# More Algorithms for Arbitrary-Precision Sampling

This version of the document is dated 2020-10-25.

**Peter Occil**

This page contains additional algorithms for arbitrary-precision sampling of continuous distributions, Bernoulli factory algorithms (biased-coin to biased-coin algorithms), and algorithms to simulate irrational probabilities. These samplers are designed to not rely on floating-point arithmetic. They may depend on algorithms given in the following pages:

- **Partially-Sampled Random Numbers for Accurate Sampling of the Beta, Exponential, and Other Continuous Distributions**
- **Bernoulli Factory Algorithms**

# 1 Contents



# 2 Bernoulli Factories and Irrational Probability Simulation

## 2.1 Certain Numbers Based on the Golden Ratio

The following algorithm given by Fishman and Miller (2013)[1] finds the continued fraction expansion of certain numbers described as—

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

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

First, define the following operations:

- **Get the previous and next Fibonacci-based number given *k*, *m*, and $\ell$**:
  1. If $k$ is 0 or less, return an error.
  2. Set $g0$ to 0, $g1$ to 1, $x$ to 0, and $y$ to 0.
  3. Do the following $k$ times: Set $y$ to $m * g1 + \ell * g0$, then set $x$ to $g0$, then set $g0$ to $g1$, then set $g1$ to $y$.
  4. Return $x$ and $y$, in that order.
- **Get the partial denominator given *pos*, *k*, *m*, and $\ell$** (this partial denominator is part of the continued fraction expansion found by Fishman and Miller):
  1. **Get the previous and next Fibonacci-based number given *k*, *m*, and $\ell$**, call them $p$ and $n$, respectively.
  2. If $\ell$ is 1 and $k$ is odd, return $p + n$.
  3. If $\ell$ is −1 and $pos$ is 0, return $n - p - 1$.
  4. If $\ell$ is 1 and $pos$ is 0, return $(n + p) - 1$.
  5. If $\ell$ is −1 and $pos$ is even, return $n - p - 2$. (The paper had an error here; the correction given here was verified by Miller via personal communication.)
  6. If $\ell$ is 1 and $pos$ is even, return $(n + p) - 2$.
  7. Return 1.

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

1. **Get the partial denominator given *pos*, *k*, *m*, and $\ell$**, call it $kp$.
2. With probability $kp/(1 + kp)$, return a number that is 1 with probability $1/kp$ and 0 otherwise.
3. Run this algorithm recursively, but with $pos = pos + 1$. If the algorithm returns 1, return 0. Otherwise, go to step 2.

## 2.2 Ratio of Lower Gamma Functions (γ(*m*, *n*)/γ(*m*, 1)).

1. Set *ret* to the result of **kthsmallest** with the two parameters $m$ and $m$.
2. Set $k$ to 1, then set $u$ to point to the same value as *ret*.
3. Generate a uniform(0, 1) random number $v$.
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, return a number that is 1 if *ret* is less than $n$ and 0 otherwise. (If *ret* is implemented as a uniform PSRN, this comparison should be done via **URandLessThanReal**.) If $k$ is even, go to step 1.

# 3 Arbitrary-Precision Samplers

## 3.1 Rayleigh Distribution

The following is an arbitrary-precision sampler for the Rayleigh distribution with

parameter $s$, which is a rational number greater than 0.

1. Set $k$ to 0, and set $y$ to $2 * s * s$.
2. With probability $\exp(-(k * 2 + 1)/y)$, go to step 3. Otherwise, add 1 to $k$ and repeat this step. (The probability check should be done with the **exp(−x/y) algorithm** in "**Bernoulli Factory Algorithms**", with $x/y = (k * 2 + 1)/y$.)
3. (Now we sample the piece located at $[k, k + 1)$.) Create a positive-sign zero-integer-part uniform PSRN, and create an input coin that returns the result of **SampleGeometricBag** on that uniform PSRN.
4. Set $ky$ to $k * k / y$.
5. (At this point, we simulate $\exp(-U^2/y)$, $\exp(-k^2/y)$ , $\exp(-U*k*2/y)$, as well as a scaled-down version of $U + k$, where $U$ is the number built up by the uniform PSRN.) Call the **exp(−x/y) algorithm** with $x/y = ky$, then call the **exp(−(λ^k * x)) algorithm** using the input coin from step 2, $x = 1/y$, and $k = 2$, then call the same algorithm using the same input coin, $x = k * 2 / y$, and $k = 1$, then call the **sub-algorithm** given later with the uniform PSRN and $k = k$. If all of these calls return 1, the uniform PSRN was accepted. Otherwise, remove all digits from the uniform PSRN's fractional part and go to step 4.
6. If the uniform PSRN, call it *ret*, was accepted by step 5, set *ret*'s integer part to $k$, then optionally fill *ret* with uniform random digits as necessary to give its fractional part the desired number of digits (similarly to **FillGeometricBag**), and return *ret*.

