# Randomization and Sampling Methods

This version of the document is dated 2020-09-05.

**Peter Occil**

Discusses many ways applications can do randomization and sampling from an underlying (pseudo-)random number generator and includes pseudocode for many of them.

# 1 Introduction

This page discusses many ways applications can sample random content by transforming the numbers produced by an underlying source of random numbers (such as a pseudorandom number generator [PRNG] or another kind of random number generator [RNG]), and offers pseudocode for many of these methods. Those methods include—

- ways to derive uniform random numbers (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
- generating non-uniform random numbers, including **weighted choice**, the **Poisson distribution**, and **other probability distributions**.

This page is focused on sampling methods that *exactly* sample from the distribution described, without introducing additional errors beyond those already present in the inputs (and assuming that we have a source of "truly" random numbers). 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 an infinite number of 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.

**Sample Python code** that implements many of the methods in this document is available, together with **documentation for the code**.

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

- This document does not cover how to choose an underlying RNG (or PRNG) for a particular application, including in terms of security, performance, and quality. I have written more on RNG recommendations in **another document**.
- 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 RNGs for correctness or adequate random number generation. This is covered, for example, in "**Testing PRNGs for High-Quality Randomness**".
- This document does not explain how to specify or generate "seeds" for certain PRNGs. This is **covered in detail** 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**.
- Variance reduction techniques, such as importance sampling or common random numbers, are not in the scope of this document.

## 1.1 Sources of Random Numbers

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

- each number is *equally likely to occur*, and
- each number is *chosen independently of any other choice*.

That is, the methods assume we have a **source of (uniform) random numbers** (also known as a *uniform discrete memoryless source*). (Thus, none of these methods *generate* random numbers themselves, strictly speaking, but 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 kind of RNG) even if they don't in theory.

Note that the source of random numbers can be generated (or 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. They can serve as a source of random numbers regardless of their predictability or statistical quality, and whether or not they use a deterministic algorithm. However, some random number generators are better than others for certain purposes, and giving advice on which RNG to choose is outside the scope of this document.

The randomization methods in this document will be deterministic (that is, produce the same values given the same state and input) whenever they are powered by a PRNG (as will generally be the case in practice), as PRNGs are deterministic. However, if a "true" random number generator powers these methods, they will not necessarily be deterministic.

## 1.2 About This Document

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

**Comments on any aspect of this document are welcome.**

# 2 Contents

# 3 Notation

In this document:

- The **pseudocode conventions** 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 where `x` is a floating-point number (Pedersen 2018)[1].
- `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. Note, however, that summing floating-point numbers naïvely can introduce round-off errors. **Kahan summation** along with parallel reductions can be a more robust way than the naïve approach to compute the sum of three or more floating-point numbers.

# 4 Uniform Random Integers

This section shows how to derive independent uniform random integers with the help of a source of random numbers (such as a source produced by a PRNG).

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 generates independent uniform random integers **in the interval [0, `maxInclusive`]** using a source of random numbers[(2)]. There are many algorithms for `RNDINT`, but in general, the algorithm can be *unbiased* only if it runs forever in the worst case (for further information, see "**A Note on Integer Generation Algorithms**"). In the pseudocode below, which implements `RNDINT`[(2)]—

- `NEXTRAND()` reads the next number generated by a source of numbers that behave like independent and identically distributed random numbers (such as a source produced by a PRNG or another kind of random number generator), as described below, and
- `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[(3)]. | The next number from a new source formed by writing the underlying source's outputs to a stream of memory units (such as 8-bit bytes) and using a ***randomness extraction*** technique to transform that stream to $n$-bit non-negative integers. | $2^n$. |
| Uniform numbers not described below. | Same as above. | $2^n$. |
| Uniform 32-bit non-negative integers. | The next number from the source. | $2^{32}$. |
| Uniform 64-bit non-negative 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 $p$ (e.g., 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 (e.g., 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
      ret = NEXTRAND()
      if ret < nPlusOne: return ret
      if ret < maxexc: return rem(ret, nPlusOne)
    end
```

```
  else
    cx = floor(maxInclusive / MODULUS) + 1
    while true
       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() ). This will
  // be a constant here, though.
  modBits = ln(MODULUS)/ln(2)
  // Fast dice roller algorithm.
  x = 1
  y = 0
  nextBit = modBits
  rngv = 0
  while true
    if nextBit >= modBits
      nextBit = 0
      rngv = NEXTRAND()
    end
    x = x * 2
    y = y * 2 + rem(rngv, 2)
    rngv = floor(rngv / 2)
    nextBit = nextBit + 1
    if x > maxInclusive
      if y <= maxInclusive: return y
      x = x - maxInclusive - 1
      y = y - maxInclusive - 1
    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
```

## 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`] (e.g., Java `int`, `short`, or `long`), or
2. every integer in the interval [-`MAXINT`, `MAXINT`] (e.g., 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
     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
```

