Randomization and Sampling Methods

Peter Occil

Randomization and Sampling Methods

This version of the document is dated 2023-06-17.

Peter Occil

Abstract: This page discusses many ways applications can sample randomized content by transforming the numbers produced by an underlying source of random numbers, such as numbers produced by a pseudorandom number generator, and offers pseudocode and Python sample code for many of these methods.

2020 Mathematics Subject Classification: 68W20.

1 Introduction

This page catalogs *randomization methods* and *sampling methods*. A randomization or sampling method is driven by a "source of random numbers" and produces numbers or other values called ***random variates***. These variates are the result of the randomization. (The "source of random numbers" is often simulated in practice by so-called pseudorandom number generators, or PRNGs.) This document covers many methods, including—

- ways to sample integers or real numbers from a uniform distribution (such as the **core method, `RNDINT(N)`**),
- ways to generate randomized content and conditions, such as **true/false conditions**, **shuffling**, and **sampling unique items from a list**, and
- non-uniform distributions, including **weighted choice**, the **Poisson distribution**, and **other probability distributions**.

This page is focused on randomization and sampling methods that *exactly* sample from the distribution described, without introducing additional errors beyond those already present in the inputs (and assuming that an ideal "source of random numbers" is available). This will be the case if there is a finite number of values to choose from. But for the normal distribution and other distributions that take on infinitely many values, there will always be some level of approximation involved; in this case, the focus of this page is on methods that *minimize* the error they introduce.

This document shows pseudocode for many of the methods, and **sample Python code**[1] that implements many of the methods in this document is available, together with **documentation for the code**[2].

The randomization methods presented on this page assume we have an endless source of numbers chosen independently at random and with a uniform distribution. For more information, see "**Sources of Random Numbers**" in the appendix.

In general, the following are outside the scope of this document:

[1] https://peteroupc.github.io/randomgen.zip
[2] https://peteroupc.github.io/randomgendoc.html

- This document does not cover how to choose an underlying PRNG (or device or program that simulates a "source of random numbers") for a particular application, including in terms of security, performance, and quality. I have written more on recommendations in **another document**[3].
- This document does not include algorithms for specific PRNGs, such as Mersenne Twister, PCG, xorshift, linear congruential generators, or generators based on hash functions.
- This document does not cover how to test PRNGs for correctness or adequacy, and the same applies to other devices and programs that simulate a "source of random numbers". Testing is covered, for example, in "**Testing PRNGs for High-Quality Randomness**[4]".
- This document does not explain how to specify or generate "seeds" for use in PRNGs. This is **covered in detail**[5] elsewhere.
- This document does not show how to generate random security parameters such as encryption keys.
- This document does not cover randomness extraction (also known as *unbiasing*, *deskewing*, or *whitening*). See my **Note on Randomness Extraction**[6].
- "Variance reduction" methods, such as importance sampling or common random numbers, are outside the scope of this document.

In addition, this page is not focused on sampling methods used for computer graphics rendering (such as Poisson disk sampling, multiple importance sampling, blue noise, and gradient noise), because this application tends to give performance and visual acceptability a greater importance than accuracy and exact sampling.

1.1 About This Document

This is an open-source document; for an updated version, see the source code[7] **or its rendering on GitHub**[8]**. You can send comments on this document either on CodeProject**[9] **or on the GitHub issues page**[10]**.**

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**[11]:

- 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] https://peteroupc.github.io/random.html
[4] https://peteroupc.github.io/randomtest.html
[5] https://peteroupc.github.io/random.html#Nondeterministic_Sources_and_Seed_Generation
[6] https://peteroupc.github.io/randextract.html
[7] https://github.com/peteroupc/peteroupc.github.io/raw/master/randomfunc.md
[8] https://github.com/peteroupc/peteroupc.github.io/blob/master/randomfunc.md
[9] https://www.codeproject.com/Articles/1190459/Random-Number-Generation-and-Sampling-Methods
[10] https://github.com/peteroupc/peteroupc.github.io/issues
[11] https://github.com/peteroupc/peteroupc.github.io/issues/18

3 Notation

In this document:

- The **pseudocode conventions**[12] apply to this document.
- The following notation for intervals is used: [`a`, `b`) means "`a` or greater, but less than `b`". (`a`, `b`) means "greater than `a`, but less than `b`". (`a`, `b`] means "greater than `a` and less than or equal to `b`". [`a`, `b`] means "`a` or greater and `b` or less".
- `log1p(x)` is equivalent to `ln(1 + x)` and is a "robust" alternative to `ln(1 + x)` where `x` is a floating-point number (Pedersen 2018)[13].
- `MakeRatio(n, d)` creates a rational number with the given numerator `n` and denominator `d`.
- `Sum(list)` calculates the sum of the numbers in the given list of integers or rational numbers. (Summing floating-point numbers naïvely can introduce round-off errors.)

4 Uniform Random Integers

This section shows how to derive independent uniform random integers with the help of a "source of random numbers" (or a device or program that simulates that source).

This section describes four methods: `RNDINT`, `RNDINTEXC`, `RNDINTRANGE`, `RNDINTEXCRANGE`. Of these, `RNDINT`, described next, can serve as the basis for the remaining methods.

4.1 `RNDINT`: Random Integers in [0, N]

In this document, **`RNDINT(maxInclusive)`** is the core method in this document; it derives independent uniform integers **in the interval [0, `maxInclusive`]** from a "source of random numbers"[14]. In the pseudocode below, which implements `RNDINT`:

[12] https://peteroupc.github.io/pseudocode.html

[13] Pedersen, K., "**Reconditioning your quantile function**", arXiv:1704.07949v3 [stat.CO], 2018. https://arxiv.org/abs/1704.07949

[14] For an exercise solved by part of the `RNDINT` pseudocode, see A. Koenig and B. E. Moo, *Accelerated C++*, 2000; see also a **blog post by Johnny Chan**.

- `NEXTRAND()` reads the next number generated by a "source of (uniform) random numbers" as defined in the appendix (an endless source of numbers, each of which is chosen independently of any other choice and with a uniform distribution). As noted in the appendix, a pseudorandom number generator can simulate this source in practice. For this method, the source can have a non-uniform instead of uniform distribution.
- `MODULUS` is the number of different outcomes possible with that source.

Specifically:

If the underlying source produces:	Then `NEXTRAND()` is:	And `MODULUS` is:
Non-uniform numbers[15].	The next bit from a new source formed by taking the underlying source's outputs as input to a ***randomness extraction***[16] technique to produce independent unbiased random bits (zeros or ones).	2.
Uniform numbers not described below.	Same as above.	2^n.
Uniform 32-bit nonnegative integers.	The next number from the source.	2^{32}.
Uniform 64-bit nonnegative integers.	The next number from the source.	2^{64}.
Uniform integers in the interval [0, n).	The next number from the source.	n.
Uniform numbers in the interval [0, 1) known to be evenly spaced by a number p (for example, dSFMT).	The next number from the source, multiplied by p.	$1/p$.
Uniform numbers in the interval [0, 1), where numbers in [0, 0.5) and those in [0.5, 1) are known to occur with equal probability (for example, Java's `Math.random()`).	0 if the source outputs a number less than 0.5, or 1 otherwise.	2.

```
METHOD RndIntHelperNonPowerOfTwo(maxInclusive)
  if maxInclusive <= MODULUS - 1:
    // NOTE: If the programming language implements
    // division with two integers by discarding the result's
    // fractional part, the division can be used as is without
    // using a "floor" function.
    nPlusOne = maxInclusive + 1
    maxexc = floor((MODULUS - 1) / nPlusOne) * nPlusOne
    while true // until a value is returned
      ret = NEXTRAND()
      if ret < nPlusOne: return ret
      if ret < maxexc: return rem(ret, nPlusOne)
    end
  else
    cx = floor(maxInclusive / MODULUS) + 1
```

[15] An example of such a source is a Gaussian noise generator. This kind of source is often called an *entropy source*.

[16] https://peteroupc.github.io/randextract.html

```
    while true // until a value is returned
       ret = cx * NEXTRAND()
       // NOTE: The addition operation below should
       // check for integer overflow and should reject the
       // number if overflow would result.
       ret = ret + RNDINT(cx - 1)
       if ret <= maxInclusive: return ret
    end
  end
END METHOD

METHOD RndIntHelperPowerOfTwo(maxInclusive)
  // NOTE: Finds the number of bits minus 1 needed
  // to represent MODULUS (in other words, the number
  // of random bits returned by NEXTRAND() ). In practice,
  // this will be a constant and can be calculated in advance.
  modBits = ln(MODULUS)/ln(2)
  // Lumbroso's Fast Dice Roller.
  x = 1
  y = 0
  nextBit = modBits
  rngv = 0
  maxIncMinus1 = maxInclusive - 1
  while true // until a value is returned
    if nextBit >= modBits
      nextBit = 0
      rngv = NEXTRAND()
    end
    nextBit = nextBit + 1
    // if modBits=1, this can read "bit=rngv"
    bit = rem(rngv, 2)
    x = x * 2
    y = y * 2 + bit
    // if modBits=1, the following line
    // can be left out
    rngv = floor(rngv / 2)
    if x > maxInclusive
      x = x - maxIncMinus1
      if y <= maxInclusive: return y
      y = y - maxIncMinus1
    end
  end
END METHOD

METHOD RNDINT(maxInclusive)
  // maxInclusive must be 0 or greater
  if maxInclusive < 0: return error
  if maxInclusive == 0: return 0
  if maxInclusive == MODULUS - 1: return NEXTRAND()
  // NOTE: Finds the number of bits minus 1 needed
  // to represent MODULUS (if it's a power of 2).
  // This will be a constant here, though.
```

```
  modBits=ln(MODULUS)/ln(2)
  if floor(modBits) == modBits // Is an integer
    return RndIntHelperPowerOfTwo(maxInclusive)
  else
    return RndIntHelperNonPowerOfTwo(maxInclusive)
  end
END METHOD
```

The table below shows algorithms that have been proposed for choosing an integer uniformly at random. Some are *unbiased* (exact) and others are biased, but in general, the algorithm can be unbiased only if it runs forever in the worst case. The algorithms listed take `n` as a parameter, where `n = maxInclusive + 1`, and thus sample from the interval [0, `n`). (The column "Unbiased?" means whether the algorithm generates random integers without bias, even if `n` is not a power of 2.)

Algorithm	Optimal?	Unbiased?	Time Complexity
Rejection sampling: Sample in a bigger range until a sampled number fits the smaller range.	Not always	Yes	Runs forever in worst case
Multiply-and-shift reduction: Generate `bignumber`, an integer made of `k` unbiased bits, where `k` is much greater than `n`, then find `(bignumber * n) >> k` (see (Lemire 2016)[17], (Lemire 2018)[18], and the "Integer Multiplication" algorithm surveyed by M. O'Neill).	No	No	Constant
Modulo reduction: Generate `bignumber` as above, then find `rem(bignumber, n)`.	No	No	Constant
Fast Dice Roller (Lumbroso 2013)[19] (see pseudocode above).	Yes	Yes	Runs forever in worst case
Algorithm FYKY (Bacher et al. 2017)[^7]. Effectively the same as replacing the lines "`if y <= maxInclusive: return y; y = y - maxIncMinus1`" in the pseudocode above with "`if y >= x: return y-x`".	Yes	Yes	Runs forever in worst case
Math Forum (2004)[20] or (Mennucci 2018)[21] (batching/recycling random bits).	Yes	Yes	Runs forever in worst case

Algorithm	Optimal?	Unbiased?	Time Complexity
"FP Multiply" surveyed by **M. O'Neill**[22].	No	No	Constant
Algorithm in "Conclusion" section by O'Neill.	No	Yes	Runs forever in worst case
"Debiased" and "Bitmask with Rejection" surveyed by M. O'Neill.	No	Yes	Runs forever in worst case

Notes:

1. **`RNDINT` as a binary tree walker.** Donald E. Knuth and Andrew C. Yao (1976)[23] showed that any algorithm that generates random integers using random unbiased bits (each bit is 0 or 1 with equal probability) can be described as a *binary tree* (also known as a *DDG tree* or *discrete distribution generating tree*). (This also applies to `RNDINT` algorithms.) Random unbiased bits trace a path in this tree, and each leaf (terminal node) in the tree represents an outcome. In the case of `RNDINT(maxInclusive)`, there are `n = maxInclusive + 1` outcomes that each occur with probability `1/n`.Knuth and Yao showed that any *optimal* DDG tree algorithm needs at least `log2(n)` and at most `log2(n) + 2` bits on average (where `log2(x) = ln(x)/ln(2)`).[24] But as they also showed, for the algorithm to be *unbiased (exact)*, it must run forever in the worst case, even if it uses few random bits on average (that is, there is no way in general to "fix" this worst case while remaining unbiased). This is because `1/n` will have an infinite run of base-2 digits except when `n` is a power of 2, so that the resulting DDG tree will have to either be infinitely deep, or include "rejection leaves" at the end of the tree.For instance, the *modulo reduction* method can be represented by a DDG tree in which rejection leaves are replaced with labeled outcomes, but the method is biased because only some outcomes can replace rejection leaves this way. For the same reason, stopping the *rejection sampler* after a fixed number of tries likewise leads to bias. However, which outcomes are biased this way depends on the algorithm.
2. **Reducing "bit waste".** Any implementation of `RNDINT` needs at least `log2(n)` bits per chosen integer on average, as noted above, but most of them use many more. There are various ways to bring an algorithm closer to `log2(n)`. They include batching, bit recycling, and randomness extraction, and they are discussed, for example, in the Math Forum page and the Lumbroso and Mennucci papers referenced above, and in Devroye and Gravel (2020, section 2.3)[25]. *Batching example*: To generate three digits from 0 through 9, we can call

[17] D. Lemire, "A fast alternative to the modulo reduction", Daniel Lemire's blog, 2016.

[18] Lemire, D., "**Fast Random Integer Generation in an Interval**", arXiv:1805.10941v4 [cs.DS], 2018. https://arxiv.org/abs/1805.10941v4

[19] Lumbroso, J., "**Optimal Discrete Uniform Generation from Coin Flips, and Applications**", arXiv:1304.1916 [cs.DS] https://arxiv.org/abs/1304.1916

[20] "**Probability and Random Numbers**", Feb. 29, 2004. https://web.archive.org/web/20170705004415/http://mathforum.org/library/drmath/view/65653.html

[21] Mennucci, A.C.G., "**Bit Recycling for Scaling Random Number Generators**", arXiv:1012.4290 [cs.IT], 2018. https://arxiv.org/abs/1012.4290

[22] https://www.pcg-random.org/posts/bounded-rands.html

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

[24] This is because the *binary entropy* of `p = 1/n` is `p * log2(1/p) = log2(n) / n`, and the sum of `n` binary entropies (for `n` outcomes with probability `1/n` each) is `log2(n) = ln(n)/ln(2)`.

[25] Devroye, L., Gravel, C., "**Random variate generation using only finitely many unbiased, independently and identically distributed random bits**", arXiv:1502.02539v6 [cs.IT], 2020. https://arxiv.org/abs/1502.02539v6

`RNDINT(999)` to generate an integer in [0, 999], then break the number it returns into three digits.

3. **Simulating dice.** If we have a (virtual) fair p-sided die, how can we use it to simulate rolls of a k-sided die? This can't be done without "wasting" randomness, unless "every prime number dividing k also divides p" (see "**Simulating a dice with a dice**[26]" by B. Kloeckner, 2008). However, *randomness extraction* (see my **Note on Randomness Extraction**[27]) can turn die rolls into unbiased bits, so that the discussion earlier in this section applies.

4.2 `RNDINTRANGE`: Random Integers in [N, M]

The naïve way of generating a **random integer in the interval [`minInclusive, maxInclusive`]**, shown below, works well for nonnegative integers and arbitrary-precision integers.

```
METHOD RNDINTRANGE(minInclusive, maxInclusive)
  // minInclusive must not be greater than maxInclusive
  if minInclusive > maxInclusive: return error
  return minInclusive + RNDINT(maxInclusive - minInclusive)
END METHOD
```

The naïve approach won't work as well, though, if the integer format can express negative and nonnegative integers and the difference between `maxInclusive` and `minInclusive` exceeds the highest possible integer for the format. For integer formats that can express—

1. every integer in the interval [-1 - `MAXINT`, `MAXINT`] (for example, Java `int`, `short`, or `long`), or
2. every integer in the interval [-`MAXINT`, `MAXINT`] (for example, Java `float` and `double` and .NET's implementation of `System.Decimal`),

where `MAXINT` is an integer greater than 0, the following pseudocode for `RNDINTRANGE` can be used.

```
METHOD RNDINTRANGE(minInclusive, maxInclusive)
  // minInclusive must not be greater than maxInclusive
  if minInclusive > maxInclusive: return error
  if minInclusive == maxInclusive: return minInclusive
  if minInclusive==0: return RNDINT(maxInclusive)
  // Difference does not exceed maxInclusive
  if minInclusive > 0 or minInclusive + MAXINT >= maxInclusive
      return minInclusive + RNDINT(maxInclusive - minInclusive)
  end
  while true // until a value is returned
    ret = RNDINT(MAXINT)
    // NOTE: For case 1, use the following line:
    if RNDINT(1) == 0: ret = -1 - ret
    // NOTE: For case 2, use the following three lines
    // instead of the preceding line; these lines
    // avoid negative zero
    // negative = RNDINT(1) == 0
    // if negative: ret = 0 - ret
    // if negative and ret == 0: continue
    if ret >= minInclusive and ret <= maxInclusive: return ret
  end
END METHOD
```

[26] https://perso.math.u-pem.fr/kloeckner.benoit/papiers/DiceSimulation.pdf
[27] https://peteroupc.github.io/randextract.html

4.3 `RNDINTEXC`: Random Integers in [0, N)

`RNDINTEXC(maxExclusive)`, which generates a **random integer in the interval [0, `maxExclusive`)**, can be implemented as follows[28]:

```
METHOD RNDINTEXC(maxExclusive)
   if maxExclusive <= 0: return error
   return RNDINT(maxExclusive - 1)
END METHOD
```

> **Note:** `RNDINTEXC` is not given as the core random generation method because it's harder to fill integers in popular integer formats with random bits with this method.

4.4 `RNDINTEXCRANGE`: Random Integers in [N, M)

`RNDINTEXCRANGE` returns a **random integer in the interval [`minInclusive`, `maxExclusive`)**. It can be implemented using **`RNDINTRANGE`**, as the following pseudocode demonstrates.

```
METHOD RNDINTEXCRANGE(minInclusive, maxExclusive)
   if minInclusive >= maxExclusive: return error
   if minInclusive >=0: return RNDINTRANGE(
      minInclusive, maxExclusive - 1)
   while true // until a value is returned
     ret = RNDINTRANGE(minInclusive, maxExclusive)
     if ret < maxExclusive: return ret
   end
END METHOD
```

4.5 Uniform Random Bits

The idiom `RNDINT((1 << b) - 1)` is a naïve way of generating a **uniform random `b`-bit integer** (with maximum $2^b - 1$).

In practice, memory is usually divided into *bytes*, or 8-bit integers in the interval [0, 255]. In this case, a block of memory can be filled with random bits—

- by setting each byte in the block to `RNDINT(255)`, or
- via a PRNG (or another device or program that simulates a "source of random numbers"), if it outputs one or more 8-bit chunks at a time.

4.6 Examples of Using the `RNDINT` Family

1. To choose either -1 or 1 with equal probability (the *Rademacher distribution*), one of the following idioms can be used: `(RNDINT(1) * 2 - 1)` or `(RNDINTEXC(2) * 2 - 1)`.
2. To generate a random integer that's divisible by a positive integer (`DIV`), generate the integer with any method (such as `RNDINT`), let `X` be that integer, then generate `X - rem(X, DIV)` if `X >= 0`, or `X - (DIV - rem(abs(X), DIV))` otherwise. (Depending on the method, the resulting integer may be out of range, in which case this procedure is to be repeated.)

[28] A naïve `RNDINTEXC` implementation often seen in certain languages like JavaScript is the idiom `floor(Math.random() * maxExclusive)`, where `Math.random()` is any method that outputs a floating-point number that behaves like an independent uniform random variate in the interval [0, 1). However, no implementation of `Math.random()` can choose from all real numbers in [0, 1), so this idiom can bias some results over others depending on the value of `maxExclusive`. For example, if `Math.random()` is implemented as `RNDINT(X - 1)/X` and `X` is not divisible by `maxExclusive`, the result will be biased. Also, an implementation might pre-round `Math.random() * maxExclusive` (before the `floor`) to the closest number it can represent; in rare cases, that might be `maxExclusive` for certain rounding modes. If an application is concerned about these issues, it should treat the `Math.random()` implementation as simulating the "source of random numbers" for `RNDINT` and implement `RNDINTEXC` through `RNDINT` instead.

3. A random 2-dimensional point on an N×M grid can be expressed as a single integer as follows:
   - To generate a random N×M point `P`, generate `P = RNDINT(N * M - 1)` (`P` is thus in the interval [0, `N * M`)).
   - To convert a point `P` to its 2D coordinates, generate `[rem(P, N), floor(P / N)]`. (Each coordinate starts at 0.)
   - To convert 2D coordinates `coord` to an N×M point, generate `P = coord[1] * N + coord[0]`.
4. To simulate rolling an N-sided die (N greater than 1): `RNDINTRANGE(1, N)`, which chooses a number in the interval [1, N] with equal probability.
5. To generate a random integer with one base-10 digit: `RNDINTRANGE(0, 9)`.
6. To generate a random integer with N base-10 digits (where N is 2 or greater), where the first digit can't be 0: `RNDINTRANGE(pow(10, N-1), pow(10, N) - 1)`.
7. To choose a number in the interval [`mn`, `mx`), with equal probability, in increments equal to `step`: `mn+step*RNDINTEXC(ceil((mx-mn)/(1.0*step)))`.
8. To choose an integer in the interval [0, `X`) at random:
   - And favor numbers in the middle: `floor((RNDINTEXC(X) + RNDINTEXC(X)) / 2)`.
   - And favor high numbers: `max(RNDINTEXC(X), RNDINTEXC(X))`.
   - And favor low numbers: `min(RNDINTEXC(X), RNDINTEXC(X))`.
   - And strongly favor high numbers: `max(RNDINTEXC(X), RNDINTEXC(X), RNDINTEXC(X))`.
   - And strongly favor low numbers: `min(RNDINTEXC(X), RNDINTEXC(X), RNDINTEXC(X))`.

5 Randomization Techniques

This section describes commonly used randomization techniques, such as shuffling, selection of several unique items, and creating random strings of text.

5.1 Boolean (True/False) Conditions

To generate a condition that is true at the specified probabilities, use the following idioms in an `if` condition:

- True or false with equal probability: `RNDINT(1) == 0`.
- True with X percent probability: `RNDINTEXC(100) < X`.
- True with probability X/Y (a *Bernoulli trial*): `RNDINTEXC(Y) < X`.
- True with odds of X to Y: `RNDINTEXC(X + Y) < X`.