The sub-algorithm below simulates a probability equal to $(U+k)/base^z$, where $U$ is the number built by the uniform PSRN, *base* is the base (radix) of digits stored by that PSRN, $k$ is an integer 0 or greater, and $z$ is the number of significant digits in $k$ (for this purpose, $z$ is 0 if $k$ is 0).

For base 2:

1. Set $N$ to 0.
2. With probability 1/2, go to the next step. Otherwise, add 1 to $N$ and repeat this step.
3. If $N$ is less than $z$, return $\mathrm{rem}(k / 2^{z - 1 - N}, 2)$. (Alternatively, shift $k$ to the right, by $z - 1 - N$ bits, then return $k$ *AND* 1, where "*AND*" is a bitwise AND-operation.)
4. Subtract $z$ from $N$. Then, if the item at position $N$ in the uniform PSRN's fractional part (positions start at 0) is not set to a digit (e.g., 0 or 1 for base 2), set the item at that position to a digit chosen uniformly at random (e.g., either 0 or 1 for base 2), increasing the capacity of the uniform PSRN's fractional part as necessary.
5. Return the item at position $N$.

For bases other than 2, such as 10 for decimal, this can be implemented as follows (based on **URandLess**):

1. Set $i$ to 0.
2. If $i$ is less than $z$:
    1. Set $da$ to $\mathrm{rem}(k / 2^{z - 1 - i}, base)$, and set $db$ to a digit chosen uniformly at random (that is, an integer in the interval $[0, base)$).
    2. Return 1 if $da$ is less than $db$, or 0 if $da$ is greater than $db$.
3. If $i$ is $z$ or greater:
    1. If the digit at position $(i - z)$ in the uniform PSRN's fractional part is not set, set the item at that position to a digit chosen uniformly at random (positions start at 0 where 0 is the most significant digit after the point, 1 is the next, etc.).
    2. Set $da$ to the item at that position, and set $db$ to a digit chosen uniformly at random (that is, an integer in the interval $[0, base)$).
    3. Return 1 if $da$ is less than $db$, or 0 if $da$ is greater than $db$.
4. Add 1 to $i$ and go to step 3.

## 3.2 Uniform Distribution Inside N-Dimensional Shapes

The following is a general way to describe an arbitrary-precision sampler for generating a point uniformly at random inside a geometric shape located entirely in the hypercube [0, 1]×[0, 1]×...×[0,1] in *N*-dimensional space, provided the shape's boundary has zero volume. Such a description has the following skeleton.

1. Generate *N* empty PSRNs, with a positive sign, an integer part of 0, and an empty fractional part. Call the PSRNs *p1*, *p2*, ..., *pN*.
2. Set *S* to *base*, where *base* is the base of digits to be stored by the PSRNs (such as 2 for binary or 10 for decimal). Then set *N* coordinates to 0, call the coordinates *c1*, *c2*, ..., *cN*, then set *d* to 1.
3. For each coordinate (*c1*, ..., *cN*), multiply that coordinate by *base* and add a digit chosen uniformly at random to that coordinate.
4. This step uses a function known as **InShape**, which takes the coordinates of a box and returns one of three values: *YES* if the box is entirely inside the shape; *NO* if the box is entirely outside the shape; and *MAYBE* if the box is partly inside and partly outside the shape, or if the function is unsure. In this step, run **InShape** using the current box, whose coordinates in this case are ((*c1*/*S*, *c2*/*S*, ..., *cN*/*S*), ((*c1*+1)/*S*, (*c2*+1)/*S*, ..., (*cN*+1)/*S*)). *Implementation notes:*

