2. (v acts as a uniform random variate greater than 0 and less than 1 to compare with $f(U)$.) Set v to $b * v + d$, where b is the base (or radix) of the uniform PSRN's digits, and d is a digit chosen uniformly at random.
3. Calculate an approximation of $f(U)$ as follows:
    1. Set n to the number of items (sampled and unsampled digits) in the uniform PSRN's fractional part.
    2. Of the first n digits (sampled and unsampled) in the PSRN's fractional part, sample each of the unsampled digits uniformly at random. Then let uk be the PSRN's digit expansion up to the first n digits after the point.
    3. Calculate the lowest and highest values of f in the interval [uk,

    $uk + b^{-n}$], call them *fmin* and *fmax*. If 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 $\pi/5$, b is 10, and k is 3, *pk* is 628.
5. If $pk + 1 \le v$, return 0. If $pk - 2 \ge v$, return 1. If neither is the case, add 1 to k and go to step 2.

> **Notes:**
>
> 1. This algorithm is related to the Bernoulli factory problem, where the input probability is unknown. However, the algorithm doesn't exactly solve that problem because it has access to the input probability's value to some extent.
> 2. This section appears in the appendix because this article is focused on algorithms that don't rely on calculations of irrational numbers.

9.2 Proof of the General Martingale Algorithm

This proof of the **general martingale algorithm** is similar to the proof for certain alternating series with only nonzero coefficients, given in Łatuszyński et al. (2019/2011)[27], section 3.1. Suppose we repeatedly flip a coin that shows heads with probability $g(\lambda)$ and we get the following results: $X_1, X_2, ...$, where each result is either 1 if the coin shows heads or 0 otherwise. Then define two sequences *U* and *L* as follows:

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

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

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

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

9.3 Algorithm for sin(*λ***π*/2)

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

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

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

This is a **convex combination** of $w_n(1)$ and $\frac{w_n(\mu)}{w_n(1)}$ — to simulate $g(\mu)$, first an integer *n* is chosen with probability $w_n(1)$ and then a coin that shows heads with probability $\frac{w_n(\mu)}{w_n(1)}$ is flipped. Finally, to simulate

$f(\lambda)$, the input coin is "inverted" ($\mu = 1-\lambda$), $g(\mu)$ is simulated using the "inverted" coin, and 1 minus the simulation result is returned.

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

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

9.4 Pushdown Automata and Algebraic Functions

Moved to **Supplemental Notes on Bernoulli Factories**.

9.5 Sampling Distributions Using Incomplete Information: Omitted Algorithms

Algorithm 2. Suppose there is an *oracle* that produces independent random real numbers whose expected value ("long-run average") is a known or unknown mean. The goal is now to produce nonnegative random variates whose expected value is the mean of $f(X)$, where X is a number produced by the oracle. This is possible whenever—

- f has a finite lower bound and a finite upper bound on its domain, and
- the mean of $f(X)$ is not less than δ, where δ is a known rational number greater than 0.

The algorithm to achieve this goal follows (see Lee et al. 2014)[28]:

1. Let m be a rational number equal to or greater than the maximum

value of abs($f(\mu)$) anywhere. Create a ν input coin that does the following: "Take a number from the oracle, call it x. With probability abs($f(x)$)/m, return a number that is 1 if $f(x) < 0$ and 0 otherwise. Otherwise, repeat this process."
2. Use one of the **linear Bernoulli factories** to simulate 2*ν (2 times the ν coin's probability of heads), using the ν input coin, with $\epsilon = \delta/m$. If the factory returns 1, return 0. Otherwise, take a number from the oracle, call it ξ, and return abs($f(\xi)$).

> **Example:** An example from Lee et al. (2014)[29]. Say the oracle produces uniform random variates in [0, 3*π], and let $f(\nu)$ = sin(ν). Then the mean of $f(X)$ is 2/(3*π), which is greater than 0 and found in SymPy by `sympy.stats.E(sin(sympy.stats.Uniform('U',0,3*pi)))`, so the algorithm can produce nonnegative random variates whose expected value ("long-run average") is that mean.
>
> **Notes:**
>
> 1. Averaging to the mean of $f(X)$ (that is, $\mathbf{E}[f(X)]$ where $\mathbf{E}[.]$ means expected or average value) is not the same as averaging to $f(\mu)$ where μ is the mean of the oracle's numbers (that is, $f(\mathbf{E}[X])$). For example, if X is 0 or 1 with equal probability, and $f(\nu) = \exp(-\nu)$, then $\mathbf{E}[f(X)] = \exp(0) + (\exp(-1) - \exp(0))*(1/2)$, and $f(\mathbf{E}[X]) = f(1/2) = \exp(-1/2)$.
>
> 2. (Lee et al. 2014, Corollary 4)[30]: If $f(\mu)$ is known to return only values in the interval [a, c], the mean of $f(X)$ is not less than δ, $\delta > b$, and δ and b are known numbers, then Algorithm 2 can be modified as follows:
>
>   - Use $f(\nu) = f(\nu) - b$, and use $\delta = \delta - b$.
>   - m is taken as max($b-a$, $c-b$).
>   - When Algorithm 2 finishes, add b to its return value.
>
> 3. The check "With probability abs($f(x)$)/m" is exact if the oracle produces only rational numbers *and* if $f(x)$ outputs only rational numbers. If the oracle or f can produce irrational numbers (such as numbers that follow a beta distribution or another non-discrete distribution), then this check should be implemented using uniform **partially-sampled random numbers (PSRNs)**.