The following helper method generates 1 with probability `x`/`y` and 0 otherwise:

```
METHOD ZeroOrOne(x,y)
  if RNDINTEXC(y)<x: return 1
  return 0
END METHOD
```

The method can also be implemented in the following way (as pointed out by Lumbroso (2013, Appendix B)[29]):

```
// NOTE: Modified from Lumbroso
// Appendix B to add 'z==0' and error checks
METHOD ZeroOrOne(x,y)
  if y <= 0: return error
  if x==y: return 1
  z = x
  while true // until a value is returned
    z = z * 2
```

[29] Lumbroso, J., "**Optimal Discrete Uniform Generation from Coin Flips, and Applications**", arXiv:1304.1916 [cs.DS] https://arxiv.org/abs/1304.1916

```
    if z >= y
      if RNDINT(1) == 0: return 1
      z = z - y
    else if z == 0 or RNDINT(1) == 0: return 0
  end
END METHOD
```

Note: Probabilities can be rational or irrational numbers. Rational probabilities are of the form `n/d` and can be simulated with `ZeroOrOne` above. Irrational probabilities (such as `exp(-x/y)` or `ln(2)`) have an infinite digit expansion (`0.ddddd....`), and they require special algorithms to simulate; see "**Algorithms for General Irrational Constants**[30]" and "**Algorithms for Specific Constants**[31]" in my page on Bernoulli Factory algorithms.

Examples:

- True with probability 3/8: `RNDINTEXC(8) < 3`.
- True with odds of 100 to 1: `RNDINTEXC(101) < 1`.
- True with 20% probability: `RNDINTEXC(100) < 20`.
- To generate a random integer in [0, `y`), or -1 instead if that number would be less than `x`, using fewer random bits than the naïve approach: `if ZeroOrOne(x, y) == 1: return -1; else: return RNDINTEXCRANGE(x, y)`.

5.2 Random Sampling

This section contains ways to choose one or more items from among a collection of them, where each item in the collection has the same chance to be chosen as any other. This is called *random sampling* and can be done *with replacement* or *without replacement*.

5.2.1 Sampling With Replacement: Choosing a Random Item from a List

Sampling with replacement essentially means taking a random item and putting it back. To choose a random item from a list—

- whose size is known in advance, use the idiom `list[RNDINTEXC(size(list))]`; or
- whose size is not known in advance, generate `RandomKItemsFromFile(file, 1)`, in **pseudocode given later** (the result will be a 1-item list or be an empty list if there are no items).

5.2.2 Sampling Without Replacement: Choosing Several Unique Items

Sampling without replacement essentially means taking a random item *without* putting it back. There are several approaches for doing a uniform random choice of `k` unique items or values from among `n` available items or values, depending on such things as whether `n` is known and how big `n` and `k` are.

1. **If `n` is not known in advance:** Use the *reservoir sampling* method; see the `RandomKItemsFromFile` method, in **pseudocode given later**.
2. **If `n` is relatively small (for example, if there are 200 available items, or there is a range of numbers from 0 through 200 to choose from):**
   - If items have to be chosen from a list **in relative (index) order**, or if `n` is 1, then use `RandomKItemsInOrder` (given later).
   - Otherwise, if the order of the sampled items is unimportant, and each item can be derived from its *index* (the item's position as an integer starting at 0) without looking it up in a list: Use the `RandomKItemsFromFile` method.[32]

[30] https://peteroupc.github.io/bernoulli.html#Algorithms_for_General_Irrational_Constants
[31] https://peteroupc.github.io/bernoulli.html#Algorithms_for_Specific_Constants
[32] The user "BVtp" from the *Stack Overflow* community led me to this insight.

- Otherwise, if `k` is much smaller than `n`, proceed as in item 3 instead.
- Otherwise, any of the following will choose `k` items in random order:
  - Store all the items in a list, **shuffle** that list, then choose the first `k` items from that list.
  - If the items are already stored in a list and the list's order can be changed, then shuffle that list and choose the first `k` items from the shuffled list.
  - If the items are already stored in a list and the list's order can't be changed, then store the indices to those items in another list, shuffle the latter list, then choose the first `k` indices (or the items corresponding to those indices) from the latter list.

3. **If `k` is much smaller than `n` and the order of the items must be random or is unimportant:**
   1. **If the items are stored in a list whose order can be changed:** Do a *partial shuffle* of that list, then choose the *last* `k` items from that list. A *partial shuffle* proceeds as given in the section "**Shuffling**", except the partial shuffle stops after `k` swaps have been made (where swapping one item with itself counts as a swap).
   2. **Otherwise**: Create another empty list `newlist`, and create a key/value data structure such as a hash table. Then, for each integer `i` in the interval [0, $k - 1$], do `j = RNDINTEXC(n-i); AddItem(newlist, HGET(j,j)); HSET(j,HGET(n-i-1,n-i-1))`, where `HSET(k,v)` sets the item with key `k` in the hash table to `v`, and `HGET(k,v)` gets the item with key `k` in that table, or returns `v` if there is no such item (Ting 2021)[33]. The new list stores the indices to the chosen items, in random order.
4. **If `n - k` is much smaller than `n`, the items are stored in a list, and the order of the sampled items is unimportant:**
   1. **If the list's order can be changed:** Do a *partial shuffle* of that list, except that `n-k` rather than `k` swaps are done, then choose the *first* `k` items from that list. (Note 5 in "Shuffling" can't be used.)
   2. Otherwise, **if `n` is not very large (for example, less than 5000):** Store the indices to those items in another list, do a *partial shuffle* of the latter list, except that `n-k` rather than `k` swaps are done, then choose the *first* `k` indices (or the items corresponding to those indices) from the latter list. (Note 5 in "Shuffling" can't be used.)
   3. **Otherwise**: Proceed as in item 5 instead.
5. **Otherwise (for example, if 32-bit or larger integers will be chosen so that `n` is 2^{32}, or if `n` is otherwise very large):**
   - If the items have to be chosen **in relative (index) order**: Let `n2 = floor(n/2)`. Generate `h = PolyaEggenberger(k, n2, n, -1)`. Sample `h` integers in relative order from the list `[0, 1, ..., n2 - 1]` by doing a recursive run of this algorithm (items 1 to 5), then sample `k - h` integers in relative order from the list `[n2, n2 + 1, ..., n - 1]` by running this algorithm recursively. The integers chosen this way are the indices to the desired items in relative (index) order (Sanders et al. 2019)[34].
   - Otherwise, create a data structure to store the indices to items already chosen. When a new index to an item is randomly chosen, add it to the data structure if it's not already there, or if it is, choose a new random index. Repeat this process until `k` indices were added to the data structure this way. Examples of suitable data structures are—
     - a **hash table**[35],
     - a compressed bit set (e.g, "roaring bitmap", EWAH), and
     - a self-sorting data structure such as a **red–black tree**[36], if the random items are to be retrieved in sorted order or in index order.

     Many applications require generating "unique random" values to identify database records or other shared resources, among other reasons. For ways to generate such values, see my **recom-**

[33]Daniel Ting, "**Simple, Optimal Algorithms for Random Sampling Without Replacement**", arXiv:2104.05091, 2021. https://arxiv.org/abs/2104.05091

[34]Sanders, P., Lamm, S., et al., "**Efficient Parallel Random Sampling – Vectorized, Cache-Efficient, and Online**", arXiv:1610.0514v2 [cs.DS], 2019. https://arxiv.org/abs/1610.0514v2

[35]https://en.wikipedia.org/wiki/Hash_table

[36]https://en.wikipedia.org/wiki/Red%E2%80%93black_tree

mendation document[37].

5.2.3 Shuffling

The **Fisher–Yates shuffle method**[38] shuffles a list (puts its items in a random order) such that all permutations (arrangements) of that list occur with the same probability. However, that method is also easy to write incorrectly — see also (Atwood 2007)[39]. The following pseudocode is designed to shuffle a list's contents.

```
METHOD Shuffle(list)
   // NOTE: Check size of the list early to prevent
   // `i` from being less than 0 if the list's size is 0 and
   // `i` is implemented using a nonnegative integer
   // type available in certain programming languages.
   if size(list) >= 2
      // Set i to the last item's index
      i = size(list) - 1
      while i > 0
         // Choose an item ranging from the first item
         // up to the item given in `i`. Observe that the item
         // at i+1 is excluded.
         j = RNDINTEXC(i + 1)
         // Swap item at index i with item at index j;
         // in this case, i and j may be the same
         tmp = list[i]
         list[i] = list[k]
         list[k] = tmp
         // Move i so it points to the previous item
         i = i - 1
      end
   end
   // NOTE: An implementation can return the
   // shuffled list, as is done here, but this is not required.
   return list
END METHOD
```

Notes:

1. `j = RNDINTEXC(i + 1)` can't be replaced with `j = RNDINTEXC(size(list))` since it introduces biases. If that line is replaced with `j = RNDINTEXC(i)`, the result is Sattolo's algorithm (which chooses from among permutations with cycles), rather than a Fisher–Yates shuffle.
2. When it comes to shuffling, the choice of pseudorandom number generator (or whatever is simulating a "source of random numbers") is important; see my **recommendation document on shuffling**[40].
3. A shuffling algorithm that can be carried out in parallel is described in (Bacher et al., 2015)[41].

[37] https://peteroupc.github.io/random.html#Unique_Random_Identifiers
[38] https://en.wikipedia.org/wiki/Fisher-Yates_shuffle
[39] Jeff Atwood, "**The danger of naïveté**", Dec. 7, 2007. https://blog.codinghorror.com/the-danger-of-naivete/
[40] https://peteroupc.github.io/random.html#Shuffling
[41] Bacher, A., Bodini, O., et al., "**MergeShuffle: A Very Fast, Parallel Random Permutation Algorithm**", arXiv:1508.03167 [cs.DS], 2015. https://arxiv.org/abs/1508.03167

4. A *derangement* is a permutation where every item moves to a different position. A random derangement can be generated as follows (Merlini et al. 2007)[42]: (1) modify `Shuffle` by adding the following line after `j = RNDINTEXC(i + 1): if i == list[j]: return nothing`, and changing `while i > 0` to `while i >= 0`; (2) use the following pseudocode with the modified `Shuffle` method: `while True; list = []; for i in 0...n: AddItem(list, i); s=Shuffle(list); if s!=nothing: return s; end`.
5. Ting (2021)[43] showed how to reduce the space complexity of shuffling via a *hash table*: Modify `Shuffle` as follows: (1) Create a hash table at the start of the method; (2) instead of the swap `tmp = list[i]; list[i] = list[j]; list[j] = tmp`, use the following: `list[i] = HGET(j,j); HSET(k,HGET(i,i)); if k==i: HDEL(i)`, where `HSET(k,v)` sets the item with key `k` in the hash table to `v`; `HGET(k,v)` gets the item with key `k` in that table, or returns `v` if there is no such item; and `HDEL(k)` deletes the item with key `k` from that table. (The hash table can instead be any key/value map structure, including a red–black tree. This can be combined with note 4 to generate derangements, except replace `list[j]` in note 4 with `HGET(j,j)`.)

5.2.4 Random Character Strings

To generate a random string of characters:

1. Prepare a list of the characters (including letters or digits) the string can have. Examples are given later in this section.
2. Build a new string whose characters are chosen at random from that character list. The method, shown in the pseudocode below, demonstrates this. The method samples characters at random with replacement, and returns a list of the sampled characters. (How to convert this list to a text string depends on the programming language and is outside the scope of this page.) The method takes two parameters: `characterList` is the list from step 1, and `stringSize` is the number of random characters.

```
METHOD RandomString(characterList, stringSize)
  i = 0
  newString = NewList()
  while i < stringSize
    // Choose a character from the list
    randomChar = characterList[RNDINTEXC(size(characterList))]
    // Add the character to the string
    AddItem(newString, randomChar)
    i = i + 1
  end
  return newString
END METHOD
```

The following are examples of character lists:

1. For an *alphanumeric string*, or string of letters and digits, the characters can be the basic digits "0" to "9" (U+0030-U+0039, nos. 48-57), the basic upper case letters "A" to "Z" (U+0041-U+005A, nos. 65-90), and the basic lower case letters "a" to "z" (U+0061-U+007A, nos. 96-122), as given in the Unicode Standard.
2. For a base-10 digit string, the characters can be the basic digits only.

[42] Merlini, D., Sprugnoli, R., Verri, M.C., "An Analysis of a Simple Algorithm for Random Derangements", 2007.

[43] Daniel Ting, "**Simple, Optimal Algorithms for Random Sampling Without Replacement**", arXiv:2104.05091, 2021. https://arxiv.org/abs/2104.05091

3. For a base-16 digit (hexadecimal) string, the characters can be the basic digits as well as the basic letters "A" to "F" or "a" to "f" (not both).

   Notes:

   1. If the list of characters is fixed, the list can be created in advance at runtime or compile time, or (if every character takes up the same number of code units) a string type as provided in the programming language can be used to store the list as a string.
   2. **Unique random strings:** Generating character strings that are not only random, but also unique, can be done by storing a list (such as a hash table) of strings already generated and checking newly generated strings against that list. However, if the unique values will identify something, such as database records or user accounts, an application may care about the choice of PRNG (or other device or program that simulates a "source of random numbers"), so that using *random* unique values might not be best; see my **recommendation document**[44].
   3. **Word generation:** This technique could also be used to generate "pronounceable" words, but this is less flexible than other approaches; see also "**Markov Chains**".

5.2.5 Pseudocode for Random Sampling

The following pseudocode implements two methods:

1. `RandomKItemsFromFile` implements ***reservoir sampling***[45]; it chooses up to `k` random items from a file of indefinite size (`file`). Although the pseudocode refers to files and lines, the technique works whenever items are retrieved one at a time from a data set or list whose size is not known in advance. In the pseudocode, `ITEM_OUTPUT(item, thisIndex)` is a placeholder for code that returns the item to store in the list; this can include the item's value, its index starting at 0, or both.
2. `RandomKItemsInOrder` returns a list of up to `k` random items from the given list (`list`), in the order in which they appeared in the list. It is based on a technique presented in (Devroye 1986)[46], p. 620.

```
METHOD RandomKItemsFromFile(file, k)
  list = NewList()
  j = 0
  index = 0
  while true // until a value is returned
    // Get the next line from the file
    item = GetNextLine(file)
    thisIndex = index
    index = index + 1
    // If the end of the file was reached, break
    if item == nothing: break
    // NOTE: The following line is OPTIONAL
    // and can be used to choose only random lines
    // in the file that meet certain criteria,
    // expressed as MEETS_CRITERIA below.
    // ------
    // if not MEETS_CRITERIA(item): continue
    // ------
    if j < k // phase 1 (fewer than k items)
      AddItem(list, ITEM_OUTPUT(item, thisIndex))
```

[44] https://peteroupc.github.io/random.html#Unique_Random_Identifiers
[45] https://en.wikipedia.org/wiki/Reservoir_sampling
[46] Devroye, L., ***Non-Uniform Random Variate Generation***, 1986.

```
      j = j + 1
    else // phase 2
      j = RNDINT(thisIndex)
      if j < k: list[j] = ITEM_OUTPUT(item, thisIndex)
    end
  end
  // NOTE: Shuffle at the end in case k or
  // fewer lines were in the file, since in that
  // case the items would appear in the same
  // order as they appeared in the file
  // if the list weren't shuffled.  This line
  // can be removed, however, if the order of
  // the items in the list is unimportant.
  if size(list)>=2: Shuffle(list)
  return list
end

METHOD RandomKItemsInOrder(list, k)
  n = size(list)
  // Special case if k is 1
  if k==1: return [list[RNDINTEXC(n)]]
  i = 0
  kk = k
  ret = NewList()
  while i < n and size(ret) < k
    u = RNDINTEXC(n - i)
    if u <= kk
      AddItem(ret, list[i])
      kk = kk - 1
    end
    i = i + 1
  end
  return ret
END METHOD
```

Examples:

1. Removing `k` random items from a list of `n` items (`list`) is equivalent to generating a new list by `RandomKItemsInOrder(list, n - k)`.
2. **Filtering:** If an application needs to sample the same list (with or without replacement) repeatedly, but only from among a selection of that list's items, it can create a list of items it wants to sample from (or a list of indices to those items), and sample from the new list instead.[47] This won't work well, though, for lists of indefinite or very large size.

5.3 Rejection Sampling

Rejection sampling is a simple and flexible approach for generating random content that meets certain requirements. To implement rejection sampling:

1. Generate the random content (such as a number or text string) by any method and with any distribution and range.
2. If the content doesn't meet predetermined criteria, go to step 1.

[47] See also the *Stack Overflow* question "Random index of a non zero value in a numpy array".

Example criteria include checking—

- whether a number generated this way—
  - is not less than a minimum threshold (*left-truncation*),
  - is not greater than a maximum threshold (*right-truncation*),
  - is prime,
  - is divisible or not by certain numbers,
  - is not among numbers chosen recently by the sampling method,
  - was not already chosen (with the aid of a hash table, red–black tree, or similar structure),
  - was not chosen more often in a row than desired, or
  - is not included in a "blacklist" of numbers,
- whether a point generated this way is sufficiently distant from previous random points (with the aid of a KD-tree or similar structure),
- whether a point generated this way lies in a simple or complex shape,
- whether a text string generated this way matches a regular expression, or
- two or more of the foregoing criteria.

(KD-trees, hash tables, red–black trees, prime-number testing algorithms, and regular expressions are outside the scope of this document.)

Notes:

1. The running time for rejection sampling depends on the acceptance rate, that is, how often the sampler accepts a sampled outcome rather than rejecting it. In general, this rate is the number of acceptable outcomes divided by the total number of outcomes.
2. All rejection sampling strategies have a chance to reject data, so they all have a *variable running time* (in fact, they could run indefinitely). But graphics processing units (GPUs) and other devices that run multiple tasks at once work better if all the tasks finish their work at the same time. This is not possible if they all implement a rejection sampling technique because of its variable running time. If each iteration of the rejection sampler has a low rejection rate, one solution is to have each task run one iteration of the sampler, with its own "source of random numbers" (such as numbers generated from its own PRNG), then to take the first sampled number that hasn't been rejected this way by a task (which can fail at a very low rate).[48]

5.4 Random Walks

A *random walk* is a process with random behavior over time. A simple form of random walk involves choosing, at random, a number that changes the state of the walk. The pseudocode below generates a random walk with n steps, where `STATEJUMP()` is the next number to add to the current state (see examples later in this section).

```
METHOD RandomWalk(n)
  // Create a new list with an initial state
  list=[0]
  // Add 'n' new numbers to the list.
  for i in 0...n: AddItem(list, list[i] + STATEJUMP())
  return list
END METHOD
```

Notes:

[48] S. Linderman, "A Parallel Gamma Sampling Implementation", Laboratory for Independent Probabilistic Systems Blog, Feb. 21, 2013, illustrates one example, a GPU-implemented sampler of gamma-distributed random variates.

1. A **white noise process** is simulated by creating a list of independent random variates generated in the same way. Such a process generally models behavior over time that does not depend on the time or the current state. One example is `ZeroOrOne(px,py)` (for modeling a *Bernoulli process*, where each number is either 1 with probability `px/py` or 0 otherwise).
2. A useful reference here is De Bruyne et al. (2021)[49].

Examples:

1. If `STATEJUMP()` is `RNDINT(1) * 2 - 1`, the random walk generates numbers that each differ from the last by -1 or 1, chosen at random.
2. If `STATEJUMP()` is `ZeroOrOne(px,py) * 2 - 1`, the random walk generates numbers that each differ from the last by either 1 with probability `px/py` or -1 otherwise.
3. **Binomial process:** If `STATEJUMP()` is `ZeroOrOne(px,py)`, the random walk advances the state with probability `px/py`.

5.5 Random Dates and Times

Pseudocode like the following can be used to choose a **random date-and-time** bounded by two dates-and-times (`date1`, `date2`): `dtnum1 = DATETIME_TO_NUMBER(date1); dtnum2 = DATETIME_TO_NUMBER(date2); num = RNDINTRANGE(date1, date2); result = NUMBER_TO_DATETIME(num)`.

In that pseudocode, `DATETIME_TO_NUMBER` and `NUMBER_TO_DATETIME` convert a date-and-time to or from a number, respectively, at the required granularity, for instance, month, day, or hour granularity (the details of such conversion depend on the date-and-time format and are outside the scope of this document). Instead of `RNDINTRANGE(date1, date2)`, any other random selection strategy can be used.

5.6 Randomization in Statistical Testing

Statistical testing uses shuffling and *bootstrapping* to help draw conclusions on data through randomization.

- **Shuffling** is used when each item in a data set belongs to one of several mutually exclusive groups. Here, one or more **simulated data sets** are generated by shuffling the original data set and regrouping each item in the shuffled data set in order, such that the number of items in each group for the simulated data set is the same as for the original data set.
- ***Bootstrapping***[50] is a method of creating one or more random samples (simulated data sets) of an existing data set, where the items in each sample are chosen **at random with replacement**. (Each random sample can contain duplicates this way.) See also (Brownlee 2018)[51].

After creating the simulated data sets, one or more statistics, such as the mean, are calculated for each simulated data set as well as the original data set, then the statistics for the simulated data sets are compared with those of the original (such comparisons are outside the scope of this document).

5.7 Markov Chains

A **Markov chain**[52] models one or more *states* (for example, individual letters or syllables), and stores the probabilities to transition from one state to another (for example, "b" to "e" with a chance of 20 percent, or "b" to "b" with a chance of 1 percent). Thus, each state can be seen as having its own list of *weights* for each relevant state transition (see "**Weighted Choice With Replacement**). For example, a Markov chain for generating **"pronounceable" words**, or words similar to natural-language words, can include"start" and "stop" states for the start and end of the word, respectively.

[49] De Bruyne, B., et al., "Generating discrete-time constrained random walks and Lévy flights, arXiv:2104.06145 (2021).

[50] https://en.wikipedia.org/wiki/Bootstrapping_%28statistics%29

[51] Brownlee, J. "**A Gentle Introduction to the Bootstrap Method**", *Machine Learning Mastery*, May 25, 2018. https://machinelearningmastery.com/a-gentle-introduction-to-the-bootstrap-method/

[52] https://en.wikipedia.org/wiki/Markov_chain

An algorithm called *coupling from the past* (Propp and Wilson 1996)[53] can sample a state from a Markov chain's *stationary distribution*, that is, the chain's steady state, by starting multiple chains at different states and running them until they all reach the same state at the same time. However, stopping the algorithm early can introduce bias unless precautions are taken (Fill 1998)[54]. The algorithm works correctly if the chain has a finite number of states and is not periodic, and each state is reachable from every other.