    - **InShape**, as well as the divisions of the coordinates by *S*, should be implemented using rational arithmetic. Instead of dividing those coordinates this way, an implementation can pass *S* as a separate parameter to **InShape**.
    - If the shape in question is convex, and the point (0, 0, ..., 0) is on or inside that shape, **InShape** can return—
        - *YES* if all the box's corners are in the shape;
        - *NO* if none of the box's corners are in the shape and if the shape's boundary does not intersect with the box's boundary; and
        - *MAYBE* in any other case, or if the function is unsure.

      In the case of two-dimensional shapes, the shape's corners are (*c1*/*S*, *c2*/*S*), ((*c1*+1)/*S*, *c2*/*S*), (*c1*,(*c2*+1)/*S*), and ((*c1*+1)/*S*, (*c2*+1)/*S*).
    - **InShape** implementations often involve a shape's *signed distance field*, its *implicit curve* or *algebraic curve* equation (for closed curves), or its *implicit surface* equation (for closed surfaces).
5. If the result of **InShape** is *YES*, then the current box was accepted. If the box is accepted this way, then at this point, *c1*, *c2*, etc., will each store the *d* digits of a coordinate in the shape, expressed as a number in the interval [0, 1], or more precisely, a range of numbers. (For example, if *base* is 10, *d* is 3, and *c1* is 342, then the first coordinate is 0.342, or more precisely, a number in the interval [0.342, 0.343].) In this case, do the following:
    1. For each coordinate (*c1*, ..., *cN*), transfer that coordinate's digits to the corresponding PSRN's fractional part. The variable *d* tells how many digits to transfer this way. (For example, if *base* is 10, *d* is 3, and *c1* is 342, set *p1*'s fractional part to [3, 4, 2].)
    2. For each PSRN (*p1*, ..., *pN*), optionally fill that PSRN with uniform random digits as necessary to give its fractional part the desired number of digits (similarly to **FillGeometricBag**).
    3. For each PSRN, optionally do the following: With probability 1/2, set that

PSRN's sign to negative. (This will result in a symmetric shape in the corresponding dimension. This step can be done for some PSRNs and not others.)

4. Return the PSRNs *p1*, ..., *pN*, in that order.

6. If the result of **InShape** is *NO*, then the current box lies outside the shape and is rejected. In this case, go to step 2.
7. If the result of **InShape** is *MAYBE*, it is not known whether the current box lies fully inside the shape, so multiply *S* by *base*, then add 1 to *d*, then go to step 3.

**Notes:**

- See (Li and El Gamal 2016)[2] and (Oberhoff 2018)[3] for related work on encoding random points uniformly distributed in a shape.
- Rejection sampling on a shape is subject to the "curse of dimensionality", since typical shapes of high dimension will tend to cover much less volume than their bounding boxes, so that it would take a lot of time on average to accept a high-dimensional box. Moreover, the more area the shape takes up in the bounding box, the higher the acceptance rate.
- An **InShape** function can implement a set operation (such as a union, intersection, or difference) of several simpler shapes, each with its own **InShape** function. The final result depends on the shape operation (such as union or intersection) as well as the result returned by each component for a given box (for example, for unions, the final result is *YES* if any component returns *YES*; *NO* if all components return *NO*; and *MAYBE* otherwise).
- (Devroye 1986, chapter 8, section 3)[4] describes grid-based methods to optimize random point generation. In this case, the result of **InShape** could be precalculated for all boxes whose coordinates begin with a *k*-digit prefix; each such box is labeled with the result; all boxes labeled *NO* are discarded; and the algorithm is modified by adding the following after step 2: "2a. Choose a precalculated box uniformly at random, then set *c1*, ..., *cN* to that box's coordinates, then set *d* to *k* and set *S* to $base^k$. If a box labeled *YES* was chosen, follow the substeps in step 5. If a box labeled *MAYBE* was chosen, multiply *S* by *base* and add 1 to *d*." (For example, if *base* is 10, *k* is 1, and *N* is 2, the space could be divided into a 10×10 grid, made up of 100 boxes. Then, **InShape** is precalculated for the box with coordinates ((0, 0), (1, 1)), the box ((0, 1), (1, 2)), and so on, each such box is labeled with the result, and boxes labeled *NO* are discarded. Finally the algorithm above is modified as just given.)