Algorithm 3. Suppose there is an *oracle* that produces independent random real numbers that are all greater than or equal to a (which is a known rational number), whose mean (μ) is unknown. The goal is to use the oracle to produce nonnegative random variates with mean $f(\mu)$. This is possible only if f is 0 or greater everywhere in the interval $[a, \infty)$ and is nowhere decreasing in that interval (Jacob and Thiery 2015)[31]. This can be done using the algorithm below. In the algorithm:

- $f(\mu)$ must be a function that can be written as—
  $c[0]*z^0 + c[1]*z^1 + ...$,
  which is an infinite series where $z = \mu-a$ and all $c[i]$ are 0 or greater.
- ψ is a rational number close to 1, such as 95/100. (The exact choice is arbitrary and can be less or greater for efficiency purposes, but must be greater than 0 and less than 1.)

The algorithm follows.

1. Set *ret* to 0, *prod* to 1, k to 0, and w to 1. (w is the probability of generating k or more random variates in a single run of the algorithm.)
2. If k is greater than 0: Take a number from the oracle, call it x, and multiply *prod* by $x-a$.
3. Add $c[k]$**prod*/w to *ret*.
4. Multiply w by ψ and add 1 to k.
5. With probability ψ, go to step 2. Otherwise, return *ret*.

Now, assume the oracle's numbers are all less than or equal to b (rather than greater than or equal to a), where b is a known rational number. Then f must be 0 or greater everywhere in $(-\infty, b]$ and be nowhere increasing there (Jacob and Thiery 2015)[32], and the algorithm above can be used with the following modifications: (1) In the note on the infinite series, $z = b - \mu$; (2) in step 2, multiply *prod* by $b - x$ rather than $x - a$.

> **Note:** This algorithm is exact if the oracle produces only rational numbers *and* if all $c[i]$ are rational numbers. If the oracle can produce irrational numbers, then they should be implemented using uniform PSRNs. See also note 3 on Algorithm 2.

10 License

---

1. Thomas, A.C., Blanchet, J., "**A Practical Implementation of the Bernoulli Factory**", arXiv:1106.2508v3 [stat.AP], 2012.↩
2. Nacu, Şerban, and Yuval Peres. "**Fast simulation of new coins from old**", The Annals of Applied Probability 15, no. 1A (2005): 93-115.↩
3. Flajolet, P., Pelletier, M., Soria, M., "**On Buffon machines and numbers**", arXiv:0906.5560 [math.PR], 2010↩
4. Mossel, Elchanan, and Yuval Peres. New coins from old: computing with unknown bias. Combinatorica, 25(6), pp.707-724, 2005.↩
5. Mossel, Elchanan, and Yuval Peres. New coins from old: computing with unknown bias. Combinatorica, 25(6), pp.707-724, 2005.↩
6. "*x* is odd" means that *x* is an integer and not divisible by 2. This is true if *x* − 2*floor(*x*/2) equals 1, or if *x* is an integer and the least significant bit of abs(*x*) is 1.↩
7. "*x* is even" means that *x* is an integer and divisible by 2. This is true if *x* − 2*floor(*x*/2) equals 0, or if *x* is an integer and the least significant bit of abs(*x*) is 0.↩
8. Citterio, M., Pavani, R., "A Fast Computation of the Best k-Digit Rational Approximation to a Real Number", Mediterranean Journal of Mathematics 13 (2016).↩
9. 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.↩
10. "*x* is odd" means that *x* is an integer and not divisible by 2. This is true if *x* − 2*floor(*x*/2) equals 1, or if *x* is an integer and the least significant bit of abs(*x*) is 1.↩
11. Jacob, P.E., Thiery, A.H., "On nonnegative unbiased estimators", Ann. Statist., Volume 43, Number 2 (2015), 769-784.↩

12. Duvignau, R., 2015. Maintenance et simulation de graphes aléatoires dynamiques (Doctoral dissertation, Université de Bordeaux).↩

13. Duvignau, R., 2015. Maintenance et simulation de graphes aléatoires dynamiques (Doctoral dissertation, Université de Bordeaux).↩

14. Duvignau, R., 2015. Maintenance et simulation de graphes aléatoires dynamiques (Doctoral dissertation, Université de Bordeaux).↩

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

16. *Summation notation*, involving the Greek capital sigma (Σ), is a way to write the sum of one or more terms of similar form. For example, $\sum_{k=0}^n g(k)$ means $g(0)+g(1)+...+g(n)$, and $\sum_{k\ge 0} g(k)$ means $g(0)+g(1)+...$.↩

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

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

19. 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.↩

20. 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.↩

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

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

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

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

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

26. Łatuszyński, K., Kosmidis, I., Papaspiliopoulos, O., Roberts, G.O., "**Simulating events of unknown probabilities via reverse time martingales**", arXiv:0907.4018v2 [stat.CO], 2009/2011.↩

27. Łatuszyński, K., Kosmidis, I., Papaspiliopoulos, O., Roberts, G.O., "**Simulating events of unknown probabilities via reverse time martingales**", arXiv:0907.4018v2 [stat.CO], 2009/2011.↩

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

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

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

31. Jacob, P.E., Thiery, A.H., "On nonnegative unbiased estimators", Ann. Statist., Volume 43, Number 2 (2015), 769-784.↩

32. Jacob, P.E., Thiery, A.H., "On nonnegative unbiased estimators", Ann. Statist., Volume 43, Number 2 (2015), 769-784.↩