The following pseudocode implements coupling from the past. In the method, `StateCount` is the number of states in the Markov chain, `UPDATE(chain, state, random)` transitions the Markov chain to the next state given the current state and random variates, and `RANDOM()` generates one or more random variates needed by `UPDATE`.

```
METHOD CFTP(chain)
   states=[]
   numstates=StateCount(chain)
   done=false
   randoms=[]
   while not done
      // Start multiple chains at different states.  NOTE:
      // If the chain is monotonic (meaning the states
      // are ordered and, whenever state A is less
      // than state B, A's next state is never higher than
      // B's next state), then just two chains can be
      // created instead, starting
      // at the first and last state, respectively.
      for i in 0...numstates: AddItem(states, i)
      // Update each chain with the same randomness
      AddItem(randoms, RANDOM())
      for k in 0...size(randoms):
         for i in 0...numstates: states[i]=
            UPDATE(chain, states[i], randoms[size(randoms)-1-k])
      end
      // Stop when all states are the same
      fs=states[0]
      done=true
      for i in 1...numstates: done=(done and states[i]==fs)
   end
   return states[0] // Return the steady state
END METHOD
```

> **Note:** A **discrete phase-type distribution** consists of a Markov chain, a start state, and an end state. It models the (random) number of steps, minus 1, needed for the Markov chain to move from the start state to the end state.

5.8 Random Graphs

A *graph* is a listing of points and the connections between them. The points are called *vertices* and the connections, *edges*.

A convenient way to represent a graph is an *adjacency matrix*. This is an $n \times n$ matrix with n rows and n columns (signifying n vertices in the graph). For simple graphs, an adjacency matrix contains only 1s and

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

[54] Fill, J.A., "**An interruptible algorithm for perfect sampling via Markov chains**", *Annals of Applied Probability* 8(1), 1998. https://projecteuclid.org/euclid.aoap/1027961037

0s — for the cell at row r and column c, a 1 means there is an edge pointing from vertex r to vertex c, and a 0 means there are none.

In this section, `Zeros(n)` creates an `n`×`n` zero matrix (such as a list consisting of `n` lists, each of which contains `n` zeros).

The following method generates a random `n`-vertex graph that follows the model G(n, p) (also known as the *Gilbert model* (Gilbert 1959)[55]), where each edge is drawn with probability `px/py` (Batagelj and Brandes 2005)[56].

```
METHOD GNPGraph(n, px, py)
    graph=Zeros(n)
    for i in 2...n
       j = i
       while j > 0
          j = j - 1 - min(NegativeBinomialInt(1, px, py), j - 1)
          if j > 0
             // Build an edge
             graph[i][j]=1
             graph[j][i]=1
          end
       end
    end
    return graph
END METHOD
```

Other kinds of graphs are possible, including *Erdős–Rényi graphs* (choose `m` random edges uniformly without replacement, given an n×n adjacency matrix), Chung–Lu graphs, preferential attachment graphs, and more. For example, a *mesh graph* is a graph in the form of a rectangular mesh, where vertices are the corners and edges are the sides of the mesh's rectangles. A random *maze* is a random spanning tree (Costantini 2020)[^29] of a mesh graph. Penschuck et al. (2020)[57] give a survey of random graph generation techniques.

5.9 A Note on Sorting Random Variates

In general, sorting random variates is no different from sorting any other data. (Sorting algorithms are outside this document's scope.)[58]

6 General Non-Uniform Distributions

Some applications need to choose random values such that some of them have a greater chance to be chosen than others (a *non-uniform* distribution). Most of the techniques in this section show how to use the **uniform random integer methods** to generate such random values.

6.1 Weighted Choice

The weighted choice method generates a random item or value from among a collection of them with separate probabilities of each item or value being chosen. There are several kinds of weighted choice.

[55] E. N. Gilbert, "Random Graphs", *Annals of Mathematical Statistics* 30(4), 1959.

[56] V. Batagelj and U. Brandes, "Efficient generation of large random networks", Phys.Rev. E 71:036113, 2005.

[57] Penschuck, M., et al., "**Recent Advances in Scalable Network Generation**", arXiv:2003.00736v1 [cs.DS], 2020. https://arxiv.org/abs/2003.00736v1

[58] Jon Louis Bentley and James B. Saxe, "Generating Sorted Lists of Random Numbers", *ACM Trans. Math. Softw.* 6 (1980), pp. 359-364, describes a way to generate certain kinds of random variates in sorted order, but it's not given here because it relies on generating real numbers in the interval [0, 1], which is inherently imperfect because computers can't choose among all real numbers between 0 and 1, and there are infinitely many of them.

6.1.1 Weighted Choice With Replacement

The first kind is called weighted choice *with replacement* (which can be thought of as drawing a ball, then putting it back) or a *categorical distribution*, where the probability of choosing each item doesn't change as items are chosen. In the following pseudocode:

- `WeightedChoice` takes a single list `weights` of weights (integers 0 or greater) and returns the *index* of a weight from that list. The greater the weight, the greater the chance its index will be chosen.
- `CumulativeWeightedChoice` takes a single list `weights` of N *cumulative weights*; they start at 0 and the next weight is not less than the previous. Returns a number in the interval [0, N - 1).
- `NormalizeRatios` calculates a list of integers with the same proportions as the given list of rational numbers (numbers of the form `x/y`). This is useful for converting rational weights to integer weights for use in `WeightedChoice`.
- `gcd(a, b)` is the greatest common divisor between two numbers (where `gcd(0, a) = gcd(a, 0) = a` whenever `a >= 0`).

```
METHOD WChoose(weights, value)
    // Choose the index according to the given value
    runningValue = 0
    for i in 0...size(weights) - 1
       if weights[i] > 0
          newValue = runningValue + weights[i]
          // NOTE: Includes start, excludes end
          if value < newValue: break
          runningValue = newValue
       end
    end
    // Should not happen with integer weights
    return error
END METHOD

METHOD WeightedChoice(weights)
    return WChoose(weights,
        RNDINTEXC(Sum(weights)))
END METHOD

METHOD CumulativeWeightedChoice(weights)
    if size(weights)==0 or weights[0]!=0: return error
    value = RNDINTEXC(weights[size(weights) - 1])
    // Choose the index according to the given value
    for i in 0...size(weights) - 1
       // Choose only if difference is positive
       if weights[i] < weights[i+1] and
           weights[i]>=value: return i
    end
    return 0
END METHOD

METHOD NormalizeRatios(ratios)
  prod=1; gc=0
  for r in ratios: prod*=r[1] // Multiply denominators
  weights=[]
```

```
  for r in ratios
     rn = floor(r * prod)
     gc = gcd(rn, gc); AddItem(weights, rn)
  end
  if gc==0: return weights
  for i in 0...size(weights): weights[i]=floor(weights[i]/gc)
  return weights
END METHOD
```

The following are various ways to implement `WeightedChoice`. Many of them require using a special data structure.

Algorithm	Notes
Linear search	See the pseudocode for `WeightedChoice` above.
Linear search with cumulative weights	Calculates a list of cumulative weights (also known as a *cumulative distribution table* or *CDT*), then generates, at random, a number less than the sum of (original) weights (which should be an integer if the weights are integers), then does a linear scan of the new list to find the first item whose cumulative weight exceeds the generated number.
Fast Loaded Dice Roller (Saad et al., 2020a)[59].	Uses integer weights only, and samples using random bits ("fair coins"). This sampler comes within 6 bits, on average, of the optimal number of random bits per sample.
Samplers described in (Saad et al., 2020b)[60]	Uses integer weights only, and samples using random bits. The samplers come within 2 bits, on average, of the optimal number of random bits per sample as long as the sum of the weights is of the form 2^k or $2^k - 2^m$.
Rejection sampling	Given a list (`weights`) of `n` weights: (1) find the highest weight and call it *max*; (2) set *i* to `RNDINT(n - 1)`; (3) With probability `weights[i]/max` (for example, if `ZeroOrOne(weights[i], max)==1` for integer weights), return *i*, and go to step 2 otherwise. (See, for example, sec. 4 of the Fast Loaded Dice Roller paper, or the Tang or Klundert papers. `weights[i]` can also be a function that calculates the weight for `i` "on the fly"; in that case, `max` is the maximum value of `weights[i]` for every `i`.)If the weights are instead log-weights (that is, each weight is `ln(x)/ln(b)` where `x` is the original weight and `b` is the logarithm base), step 3 changes to: "(3) If `Expo(ln(b)) > max - weights[i]` (which happens with probability `pow(b, -(max - weights[i]))`), return *i*, and go to step 2 otherwise."If the weights are instead "coins", each with a separate but unknown probability of heads, the algorithm is also called *Bernoulli race* (Dughmi et al. 2017)[61]: (1) set *i* to `RNDINT(n - 1)`; (2) flip coin *i* (the first coin is 0, the second is 1, and so on), then return *i* if it returns 1 or heads, or go to step 1 otherwise.

Algorithm	Notes
Bringmann and Panagiotou (2012/2017)[62].	Shows a sampler designed to work on a sorted list of weights.
Alias method (Walker 1977)[63]	Michael Vose's version of the alias method (Vose 1991)[64] is described in "**Darts, Dice, and Coins: Sampling from a Discrete Distribution**[65]". Weights should be rational numbers.
(Klundert 2019)[66]	Various data structures, with emphasis on how they can support changes in weights.
The Bringmann–Larsen succinct data structure (Bringmann and Larsen 2013)[67]	Uses rejection sampling if the sum of weights is large, and a compressed structure otherwise.
Hübschle-Schneider and Sanders (2019)[68].	Parallel weighted random samplers.
(Tang 2019)[69].	Presents various algorithms, including two- and multi-level search, binary search (with cumulative weights), and a new "flat" method.
"Loaded Die from Biased Coins"	Given a list of probabilities `probs` that must sum to 1 and should be rational numbers: (1) Set `cumu` to 1 and `i` to 0; (2) with probability `probs[i]/cumu`, return `i`; (3) subtract `probs[i]` from `cumu`, then add 1 to `i`, then go to step 2. For a correctness proof, see "Darts, Dice, and Coins". If each probability in `probs` is calculated "on the fly", this is also called sequential search; see chapter 10 of Devroye (1986)[70] (but this generally won't be exact unless all the probabilities involved are rational numbers).
Knuth and Yao (1976)[71]	Generates a binary DDG tree from the binary expansions of the probabilities (that is, they have the base-2 form 0.bbbbbb… where b is 0 or 1). Comes within 2 bits, on average, of the optimal number of random bits per sample. This is suggested in exercise 3.4.2 of chapter 15 of Devroye (1986)[72], implemented in *randomgen.py* as the `discretegen` method, and also described in (Devroye and Gravel 2020)[73]. `discretegen` can work with probabilities that are irrational numbers (which have infinite binary expansions) as long as there is a way to calculate the binary expansion "on the fly".
Han and Hoshi (1997)[74]	Uses cumulative probabilities as input and comes within 3 bits, on average, of the optimal number of random bits per sample. Also described in (Devroye and Gravel 2020)[75].

[59](2020a) Saad, F.A., Freer C.E., et al., "**The Fast Loaded Dice Roller: A Near-Optimal Exact Sampler for Discrete Probability Distributions**", arXiv:2003.03830v2 [stat.CO], also in *AISTATS 2020: Proceedings of the 23rd International Conference on Artificial Intelligence and Statistics, Proceedings of Machine Learning Research* 108, Palermo, Sicily, Italy, 2020. https://arxiv.org/abs/2003.03830v2

[60]Feras A. Saad, Cameron E. Freer, Martin C. Rinard, and Vikash K. Mansinghka, "**Optimal Approximate Sampling From Discrete Probability Distributions**", arXiv:2001.04555v1 [cs.DS], also in Proc. ACM Program. Lang. 4, POPL, Article 36 (January 2020), 33 pages. https://arxiv.org/abs/2001.04555v1

[61]Shaddin Dughmi, Jason D. Hartline, Robert Kleinberg, and Rad Niazadeh. 2017. Bernoulli Factories and Black-Box

Notes:

1. **Weighted choice algorithms as binary tree walkers.** Just like `RNDINT` algorithms (see the **`RNDINT` section**), weighted choice algorithms can all be described as random walks on a binary DDG tree. In this case, though, the probabilities are not necessarily uniform, and on average, the algorithm needs at least as many unbiased random bits as the sum of *binary entropies* of all the probabilities involved. For example, say we give the four integers 1, 2, 3, 4 the following weights: 3, 15, 1, 2. The binary entropies of these weights add up to 0.4010… + 0.3467… + 0.2091… + 0.3230… = 1.2800… (because the sum of the weights is 21 and the binary entropy of 3/21 is `0 - (3/21) * log2(3/21) = 0.4010...`, where `log2(x) = ln(x)/ln(2)`, and so on for the other weights). Thus, any weighted sampler will require at least 1.2800… bits on average to generate a number with probability proportional to these weights.[76] The note "Reducing 'bit waste' " from the `RNDINT` section also applies here.
2. For best results, weights passed to the algorithms in the table above should first be **converted to integers** (for example, using `NormalizeRatios` in the pseudocode above), **or rational numbers** when indicated. (Obviously, if the weights were originally irrational numbers, this conversion will be lossy and the algorithm won't be exact, unless noted otherwise in the table.) Also, using floating-point numbers in the algorithms can introduce unnecessary rounding errors, so such numbers should be avoided.
3. The **Python sample code**[77] contains a variant of the `WeightedChoice` pseudocode for generating multiple random points in one call.

Examples:

1. Assume we have the following list: `["apples", "oranges", "bananas", "grapes"]`, and `weights` is the following: `[3, 15, 1, 2]`. The weight for "apples" is 3, and the weight for "oranges" is 15. Since "oranges" has a higher weight than "apples", the index for "oranges" (1) has a greater probability of being chosen than the index for "apples" (0) with the

Reductions in Mechanism Design. In *Proceedings of 49th Annual ACM SIGACT Symposium on the Theory of Computing*, Montreal, Canada, June 2017 (STOC'17).

[62] K. Bringmann and K. Panagiotou, "**Efficient Sampling Methods for Discrete Distributions**." *Algorithmica* 79 (2007), also in Proc. 39th International Colloquium on Automata, Languages, and Programming (ICALP'12), 2012. https://link.springer.com/article/10.1007/s00453-016-0205-0

[63] A.J. Walker, "An efficient method for generating discrete random variables with general distributions", *ACM Transactions on Mathematical Software* 3, 1977.

[64] Vose, Michael D. "A linear algorithm for generating random numbers with a given distribution." IEEE Transactions on software engineering 17, no. 9 (1991): 972-975.

[65] https://www.keithschwarz.com/darts-dice-coins/

[66] Klundert, B. van de, "**Efficient Generation of Discrete Random Variates**", Faculty of Science Theses, Universiteit Utrecht, 2019. https://dspace.library.uu.nl/handle/1874/393383

[67] K. Bringmann and K. G. Larsen, "Succinct Sampling from Discrete Distributions", In: Proc. 45th Annual ACM Symposium on Theory of Computing (STOC'13), 2013.

[68] L. Hübschle-Schneider and P. Sanders, "**Parallel Weighted Random Sampling**", arXiv:1903.00227v2 [cs.DS], 2019. https://arxiv.org/abs/1903.00227v2

[69] Y. Tang, "An Empirical Study of Random Sampling Methods for Changing Discrete Distributions", Master's thesis, University of Alberta, 2019.

[70] Devroye, L., ***Non-Uniform Random Variate Generation***, 1986.

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

[72] Devroye, L., ***Non-Uniform Random Variate Generation***, 1986.

[73] Devroye, L., Gravel, C., "**Random variate generation using only finitely many unbiased, independently and identically distributed random bits**", arXiv:1502.02539v6 [cs.IT], 2020. https://arxiv.org/abs/1502.02539v6

[74] T. S. Han and M. Hoshi, "Interval algorithm for random number generation", *IEEE Transactions on Information Theory* 43(2), March 1997.

[75] Devroye, L., Gravel, C., "**Random variate generation using only finitely many unbiased, independently and identically distributed random bits**", arXiv:1502.02539v6 [cs.IT], 2020. https://arxiv.org/abs/1502.02539v6

[76] This is because the *binary entropy* of `p = 1/n` is `p * log2(1/p) = log2(n) / n`, and the sum of `n` binary entropies (for `n` outcomes with probability `1/n` each) is `log2(n) = ln(n)/ln(2)`.

[77] https://peteroupc.github.io/randomgen.zip

`WeightedChoice` method. The following idiom implements how to get a randomly chosen item from the list with that method: `item = list[WeightedChoice(weights)]`.

2. Example 1 can be implemented with `CumulativeWeightedChoice` instead of `WeightedChoice` if `weights` is the following list of cumulative weights: `[0, 3, 18, 19, 21]`.
3. **Piecewise constant distribution.** Assume the weights from example 1 are used and the list contains the following: `[0, 5, 10, 11, 13]` (one more item than the weights). This expresses four intervals: [0, 5), [5, 10), and so on. Choose a random index (and thus interval) with `index = WeightedChoice(weights)`. Then independently, choose a number in the chosen interval uniformly at random (for example, code like the following chooses a random integer this way: `number = RNDINTEXCRANGE(list[index], list[index + 1])`).

6.1.2 Weighted Choice Without Replacement

Weighted choice *without replacement* can be thought of as drawing a ball *without* putting it back.

The following are ways to implement weighted choice without replacement, where each item **can be chosen no more than once** at random. The weights have the property that higher-weighted items have a greater chance to appear first.

- Use `WeightedChoice` to choose random indices. Each time an index is chosen, set the weight for the chosen index to 0 to keep it from being chosen again. Or…
- Assign each index a random exponential random variate (with a rate equal to that index's weight, which must be an integer 1 or greater). Make a list of pairs assigning each number to an index, then sort that list in ascending order (from low to high) by those numbers. Example: `v=[]; for i in 0...size(weights): AddItem(v, [ExpoNew(weights[i]), i]); Sort(v)` (see note 2 below). The sorted list of indices will then correspond to a weighted choice without replacement. See "**Algorithms for sampling without replacement**[78]" and see also (Efraimidis 2015)[79]. (Efraimidis and Spirakis 2005)[80] describes how to implement this approach without having to store every item using reservoir sampling (essentially a *priority queue* of the `k` items with the lowest random variates associated with them).

  Notes:

  1. Some applications (particularly some games) wish to control which random outcomes appear, to make those outcomes appear "fairer" to users (for example, to avoid long streaks of good outcomes or of bad outcomes). Weighted choice without replacement of a list of outcomes can be used for this purpose as long as the list is replenished once all the outcomes are chosen. However, this kind of sampling should be avoided in applications that care about information security, including when a player or user would have a significant and unfair advantage if the outcomes were easy to guess.
  2. `ExpoNew(weight)` creates an "empty" exponential random variate with rate `weight`, whose bits are not yet determined (see "**Partially-Sampled Random Numbers**[81]"). Sorting the variates relies on comparing two variates via `ExpoLess(a, b)`, which returns whether one exponential random variate (`a`) is less than another (`b`), building up the bits of both as necessary.

6.1.3 Unequal Probability Sampling

For the methods given in the previous section, the weights have the property that higher-weighted items are chosen first, but each item's weight is not necessarily the chance that a given sample of `n` items will include

[78] https://timvieira.github.io/blog/post/2019/09/16/algorithms-for-sampling-without-replacement/

[79] Efraimidis, P. "**Weighted Random Sampling over Data Streams**", arXiv:1012.0256v2 [cs.DS], 2015. https://arxiv.org/abs/1012.0256v2

[80] Efraimidis, P. and Spirakis, P. "**Weighted Random Sampling (2005; Efraimidis, Spirakis)**", 2005.

[81] https://peteroupc.github.io/exporand.html

that item (an *inclusion probability*). The following method chooses a random sample of `n` indices from a list of items (whose weights are integers stored in a list called `weights`), such that the chance that index `k` is in the sample is given as `weights[k]*n/Sum(weights)`. The chosen indices will not necessarily be in random order. The method implements the "**splitting method**[82]" (Deville and Tillé 1998)[83].

```
METHOD InclusionSelect(weights, n)
  if n>size(weights): return error
  if n==0: return []
  ws=Sum(weights)
  wts=[]
  items=[]
  // Calculate inclusion probabilities
  for i in 0...size(weights):
     AddItem(wts,[MakeRatio(weights[i],ws)*n, i])
  Sort(wts)
  // Check for invalid inclusion probabilities
  if wts[size(wts)-1][0]>1: return error
  last=size(wts)-n
  if n==size(wts)
    for i in 0...n: AddItem(items,i)
    return items
  end
  while true // until a value is returned
    lamda=min(MakeRatio(1,1)-wts[last-1][0],wts[last][0])
    if lamda==0: return error
    if ZeroOrOne(lamda[0],lamda[1])
      for k in 0...size(wts)
        if k+1>ntotal-n:AddItem(items,wts[k][1])
      end
      return items
    end
    newwts=[]
    for k in 0...size(wts)
      newwt=(k+1<=last) ?
          wts[k][0]/(1-lamda) : (wts[k][0]-lamda)/(1-lamda)
      AddItem(newwts,[newwt,wts[k][1]])
    end
    wts=newwts
    Sort(wts)
  end
END METHOD
```

For the case when the list of items has an unknown size and its weight can be calculated "on the fly", see (Chao 1982)[84]; (Cohen et al. 2010)[85] (VarOpt$_k$).

[82] https://www.eustat.eus/productosServicios/52.1_Unequal_prob_sampling.pdf#page=68

[83] Deville, J.-C. and Tillé, Y. Unequal probability sampling without replacement through a splitting method. Biometrika 85 (1998).

[84] Chao, M-T., "A general purpose unequal probability sampling plan", *Biometrika* 69 (1982).

[85] Cohen E., Duffield N., Kaplan H., Lund C., Thorup M., "**Stream sampling for variance-optimal estimation of subset sums**", arXiv:0803.0473, 2010. https://arxiv.org/abs/0803.0473

6.2 Mixtures of Distributions

A *mixture* consists of two or more probability distributions with separate probabilities of being sampled. To generate random content from a mixture—

1. generate `index = WeightedChoice(weights)`, where `weights` is a list of relative probabilities that each distribution in the mixture will be sampled, then
2. based on the value of `index`, generate the random content from the corresponding distribution.

Examples:

1. One mixture consists of the sum of three six-sided virtual die rolls and the result of one six-sided die roll, but there is an 80% chance to roll one six-sided virtual die rather than three. The following pseudocode shows how this mixture can be sampled: `index = WeightedChoice([80, 20]); number = 0; if index==0: number = RNDINTRANGE(1,6); else: number = RNDINTRANGE(1,6) + RNDINTRANGE(1,6) + RNDINTRANGE(1,6)`.
2. Choosing an independent uniform random point, from a complex shape (in any number of dimensions) is equivalent to doing such sampling from a mixture of simpler shapes that make up the complex shape (here, the `weights` list holds the n-dimensional "volume" of each simpler shape). For example, a simple closed 2D polygon can be ***triangulated***[86], or decomposed into **triangles**, and a mixture of those triangles can be sampled.[87]
3. Take a set of nonoverlapping integer ranges (for example, [0, 5], [7, 8], [20, 25]). To choose an independent uniform random integer from those ranges:
   - Create a list (`weights`) of weights for each range. Each range is given a weight of `(mx - mn) + 1`, where `mn` is that range's minimum and `mx` is its maximum.
   - Choose an index using `WeightedChoice(weights)`, then generate `RNDINTRANGE(mn, mx)`, where `mn` is the corresponding range's minimum and `mx` is its maximum.