**Examples:**

- The following example generates a point inside a quarter diamond (centered at (0, ..., 0), "radius" 1): Let **InShape** return *YES* if ((*c1*+1) + ... + (*cN*+1)) < *S*; *NO* if (*c1* + ... + *cN*) > *S*; and *MAYBE* otherwise. For a full circle, step 5.3 in the algorithm is done for all *N* dimensions.
- The following example generates a point inside a quarter hypersphere (centered at (0, ..., 0), radius 1): Let **InShape** return *YES* if ((*c1*+1) + ... + (*cN*+1)) < $S^2$; *NO* if (*c1* + ... + *cN*) > $S^2$; and *MAYBE* otherwise. For a full hypersphere, step 5.3 in the algorithm is done for all *N* dimensions. In the case of a 2-dimensional circle, this algorithm thus adapts the well-known rejection technique of generating X and Y coordinates until $X^2+Y^2 < 1$ (e.g., (Devroye 1986, p. 230 et seq.)[4]).

## 3.3 Sum of Exponential Random Numbers

An arbitrary-precision sampler for the sum of $n$ exponential random numbers (also known as the Erlang($n$) or gamma($n$) distribution) is doable via partially-sampled uniform random numbers, though it is obviously inefficient for large values of $n$.

1. Generate $n$ uniform PSRNs, and turn each of them into an exponential random number with a rate of 1, using an algorithm that employs rejection from the uniform distribution (such as the von Neumann algorithm or Karney's improvement to that algorithm (Karney 2014)[5]). This algorithm won't work for exponential PSRNs (e-rands), described in my article on **partially-sampled random numbers**, because the sum of two e-rands may follow a subtly wrong distribution. By contrast, generating exponential random numbers via rejection from the uniform distribution will allow unsampled digits to be sampled uniformly at random without deviating from the exponential distribution.
2. Generate the sum of the random numbers generated in step 1 by applying the **algorithm to add two PSRNs** given in another document.

## 3.4 Hyperbolic Secant Distribution

The following algorithm adapts the rejection algorithm from p. 472 in (Devroye 1986)[4] for arbitrary-precision sampling.

1. Generate a uniform PSRN, call it *ret*, and turn it into an exponential random number with a rate of 1, using an algorithm that employs rejection from the uniform distribution.
2. Set *ip* to 1 plus *ret*'s integer part.
3. (The rest of the algorithm accepts *ret* with probability 1/(1+*ret*).) With probability *ip*/(1+*ip*), generate a number that is 1 with probability 1/*ip* and 0 otherwise. If that number is 1, *ret* was accepted, in which case optionally fill it with uniform random digits as necessary to give its fractional part the desired number of digits (similarly to **FillGeometricBag**), then set *ret*'s sign to positive or negative with equal probability, then return *ret*.
4. Call **SampleGeometricBag** on *ret*'s fractional part (ignore *ret*'s integer part and sign). If the call returns 1, go to step 1. Otherwise, go to step 3.

## 3.5 Mixtures

A *mixture* involves sampling one of several distributions, where each distribution has a separate probability of being sampled. In general, an arbitrary-precision sampler is possible if all of the following conditions are met:

- There is a finite number of distributions to choose from.
- The probability of sampling each distribution is a rational number, or it can be expressed as a function for which a **Bernoulli factory algorithm** exists.
- For each distribution, an arbitrary-precision sampler exists.

One example of a mixture is two beta distributions, with separate parameters. One beta distribution is chosen with probability exp(−3) (a probability for which a Bernoulli factory algorithm exists) and the other is chosen with the opposite probability. For the two beta distributions, an arbitrary-precision sampling algorithm exists (see my article on **partially-sampled random numbers** for details).