   This method can be adapted for rational numbers with a common denominator by treating the integers involved as the numerators for such numbers. For example, [0/100, 5/100], [7/100, 8/100], [20/100, 25/100], where the numerators are the same as in the previous example.
4. In the pseudocode `index = WeightedChoice([80, 20]); list = [[0, 5], [5, 10]]; number = RNDINTEXCRANGE(list[index][0], list[index][1])`, a random integer in [0, 5) is chosen at an 80% chance, and a random integer in [5, 10) at a 20% chance.
5. A **hyperexponential distribution** is a mixture of **exponential distributions**, each one with a separate weight and separate rate parameter.

6.3 Transformations of Random Variates

Random variates can be generated by combining one or more random variates, by transforming them, by discarding some of them, or any combination of these.

As an example, **"Probability and Games: Damage Rolls"** by Red Blob Games includes interactive graphics showing score distributions for lowest-of, highest-of, drop-the-lowest, and reroll game mechanics.[88] These and similar distributions can be generalized as follows.

Generate one or more random variates (numbers), each with a separate probability distribution, then:

1. **Highest-of:** Choose the highest generated variate.

[86] https://en.wikipedia.org/wiki/Polygon_triangulation

[87] The **Python sample code** includes a `ConvexPolygonSampler` class that implements this kind of sampling for convex polygons; unlike other polygons, convex polygons are trivial to decompose into triangles. https://peteroupc.github.io/randomgen.zip

[88] That article also mentions a critical-hit distribution, which is actually a **mixture** of two distributions: one roll of dice and the sum of two rolls of dice.

2. **Drop-the-lowest:** Add all generated variates except the lowest.
3. **Reroll-the-lowest:** Add all generated variates except the lowest, then add a number generated randomly by a separate probability distribution.
4. **Lowest-of:** Choose the lowest generated number.
5. **Drop-the-highest:** Add all generated variates except the highest.
6. **Reroll-the-highest:** Add all generated variates except the highest, then add a number generated randomly by a separate probability distribution.
7. **Sum:** Add all generated variates.
8. **Mean:** Add all generated variates, then divide the sum by the number of variates.
9. **Geometric transformation:** Treat the variates as an n-dimensional point, then apply a geometric transformation, such as a rotation or other *affine transformation*[89], to that point.

If the probability distributions are the same, then strategies 1 to 3 give higher numbers a greater probability, and strategies 4 to 6, lower numbers.

> **Note:** Variants of strategy 4 — for example, choosing the second-, third-, or nth-lowest number — are formally called second-, third-, or nth-**order statistics distributions**, respectively.
>
> **Examples:**
>
> 1. The idiom `min(RNDINTRANGE(1, 6), RNDINTRANGE(1, 6))` takes the lowest of two six-sided die results (strategy 4). Due to this approach, 1 has a greater chance of occurring than 6.
> 2. The idiom `RNDINTRANGE(1, 6) + RNDINTRANGE(1, 6)` takes the result of two six-sided dice (see also "**Dice**") (strategy 7).
> 3. A **binomial distribution** models the sum of `n` numbers each generated by `ZeroOrOne(px,py)` (strategy 7) (see "**Binomial Distribution**").
> 4. A **Poisson binomial distribution**[90] models the sum of `n` numbers each with a separate probability of being 1 as opposed to 0 (strategy 7). Given `probs`, a list of the `n` probabilities as rational numbers, the pseudocode is: `for i in 0...n: x=x+ZeroOrOne(probs[i][0],probs[i][1]); return x`.
> 5. **Clamped random variates.** These are one example of transformed random variates. To generate a clamped random variate, generate a number at random as usual, then—
>
>   - if that number is less than a minimum threshold, use the minimum threshold instead (*left-censoring*), or
>   - if that number is greater than a maximum threshold, use the maximum threshold instead (*right-censoring*),
>
>   or both.
>
>   An example of a clamped random variate is `min(200, RNDINT(255))`.
> 6. A **compound Poisson distribution** models the sum of n numbers each chosen at random in the same way, where n follows a **Poisson distribution** (for example, `n = PoissonInt(10, 1)` for an average of 10 numbers) (strategy 7, sum).
> 7. A **Pólya–Aeppli distribution** is a compound Poisson distribution in which the numbers are generated by `NegativeBinomial(1, 1-p)+1` for a fixed `p`.

[89] An *affine transformation* is one that keeps straight lines straight and parallel lines parallel.
[90] https://en.wikipedia.org/wiki/Poisson_binomial_distribution

7 Specific Non-Uniform Distributions

This section contains information on some of the most common non-uniform probability distributions.

7.1 Dice

The following method generates a random result of rolling virtual dice. It takes three parameters: the number of dice (`dice`), the number of sides in each die (`sides`), and a number to add to the result (`bonus`) (which can be negative, but the result of the method is 0 if that result is greater). See also Red Blob Games, **"Probability and Games: Damage Rolls"**.

```
METHOD DiceRoll(dice, sides, bonus)
    if dice < 0 or sides < 1: return error
    ret = 0
    for i in 0...dice: ret=ret+RNDINTRANGE(1, sides)
    return max(0, ret + bonus)
END METHOD
```

> **Examples:** The result of rolling— - four six-sided virtual dice ("4d6") is `DiceRoll(4,6,0)`, - three ten-sided virtual dice, with 4 added ("3d10 + 4"), is `DiceRoll(3,10,4)`, and - two six-sided virtual dice, with 2 subtracted ("2d6 - 2"), is `DiceRoll(2,6,-2)`.

7.2 Binomial Distribution

The *binomial distribution* uses two parameters: `trials` and `p`. This distribution models the number of successes in a fixed number of independent trials (equal to `trials`), each with the same probability of success (equal to `p`, where `p <= 0` means never, `p >= 1` means always, and `p = 1/2` means an equal chance of success or failure). In this document, `Binomial(trials, p)` is a binomial random variate with the given parameters.

This distribution has a simple implementation: `count = 0; for i in 0...trials: count=count+ZeroOrOne(px, py)`. But for large numbers of trials, this can be very slow.

The pseudocode below implements an exact sampler of this distribution, with certain optimizations based on (Farach-Colton and Tsai 2015)[91]. (Another exact sampler is given in (Bringmann et al. 2014)[92] and described in my "**Miscellaneous Observations on Randomization**[93]".) Here, the parameter `p` is expressed as a ratio `px`/`py`.

```
METHOD BinomialInt(trials, px, py)
  if trials < 0: return error
  if trials == 0: return 0
  // Always succeeds
  if mx: return trials
  // Always fails
  if p <= 0.0: return 0
  count = 0
  ret = 0
  recursed = false
  if py*2 == px // Is half
    if i > 200
```

[91] Farach-Colton, M. and Tsai, M.T., 2015. Exact sublinear binomial sampling. *Algorithmica* 73(4), pp. 637-651.

[92] K. Bringmann, F. Kuhn, et al., "Internal DLA: Efficient Simulation of a Physical Growth Model." In: *Proc. 41st International Colloquium on Automata, Languages, and Programming (ICALP'14)*, 2014.

[93] https://peteroupc.github.io/randmisc.html#On_a_Binomial_Sampler

```
      // Divide and conquer
      half = floor(trials / 2)
      return BinomialInt(half, 1, 2) + BinomialInt(trials - half, 1, 2)
    else
      if rem(trials,2)==1
        count=count+RNDINT(1)
        trials=trials-1
      end
      // NOTE: This step can be made faster
      // by precalculating an alias table
      // based on a list of n + 1 binomial(1/2)
      // weights, which consist of n-choose-i
      // for every i in [0, n], and sampling based on
      // that table (see Farach-Colton and Tsai).
      for i in 0...trials: count=count+RNDINT(1)
    end
  else
    // Based on proof of Theorem 2 in Farach-Colton and Tsai.
    // Decompose px/py into its base-2 digits.
    pw = MakeRatio(px, py)
    pt = MakeRatio(1, 2)
    while trials>0 and pw>0
      c=BinomialInt(trials, 1, 2)
      if pw>=pt
        count=count+c
        trials=trials-c
        pw=pw-pt
      else
        trials=c
      end
      pt=pt/2 // NOTE: Not rounded
    end
  end
  if recursed: return count+ret
  return count
END METHOD
```

> **Note:** If `px/py` is `1/2`, the binomial distribution models the task "Flip N coins, then count the number of heads", and the random sum is known as ***Hamming distance***[94] (treating each trial as a "bit" that's set to 1 for a success and 0 for a failure). If `px` is `1`, then this distribution models the task "Roll `n` `py`-sided dice, then count the number of dice that show the number 1."

7.3 Negative Binomial Distribution

In this document, the *negative binomial distribution* models the number of failing trials that happen before a fixed number of successful trials (`successes`). Each trial is independent and has a success probability of `px/py` (where 0 means never and 1 means always). The following is a naïve implementation; see also the notes for the geometric distribution, a special case of this one.

```
METHOD NegativeBinomialInt(successes, px, py)
    // successes>=0; px/py needs to be greater than 0
    if successes < 0 or px == 0: return error
```

[94] https://en.wikipedia.org/wiki/Hamming_distance

```
    if successes == 0 or px >= py: return 0
    total = 0
    count = 0
    while total < successes
        if ZeroOrOne(px, py) == 1: total = total + 1
        else: count = count + 1
    end
    return count
END METHOD
```

If `successes` is a non-integer, the distribution is often called a *Pólya distribution*. In that case, it can be sampled using the following pseudocode (Heaukulani and Roy 2019)[95]:

```
METHOD PolyaInt(sx, sy, px, py)
   isinteger=rem(sx,sy)==0
   sxceil=ceil(sx/sy)
   while true // until a value is returned
      w=NegativeBinomialInt(sxceil, px, py)
      if isinteger or w==0: return w
      tmp=MakeRatio(sx,sy)
      anum=tmp
      for i in 1...w: anum=anum*(tmp+i)
      tmp=sxceil
      aden=tmp
      for i in 1...w: aden=aden*(tmp+i)
      a=anum/aden
      if ZeroOrOne(a[0], a[1])==1: return w
   end
END METHOD
```

7.4 Geometric Distribution

The geometric distribution is a negative binomial distribution with `successes = 1`. In this document, a geometric random variate is the number of failures that have happened before one success happens. For example, if `p` is 1/2, the geometric distribution models the task "Flip a coin until you get tails, then count the number of heads." As a unique property of the geometric distribution, given that `n` trials have failed, the number of new failing trials has the same distribution (where `n` is an integer greater than 0).

Notes:

1. The negative binomial and geometric distributions are defined differently in different works. For example, *Mathematica*'s definition excludes the last success, but the definition in (Devroye 1986, p. 498)[96] includes it. And some works may define a negative binomial number as the number of successes before N failures, rather than vice versa.
2. A *bounded geometric* random variate is either n (an integer greater than 0) or a geometric random variate, whichever is less. Exact and efficient samplers for the geometric and bounded geometric distributions, such as the ones described in (Bringmann and Friedrich 2013)[97], are described in my "**Miscellaneous Observations on Randomization**[98]".

[95] Heaukulani, C., Roy, D.M., "**Black-box constructions for exchangeable sequences of random multisets**", arXiv:1908.06349v1 [math.PR], 2019. Note however that this reference defines a negative binomial distribution as the number of successes before N failures (not vice versa). https://arxiv.org/abs/1908.06349v1

[96] Devroye, L., ***Non-Uniform Random Variate Generation***, 1986.

[97] Bringmann, K., and Friedrich, T., 2013, July. Exact and efficient generation of geometric random variates and random graphs, in *International Colloquium on Automata, Languages, and Programming* (pp. 267-278).

[98] https://peteroupc.github.io/randmisc.html#On_ _Geometric_Samplers

7.5 Exponential Distribution

The *exponential distribution* uses a parameter known as λ , the rate, or the inverse scale. Usually, λ is the probability that an independent event of a given kind will occur in a given span of time (such as in a given day or year), and the random result is the number of spans of time until that event happens. Usually, λ is equal to 1, or 1/1. $1/\ \lambda$ is the scale (mean), which is usually the average waiting time between two independent events of the same kind.

In this document, `Expo(lamda)` is an exponentially-distributed random variate with the rate `lamda`. For algorithms to sample exponential random variates to arbitrary precision, see "**Partially-Sampled Random Numbers**[99]".

7.6 Poisson Distribution

The *Poisson distribution* uses a parameter `mean` (also known as λ). λ is the average number of independent events of a certain kind per fixed unit of time or space (for example, per day, hour, or square kilometer). A Poisson-distributed number is the number of such events within one such unit.

In this document, `Poisson(mean)` is a Poisson-distributed number if `mean` is greater than 0, or 0 if `mean` is 0.

The method `PoissonInt` generates a Poisson random variate with mean `mx/my`, with the `Poisson1` method using an algorithm by Duchon and Duvignau (2016)[100],

```
METHOD Poisson1()
  ret=1; a=1; b=0
  while true // until this method returns
    j=RNDINT(a)
    if j<a and j<b: return ret
    if j==a: ret=ret+1
    else
      ret=ret-1; b=a+1
    end
    a=a+1
  end
END METHOD

METHOD PoissonInt(mx, my)
  if my == 0: return error
  if mx == 0 or (mx < 0 and my < 0) or (mx > 0 and my < 0): return 0
  r=0
  while mx>=my
     r=r+Poisson1(); mx=mx-my
  end
  if mx>0
    // At this point, mx/my < 1, so sum a Poisson number
    // of coin flips with heads prob. mx/my; see Devroye 1986, p. 487
    r=r+BinomialInt(Poisson1(), mx, my)
  end
  return r
END METHOD
```

[99] https://peteroupc.github.io/exporand.html

[100] Duchon, P., Duvignau, D., "Preserving the number of cycles of length k in a growing uniform permutation", *Electronic Journal of Combinatorics* 23(4), 2016.

Note: To generate a sum of `n` independent Poisson random variates with separate means, generate a Poisson random variate whose mean is the sum of those means (see (Devroye 1986)[101], p. 501). For example, to generate a sum of 1000 independent Poisson random variates with a mean of 1/1000000, simply generate `PoissonInt(1, 1000)` (because 1/1000000 * 1000 = 1/1000).

7.7 Pólya–Eggenberger Distribution

Suppose items are drawn at random from a collection of items each labeled either `1` or `0`, and after drawing an item, it's put back and `m` more items of the same label as the drawn item are added. Then:

- The *Pólya–Eggenberger distribution* models the number of items drawn this way that are labeled `1`.
- The *inverse Pólya–Eggenberger distribution* models the number of `0`-labeled items drawn before `successes` many `1`-labeled items are drawn.

(Johnson and Kotz 1969)[102]. In the methods below, `trials` is the number of items drawn at random, `ones` is the number of items labeled `1` in the collection, `count` is the number of items labeled `1` or `0` in that collection, `m` is the number of items added after each draw (or -1 for sampling *without replacement*), and `successes` is the number of `1`-labeled items drawn.

```
METHOD PolyaEggenberger(trials, ones, count, m)
    if ones < 0 or count < 0 or trials < 0 or
       ones > count or trials > count
      return error
    end
    if ones == 0: return 0
    zeros=count-ones
    ret=0
    for i in 0...trials
       if zeros==0 or ZeroOrOne(ones,zeros)==1
          ones=ones+m
          ret=ret+1
       else: zeros=zeros+m
    end
    return ret
END METHOD

METHOD InversePolyaEggenberger(successes, ones, count, m)
    if ones <= 0 or count < 0 or successes < 0 or
       ones > count or successes > count
      return error
    end
    zeros=count-ones
    ret=0; trials=0
    while ret<successes
       if zeros==0 or ZeroOrOne(ones,zeros)==1
          ones=ones+m
          ret=ret+1
       else: zeros=zeros+m
       trials=trials+1
    end
    return trials-successes
```

[101] Devroye, L., ***Non-Uniform Random Variate Generation***, 1986.
[102] Johnson and Kotz, "Discrete Distributions", 1969.

```
END METHOD
```

Notes:

1. A **hypergeometric distribution** is a Pólya–Eggenberger distribution with `m=-1`. For example, in a 52-card deck of Anglo-American playing cards, 12 of the cards are face cards (jacks, queens, or kings). After the deck is shuffled and seven cards are drawn, the number of face cards drawn this way follows a hypergeometric distribution where `trials` is 7, `ones` is 12, `count` is 52, and `m` is -1.
2. A **negative hypergeometric distribution** is an inverse Pólya–Eggenberger distribution with `m=-1`.

7.8 Random Integers with a Given Positive Sum

The following pseudocode shows how to generate `n` random integers with a given positive sum, in random order (specifically, a uniformly chosen random partition of that sum into `n` parts with repetition and in random order). (The algorithm for this was presented in (Smith and Tromble 2004)[103].) In the pseudocode below—

- the method `PositiveIntegersWithSum` returns `n` integers greater than 0 that sum to `total`, in random order,
- the method `IntegersWithSum` returns `n` integers 0 or greater that sum to `total`, in random order, and
- `Sort(list)` sorts the items in `list` in ascending order (however, sort algorithms are outside the scope of this document).

```
METHOD PositiveIntegersWithSum(n, total)
    if n <= 0 or total <=0: return error
    ls = [0]
    ret = NewList()
    while size(ls) < n
      c = RNDINTEXCRANGE(1, total)
      found = false
      for j in 1...size(ls)
        if ls[j] == c
          found = true
          break
        end
      end
      if found == false: AddItem(ls, c)
    end
    Sort(ls)
    AddItem(ls, total)
    for i in 1...size(ls): AddItem(ret,
        ls[i] - ls[i - 1])
    return ret
END METHOD

METHOD IntegersWithSum(n, total)
  if n <= 0 or total <=0: return error
  ret = PositiveIntegersWithSum(n, total + n)
```

[103] Smith, Noah A., and Roy W. Tromble. "**Sampling uniformly from the unit simplex**." Johns Hopkins University, Tech. Rep 29 (2004).

```
  for i in 0...size(ret): ret[i] = ret[i] - 1
  return ret
END METHOD
```

Notes:

1. To generate `N` random integers with a given positive average `avg` (an integer), in random order, generate `IntegersWithSum(N, N * avg)`.
2. To generate `N` random integers `min` or greater and with a given positive sum `sum` (an integer), in random order, generate `IntegersWithSum(N, sum - N * min)`, then add `min` to each number generated this way. The **Python sample code**[104] implements an efficient way to generate such integers if each one can't exceed a given maximum; the algorithm is thanks to a *Stack Overflow* answer (`questions/61393463`) by John McClane.
3. To generate `N` rational numbers that sum to `tx/ty`, call `IntegersWithSum(N, tx * ty * x)` or `PositiveIntegersWithSum(N, tx * ty * x)` as appropriate (where `x` is the desired accuracy as an integer, such as `pow(2, 32)` or `pow(2, 53)`, so that the results are accurate to `1/x` or less), then for each number `c` in the result, convert it to `MakeRatio(c, tx * ty * x) * MakeRatio(tx, ty)`.

7.9 Multinomial Distribution

The *multinomial distribution* uses two parameters: `trials` and `weights`. It models the number of times each of several mutually exclusive events happens among a given number of trials (`trials`), where each event can have a separate probability of happening (given as a list of `weights`).

A trivial implementation is to fill a list with as many zeros as `weights`, then for each trial, choose `index = WeightedChoice(weights)` and add 1 to the item in the list at the chosen `index`. The resulting list follows a multinomial distribution. The pseudocode below shows an optimization suggested in (Durfee et al., 2018, Corollary 45)[105], but assumes all weights are integers.

```
METHOD Multinomial(trials, weights)
    if trials < 0: return error
    // create a list of successes
    list = []
    ratios = []
    sum=Sum(weights)
    for i in 0...size(weights): AddItem(ratios,
           MakeRatio(weights[i], sum))
    end
    for i in 0...size(weights)
        r=ratios[i]
        b=BinomialInt(t,r[0],r[1])
        AddItem(list, b)
        trials=trials-b
        if trials>0: for j in range(i+1,
            len(weights)): ratios[j]=ratios[j]/(1-r)
    end
    return list
END METHOD
```

[104] https://peteroupc.github.io/randomgen.zip

[105] Durfee, et al., "l1 Regression using Lewis Weights Preconditioning and Stochastic Gradient Descent", *Proceedings of Machine Learning Research* 75(1), 2018.

8 Randomization with Real Numbers

This section describes randomization methods that use random real numbers, not just random integers. These include random rational numbers, fixed-point numbers, and floating-point numbers.

But whenever possible, **applications should work with random integers**, rather than other random real numbers. This is because:

- No computer can choose from among all real numbers between two others, since there are infinitely many of them.
- Algorithms that work with integers are more portable than those that work with other real numbers, especially floating-point numbers.[106] Integer algorithms are easier to control for their level of accuracy.
- For applications that may care about reproducible "random" numbers (unit tests, simulations, machine learning, and so on), using non-integer numbers (especially floating-point numbers) can complicate the task of making a method reproducible from run to run or across computers.

The methods in this section should not be used to sample at random for information security purposes, even if a secure "source of random numbers" is available. See "Security Considerations" in the appendix.

8.1 Uniform Random Real Numbers

This section defines a method, namely `RNDRANGEMinMaxExc(a, b)`, to generate independent "uniform" random real numbers greater than `a` and less than `b`.[107]

The section shows how this method can be implemented for fixed-point, rational, and floating-point numbers. However, all three formats use a predetermined and fixed precision. Other formats for random real numbers don't have this limitation and include **partially-sampled random numbers**[108] and "constructive reals" or "recursive reals" (Boehm 2020)[109].

8.1.1 For Fixed-Point Number Formats

For fixed-point number formats representing multiples of 1/`n`, this method is trivial. The following returns an integer that represents a fixed-point number. In the method below (and in the note), `fpa` and `fpb` are the bounds of the fixed-point number generated and are integers that represent fixed-point numbers (such that `fpa = a * n` and `fpb = b * n`). For example, if `n` is 100, to generate a number in the open interval (6.35, 9.96), generate `RNDRANGEMinMaxExc(6.35, 9.96)` or `RNDINTRANGE(635 + 1, 996 - 1)`.

- `RNDRANGEMinMaxExc(a, b)`: `RNDINTRANGE(fpa + 1, fpb - 1)`, or an error if `fpa >= fpb or a == fpb - 1`. But if `a` is 0 and `b` is 1: `(RNDINT(n - 2) + 1)` or `(RNDINTEXC(n - 1) + 1)`.

  Note: Additional methods to sample fixed-point numbers in a different interval are given below, but are not used in the rest of this article.

  - `RNDRANGE(a, b)`, interval [`a`, `b`]: `RNDINTRANGE(fpa, fpb)`. But if `a` is 0 and `b` is 1: `RNDINT(n)`.
  - `RNDRANGEMinExc(a, b)`, interval (`a`, `b`]: `RNDINTRANGE(fpa + 1, fpb)`, or an error if `fpa >= fpb`. But if `a` is 0 and `b` is 1: `(RNDINT(n - 1) + 1)` or `(RNDINTEXC(n) + 1)`.
  - `RNDRANGEMaxExc(a, b)`, interval [`a`, `b`): `RNDINTEXCRANGE(fpa, fpb)`. But if `a` is 0 and `b` is 1: `RNDINTEXC(n)` or `RNDINT(n - 1)`.

[106] The NVIDIA white paper "**Floating Point and IEEE 754 Compliance for NVIDIA GPUs**", and "**Floating-Point Determinism**" by Bruce Dawson, discuss issues with floating-point numbers in much more detail. https://docs.nvidia.com/cuda/floating-point/ https://randomascii.wordpress.com/2013/07/16/floating-point-determinism/

[107] "Uniform" in quotes means, as close to the uniform distribution as possible for the number format. Both bounds are excluded because, mathematically, any specific real number from the uniform distribution occurs with probability 0.

[108] https://peteroupc.github.io/exporand.html

[109] Boehm, Hans-J. "Towards an API for the real numbers." In Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 562-576. 2020.

8.1.2 For Rational Number Formats

A *rational number* is a ratio of integers. If the rational number's denominator is `n` (which must be 1 or greater), use the previous section to generate its numerator, so that the rational number is a multiple of 1/`n`.

8.1.3 For Floating-Point Number Formats

For floating-point number formats representing numbers of the form `FPSign` * `FPSignificand` * `FPRADIX` `^e` [110], the following pseudocode implements `RNDRANGEMinMaxExc(lo, hi)`. In the pseudocode:

- `MINEXP` is the lowest exponent a number can have in the floating-point format. For the IEEE 754 binary64 format (Java `double`), `MINEXP = -1074`. For the IEEE 754 binary32 format (Java `float`), `MINEXP = -149`.
- `FPPRECISION` is the number of significant digits in the floating-point format, whether the format stores them as such or not. Equals 53 for binary64, or 24 for binary32.
- `FPRADIX` is the digit base of the floating-point format. Equals 2 for binary64 and binary32.
- `FPExponent(x)` returns the value of `e` for the number `x` such that the number of digits in `s` equals `FPPRECISION`. Returns `MINEXP` if `x = 0` or if `e` would be less than `MINEXP`.
- `FPSignificand(x)` returns the significand (which is nonnegative) of the number `x`. Returns 0 if `x = 0`. Has `FPPRECISION` digits, but may have fewer if `FPExponent(x) == MINEXP`.
- `FPSign(x)` returns either -1 or 1 indicating whether the number is positive or negative. Can be − 1 even if `s` is 0.

See also (Downey 2007)[111] and the **Rademacher Floating-Point Library**[112].

```
METHOD RNDRANGEMinMaxExc(lo, hi)
   if mn >= mx: return error
   return RNDRANGEHelper(lo, hi)
END METHOD

METHOD RNDRANGEHelper(lo, hi)
  losgn = FPSign(lo)
  hisgn = FPSign(hi)
  loexp = FPExponent(lo)
  hiexp = FPExponent(hi)
  losig = FPSignificand(lo)
  hisig = FPSignificand(hi)
  if lo > hi: return error
  if losgn == 1 and hisgn == -1: return error
  if losgn == -1 and hisgn == 1
    // Straddles negative and positive ranges
    // NOTE: Changes negative zero to positive
    mabs = max(abs(lo),abs(hi))
    while true // until a value is returned
       ret=RNDRANGEHelper(0, mabs)
       neg=RNDINT(1)
       if neg==0: ret=-ret
       if ret>=lo and ret<=hi: return ret
    end
  end
  if lo == hi: return lo
```

[110] This includes integers if `e` is limited to 0, and fixed-point numbers if `e` is limited to a single exponent less than 0.
[111] Downey, A. B. "**Generating Pseudo-random Floating Point Values**", 2007.
[112] https://gitlab.com/christoph-conrads/rademacher-fpl

```
  if losgn == -1
    // Negative range
    return -RNDRANGEHelper(abs(lo), abs(hi))
  end
  // Positive range
  expdiff=hiexp-loexp
  if loexp==hiexp
    // Exponents are the same
    // NOTE: Automatically handles
    // subnormals
    s=RNDINTRANGE(losig, hisig)
    return s*1.0*pow(FPRADIX, loexp)
  end
  while true // until a value is returned
    ex=hiexp
    while ex>MINEXP
      v=RNDINTEXC(FPRADIX)
      if v==0: ex=ex-1
      else: break
    end
    s=0
    if ex==MINEXP
      // Has FPPRECISION or fewer digits
      // and so can be normal or subnormal
      s=RNDINTEXC(pow(FPRADIX,FPPRECISION))
    else if FPRADIX != 2
      // Has FPPRECISION digits
      s=RNDINTEXCRANGE(
        pow(FPRADIX,FPPRECISION-1),
        pow(FPRADIX,FPPRECISION))
    else
      // Has FPPRECISION digits (bits), the highest
      // of which is always 1 because it's the
      // only nonzero bit
      sm=pow(FPRADIX,FPPRECISION-1)
      s=RNDINTEXC(sm)+sm
    end
    ret=s*1.0*pow(FPRADIX, ex)
    if ret>=lo and ret<=hi: return ret
  end
END METHOD
```

Notes:

1. Additional methods to sample "uniform" floating-point numbers in a different interval are given below, but are not used in the rest of this article.
   - `RNDRANGE(mn, mx)`, interval [`mn`, `mx`]: Generate `RNDRANGEHelper(mn, mx)`.
   - `RNDRANGEMaxExc(mn, mx)`, interval [`mx`, `mx`): If `mn >= mx`, return an error. Otherwise, generate `RNDRANGEHelper(mn, mx)` in a loop until a number other than `mx` is generated this way.
   - `RNDRANGEMinExc(mn, mx)`, interval (`mn`, `mx`]: If `mn >= mx`, return an error. Otherwise, generate `RNDRANGEHelper(mn, mx)` in a loop until a number other than `mn` is generated

this way.

2. Many software libraries sample "uniform" real numbers by multiplying or dividing a uniform random integer by a constant. For example, a method to sample "uniformly" at random from the half-open interval [0, 1) is often implemented like `RNDINTEXC(X) * (1.0/X)` or `RNDINTEXC(X) / X`, where X varies based on the software library.[113] The disadvantage here is that doing so does not necessarily cover all numbers a floating-point format can represent in the range (Goualard 2020)[^65]. As another example, a method to sample "uniformly" at random from the half-open interval [`a`, `b`) is often implemented like `a + Math.random() * (b - a)`, where `Math.random()` is a "uniform" random floating-point number in [0, 1); however, this not only has the same disadvantage, but has many other issues where floating-point numbers are involved (Monahan 1985)[114].

8.2 Monte Carlo Sampling: Expected Values, Integration, and Optimization

Requires random real numbers.

Randomization is the core of **Monte Carlo sampling**. There are three main uses of Monte Carlo sampling: estimation, integration, and optimization.

1. **Estimating expected values.** Monte Carlo sampling can help estimate the **expected value** (mean or "long-run average") of a sampling distribution, or of a *function* of values sampled from that distribution. This function is called `EFUNC(x)` in this section, where `x` is one of the values in the sample. Algorithms to estimate expected values are called *estimators*. One such estimator is to sample `n` values, apply `EFUNC(x)` to each sampled value `x`, add the values, and divide by `n` (see note below). However, this estimator won't work for all distributions, since they may have an infinite expected value, and it also doesn't allow controlling for the estimate's error. This estimator is called:

    - The **nth sample raw moment** (a raw moment is a mean of nth powers) if `EFUNC(x)` is `pow(x, n)`.
    - The **sample mean**, if `EFUNC(x)` is `x` or `pow(x, 1)`.
    - The **nth sample central moment** (a central moment is a moment about the mean) if `EFUNC(x)` is `pow(x-m, n)`, where `m` is the sample mean.
    - The (biased) **sample variance**, the second sample central moment.
    - The **sample probability**, if `EFUNC(x)` is `1` if some condition is met or `0` otherwise.

    There are two sources of error in Monte Carlo estimators: bias and variance. An estimator is *unbiased* (has bias 0) if its expected value equals the distribution's expected value. For example, any `n`th sample *raw* moment is an unbiased estimator provided the sample size is at least `n`, but the sample variance is not unbiased, and neither is one for any sample *central* moment other than the first (Halmos 1946)[115]. ("Variance reduction" methods are outside the scope of this document.) An estimator's *mean squared error* equals variance plus square of bias.

    For Monte Carlo estimators with accuracy guarantees, see "**Randomized Estimation Algorithms**[116]".

2. **Monte Carlo integration**[117]. This is usually a special case of Monte Carlo estimation that approximates a multidimensional integral over a sampling domain; here, `EFUNC(z)` is the function to find the integral of, where `z` is a randomly chosen point in the sampling domain. For example, `EFUNC(z)` can be 1 if `z` is in the true volume and 0 if not.

[113] Ideally, `X` is the highest integer `p` such that all multiples of `1/p` in the interval [0, 1] are representable in the number format in question. For example, `X` is 2^53 (9007199254740992) for binary64, and 2^24 (16777216) for binary32.

[114] Monahan, J.F., "Accuracy in Random Number Generation", *Mathematics of Computation* 45(172), 1985.

[115] Halmos, P.R., "The theory of unbiased estimation", *Annals of Mathematical Statistics* 17(1), 1946.

[116] https://peteroupc.github.io/estimation.html

[117] https://en.wikipedia.org/wiki/Monte_Carlo_integration

3. **Stochastic optimization**. This uses randomness to help find the minimum or maximum value of a function with one or more variables; examples include ***simulated annealing***[118] and ***simultaneous perturbation stochastic approximation***[119] (see also (Spall 1998)[120]).

   Note: Assuming the true population has a finite mean and variance, the *sample mean* is an unbiased estimator of the mean, but the *sample variance* is generally a biased estimator of variance for every sample smaller than the whole population (Halmos 1946)[121]. The following pseudocode returns a two-item list containing the sample mean and an **unbiased estimator of the variance**, in that order, of a list of real numbers (`list`), using the **Welford method**[122] presented by J. D. Cook. The square root of the variance calculated here is what many APIs call a standard deviation (for example, Python's `statistics.stdev`). For the usual (biased) sample variance, replace `(size(list)-1)` with `size(list)` in the pseudocode shown next. The pseudocode follows: `if size(list)==0: return [0, 0]; if size(list)==1: return [list[0], 0]; xm=list[0]; xs=0; i=1; while i < size(list); c = list[i]; i = i + 1; cxm = (c - xm); xm = xm + cxm *1.0/ i; xs = xs + cxm * (c - xm); end; return [xm, xs*1.0/(size(list)-1)]`.

8.3 Point Sample Selection

Requires random real numbers.

Various methods have been developed for selecting a uniform-behaving sample of points, especially for Monte Carlo methods.

Among these methods, a ***low-discrepancy sequence***[123] (or *quasirandom point set* or *quasi-Monte Carlo point set*) is a deterministic sequence of points with a *low discrepancy* to the uniform distribution on the box, as compared to independent points from that distribution. The following are examples:

- A base-N *van der Corput sequence* is generated as follows: For each nonnegative integer index in the sequence, take the index as a base-N number, then divide the least significant base-N digit by N, the next digit by N^2, the next by N^3, and so on, and add together these results of division.
- A *Halton sequence* is a set of two or more van der Corput sequences with different prime bases; a Halton point at a given index has coordinates equal to the points for that index in the van der Corput sequences.
- Roberts, M., in "**The Unreasonable Effectiveness of Quasirandom Sequences**", presents a low-discrepancy sequence based on a "generalized" version of the golden ratio.
- Sobol sequences are explained in "**Sobol sequence generator**[124]" by S. Joe and F. Kuo.

The points of a low-discrepancy sequence can be "scrambled" with the help of a pseudorandom number generator (or another device or program that simulates a "source of random numbers"). In Monte Carlo sampling, low-discrepancy sequences are often used to achieve more efficient "random" sampling, but in general, they can be safely used this way only if none of their points is skipped (Owen 2020)[125].

Other methods that likewise produce a uniform-behaving point sample include the following.

[118] https://en.wikipedia.org/wiki/Simulated_annealing

[119] https://en.wikipedia.org/wiki/Simultaneous_perturbation_stochastic_approximation

[120] Spall, J.C., "An Overview of the Simultaneous Perturbation Method for Efficient Optimization", *Johns Hopkins APL Technical Digest* 19(4), 1998, pp. 482-492.

[121] Halmos, P.R., "The theory of unbiased estimation", *Annals of Mathematical Statistics* 17(1), 1946.

[122] https://www.johndcook.com/blog/standard_deviation/

[123] https://en.wikipedia.org/wiki/Low-discrepancy_sequence

[124] https://web.maths.unsw.edu.au/~fkuo/sobol/

[125] Owen, Art B. "On dropping the first Sobol'point." In International Conference on Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing, pp. 71-86. Springer, Cham, 2022. "**Preprint**": arXiv:2008.08051. https://arxiv.org/abs/2008.08051

- *Stratified sampling* divides an N-dimensional box into smaller boxes of the same size and chooses one or more points uniformly at random in each box.
- *Latin hypercube sampling* can be implemented using the following pseudocode for an `n`-number sequence: `lhs = []; for i in 0...n: AddItem(RNDRANGEMinMaxExc(i*1.0/n,(i+1)*1.0/n)); lhs = Shuffle(lhs)`.
- Special versions of pseudorandom number generators. One example is linear congruential generators with modulus `m`, a full period, and "good lattice structure"; a sequence of `n`-dimensional points is then `[MLCG(i), MLCG(i+1), ..., MLCG(i+n-1)]` for each integer `i` in the interval [1, `m`] (L'Ecuyer 1999)[126]. One example is `MLCG(seed): rem(92717*seed,262139)/262139.0`. Another example is certain linear feedback shift register generators (Harase 2020)[127].
- If a low-discrepancy sequence outputs numbers in the interval [0, 1], the **Baker's map** of the sequence is `2 * (MakeRatio(1,2)-abs(x - MakeRatio(1,2)))`, where `x` is each number in the sequence.
- Other random point sampling methods, including Poisson disk sampling, the "best candidate algorithm", and N-farthest-points, are described in Kamath (2022)[^72].

8.4 Notes on Randomization Involving Real Numbers

Requires random real numbers.

8.4.1 Random Walks: Additional Examples

- One example of a white noise process is a list of `Normal(0, 1)` numbers (*Gaussian white noise*).
- If `STATEJUMP()` is `RNDRANGEMinMaxExc(-1, 1)`, the random state is advanced by a random real number in the interval (-1, 1).
- A **continuous-time process** models random behavior at every moment, not just at discrete times. There are two popular examples:
  - A *Wiener process* (also known as *Brownian motion*) has random states and jumps that are normally distributed. For a random walk that follows a Wiener process, `STATEJUMP()` is `Normal(mu * timediff, sigma * sqrt(timediff))`, where `mu` is the drift (or average value per time unit), `sigma` is the volatility, and `timediff` is the time difference between samples. A *Brownian bridge* (Revuz and Yor 1999)[128] modifies a Wiener process as follows: For each time X, calculate `W(X) - W(E) * (X - S) / (E - S)`, where `S` and `E` are the starting and ending times of the process, respectively, and `W(X)` and `W(E)` are the state at times X and E, respectively.
  - In a *Poisson point process*, the time between each event is its own exponential random variate with its own rate parameter (for example, `Expo(rate)`) (see "**Exponential Distribution**"). The process is *homogeneous* if all the rates are the same, and *inhomogeneous* if the rate is a function of the "timestamp" before each event jump (the *hazard rate function*); to generate arrival times here, potential arrival times are generated at the maximum possible rate (`maxrate`) and each one is accepted if `RNDRANGEMinMaxExc(0, maxrate) < thisrate`, where `thisrate` is the rate for the given arrival time (Lewis and Shedler 1979)[129].

8.4.2 Transformations: Additional Examples

1. **Bates distribution**: Find the mean of n uniform random variates in a given range (such as variates generated by `RNDRANGEMinMaxExc(minimum, maximum)`) (strategy 8, mean).

[126] P. L'Ecuyer, "Tables of Linear Congruential Generators of Different Sizes and Good Lattice Structure", *Mathematics of Computation* 68(225), January 1999, with **errata**.

[127] Harase, S., "**A table of short-period Tausworthe generators for Markov chain quasi-Monte Carlo**", arXiv:2002.09006 [math.NA], 2020. https://arxiv.org/abs/2002.09006

[128] D. Revuz, M. Yor, "Continuous Martingales and Brownian Motion", 1999.

[129] Lewis, P.W., Shedler, G.S., "Simulation of nonhomogeneous Poisson processes by thinning", *Naval Research Logistics Quarterly* 26(3), 1979.

2. A random point (`x`, `y`) can be transformed (strategy 9, geometric transformation) to derive a point with **correlated random** coordinates (old `x`, new `x`) as follows (see (Saucier 2000)[130], sec. 3.8): `[x, y*sqrt(1 - rho * rho) + rho * x]`, where `x` and `y` are independent numbers chosen at random in the same way, and `rho` is a *correlation coefficient* in the interval [-1, 1] (if `rho` is 0, `x` and `y` are uncorrelated).
3. It is reasonable to talk about sampling the sum or mean of N random variates, where N has a fractional part. In this case, `ceil(N)` random variates are generated and the last variate is multiplied by that fractional part. For example, to sample the sum of 2.5 random variates, generate three random variates, multiply the last by 0.5 (the fractional part of 2.5), then add together all three variates.
4. A **hypoexponential distribution** models the sum of n random variates that follow an exponential distribution and each have a separate rate parameter (see "**Exponential Distribution**").
5. The **maximal coupling** method mentioned by **P. Jacob**[131] generates correlated random variates from two distributions, P and Q, with known probability density functions or PDFs (`PPDF` and `QPDF`, respectively); this works only if the area under each PDF is 1: Sample a number `x` at random from distribution P, and if `RNDRANGEMinMaxExc(0, PPDF(x)) < QPDF(x)`, return `[x, x]`. Otherwise, sample a number `y` at random from distribution Q until `PPDF(y) < RNDRANGEMinMaxExc(0, QPDF(y))`, then return `[x, y]`.

8.5 Sampling from a Distribution of Data Points

Requires random real numbers.

Generating random data points based on how a list of data points is distributed involves the field of **machine learning**: *fit a data model* to the data points, then *predict* a new data point based on that model, with randomness added to the mix. Three kinds of data models, described below, serve this purpose. (How fitting works is outside the scope of this page. Moreover, the variety of machine learning models available makes clear that sampling using only preexisting data points is an ill-posed problem.)

1. **Density estimation models. Density estimation** models seek to describe the distribution of data points in a given data set, where areas with more points have a greater chance to be sampled.[132] The following are examples.
    - **Histograms** are sets of one or more non-overlapping *bins*, which are generally of equal size. Histograms are ***mixtures***, where each bin's weight is the number of data points in that bin. After a bin is randomly chosen, a random data point that could fit in that bin is generated (that point need not be an existing data point).
    - **Gaussian mixture models**[133] are also mixtures, in this case, mixtures of one or more **Gaussian (normal) distributions**.
    - **Kernel distributions** are mixtures of sampling distributions, one for each data point. Estimating a kernel distribution is called ***kernel density estimation***[134]. To sample from a kernel distribution:
        1. Choose one of the numbers or points in the list uniformly at random **with replacement**.
        2. Add a randomized "jitter" to the chosen number or point; for example, add a separately generated `Normal(0, sigma)` to the chosen number or each component of the chosen point, where `sigma` is the *bandwidth*[135].

[130] Saucier, R. "Computer Generation of Statistical Distributions", March 2000.

[131] https://statisfaction.wordpress.com/2017/09/06/sampling-from-a-maximal-coupling/

[132] Other references on density estimation include **a Wikipedia article on multiple-variable kernel density estimation**, and a **blog post by M. Kay**. https://en.wikipedia.org/wiki/Multivariate_kernel_density_estimation https://web.archive.org/web/20160501200206/http://mark-kay.net/2013/12/24/kernel-density-estimation

[133] https://en.wikipedia.org/wiki/Mixture_model

[134] https://en.wikipedia.org/wiki/Kernel_density_estimation

[135] "Jitter", as used in this step, follows a distribution formally called a *kernel*, of which the normal distribution is one example. *Bandwidth* should be set so that the estimated distribution fits the data and remains smooth. A more complex kind of "jitter" (for multi-component data points) consists of a point generated from a **multinormal distribution** with all the means equal to

- **Stochastic interpolation** is described in (Saucier 2000)[136], sec. 5.3.4.
- **Fitting a known distribution** (such as the normal distribution), with unknown parameters, to data can be done by **maximum likelihood estimation**[137], among other ways.

2. **Regression models.** A *regression model* is a model that summarizes data as a formula and an error term. If an application has data in the form of inputs and outputs (for example, monthly sales figures) and wants to sample a random but plausible output given a known input point (for example, sales for a future month), then the application can fit and sample a regression model for that data. For example, a *linear regression model*, which simulates the value of `y` given known inputs `a` and `b`, can be sampled as follows: `y = c1 * a + c2 * b + c3 + Normal(0, sqrt(mse))`, where `mse` is the mean squared error and `c1`, `c2`, and `c3` are the coefficients of the model. (Here, `Normal(0, sqrt(mse))` is the error term.)

3. **Generative models.** These are machine learning models that take random variates as input and generate outputs (such as images or sounds) that are similar to examples they have already seen.

   Notes:

   1. Usually, more than one kind of data model or machine learning model is a possible choice to fit to a given data set (for example, multiple kinds of density estimation models, regression models, parametric distributions, decision trees, or combinations of these). If several kinds of model are fitting choices, then the simplest kind that shows an acceptable *predictive performance* for the data set (for example, information criterion, precision, recall) should be chosen.
   2. If the existing data points each belong in one of several *categories*, choosing a random category could be done by choosing a number at random with probability proportional to the number of data points in each category (see "**Weighted Choice**").
   3. If the existing data points each belong in one of several *categories*, choosing a random data point *and* its category could be done—
      - by choosing a random data point based on all the existing data points, then finding its category (for example, via machine learning models known as *classification models*), or
      - by choosing a random category as given in note 2, then by choosing a random data point based only on the existing data points of that category.

8.6 Sampling from an Arbitrary Distribution

Requires random real numbers.