## 3.6 Building an Arbitrary-Precision Sampler

In many cases, if a continuous distribution—

- has a probability density function (PDF), or a function proportional to the PDF, with a known symbolic form,
- has a cumulative distribution function (CDF) with a known symbolic form,
- takes on only values 0 or greater, and
- has a PDF that has an infinite tail to the right, is bounded from above (that is, *PDF(0)* is other than infinity), and decreases monotonically,

it may be possible to describe an arbitrary-precision sampler for that distribution. Such a description has the following skeleton.

1. With probability $A$, set *intval* to 0, then set *size* to 1, then go to step 4.
    - $A$ is calculated as ($CDF(1) - CDF(0)$) / ($1-CDF(0)$), where $CDF$ is the distribution's CDF. This should be found analytically using a computer algebra system such as SymPy.
    - The symbolic form of $A$ will help determine which Bernoulli factory algorithm, if any, will simulate the probability; if a Bernoulli factory exists, it should be used.
2. Set *intval* to 1 and set *size* to 1.
3. With probability $B(size, intval)$, go to step 4. Otherwise, add *size* to *intval*, then multiply *size* by 2, then repeat this step.
    - This step chooses an interval beyond 1, and grows this interval by geometric steps, so that an appropriate interval is chosen with the correct probability.
    - The probability $B(size, intval)$ is the probability that the interval is chosen given that the previous intervals weren't chosen, and is calculated as ($CDF(size + intval) - CDF(intval)$) / ($1-CDF(intval)$). This should be found analytically using a computer algebra system such as SymPy.
    - The symbolic form of $B$ will help determine which Bernoulli factory algorithm, if any, will simulate the probability; if a Bernoulli factory exists, it should be used.
4. Generate an integer in the interval [*intval*, *intval* + *size*) uniformly at random, call it $i$.
5. Create a positive-sign zero-integer-part uniform PSRN, *ret*.
6. Create an input coin that calls **SampleGeometricBag** on the PSRN *ret*. Run a Bernoulli factory algorithm that simulates the probability $C(i, \lambda)$, using the input coin (here, $\lambda$ is the probability built up in *ret* via **SampleGeometricBag**, and lies in the interval [0, 1]). If the call returns 0, go to step 4.
    - The probability $C(i, \lambda)$ is calculated as $PDF(i + \lambda) / M$, where $PDF$ is the distribution's PDF or a function proportional to the PDF, and should be found analytically using a computer algebra system such as SymPy.
    - In this formula, $M$ is any convenient number in the interval [$PDF(intval)$, max(1, $PDF(intval)$)], and should be as low as feasible. $M$ serves to ensure that $C$ is as close as feasible to 1 (to improve acceptance rates), but no higher than 1. The choice of $M$ can vary for each interval (each value of *intval*, which can only be 0, 1, or a power of 2). Any such choice for $M$ preserves the algorithm's correctness because the PDF has to be monotonically decreasing and a new interval isn't chosen when $\lambda$ is rejected.
    - The symbolic form of $C$ will help determine which Bernoulli factory algorithm, if any, will simulate the probability; if a Bernoulli factory exists, it should be used.
7. The PSRN *ret* was accepted, so set *ret*'s integer part to $i$, then 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*.

Examples of algorithms that use this skeleton are the algorithm for the **ratio of two uniform random numbers**, the algorithm for the Rayleigh distribution given above, and

the algorithm for the reciprocal of power of uniform, given later.

Perhaps the most difficult part of describing an arbitrary-precision sampler with this skeleton is finding the appropriate Bernoulli factory for the probabilities $A$, $B$, and $C$, especially when these probabilities have a non-trivial symbolic form.

> **Note:** The algorithm skeleton uses ideas similar to the inversion-rejection method described in (Devroye 1986, ch. 7, sec. 4.6)**(4)**; an exception is that instead of generating a uniform random number and comparing it to calculations of a CDF, this algorithm uses conditional probabilities of choosing a given piece, probabilities labeled $A$ and $B$. This approach was taken so that the CDF of the distribution in question is never directly calculated in the course of the algorithm, which furthers the goal of sampling with arbitrary precision and without using floating-point arithmetic.