Many probability distributions can be defined in terms of any of the following:

- The ***cumulative distribution function***[138], or *CDF*, `CDF(x)`, is the probability of choosing a number less than or equal to `x` at random. The probabilities are in the interval [0, 1].
- *Discrete distributions*[139] have a *probability mass function*, or *PMF*, which gives the probability that each number is randomly chosen.

0 and a *covariance matrix* that, in this context, serves as a *bandwidth matrix*. "Jitter" and bandwidth are not further discussed in this document. https://en.wikipedia.org/wiki/Multivariate_normal_distribution

[136] Saucier, R. "Computer Generation of Statistical Distributions", March 2000.

[137] https://en.wikipedia.org/wiki/Maximum_likelihood_estimation

[138] https://en.wikipedia.org/wiki/Cumulative_distribution_function

[139] A *discrete distribution* is a distribution that takes on values that can map to integers and back without loss. These values are usually integers, but they need not be. For example, the values can be non-integer values (for example, `x/y` with probability `x/(1+y)`) as long as the values can be converted to and from integers without loss. Two examples: - A rational number in lowest terms can be converted to an integer by interleaving the bits of the numerator and denominator. - Integer-quantized numbers (popular in "deep-learning" neural networks) take a relatively small number of bits (usually 8 bits or even smaller). An 8-bit quantized number format is effectively a "look-up table" that maps 256 integers to real numbers.

- *Absolutely continuous distributions* have a ***probability density function***[140], or *PDF*, `PDF(x)`, which is the "slope" function of the CDF, or the relative probability of choosing a number "close" to `x` at random. The relative probabilities are 0 or greater, and the area under the PDF is 1.
- The *quantile function* (also known as *inverse cumulative distribution function* or *inverse CDF*) maps numbers in the interval (0, 1) to numbers in the distribution, from low to high.

In this section, a **PDF-like function** is the PDF, the PMF, or either function times a (possibly unknown) positive constant.

The following sections show different ways to generate random variates based on a distribution, depending on what is known about that distribution.

> **Note:** Lists of CDFs, PDF-like functions, or quantile functions are outside the scope of this page.

8.6.1 Sampling for Discrete Distributions

If the distribution **is discrete**, numbers that closely follow it can be sampled by choosing points that cover all or almost all of the distribution, finding their weights or cumulative weights, and choosing a random point based on those weights.

If—

- the discrete distribution has a **known PDF-like function** `PDF(x)`, where `x` must be an integer,
- the interval [`mini`, `maxi`] covers all the distribution, and
- the function's values are all rational numbers (numbers of the form `y/z` where `y` and `z` are integers),

the following method samples exactly from that distribution:

```
METHOD SampleDiscrete(mini, maxi)
      // Setup
      ratios=[]
      for i in mini..maxi: AddItem(ratios, PDF(i))
      ratios=NormalizeRatios(ratios)
      // Sampling
      return mini + WeightedChoice(ratios)
END METHOD
```

If—

- the discrete distribution has a **known CDF** `CDF(x)`, where `x` must be an integer,
- the interval [`mini`, `maxi`] covers all the distribution, and
- the CDF's values are all rational numbers,

the following method samples exactly from that distribution:

```
METHOD SampleDiscreteCDF(mini, maxi)
  // Setup
  ratios=[MakeRatio(0,1)]
  for i in mini..maxi: AddItem(ratios, CDF(i))
  ratios=NormalizeRatios(ratios)
  // Sampling
  value=ratios[size(ratios) - 1]
  for i in 0...size(ratios) - 1
         if ratios[i] < ratios[i+1] and
             ratios[i]>=value: return mini + i
  end
```

[140] https://en.wikipedia.org/wiki/Probability_density_function

```
  return mini
END METHOD
```

In other cases, the discrete distribution can still be approximately sampled. The following cases will lead to an approximate sampler unless the values of the CDF or PDF-like function cover all the distribution and are calculated exactly (without error).

- The values of the CDF or PDF-like function are often calculated in practice as **floating-point numbers** of the form `FPSignificand * FPRadix ^FPExponent` (which include Java's `double` and `float`)[141]. (In general, calculating the values this way will already lead to an approximate sampling algorithm that doesn't allow controlling for the approximation error.) In that case, there are various ways to turn these numbers to rational numbers or integers.
  1. One way is to use `FPRatio(x)` (in the pseudocode below), which is lossless and calculates the rational number for the given floating-point number `x`.
  2. Another way is to scale and round the values to integers (for example, `floor(x * mult)` if `floor(x * mult) < 0.5` and `ceil(x * mult)` otherwise, where `mult` is a large integer); this is not lossless.
- An application can approximate the values of the PDF-like function as integers in a way that bounds the sampling error, such as given in (Saad et al., 2020)[142]. Although this is not lossless and works only for PDF-like functions, this may allow controlling for the approximation error, especially if the values of the PDF-like function cover all the distribution.
- The values of the CDF or PDF-like function may be calculated approximately as **rational numbers**. (In general, calculating the values this way will already lead to an approximate sampling algorithm that doesn't allow controlling for the approximation error.) These rational numbers can be turned into integer weights using `NormalizeRatios`, which is lossless.
- If the distribution takes on an **infinite number of values**, an appropriate interval [`mini`, `maxi`] can be chosen that covers almost all of the distribution. In general, this does not allow controlling for the approximation error in sampling the distribution.

```
METHOD FPRatio(fp)
  expo=FPExponent(fp)
  sig=FPSignificand(fp)
  radix=FPRadix(fp)
  if expo>=0: return MakeRatio(sig * pow(radix, expo), 1)
  return MakeRatio(sig, pow(radix, abs(expo)))
END METHOD
```

> **Note:** In addition, some distributions are known only through an *oracle* (or "black box") that produces random variates that follow that distribution. Algorithms can use this oracle to produce new random variates that follow a different distribution. One example is the Bernoulli factory (see my article "**Bernoulli Factory Algorithms**[143]"), which takes flips of a coin with one probability of heads (the oracle) and produces the flip of a new "coin" with a different probability of heads. Another example is the "Bernoulli race" described in **Weighted Choice**.

8.6.2 Inverse Transform Sampling

Inverse transform sampling[144] (or simply *inversion*) is the most generic way to sample a number from a probability distribution.

[141] This includes integers if `FPExponent` is limited to 0, and fixed-point numbers if `FPExponent` is limited to a single exponent less than 0.

[142] Saad, F.A., et al., "**Optimal Approximate Sampling from Discrete Probability Distributions**", arXiv:2001.04555 [cs.DS], 2020. See also the **associated source code**. https://arxiv.org/abs/2001.04555 https://github.com/probcomp/optimal-approximate-sampling

[143] https://peteroupc.github.io/bernoulli.html

[144] https://en.wikipedia.org/wiki/Inverse_transform_sampling

If the distribution **has a known quantile function**, generate a uniform random variate between 0 and 1 if that number wasn't already pregenerated, and take the quantile of that number. However:

- In most cases, the quantile function is not available. Thus, it has to be approximated.
- Even if the quantile function is available, a naïve quantile calculation (for example, `ICDF(RNDRANGEMinMaxExc(0, 1))`) may mean that small changes in the uniform number lead to huge changes in the quantile, leading to gaps in sampling coverage (Monahan 1985, sec. 4 and 6)[145].

The following method samples from a distribution via inversion, with an accuracy of 1/`BASE` `` `precision` `` ((Devroye and Gravel 2020)[146], but extended for any base; see also (Bringmann and Friedrich 2013, Appendix A)[147]). In the method, `ICDF(u, ubits, prec)` returns a two-item list containing upper and lower bounds, respectively, of a number that is within 1/`BASE` `` `prec` `` of the true quantile of `u`/`BASE` `` `ubits` `` , and `BASE` is the digit base (for example, 2 for binary or 10 for decimal). For this method to work, the quantile function must be continuous on the interval (0, 1) except at a countable number of points (countable means each discontinuous point can be mapped to a different integer).

```
METHOD Inversion(precision)
   u=0
   ubits=0
   threshold=MakeRatio(1,pow(BASE, precision))*2
   incr=8
   while true // until a value is returned
      incr=8
      if ubits==0: incr=precision
      // NOTE: If a uniform number (`n`) is already pregenerated,
      // use the following instead:
      // u = rem(floor(n*pow(BASE, ubits+incr)), pow(BASE, incr))
      u=u*pow(BASE,incr)+RNDINTEXC(pow(BASE,incr))
      ubits=ubits+incr
      // Get upper and lower bound
      bounds=ICDF(u,ubits,precision)
      if lower>upper: return error
      diff=bounds[1]-bounds[0]
      if diff<=threshold: return bounds[1]+diff/2
   end
end
```

Devroye and Gravel (2020, Theorem 8)[148] proved the following statement. If `X` is a random variate with quantile function `QX(x)`, and `Y` is a variate that approximates `X` and has quantile function `QY(x)`, then the *Wasserstein distance* between X and Y is the least upper bound of `abs(QX(x)-QY(x))` for every `x` greater than 0 and less than 1. This means that, if `QY(x)` is within `epsilon` of `QX(x)` where `0 < x < 1`, then (in theory) an application can sample a random variate that is close to `X` with an accuracy of `epsilon` by sampling `Y = QY(RNDRANGEMinMaxExc(0, 1))`.

Some applications need to convert a pregenerated number between 0 and 1 (usually a number sampled from a uniform distribution), called `u01` below, to a non-uniform distribution via quantiles. Notable cases include copula methods, order statistics, and Monte Carlo methods involving low-discrepancy sequences. The following way to compute quantiles is exact in theory:

[145]Monahan, J.F., "Accuracy in Random Number Generation", *Mathematics of Computation* 45(172), 1985.

[146]Devroye, L., Gravel, C., "**Random variate generation using only finitely many unbiased, independently and identically distributed random bits**", arXiv:1502.02539v6 [cs.IT], 2020. https://arxiv.org/abs/1502.02539v6

[147]Bringmann, K., and Friedrich, T., 2013, July. Exact and efficient generation of geometric random variates and random graphs, in *International Colloquium on Automata, Languages, and Programming* (pp. 267-278).

[148]Devroye, L., Gravel, C., "**Random variate generation using only finitely many unbiased, independently and identically distributed random bits**", arXiv:1502.02539v6 [cs.IT], 2020. https://arxiv.org/abs/1502.02539v6

- Distribution is **discrete, with known PMF** (and the distribution takes on integers): Sequential search (Devroye 1986, p. 85)[149]: `i = 0; p = PMF(i); while u01 > p; u01 = u01 - p; i = i + 1; p = PMF(i); end; return p`, but this is not always fast. (This works only if `PMF`'s values sum to 1, which is why a PMF and not a PDF-like function is allowed here.)

In addition, the following methods approximate the quantile:

- Distribution is **discrete, with known PDF-like function** (and the distribution takes on integers): If the interval [a, b] covers all or almost all the distribution, then the application can store the PDF-like function's values in that interval in a list and call `WChoose`: `wsum = 0; for i in a..b: wsum=wsum+PDF(i); for i in a..b: AddItem(weights, PDF(i)); return a + WChoose(weights, u01 * wsum)`. [150] (In this case, the method is exact in theory for sampling the original distribution restricted to [a, b].) See also `integers_from_u01` in the **Python sample code**[151].
- Distribution is **absolutely continuous, with known PDF-like function**: `ICDFFromContPDF(u01, mini, maxi, step)`, below, finds an approximate quantile based on a piecewise linear approximation of the PDF-like function in [`mini`, `maxi`], with pieces up to `step` wide. This method does not currently allow controlling for the approximation error in sampling the distribution. See also `DensityInversionSampler` and `numbers_from_dist_inversion` (Derflinger et al. 2010)[152], (Devroye and Gravel 2020)[153] in the Python sample code [154].
- Distribution is **absolutely continuous, with known CDF**: If the interval [a, b] covers all or almost all the distribution, and the CDF is continuous and strictly increasing on that interval, then let `D` be the original distribution restricted to [a, b]. Then it's possible to sample from a distribution that is close to `D` by a Wasserstein distance of no more than `eps` (Devroye and Gravel 2020, especially Theorem 8)[155] by the following method [156]:
  - In a setup phase: Create an empty list. Then, at values of `x` in [a, b] spaced evenly with a step size of `eps` or less, starting at `a` and ending at `b`, add the sublist `[x, (CDF(x)-a)/(b-a)]` to the list. The first item in the sublist is the *sampled point* `x`, and the second item is the *adjusted CDF value*.
  - In a sampling phase: Find the greatest adjusted CDF value less than or equal to `u01`, and the largest one greater than or equal to `u01`, such as by a binary search. (These two values will be the same if `u01` is one of the adjusted CDF values.) Call their sampled points `y` and `z`, respectively. Then return either `y`, if `y=z`; or `y+(z-y)*(u01-y)/(z-y)` otherwise.

```
METHOD ICDFFromContPDF(u01, mini, maxi, step)
  pieces=[]
```

[149] Devroye, L., ***Non-Uniform Random Variate Generation***, 1986.

[150] In floating-point arithmetic, finding the quantile based on the **CDF** instead of a PDF-like function can introduce more error (Walter 2019)[^81].

[151] https://peteroupc.github.io/randomgen.zip

[152] Gerhard Derflinger, Wolfgang Hörmann, and Josef Leydold, "Random variate generation by numerical inversion when only the density is known", ACM Transactions on Modeling and Computer Simulation 20(4) article 18, October 2010.

[153] Devroye, L., Gravel, C., "**Random variate generation using only finitely many unbiased, independently and identically distributed random bits**", arXiv:1502.02539v6 [cs.IT], 2020. https://arxiv.org/abs/1502.02539v6

[154] Part of `numbers_from_u01` uses algorithms described in Arnas, D., Leake, C., Mortari, D., "Random Sampling using k-vector", *Computing in Science & Engineering* 21(1) pp. 94-107, 2019, and Mortari, D., Neta, B., "k-Vector Range Searching Techniques".

[155] Devroye, L., Gravel, C., "**Random variate generation using only finitely many unbiased, independently and identically distributed random bits**", arXiv:1502.02539v6 [cs.IT], 2020. https://arxiv.org/abs/1502.02539v6

[156] There is a paper by Arnas et al. that describes approximate random sampling using the values of the CDF by the so-called *k-vector* technique, but the paper doesn't formally prove how good the approximation is. Arnas, D., Leake, C., Mortari, D., "Random Sampling using k-vector", Computing in Science & Engineering 21(1) pp. 94-107, 2019. See also Mortari, D., Neta, B., "k-Vector Range Searching Techniques".

```
  areas=[]
  // Setup
  lastvalue=i
  lastweight=PDF(i)
  cumuarea=0
  i = mini+step; while i <= maxi
     weight=i; value=PDF(i)
     cumuarea=cumuarea+abs((weight + lastweight) * 0.5 *
            (value - lastvalue))
     AddItem(pieces,[lastweight,weight,lastvalue,value])
     AddItem(areas,cumuarea)
     lastweight=weight;lastvalue=value
     if i==maxi: break
     i = min(i + step, maxi)
  end
  for i in 0...size(areas): areas[i]=areas[i]/cumuarea
  // Sampling
  prevarea=0
  for i in 0...size(areas)
     cu=areas[i]
     if u01<=cu
         p=pieces[i]; u01=(u01-prevarea)/(cu-prevarea)
         s=p[0]; t=p[1]; v=u01
         if s!=t: v=(s-sqrt(t*t*u01-s*s*u01+s*s))/(s-t)
         return p[2]+(p[3]-p[2])*v
     end
     prevarea=cu
  end
  return error
END METHOD
```

Notes:

1. If only percentiles of data (such as the median or 50th percentile, the minimum or 0th percentile, or the maximum or 100th percentile) are available, the quantile function can be approximated via those percentiles. The Nth percentile corresponds to the quantile for `N/100.0`. Missing values for the quantile function can then be filled in by interpolation (such as spline fitting). Sampling using only percentiles this way is an ill-posed problem, though. If the raw data points are available, see "**Sampling from a Distribution of Data Points**" instead.
2. Taking the `k`th smallest of `n` random variates distributed the same way is the same as taking the `k`th smallest of `n` *uniform* random variates in the interval [0, 1) (also known as the `k`th *order statistic*; for example, `BetaDist(k, n+1-k)`) and finding its quantile (Devroye 2006)[157]; (Devroye 1986, p. 30)[158].

8.6.3 Rejection Sampling with a PDF-Like Function

If the distribution **has a known PDF-like function** (`PDF`), and that function can be more easily sampled by another distribution with its own PDF-like function (`PDF2`) that "dominates" `PDF` in the sense that `PDF2(x) >= PDF(x)` at every valid `x`, then generate random variates with the latter distribution until a variate (call

[157] Devroye, L., "Non-Uniform Random Variate Generation". In *Handbooks in Operations Research and Management Science: Simulation*, Henderson, S.G., Nelson, B.L. (eds.), 2006, p.83.
[158] Devroye, L., ***Non-Uniform Random Variate Generation***, 1986.

it `n`) that satisfies `r < PDF(n)`, where `r = RNDRANGEMinMaxExc(0, PDF2(n))`, is generated this way (that is, sample points in `PDF2` until a point falls within `PDF`).

A variant of rejection sampling is the *squeeze principle*, in which a third PDF-like function (`PDF3`) is chosen that is "dominated" by the first one (`PDF`) and easier to sample than `PDF`. Here, a number is accepted if `r < PDF3(n)` or `r < PDF(n)` , where `r = RNDRANGEMinMaxExc(0, PDF2(n))` (Devroye 1986, p. 53)[159].

See also (von Neumann 1951)[160]; (Devroye 1986)[161], pp. 41-43; "**Rejection Sampling**"; and "**Generating Pseudorandom Numbers**[162]".

Examples:

1. To sample a random variate in the interval [`low`, `high`) from a PDF-like function with a positive maximum value no greater than `peak` at that interval, generate `x = RNDRANGEMinMaxExc(low, high)` and `y = RNDRANGEMinMaxExc(0, peak)` until `y < PDF(x)`, then take the last `x` generated this way. (See also Saucier 2000, pp. 6-7.) If the distribution **is discrete** and integer-valued, generate `x` with `x = RNDINTEXCRANGE(low, high)` instead.
2. A PDF-like function for a custom distribution, `PDF`, is `exp(-abs(x*x*x))`, and the exponential distribution's, `PDF2`, is `exp(-x)`. The exponential PDF-like function `PDF2` "dominates" `PDF` (at every `x` 0 or greater) if we multiply it by 1.5, so that `PDF2` is now `1.5 * exp(-x)`. Now we can generate numbers from our custom distribution by sampling exponential points until a point falls within `PDF`. This is done by generating `n = Expo(1)` until `RNDRANGEMinMaxExc(0, PDF2(n)) < PDF(n)`.
3. The normal distribution's upside-down bell curve has the PDF-like function `1-exp(-(x*x))`, and the highest point for this function is `peak = max(1-exp(-(low*low)), 1-exp(-(high*high)))`. Sampling this distribution then uses the algorithm in example 1.

Note: In the Python sample code, **moore.py**[163] and `numbers_from_dist` sample from a distribution via rejection sampling (Devroye and Gravel 2020)[164], (Sainudiin and York 2013)[165].

8.6.4 Alternating Series

If a PDF-like function for the target distribution is not known exactly, but can be approximated from above and below by two series expansions that converge to that function as more terms are added, the *alternating series method* (which is exact in theory) can be used. This still requires a "dominating" PDF-like function (`PDF2(x)`) to serve as the "easy-to-sample" distribution. Call the series expansions `UPDF(x, n)` and `LPDF(x, n)`, respectively, where `n` is the number of terms in the series to add. To sample the distribution using this method (Devroye 2006)[166]: (1) Sample from the "dominating" distribution, and let `x` be the sampled number; (2) set `n` to 0; (3) accept `x` if `r < LPDF(x, n)`, or go to step 1 if `r >= UPDF(x, n)`, or repeat this step with `n` increased by 1 if neither is the case, where `r = RNDRANGEMinMaxExc(0, PDF2(n))`.

[159]Devroye, L., ***Non-Uniform Random Variate Generation***, 1986.
[160]von Neumann, J., "Various techniques used in connection with random digits", 1951.
[161]Devroye, L., ***Non-Uniform Random Variate Generation***, 1986.
[162]https://mathworks.com/help/stats/generating-random-data.html
[163]https://github.com/peteroupc/peteroupc.github.io/blob/master/moore.py
[164]Devroye, L., Gravel, C., "**Random variate generation using only finitely many unbiased, independently and identically distributed random bits**", arXiv:1502.02539v6 [cs.IT], 2020. https://arxiv.org/abs/1502.02539v6
[165]Sainudiin, Raazesh, and Thomas L. York. "An Auto-Validating, Trans-Dimensional, Universal Rejection Sampler for Locally Lipschitz Arithmetical Expressions," *Reliable Computing* 18 (2013): 15-54.
[166]Devroye, L., "Non-Uniform Random Variate Generation". In *Handbooks in Operations Research and Management Science: Simulation*, Henderson, S.G., Nelson, B.L. (eds.), 2006, p.83.

8.6.5 Markov-Chain Monte Carlo

Markov-chain Monte Carlo[167] (MCMC) is a family of algorithms for sampling from a probability distribution by building a *Markov chain* of random values that approach the given distribution as the chain takes more steps. In general, however, MCMC is approximate, it doesn't allow for controlling the approximation error, and the values generated by a given chain will have a statistical *dependence* on each other (which is why techniques such as "thinning" — keeping only every Nth sample — are often employed).[168]

MCMC algorithms[169] include *Metropolis–Hastings*, *slice sampling*, and *Gibbs sampling* (see also the **Python sample code**[170]). The latter is special in that it uses not a PDF-like function, but two or more distributions, each of which uses a number sampled at random from the previous distribution (*conditional distributions*), that converge to a *joint distribution*.

> **Example:** In one Gibbs sampler, an initial value for `y` is chosen, then multiple `x`, `y` pairs of random variates are generated, where `x = BetaDist(y, 5)` then `y = Poisson(x * 10)`.

8.7 Piecewise Linear Distribution

Requires random real numbers.

A ***piecewise linear distribution*** describes an absolutely continuous distribution with weights at known points and other weights determined by linear interpolation (smoothing). The `PiecewiseLinear` method (in the pseudocode below) takes two lists as follows (see also (Kscischang 2019)[171]):

- `values` is a list of rational numbers. The numbers should be arranged in ascending order.
- `weights` is a list of rational-valued weights for the given numbers (where each number and its weight have the same index in both lists). The greater a number's weight, the greater the probability that a number close to that number will be chosen. Each weight should be 0 or greater.

```
METHOD PiecewiseLinear(values, weights)
  if size(values)!=size(weights) or size(values)==0: return error
  if size(values)==1: return values[0]
  areas=[]
  for i in 1...size(values)
     area=abs((weights[i] + weights[i-1]) *
         (values[i] - values[i-1]) / 2) // NOTE: Not rounded
     AddItem(areas,area)
  end
  // NOTE: If values and weights are rational
  // numbers, use `areas=NormalizeRatios(areas)` instead
  // of finding `areas` as given below.
  ratios=[]
  for w in areas: AddItem(ratios, FPRatio(w))
  areas=NormalizeRatios(ratios)
```

[167] https://en.wikipedia.org/wiki/Markov_chain_Monte_Carlo

[168] Many Markov chains converge to a *stationary distribution*. The chain's *mixing time* is how fast this happens; it's the number of steps after which the next draw will follow a distribution within ε of the stationary distribution. This approximate distribution is then sampled by initializing several Markov chains (independently at random), then running each of them for their mixing time ("burn-in"), then taking the next draw of each chain. For further information see Levin and Peres, *Markov chains and mixing times*, second edition, 2017.

[169] Tran, K.H., "**A Common Derivation for Markov Chain Monte Carlo Algorithms with Tractable and Intractable Targets**", arXiv:1607.01985v5 [stat.CO], 2018, gives a common framework for describing many MCMC algorithms, including Metropolis–Hastings, slice sampling, and Gibbs sampling. https://arxiv.org/abs/1607.01985v5

[170] https://peteroupc.github.io/randomgen.zip

[171] Kschischang, Frank R. "A Trapezoid-Ziggurat Algorithm for Generating Gaussian Pseudorandom Variates." (2019).

```
  index=WeightedChoice(areas)
  w=values[index+1]-values[index]
  if w==0: return values[index]
  m=(weights[index+1]-weights[index])/w
  h2=(weights[index+1]+weights[index])
  ww=w/2.0; hh=h2/2.0
  x=RNDRANGEMinMaxExc(-ww, ww)
  if RNDRANGEMinMaxExc(-hh, hh)>x*m: x=-x
  return values[index]+x+ww
END METHOD
```

> **Note:** The **Python sample code**[172] contains a variant to the method above for returning more than one random variate in one call.
>
> **Example**: Assume `values` is the following: `[0, 1, 2, 2.5, 3]`, and `weights` is the following: `[0.2, 0.8, 0.5, 0.3, 0.1]`. The weight for 2 is 0.5, and that for 2.5 is 0.3. Since 2 has a higher weight than 2.5, numbers near 2 have a greater probability of being chosen than numbers near 2.5 with the `PiecewiseLinear` method.

8.8 Specific Distributions

Methods to sample additional distributions are given in a **separate page**[173]. They cover the normal, gamma, beta, von Mises, stable, and multivariate normal distributions as well as copulas.

8.9 Index of Non-Uniform Distributions

Many distributions here require random real numbers.

A † symbol next to a distribution means that a sample from the distribution can be shifted by a location parameter (`mu`) then scaled by a scale parameter greater than 0 (`sigma`). Example: `num * sigma + mu`.

A symbol next to a distribution means the sample can be scaled to any range, which is given with the minimum and maximum values `mini` and `maxi`. Example: `mini + (maxi - mini) * num`.

For further examples and distributions, see (Devroye 1996)[174] and (Crooks 2019)[175].

Most commonly used:

- **Beta distribution** : See **Beta Distribution**[176].
- **Binomial distribution**: See **Binomial Distribution**.
- **Binormal distribution**: See **Multivariate Normal (Multinormal) Distribution**[177].
- **Cauchy (Lorentz) distribution**†: `Stable(1, 0)`. This distribution is similar to the normal distribution, but with "fatter" tails. Alternative algorithm based on one mentioned in (McGrath and Irving 1975)[178]: Generate `x = RNDRANGEMinMaxExc(0,1)` and `y = RNDRANGEMinMaxExc(0,1)` until `x * x + y * y <= 1`, then generate `(RNDINT(1) * 2 - 1) * y / x`.

[172] https://peteroupc.github.io/randomgen.zip
[173] https://peteroupc.github.io/randomnotes.html
[174] Devroye, L., 1996, December, "Random variate generation in one line of code" In *Proceedings Winter Simulation Conference* (pp. 265-272). IEEE.
[175] Crooks, G.E., ***Field Guide to Continuous Probability Distributions***, 2019. https://threeplusone.com/pubs/FieldGuide.pdf
[176] https://peteroupc.github.io/randomnotes.html#Beta_Distribution
[177] https://peteroupc.github.io/randomnotes.html#Multivariate_Normal_Multinormal_Distribution
[178] McGrath, E.J., Irving, D.C., "Techniques for Efficient Monte Carlo Simulation, Volume II", Oak Ridge National Laboratory, April 1975.

- **Chi-squared distribution**: `GammaDist(df * 0.5 + Poisson(sms * 0.5))*2`, where `df` is the number of degrees of freedom and `sms` is the sum of mean squares (where `sms` other than 0 indicates a *noncentral* distribution).
- **Dice**: See **Dice**.
- **Exponential distribution**: See **Exponential Distribution**. The naïve implementation `-ln(1-RNDRANGEMinMaxExc(0, 1)) / lamda` has several problems, such as being ill-conditioned at large values because of the distribution's right-sided tail (Pedersen 2018) [179]. An application can reduce some of these problems by applying Pedersen's suggestion of using either `-ln(RNDRANGEMinMaxExc(0, 0.5))` or `-log1p(-RNDRANGEMinMaxExc(0, 0.5))` (rather than `-ln(1-RNDRANGEMinMaxExc(0, 1))`), chosen uniformly at random each time; an alternative is `ln(1/RNDRANGEMinMaxExc(0,1))` mentioned in (Devroye 2006)[180].
- **Extreme value distribution**: See generalized extreme value distribution.
- **Gamma distribution**: See **Gamma Distribution**[181]. Generalized gamma distributions include the **Stacy distribution** (`pow(GammaDist(a), 1.0 / c) * b`, where `c` is another shape parameter) and the **Amoroso distribution** (Crooks 2015)[182], (`pow(GammaDist(a), 1.0 / c) * b + d`, where `d` is the minimum value).
- **Gaussian distribution**: See **Normal (Gaussian) Distribution**[183].
- **Geometric distribution**: See **Geometric Distribution**. The following is "exact" assuming computers can operate "exactly" on real numbers: `floor(-Expo(1)/ln(1-p))` (Devroye 1986, p. 500)[184] (ceil replaced with floor because this page defines geometric distribution differently).
- **Gumbel distribution**: See generalized extreme value distribution.
- **Inverse gamma distribution**: `b / GammaDist(a)`, where `a` and `b` have the same meaning as in the gamma distribution. Alternatively, `1.0 / (pow(GammaDist(a), 1.0 / c) / b + d)`, where `c` and `d` are shape and location parameters, respectively.
- **Laplace (double exponential) distribution**†: `(Expo(1) - Expo(1))`. Also, `Normal(0,1) * Normal(0, 1) - Normal(0, 1) * Normal(0, 1)` (Kotz et al. 2012)[185].
- **Logarithmic distribution**: `RNDRANGEMinMaxExc(0, 1) * RNDRANGEMinMaxExc(0, 1)` (Saucier 2000, p. 26). In this distribution, lower numbers are exponentially more probable than higher numbers.
- **Logarithmic normal distribution**: `exp(Normal(mu, sigma))`, where `mu` and `sigma` are the underlying normal distribution's parameters.
- **Multinormal distribution**: See multivariate normal distribution.
- **Multivariate normal distribution**: See **Multivariate Normal (Multinormal) Distribution**[186].
- **Normal distribution**: See **Normal (Gaussian) Distribution**[187].
- **Poisson distribution**: See "**Poisson Distribution**". The following is "exact" assuming computers can operate "exactly" on real numbers (Devroye 1986, p. 504)[188]: `c = 0; s = 0; while true; sum = sum + Expo(1); if sum>=mean: return c; else: c = c + 1; end`; and in addition the following optimization from (Devroye 1991)[189] can be used: `while mean > 20; n=ceil(mean-pow(mean,0.7)); g=GammaDist(n); if g>=mean: return c+(n-1-Binomial(n-1,(g-mean)/g)); mean = mean - g; c = c + n; end`.

[179] Pedersen, K., "**Reconditioning your quantile function**", arXiv:1704.07949v3 [stat.CO], 2018. https://arxiv.org/abs/1704.07949

[180] Devroye, L., "Non-Uniform Random Variate Generation". In *Handbooks in Operations Research and Management Science: Simulation*, Henderson, S.G., Nelson, B.L. (eds.), 2006, p.83.

[181] https://peteroupc.github.io/randomnotes.html#Gamma_Distribution

[182] Crooks, G.E., "**The Amoroso Distribution**", arXiv:1005.3274v2 [math.ST], 2015. https://arxiv.org/abs/1005.3274v2

[183] https://peteroupc.github.io/randomnotes.html#Normal_Gaussian_Distribution

[184] Devroye, L., ***Non-Uniform Random Variate Generation***, 1986.

[185] Kotz, Samuel, Tomasz Kozubowski, and Krzystof Podgórski. The Laplace distribution and generalizations: a revisit with applications to communications, economics, engineering, and finance. Springer Science & Business Media, 2012.

[186] https://peteroupc.github.io/randomnotes.html#Multivariate_Normal_Multinormal_Distribution

[187] https://peteroupc.github.io/randomnotes.html#Normal_Gaussian_Distribution

[188] Devroye, L., ***Non-Uniform Random Variate Generation***, 1986.

[189] Devroye, L., "Expected Time Analysis of a Simple Recursive Poisson Random Variate Generator", *Computing* 46, pp. 165-173, 1991.

- **Pareto distribution**: `pow(RNDRANGEMinMaxExc(0, 1), -1.0 / alpha) * minimum`, where `alpha` is the shape and `minimum` is the minimum.
- **Rayleigh distribution**†: `sqrt(Expo(0.5))`. If the scale parameter (`sigma`) follows a logarithmic normal distribution, the result is a *Suzuki distribution*.
- **Standard normal distribution**†: `Normal(0, 1)`. See also **Normal (Gaussian) Distribution**[190].
- **Student's *t*-distribution**: `Normal(cent, 1) / sqrt(GammaDist(df * 0.5)*2 / df)`, where `df` is the number of degrees of freedom, and *cent* is the mean of the normally-distributed random variate. A `cent` other than 0 indicates a *noncentral* distribution. Alternatively, `cos(RNDRANGEMinMaxExc(0, pi * 2)) * sqrt((pow(RNDRANGEMinMaxExc(0, 1),-2.0/df)-1) * df)` (Bailey 1994)[191].
- **Triangular distribution**† (Stein and Keblis (2009)[192]): `(1-alpha) * min(a, b) + alpha * max(a, b)`, where `alpha` is in [0, 1], `a = RNDRANGEMinMaxExc(0, 1)`, and `b = RNDRANGEMinMaxExc(0, 1)`.
- **Weibull distribution**: See generalized extreme value distribution.

Miscellaneous:

- **Archimedean copulas**: See **Gaussian and Other Copulas**[193].
- **Arcsine distribution** : `BetaDist(0.5, 0.5)` (Saucier 2000, p. 14).
- **Bates distribution**: See **Transformations of Random Variates: Additional Examples**[194].
- **Beckmann distribution**: See **Multivariate Normal (Multinormal) Distribution**[195].
- **Beta binomial distribution**: `Binomial(trials, BetaDist(a, b))`, where `a` and `b` are the two parameters of the beta distribution, and `trials` is a parameter of the binomial distribution.
- **Beta negative binomial distribution**: `NegativeBinomial(successes, BetaDist(a, b))`, where `a` and `b` are the two parameters of the beta distribution, and `successes` is a parameter of the negative binomial distribution. If *successes* is 1, the result is a *Waring–Yule distribution*. A *Yule–Simon distribution* results if *successes* and *b* are both 1 (for example, in *Mathematica*) or if *successes* and *a* are both 1 (in other works).
- **Beta-PERT distribution**: `startpt + size * BetaDist(1.0 + (midpt - startpt) * shape / size, 1.0 + (endpt - midpt) * shape / size)`. The distribution starts at `startpt`, peaks at `midpt`, and ends at `endpt`, `size` is `endpt - startpt`, and `shape` is a shape parameter that's 0 or greater, but usually 4. If the mean (`mean`) is known rather than the peak, `midpt = 3 * mean / 2 - (startpt + endpt) / 4`.
- **Beta prime distribution**†: `pow(GammaDist(a), 1.0 / alpha) / pow(GammaDist(b), 1.0 / alpha)`, where `a`, `b`, and `alpha` are shape parameters. If *a* is 1, the result is a *Singh–Maddala distribution*; if *b* is 1, a *Dagum distribution*; if *a* and *b* are both 1, a *logarithmic logistic distribution*.
- **Birnbaum–Saunders distribution**: `pow(sqrt(4+x*x)+x,2)/(4.0*lamda)`, where `x = Normal(0,gamma)`, `gamma` is a shape parameter, and `lamda` is a scale parameter.
- **Borel distribution** (Borel (1942)[196]): `r=0; q=1; while q>=1; q+=Poisson(la); q-=1; r+=1; end; return r`. `la`, the mean number of arrivals, should be in the interval (0, 1).
- **Chi distribution**: Square root of a chi-squared random variate. See chi-squared distribution.
- **Compound Poisson distribution**: See **Transformations of Random Variates: Additional Examples**.
- **Cosine distribution** : `atan2(x, sqrt(1 - x * x)) / pi`, where `x = (RNDINT(1) * 2 - 1) * RNDRANGEMinMaxExc(0, 1)` (Saucier 2000, p. 17; inverse sine replaced with `atan2` equivalent).

[190] https://peteroupc.github.io/randomnotes.html#Normal_Gaussian_Distribution

[191] Bailey, R.W., "Polar generation of random variates with the t distribution", *Mathematics of Computation* 62 (1994).

[192] Stein, W.E. and Keblis, M.F., "A new method to simulate the triangular distribution", *Mathematical and Computer Modelling* 49(5-6), 2009, pp.1143-1147.

[193] https://peteroupc.github.io/randomnotes.html#Gaussian_and_Other_Copulas

[194] https://peteroupc.github.io/randomnotes.html#Transformations_of_Random_Numbers_Additional_Examples

[195] https://peteroupc.github.io/randomnotes.html#Multivariate_Normal_Multinormal_Distribution

[196] Borel, E., "Sur l'emploi du théorème de Bernoulli pour faciliter le calcul d'un infinité de coefficients. Application au probleme de l'attente à un guichet", 1942.

- **CUB or MUB distribution** (Piccolo (2003)[197]): `if ZeroOrOne(px,py)==1: return 1+BinomialInt(m-1, zy-zx, zy); else: return RNDINTRANGE(1, m)`, where `m>=3`, `px/py` is in [0, 1], and `zx/zy` is in [0, 1].
- **Dagum distribution**: See beta prime distribution.
- **Dirichlet distribution**: Suppose we (1) generate n+1 random **gamma-distributed**[198] variates, each with separate parameters; (2) take their sum; (3) divide each of them by that sum; then (4) multiply each of them by a real number `x` greater than 0. Then:
  - After step (4), if `x` was 1, the **Dirichlet distribution**[199] (for example, (Devroye 1986)[200], p. 593-594) models the first n of those numbers.
  - If the numbers at step (1) were each generated as `Expo(1)` (a special case of the gamma distribution), the result after step (4) is a uniformly distributed sum of n+1 numbers that sum to `x` (see also linked article above).
- **Double logarithmic distribution**: `(0.5 + (RNDINT(1) * 2 - 1) * RNDRANGEMinMaxExc(0, 0.5) * RNDRANGEMinMaxExc(0, 1))` (see also Saucier 2000, p. 15, which shows the wrong X axes).
- **Erlang distribution**: `GammaDist(n)/lamda`, where `n` is an integer greater than 0. Returns a number that simulates a sum of `n` exponential random variates with the given `lamda` parameter.
- **Estoup distribution**: See zeta distribution.
- **Exponential power distribution** (generalized normal distribution version 1): `(RNDINT(1) * 2 - 1) * pow(GammaDist(1.0/a), a)`, where `a` is a shape parameter.
- **Fréchet distribution**: See generalized extreme value distribution.
- **Fréchet–Hoeffding lower bound copula**: See **Gaussian and Other Copulas**.
- **Fréchet–Hoeffding upper bound copula**: See **Gaussian and Other Copulas**.
- **Gaussian copula**: See **Gaussian and Other Copulas**.
- **Generalized extreme value (Fisher–Tippett or generalized maximum value) distribution (`GEV(c)`)**†: `(pow(Expo(1), -c) - 1) / c` if `c != 0`, or `-ln(Expo(1))` otherwise, where `c` is a shape parameter. Special cases:
  - The negative of the result expresses a generalized minimum value. In this case, a parameter of `c = 0` results in a *Gumbel distribution*.
  - A parameter of `c = 0` results in an *extreme value distribution*.
  - **Weibull distribution**: `1 - 1.0/a * GEV(-1.0/a)` (or `pow(Expo(1), 1.0/a)`), where `a` is a shape parameter.
  - **Fréchet distribution**: `1 + 1.0/a * GEV(1.0/a)` (or `pow(Expo(1), -1.0/a)`), where `a` is a shape parameter.
- **Generalized Tukey lambda distribution**: `(s1 * (pow(x, lamda1)-1.0)/lamda1 - s2 * (pow(1.0-x, lamda2)-1.0)/lamda2) + loc`, where `x` is `RNDRANGEMinMaxExc(0, 1)`, `lamda1` and `lamda2` are shape parameters, `s1` and `s2` are scale parameters, and `loc` is a location parameter.
- **Half-normal distribution**. Parameterizations include:
  - *Mathematica*: `abs(Normal(0, sqrt(pi * 0.5) / invscale))`, where `invscale` is a parameter of the half-normal distribution.
  - MATLAB: `abs(Normal(mu, sigma))`, where `mu` and `sigma` are the underlying normal distribution's parameters.
- **Hyperexponential distribution**: See **Mixtures of Distributions**.
- **Hypergeometric distribution**: See **Polya–Eggenberger Distribution**.
- **Hypoexponential distribution**: See **Transformations of Random Variates**.
- **Inverse chi-squared distribution**†: `df / (GammaDist(df * 0.5)*2)`, where `df` is the number of degrees of freedom. The scale parameter (`sigma`) is usually `1.0 / df`.
- **Inverse Gaussian distribution (Wald distribution)**: Generate `n = mu + (mu*mu*y/(2*lamda))`

[197] Piccolo, Domenico. "On the moments of a mixture of uniform and shifted binomial random variables." Quaderni di Statistica 5, no. 1 (2003): 85-104.

[198] https://peteroupc.github.io/randomnotes.md#Gamma_Distribution

[199] https://en.wikipedia.org/wiki/Dirichlet_distribution

[200] Brownlee, J. "**A Gentle Introduction to the Bootstrap Method**", *Machine Learning Mastery*, May 25, 2018. https://machinelearningmastery.com/a-gentle-introduction-to-the-bootstrap-method/

`- mu * sqrt(4 * mu * lamda * y + mu * mu * y * y) / (2 * lamda)`, where `y = pow(Normal(0, 1), 2)`, then return `n` with probability `mu / (mu + n)` (for example, if `RNDRANGEMinMaxExc(0, 1) <= mu / (mu + n)`), or `mu * mu / n` otherwise. `mu` is the mean and `lamda` is the scale; both parameters are greater than 0. Based on method published in (Devroye 1986)[201].

- **kth-order statistic**: `BetaDist(k, n+1-k)`. Returns the `k`th smallest out of `n` uniform random variates in [**0, 1). See also (Devroye 1986, p. 210)[202].
- **Kumaraswamy distribution** : `pow(BetaDist(1, b), 1.0 / a)`, where `a` and `b` are shape parameters.
- **Landau distribution**: See stable distribution.
- **Lévy distribution**†: `0.5 / GammaDist(0.5)`. The scale parameter (`sigma`) is also called dispersion.
- **Logarithmic logistic distribution**: See beta prime distribution.
- **Logarithmic series distribution**: Generate `n = NegativeBinomialInt(1, py - px, py)+1` (where `px`/`py` is a parameter in (0,1)), then return `n` if `ZeroOrOne(1, n) == 1`, or repeat this process otherwise (Flajolet et al., 2010)[203]. The following is "exact" assuming computers can operate "exactly" on real numbers: `floor(1.0 - Expo(log1p(-pow(1.0 - p, RNDRANGEMinMaxExc(0, 1)))))`, where `p` is the parameter in (0, 1); see (Devroye 1986)[204].
- **Logistic distribution**†: `(ln(x)-log1p(-x))` (***logit function***), where `x` is `RNDRANGEMinMaxExc(0, 1)`.
- **Log-multinormal distribution**: See **Multivariate Normal (Multinormal) Distribution**.
- **Max-of-uniform distribution**: `BetaDist(n, 1)`. Returns a number that simulates the largest out of `n` uniform random variates in [**0, 1). See also (Devroye 1986, p. 675)[205].
- **Maxwell distribution**†: `sqrt(GammaDist(1.5)*2)`.
- **Min-of-uniform distribution**: `BetaDist(1, n)`. Returns a number that simulates the smallest out of `n` uniform random variates in [**0, 1). See also (Devroye 1986, p. 210)[206].
- **Moyal distribution**: See the **Python sample code**[207].
- **Multinomial distribution**: See **Multinomial Distribution**.
- **Multivariate Poisson distribution**: See the **Python sample code**[208].
- **Multivariate *t*-copula**: See the **Python sample code**[209].
- **Multivariate *t*-distribution**: See the **Python sample code**[210].
- **Negative binomial distribution** (`NegativeBinomial(successes, p)`): See **Negative Binomial Distribution**. The following is "exact" assuming computers can operate "exactly" on real numbers: `Poisson(GammaDist(successes)*(1 - p) / p)` (works even if `successes` is not an integer).
- **Negative multinomial distribution**: See the **Python sample code**[211].
- **Noncentral beta distribution** : `BetaDist(a + Poisson(nc), b)`, where `nc` (a noncentrality), `a`, and `b` are greater than 0.
- **Parabolic distribution** : `BetaDist(2, 2)` (Saucier 2000, p. 30).
- **Pascal distribution**: `NegativeBinomial(successes, p) + successes`, where `successes` and `p` have the same meaning as in the negative binomial distribution, except `successes` is always an integer.
- **Pearson VI distribution**: `GammaDist(v) / GammaDist(w)`, where `v` and `w` are shape parameters greater than 0 (Saucier 2000, p. 33; there, an additional `b` parameter is defined, but that parameter is canceled out in the source code).
- **Piecewise constant distribution**: See **Weighted Choice With Replacement**.