## 3.7 Reciprocal of Power of Uniform

The following algorithm generates a PSRN of the form $1/U^{1/x}$, where $U$ is a uniform random number in [0, 1] and $x$ is an integer greater than 0.

1. Set *intval* to 1 and set *size* to 1.
2. With probability $(4^x-2^x)/4^x$, go to step 3. Otherwise, add *size* to *intval*, then multiply *size* by 2, then repeat this step.
3. Generate an integer in the interval [*intval*, *intval* + *size*) uniformly at random, call it $i$.
4. Create a positive-sign zero-integer-part uniform PSRN, *ret*.
5. Create an input coin that calls **SampleGeometricBag** on the PSRN *ret*. Call the **algorithm for $d^k$ / $(c + \lambda)^k$** in "**Bernoulli Factory Algorithms**", using the input coin, where $d$ = *intval*, $c = i$, and $k = x + 1$ (here, $\lambda$ is the probability built up in *ret* via **SampleGeometricBag**, and lies in the interval [0, 1]). If the call returns 0, go to step 3.
6. The PSRN *ret* was accepted, so set *ret*'s integer part to $i$, then 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*.

This algorithm uses the skeleton described earlier in "Building an Arbitrary-Precision Sampler". Here, the probabilities $A$, $B$, and $C$ are as follows:

- $A = 0$, since the random number can't lie in the interval [0, 1).
- $B = (4^x-2^x)/4^x$.
- $C = (x/(i + \lambda)^{x+1})$ / $M$. Ideally, $M$ is either $x$ if *intval* is 1, or $x/intval^{x+1}$ otherwise. Thus, the ideal form for $C$ is $intval^{x+1}/(i+\lambda)^{x+1}$.

## 3.8 Distribution of $U/(1-U)$

The following algorithm generates a PSRN of the form $U/(1-U)$, where $U$ is a uniform random number in [0, 1].

1. With probability 1/2, set *intval* to 0, then set *size* to 1, then go to step 4.
2. Set *intval* to 1 and set *size* to 1.
3. With probability *size*/(*size* + *intval* + 1), go to step 4. Otherwise, add *size* to *intval*, then multiply *size* by 2, then repeat this step.
4. Generate an integer in the interval [*intval*, *intval* + *size*) uniformly at random, call it $i$.

5. Create a positive-sign zero-integer-part uniform PSRN, *ret*.
6. Create an input coin that calls **SampleGeometricBag** on the PSRN *ret*. Call the **algorithm for $d^k$ / $(c + \lambda)^k$** in "**Bernoulli Factory Algorithms**", using the input coin, where $d = intval + 1$, $c = i + 1$, and $k = 2$ (here, $\lambda$ is the probability built up in *ret* via **SampleGeometricBag**, and lies in the interval [0, 1]). If the call returns 0, go to step 4.
7. The PSRN *ret* was accepted, so set *ret*'s integer part to $i$, then 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*.

This algorithm uses the skeleton described earlier in "Building an Arbitrary-Precision Sampler". Here, the probabilities $A$, $B$, and $C$ are as follows:

- $A = 1/2$.
- $B = size/(size + intval + 1)$.
- $C = (1/(i+\lambda+1)^2) / M$. Ideally, $M$ is $1/(intval+1)^2$. Thus, the ideal form for $C$ is $(intval+1)^2/(i+\lambda+1)^2$.

## 3.9 Arc-cosine Distribution

Here we reimplement an example from Devroye's book *Non-Uniform Random Variate Generation* (Devroye 1986, pp. 128–129)**(4)**. The following arbitrary-precision sampler generates a random number from a distribution with the following cumulative distribution function (CDF): `1 - cos(pi*x/2)`. The random number will be in the interval [0, 1]. Note that the result is the same as applying acos($U$)*2/π, where $U$ is a uniform [0, 1] random number, as pointed out by Devroye. The algorithm follows.