[201] Devroye, L., ***Non-Uniform Random Variate Generation***, 1986.
[202] Devroye, L., ***Non-Uniform Random Variate Generation***, 1986.
[203] Flajolet, P., Pelletier, M., Soria, M., "**On Buffon machines and numbers**", arXiv:0906.5560v2 [math.PR], 2010. https://arxiv.org/abs/0906.5560v2
[204] Devroye, L., ***Non-Uniform Random Variate Generation***, 1986.
[205] Devroye, L., ***Non-Uniform Random Variate Generation***, 1986.
[206] Devroye, L., ***Non-Uniform Random Variate Generation***, 1986.
[207] https://peteroupc.github.io/randomgen.zip
[208] https://peteroupc.github.io/randomgen.zip
[209] https://peteroupc.github.io/randomgen.zip
[210] https://peteroupc.github.io/randomgen.zip
[211] https://peteroupc.github.io/randomgen.zip

- **Piecewise linear distribution**: See **Piecewise Linear Distribution**.
- **Pólya–Aeppli distribution**: See **Transformations of Random Variates**.
- **Power distribution**: `BetaDist(alpha, 1) / b`, where `alpha` is the shape and `b` is the domain. Nominally in the interval (0, 1).
- **Power law distribution**: `pow(RNDRANGEMinMaxExc(pow(mn,n+1),pow(mx,n+1)), 1.0 / (n+1))`, where `n` is the exponent, `mn` is the minimum, and `mx` is the maximum. **Reference**.
- **Power lognormal distribution**: See the **Python sample code**[212].
- **Power normal distribution**: See the **Python sample code**[213].
- **Product copula**: See **Gaussian and Other Copulas**[214].
- **Rice distribution**: See **Multivariate Normal (Multinormal) Distribution**[215].
- **Rice–Norton distribution**: See **Multivariate Normal (Multinormal) Distribution**[216].
- **Singh–Maddala distribution**: See beta prime distribution.
- **sin^k distribution**: Generate `x = BetaDist(k+1, k+1) * pi`, then return `x` if `0-Expo(k) <= ln(pi*pi*sin(x) / ((4*x*(pi - x)))`, or repeat this process otherwise (Makalic and Schmidt 2018)[217].
- **Skellam distribution**: `Poisson(mean1) - Poisson(mean2)`, where `mean1` and `mean2` are the means used in the `Poisson` method.
- **Skew normal distribution**† (Ghorbanzadeh et al. 2014)[218]: Generate `c*max(a, b) + (1-c)*min(a, b)`, where a = `Normal(0, 1)` and independently, `b = Normal(0, 1)`, and `c = (1+th)/sqrt(2.0*(1+th))`, and `th` is a real number in [0, 1]. Special cases: If `th=0`, generate `Normal(0, 1)`; if `th=1`, generate `max(a, b)`; if `th=1`, generate `min(a, b)`.
- **Snedecor's (Fisher's) *F*-distribution**: `GammaDist(m * 0.5)*n / (GammaDist(n * 0.5 + Poisson(sms * 0.5)) * m)`, where `m` and `n` are the numbers of degrees of freedom of two random variates with a chi-squared distribution, and if `sms` is other than 0, one of those distributions is *noncentral* with sum of mean squares equal to `sms`.
- **Stable distribution**: See **Stable Distribution**[219]. *Four-parameter stable distribution*: `Stable(alpha, beta) * sigma + mu`, where `mu` is the mean and `sigma` is the scale; if `alpha` and `beta` are 1, the result is a *Landau distribution*. *"Type 0" stable distribution*: `Stable(alpha, beta) * sigma + (mu - sigma * beta * x)`, where `x` is `ln(sigma)*2.0/pi` if `alpha` is 1, and `tan(pi*0.5*alpha)` otherwise.
- **Standard complex normal distribution**: See **Multivariate Normal (Multinormal) Distribution**[220].
- **Suzuki distribution**: See Rayleigh distribution.
- **Tukey lambda distribution**: `(pow(x, lamda)-pow(1.0-x,lamda))/lamda`, where `x` is `RNDRANGEMinMaxExc(0, 1)` and `lamda` is a shape parameter.
- **Twin-*t* distribution** (Baker and Jackson 2018)[221]: Generate `x`, a random Student's *t*-distributed number (not a noncentral one). Accept `x` with probability `z = pow((1 + y) / ((1 + y * y) + y), (df + 1) * 0.5)` (for example, if `RNDRANGEMinMaxExc(0, 1) < z`), where `y = x * x / df` and `df` is the degrees of freedom used to generate the number; repeat this process otherwise.
- **von Mises distribution**: See **von Mises Distribution**[222].

[212] https://peteroupc.github.io/randomgen.zip
[213] https://peteroupc.github.io/randomgen.zip
[214] https://peteroupc.github.io/randomnotes.html#Gaussian_and_Other_Copulas
[215] https://peteroupc.github.io/randomnotes.html#Multivariate_Normal_Multinormal_Distribution
[216] https://peteroupc.github.io/randomnotes.html#Multivariate_Normal_Multinormal_Distribution
[217] Makalic, E., Schmidt, D.F., "**An efficient algorithm for sampling from sin^k(x) for generating random correlation matrices**", arXiv:1809.05212v2 [stat.CO], 2018. https://arxiv.org/abs/1809.05212v2
[218] Ghorbanzadeh, D., Jaupi, L., Durand, P., "**A Method to Simulate the Skew Normal Distribution**", *Applied Mathematics* 5(13), 2014. https://www.scirp.org/html/24-7402277_47986.htm
[219] https://peteroupc.github.io/randomnotes.html#Stable_Distribution
[220] https://peteroupc.github.io/randomnotes.html#Multivariate_Normal_Multinormal_Distribution
[221] Baker, R., Jackson, D., "**A new distribution for robust least squares**", arXiv:1408.3237 [stat.ME], 2018. https://arxiv.org/abs/1408.3237
[222] https://peteroupc.github.io/randomnotes.html#von_Mises_Distribution

- **Waring–Yule distribution**: See beta negative binomial distribution.
- **Wigner (semicircle) distribution**†: `(BetaDist(1.5, 1.5)*2-1)`. The scale parameter (`sigma`) is the semicircular radius.
- **Yule–Simon distribution**: See beta negative binomial distribution.
- **Zeta distribution**: Generate `n = floor(pow(RNDRANGEMinMaxExc(0, 1), -1.0 / r))`, and if `d / pow(2, r) < RNDRANGEMinMaxExc((d - 1) * n / (pow(2, r) - 1.0))`, where `d = pow((1.0 / n) + 1, r)`, repeat this process. The parameter `r` is greater than 0. Based on method described in (Devroye 1986)[223]. A zeta distribution **truncated** by rejecting random values greater than some integer greater than 0 is called a *Zipf distribution* or *Estoup distribution*. (Devroye uses "Zipf distribution" to refer to the untruncated zeta distribution.)
- **Zipf distribution**: See zeta distribution.

8.10 Geometric Sampling

Requires random real numbers.

This section contains ways to choose independent uniform random points in or on geometric shapes.

8.10.1 Random Points Inside a Simplex

The following pseudocode generates a random point inside an n-dimensional simplex (simplest convex figure, such as a line segment, triangle, or tetrahedron). It takes one parameter, *points*, a list consisting of the n plus one vertices of the simplex, all of a single dimension n or greater. The special case of 3 points came from Osada et al. (2002)[224].

```
METHOD VecAddProd(a, b, c)
  for j in 0...size(a): a[j]=a[j]+b[j]*c
END METHOD

METHOD RandomPointInSimplex(points):
   ret=NewList()
   if size(points) > size(points[0])+1: return error
   if size(points)==1 // Return a copy of the point
     for i in 0...size(points[0]): AddItem(ret,points[0][i])
     return ret
   end
   if size(points)==3
      // Equivalent to sqrt(RNDRANGEMinMaxExc(0, 1))
     rs=max(RNDRANGEMinMaxExc(0, 1), RNDRANGEMinMaxExc(0, 1))
     r2=RNDRANGEMinMaxExc(0, 1)
     ret=[0,0,0]
     VecAddProd(ret,points[0],1.0-rs)
     VecAddProd(ret,points[1],(1.0-r2)*rs)
     VecAddProd(ret,points[2],r2*rs)
     return ret
   end
   gammas=NewList()
   // Sample from the simplex
   for i in 0...size(points): AddItem(gammas, Expo(1))
   tsum=0 // Will store sum of all gammas
   for i in 0...size(gammas): tsum=tsum+gammas[i]
```

[223]Devroye, L., ***Non-Uniform Random Variate Generation***, 1986.
[224]Osada, R., Funkhouser, T., et al., "Shape Distributions", _ACM Transactions on Graphics 21(4), Oct. 2002.

```
   for i in 0...size(gammas): gammas[i] = gammas[i] / tsum
   gammas[size(gammas)-1]=0 // To omit last gamma in sum
   tot = 1.0 // Will store 1 minus the sum of all gammas
   for i in 0...size(gammas): tot=tot - gammas[i]
   // Build the final point
   for i in 0...size(points[0]): AddItem(ret, points[0][i]*tot)
   for i in 1...size(points): VecAddProd(
      ret, points[i], gammas[i-1])
   return ret
END METHOD
```

8.10.2 Random Points on a Sphere

The following pseudocode shows how to generate a random point on a sphere (surface of a ball) centered at the origin, with the following parameters:

- `dims`, the number of dimensions of the sphere (and of the random point).
- `radius`, the sphere's radius (if `radius` is 1, the result can also serve as a unit vector in `dims`-dimensional space).
- `p` is greater than 0, or is infinity, and describes the sphere's shape (if `p` is 2, the sphere is the usual one).

See Schechtmann and Zinn (1990)[225]. Here, EPD generates an *exponential power* random variate (Devroye 1986, pp. 174-175)[226].

```
METHOD PNorm(vec, p)
   ret=0
   if p==infinity
      for i in 0...size(vec): ret=max(ret,abs(vec[i]))
      return ret
   else
      for i in 0...size(vec): ret=ret+pow(abs(vec[i]),p)
      return pow(ret,1.0/p)
   end
END METHOD

METHOD EPD(p)
   # Infinity case is uniform in (-1,1) to be
   # appropriate for this section's purposes
   if p==infinity: return RNDRANGEMinMaxExc(-1,1)
   if p==2: return Normal(0,1)
   return (RNDINT(1) * 2 - 1)*pow(GammaDist(1/p),1/p)
END METHOD

METHOD RandomPointOnSphere(dims, radius, p)
  x=0
  while x==0
    ret=[]
    for i in 0...dims: AddItem(ret, EPD(p))
    x=PNorm(ret, p)
  end
  invnorm=radius/x
```

[225] Schechtman, G., Zinn, J., On the volume of intersection of two Lp^n balls. 1990.

[226] Devroye, L., ***Non-Uniform Random Variate Generation***, 1986.

```
  for i in 0...dims: ret[i]=ret[i]*invnorm
  return ret
END METHOD
```

Notes:

1. `PNorm(vec, p)`, also known as l `p` norm, is a generalized notion of distance. `p` can be any number 0 or greater, or can be infinity. `PNorm(vec, 2)` is the "usual" distance and, for instance, forms the "usual" versions of spheres, while `PNorm(vec, infinity)` forms a hypercube.
2. The **Python sample code**[227] contains an optimized method for points on a circle (2-dimensional sphere, `p=2`).

Example: To generate a random point on the surface of a cylinder running along the Z axis, generate random X and Y coordinates on a circle and generate a random Z coordinate by `RNDRANGEMinMaxExc(mn, mx)`, where `mn` and `mx` are the highest and lowest Z coordinates possible.

8.10.3 Random Points Inside a Box, Ball, Shell, or Cone

To generate a random point inside—

- an **N-dimensional box**, generate `RNDRANGEMinMaxExc(mn, mx)` for each coordinate, where `mn` and `mx` are the lower and upper bounds for that coordinate. For example—
  - to generate a random point inside a rectangle bounded in [0, 2) along the X axis and [3, 6) along the Y axis, generate `[RNDRANGEMinMaxExc(0,2), RNDRANGEMinMaxExc(3,6)]`, and
  - to generate a *complex number* with real and imaginary parts bounded in [0, 1], generate `[RNDRANGEMinMaxExc(0, 1), RNDRANGEMinMaxExc(0, 1)]`.
- an **N-dimensional ball**, centered at the origin, with a given radius, follow the pseudocode in `RandomPointOnSphere`, except replace `PNorm(ret, p)` with `pow(pow(PNorm(ret, p),p)+Expo(1),1.0/p)` (Barthe et al. 2005)[228]. [229]
- an **N-dimensional spherical shell** (a hollow ball), centered at the origin, with inner radius A and outer radius B (where A is less than B), generate a random point on the surface of an N-dimensional ball with radius equal to `pow(RNDRANGEMinMaxExc(pow(A, N), pow(B, N)), 1.0 / N)`[230].
- a **cone** with height `H` and radius `R` at its base, running along the Z axis, generate a random Z coordinate by `Z = max(max(RNDRANGEMinMaxExc(0, H), RNDRANGEMinMaxExc(0, H)), RNDRANGEMinMaxExc(0, H))`, then generate random X and Y coordinates inside a disc (2-dimensional ball) with radius equal to `max(RNDRANGEMinMaxExc(0,Z*(R/H)), RNDRANGEMinMaxExc(0,Z*(R/H)))`[231].

Example: To generate a random point inside a cylinder running along the Z axis, generate random X and Y coordinates inside a disc (2-dimensional ball) and generate a random Z coordinate by `RNDRANGEMinMaxExc(mn, mx)`, where `mn` and `mx` are the highest and lowest Z coordinates possible.

Notes:

1. The **Python sample code**[232] contains a method for generating a random point on the

[227] https://peteroupc.github.io/randomgen.zip

[228] Barthe, F., Guédon, O., et al., "A probabilistic approach to the geometry of the lP^N-ball", Annals of Probability 33(2), 2005.

[229] Alternatively, if `p` is an integer greater than 0, generate a random point on the surface of an ball with `N+p` dimensions and the given radius (for example, using `RandomPointOnSphere(N+p,radius,p)`), then discard the last `p` coordinates of that point (Corollary 1 of Lacko, V., & Harman, R. (2012). A conditional distribution approach to uniform sampling on spheres and balls in Lp spaces. *Metrika*, 75(7), 939-951).

[230] See the *Mathematics Stack Exchange* question titled "Random multivariate in hyperannulus", `questions/1885630`.

[231] See the *Stack Overflow* question "Uniform sampling (by volume) within a cone", `questions/41749411`. Square and cube roots replaced with maximums.

[232] https://peteroupc.github.io/randomgen.zip

surface of an ellipsoid modeling the Earth.

2. Sampling a half-ball, half-sphere, or half-shell can be done by sampling a full ball or shell and replacing one of the dimensions of the result with its absolute value.
3. Lacko and Harman (2012)[233] defined a family of *non-uniform* distributions of points inside a ball: generate `RandomPointOnSphere(dims, r*pow(BetaDist(dims/p, d/p), 1.0/p),p)` where `r>0` is the radius, `dims` and `p` are as in `RandomPointOnSphere`, and `d>=0` is a shape parameter. If `d = p`, the distribution is uniform in the ball.

8.10.4 Random Latitude and Longitude

To generate a random point on the surface of a sphere in the form of a latitude and longitude (in radians with west and south coordinates negative)[234]—

- generate the longitude `RNDRANGEMinMaxExc(-pi, pi)`, where the longitude is in the interval [- π , π), and
- generate the latitude `atan2(sqrt(1 - x * x), x) - pi / 2`, where `x = RNDRANGEMinMaxExc(-1, 1)` and the latitude is in the interval [- π /2, π /2] (the interval excludes the poles, which have many equivalent forms; if poles are not desired, generate `x` until neither -1 nor 1 is generated this way).

9 Acknowledgments

I acknowledge the commenters to the CodeProject version of this page, including George Swan, who referred me to the reservoir sampling method.

I also acknowledge Christoph Conrads, who gave suggestions in parts of this article.

10 Other Documents

The following are some additional articles I have written on the topic of randomization and pseudorandom variate generation. All of them are open-source.

- **Random Number Generator Recommendations for Applications**[235]
- **More Random Sampling Methods**[236]
- **Code Generator for Discrete Distributions**[237]
- **The Most Common Topics Involving Randomization**[238]
- **Partially-Sampled Random Numbers for Accurate Sampling of Continuous Distributions**[239]
- **Bernoulli Factory Algorithms**[240]
- **Testing PRNGs for High-Quality Randomness**[241]
- **Examples of High-Quality PRNGs**[242]

[233] Lacko, V., & Harman, R. (2012). A conditional distribution approach to uniform sampling on spheres and balls in Lp spaces. *Metrika*, 75(7), 939-951.
[234] Reference: **"Sphere Point Picking"** in MathWorld (replacing inverse cosine with `atan2` equivalent).
[235] https://peteroupc.github.io/random.html
[236] https://peteroupc.github.io/randomnotes.html
[237] https://peteroupc.github.io/autodist.html
[238] https://peteroupc.github.io/randomcommon.html
[239] https://peteroupc.github.io/exporand.html
[240] https://peteroupc.github.io/bernoulli.html
[241] https://peteroupc.github.io/randomtest.html
[242] https://peteroupc.github.io/hqprng.html

11 Notes

12 Appendix

12.1 Sources of Random Numbers

All the randomization methods presented on this page assume that we have an endless source of numbers such that—

- the numbers follow a *uniform distribution*, and
- each number is *chosen independently of any other choice*.

That is, the methods assume we have a **"source of (uniform) random numbers"**. (Thus, none of these methods *generate* random numbers themselves, strictly speaking, but rather, they assume we have a source of them already.)

However, this is an ideal assumption which is hard if not impossible to achieve in practice.

Indeed, most applications make use of *pseudorandom number generators* (PRNGs), which are algorithms that produce *random-behaving* numbers, that is, numbers that simulate the ideal "source of random numbers" mentioned above. As a result, the performance and quality of the methods on this page will depend in practice on the quality of the PRNG (or other generator of random-behaving numbers) even if they don't in theory.

The "source of random numbers" can be simulated by a wide range of devices and programs, including PRNGs, so-called "true random number generators", and application programming interfaces (APIs) that provide uniform random-behaving numbers to applications. An application ought to choose devices or programs that simulate the "source of random numbers" well enough for its purposes, including in terms of their statistical quality, "unguessability", or both. However, it is outside this document's scope to give further advice on this choice.

The randomization methods in this document are deterministic (that is, they produce the same values given the same state and input), regardless of what simulates the "source of random numbers" (such as a PRNG or a "true random number generator"). The exceptions are as follows:

- The methods do not "know" what numbers will be produced next by the "source of random numbers" (or by whatever is simulating that source).
- A few methods read lines from files of unknown size; they won't "know" the contents of those lines before reading them.

12.2 Implementation Considerations

1. **Shell scripts and Microsoft Windows batch files** are designed for running other programs, rather than general-purpose programming. However, batch files and `bash` (a shell script interpreter) might support a variable which returns a uniformly distributed "random" integer in the interval [0, 32767] (called `%RANDOM%` or `$RANDOM`, respectively); neither variable is designed for information security. Whenever possible, the methods in this document should not be implemented in shell scripts or batch files, especially if information security is a goal.
2. **Query languages such as SQL** have no procedural elements such as loops and branches. Moreover, standard SQL has no way to choose a number at random, but popular SQL dialects often do — with idiosyncratic behavior — and describing differences between SQL dialects is outside the scope of this document. Whenever possible, the methods in this document should not be implemented in SQL, especially if information security is a goal.

3. **Stateless PRNGs.** Most designs of pseudorandom number generators (PRNGs) in common use maintain an internal state and update that state each time they generate a pseudorandom number. But for ***stateless* PRNG designs**[243] (including so-called "splittable" PRNGs), `RNDINT()`, `NEXTRAND()`, and other random sampling methods in this document may have to be adjusted accordingly (usually by adding an additional parameter).
4. **Multithreading.** Multithreading can serve as a fast way to generate multiple random variates at once; it is not reflected in the pseudocode given in this page. In general, this involves dividing a block of memory into chunks, assigning each chunk to a thread, giving each thread its own instance of a pseudorandom number generator (or another program that simulates a "source of random numbers"), and letting each thread fill its assigned chunk with random variates. For an example, see "**Multithreaded Generation**[244]".
5. **Fixed amount of "randomness".** Given a k-bit integer n (which lies in the interval $[0, 2^k)$ and is chosen uniformly at random), values that approximate a probability distribution (for example, `Poisson`, `Normal`) can be generated with the integer n by—
    - **finding the quantile for** $(2n + 1)/(2^{k+1})$ (which comes from dividing the interval $[0, 1]$ into 2^k equal pieces and sampling the middle of one of the pieces), or
    - using n to help initialize a local PRNG and using the PRNG to generate a sample from that distribution.

    An application should use this suggestion only if it wants to ensure a fixed amount of "randomness" per sampled outcome is ultimately drawn, because the sampling method can return one of only 2^k different outcomes or less this way. (In general, n can't be chosen uniformly at random with a *fixed* number of randomly chosen bits, unless the number of different outcomes for n is a power of 2.) In general, neither approach given above allows for controlling the approximation error in generating a value this way.

12.3 Security Considerations

If an application samples at random for information security purposes, such as to generate passwords or encryption keys at random, the following applies:

1. **"Cryptographic generators".** The application has to use a device or program that generates random-behaving numbers that are hard to guess for information security purposes (a so-called "cryptographic generator"). Choosing such a device or program is outside the scope of this document.
2. **Timing attacks.** Certain security and privacy attacks have exploited timing and other differences to recover cleartext, encryption keys, or other secret or private data. Thus, security algorithms have been developed to have no timing differences that reveal anything about any secret or private inputs, such as keys, passwords, or "seeds" for pseudorandom number generators. But a sampling algorithm of this kind does not exist for all sampling distributions (Ben Dov et al. 2023)[245]; [246].
3. **Security algorithms out of scope.** Security algorithms that take random secrets to generate random security parameters, such as encryption keys, public/private key pairs, elliptic curves, or points on an elliptic curve, are outside this document's scope.
4. **Floating-point numbers.** Numbers chosen at random for security purposes are almost always integers (and, in very rare cases, fixed-point numbers). Even in the few security applications where those numbers are floating-point numbers (notably differential privacy and lattice-based cryptography), there are ways to avoid such floating-point numbers[247].

[243] https://peteroupc.github.io/random.html#Designs_for_PRNGs

[244] https://docs.scipy.org/doc/numpy/reference/random/multithreading.html

[245] Ben Dov, Y., David, L., et al., "Resistance to Timing Attacks for Sampling and Privacy Preserving Schemes", FORC 2023.

[246] In the privacy context, see, for example, Awan, J. and Rao, V., 2022. "**Privacy-Aware Rejection Sampling**", arXiv:2108.00965. https://arxiv.org/abs/2108.00965

[247] For example, see Balcer, V., Vadhan, S., "Differential Privacy on Finite Computers", Dec. 4, 2018; as well as Micciancio, D.

13 License

and Walter, M., "Gaussian sampling over the integers: Efficient, generic, constant-time", in Annual International Cryptology Conference, August 2017 (pp. 455-485).

[248]https://creativecommons.org/publicdomain/zero/1.0/