1. Call the **kthsmallest** algorithm with `n = 2` and `k = 2`, but without filling it with digits at the last step. Let *ret* be the result.
2. Set $m$ to 1.
3. Call the **kthsmallest** algorithm with `n = 2` and `k = 2`, but without filling it with digits at the last step. Let $u$ be the result.
4. With probability 4/(4*$m$*$m$ + 2*$m$), call the **URandLess** algorithm with parameters $u$ and *ret* in that order, and if that call returns 1, call the **algorithm for π / 4**, described in "**Bernoulli Factory Algorithms**", twice, and if both of these calls return 1, add 1 to $m$ and go to step 3. (Here, we incorporate an erratum in the algorithm on page 129 of the book.)
5. If $m$ is odd, optionally fill *ret* with uniform random digits as necessary to give its fractional part the desired number of digits (similarly to **FillGeometricBag**), and return *ret*.
6. If $m$ is even, go to step 1.

And here is Python code that implements this algorithm. Note that 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 example_4_2_1(rg, bern, precision=53):
    while True:
       ret=rg.kthsmallest_urand(2,2)
       k=1
       while True:
          u=rg.kthsmallest_urand(2,2)
          kden=4*k*k+2*k # erratum incorporated
          if randomgen.urandless(rg,u, ret) and \
             rg.zero_or_one(4, kden)==1 and \
             bern.zero_or_one_pi_div_4()==1 and \
```

```
   bern.zero_or_one_pi_div_4()==1:
   k+=1
elif (k&1)==1:
   return randomgen.urandfill(rg,ret,precision)/(1<<precision)
else: break
```

## 3.10 Logistic Distribution

The following new algorithm generates a random number that follows the logistic distribution.

1. Set $k$ to 0.
2. (Choose a 1-unit-wide piece of the logistic density.) Run the **algorithm for (1+exp(*k*))/(1+exp(*k*+1))** described in "**Bernoulli Factory Algorithms**"). If the call returns 0, add 1 to $k$ and repeat this step. Otherwise, go to step 3.
3. (The rest of the algorithm samples from the chosen piece.) Generate a uniform(0, 1) random number, call it $f$.
4. (Steps 4 through 7 succeed with probability $\exp(-(f+k))/(1+\exp(-(f+k)))^2$.) With probability 1/2, go to step 3.
5. Run the **algorithm for exp(−*k*/1)** (described in "Bernoulli Factory Algorithms"), then **sample from the number *f*** (e.g., call **SampleGeometricBag** on $f$ if $f$ is implemented as a uniform PSRN). If any of these calls returns 0, go to step 4.
6. With probability 1/2, accept $f$. If $f$ is accepted this way, set $f$'s integer part to $k$, then optionally fill $f$ with uniform random digits as necessary to give its fractional part the desired number of digits (similarly to **FillGeometricBag**), then set $f$'s sign to positive or negative with equal probability, then return $f$.
7. Run the **algorithm for exp(−*k*/1)** and **sample from the number *f*** (e.g., call **SampleGeometricBag** on $f$ if $f$ is implemented as a uniform PSRN). If both calls return 1, go to step 3. Otherwise, go to step 6.

# 4 Requests

We would like to see new implementations of the following:

- Algorithms that implement **InShape** for specific curves and surfaces. Recall that **InShape** determines whether a box lies inside, outside, or partly inside or outside a given curve or surface.
- Descriptions of new arbitrary-precision algorithms that use the skeleton given in the section "Building an Arbitrary-Precision Sampler".

# 5 Notes


- (1) Fishman, D., Miller, S.J., "Closed Form Continued Fraction Expansions of Special Quadratic Irrationals", ISRN Combinatorics Vol. 2013, Article ID 414623 (2013).
- (2) C.T. Li, A. El Gamal, "**A Universal Coding Scheme for Remote Generation of Continuous Random Variables**", arXiv:1603.05238v1 [cs.IT], 2016
- (3) Oberhoff, Sebastian, "**Exact Sampling and Prefix Distributions**", *Theses and Dissertations*, University of Wisconsin Milwaukee, 2018.
- (4) Devroye, L., ***Non-Uniform Random Variate Generation***, 1986.
- (5) Karney, C.F.F., "**Sampling exactly from the normal distribution**", arXiv:1303.6257v2 [physics.comp-ph], 2014.

# 6 License