## 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[(4)]:

```
 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
     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 kind of random number generator) that outputs one or more 8-bit chunks at a time.

## 4.6 Examples of Using the `RNDINT` Family

1. To generate a random number that's either -1 or 1 (a *Rademacher random number*), 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.)
3. A random 2-dimensional point on an NxM grid can be expressed as a single integer as follows:
    - To generate a random NxM 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 NxM point, generate `P = coord[1] * N + coord[0]`.
4. To simulate rolling an N-sided die (N greater than 1), generate a random number in the interval [1, N] by `RNDINTRANGE(1, N)`.
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 generate a random number in the interval `mn`, `mx`) in increments equal to `step`: `mn+step*RNDINTEXC(ceil((mx-mn)/(1.0*step)))`.
8. To generate a random integer in the interval [0, `X`):
    - And favor numbers in the middle: `floor((RNDINTEXC(X) + RNDINTEXC(X)) / 2)`.
    - And strongly favor low numbers: `floor(RNDINTEXC(X) * RNDINTEXC(X) / X)`.
    - And favor high numbers: `max(RNDINTEXC(X), RNDINTEXC(X))`.
    - And strongly favor high numbers: `X - 1 - floor(RNDINTEXC(X) * RNDINTEXC(X) / X)` or `max(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)[(5)]):

```
// 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
    z = z * 2
    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 Irrational Constants**" 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.
    - Otherwise, the first three cases below 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.
        - If `k` is much smaller than `n`, proceed as in item 3 instead.
3. **If `k` is much smaller than `n` and the order of the items must be random or is unimportant:**
    - **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).
    - Otherwise, **if the items are stored in a list and `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, then choose the *last* `k` indices (or the items corresponding to those indices) from the latter list.
    - Otherwise, **if `n` is not very large:** Store all the items in a list, do a *partial shuffle* of that list, then choose the *last* `k` items from that list.

4. **If `n - k` is much smaller than `n` and the order of the sampled items is unimportant:** Proceed as in item 3, except the partial shuffle involves `n - k` swaps and the *first* `k` items are chosen rather than the last `k`.
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 = Hypergeometric(k, n2, n)`. Sample `h` integers in relative order from the list `[0, 1, ..., n2 - 1]` by running this algorithm (items 1 to 5) recursively, 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)[6].

    - 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**,
        - a compressed bit set (e.g, "roaring bitmap", EWAH), and
        - a self-sorting data structure such as a **red–black tree**, if the random items are to be retrieved in sorted order or in index order.

      Many applications require generating unique random numbers to identify database records or other shared resources, among other reasons. For ways to generate such numbers, see my **RNG recommendation document**.

### 5.2.3 Shuffling

The **Fisher–Yates shuffle method** shuffles a list (puts its items in a random order) such that all permutations (arrangements) of that list are equally likely to occur. However, that method is also easy to write incorrectly — see also (Atwood 2007)[7]. 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 an 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`. Note that the item
         // at i+1 is excluded.
         k = RNDINTEXC(i + 1)
         // The following is wrong since it introduces biases:
         // "k = RNDINTEXC(size(list))"
         // The following produces Sattolo's algorithm (which
         // chooses from among permutations with cycles)
         // rather than a Fisher-Yates shuffle:
         // "k = RNDINTEXC(i)"
         // Swap item at index i with item at index k;
         // in this case, i and k 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. The choice of random number generator (including PRNGs) is important when it comes to shuffling; see my **RNG recommendation document on shuffling**.
2. A shuffling algorithm that can be carried out in parallel is described in (Bacher et al., 2015)**(8)**.
3. 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)**(9)**: (1) modify `Shuffle` by adding the following line after `k = RNDINTEXC(i + 1)`: `if i == list[k]: 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, n); s=Shuffle(list); if s!=nothing: return s; end`.

## 5.2.4 Random Character Strings

To generate a random string of characters:

1. Prepare a list of the letters, digits, and/or other characters 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.
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, the choice of random number generator (including PRNGs) is important, and using *random* unique values might not be best; see my **RNG recommendation document**.
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***; it chooses up to `k` random items from a file of indefinite size (`file`). Although the pseudocode refers to files and lines, the technique applies to any situation when 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)**(10)**, p. 620.

```
METHOD RandomKItemsFromFile(file, k)
  list = NewList()
  j = 0
  index = 0
  while true
    // 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))
      j = j + 1
    else // phase 2
      j = RNDINT(thisIndex)
      if j < k: list[j] = ITEM_OUTPUT(item, thisIndex)
    end
  end
  // NOTE: We 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.[(11)] 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 random number) by any method and with any distribution and range.
2. If the content doesn't meet predetermined criteria, go to step 1.

Example criteria include checking—

- whether a random number—
  - 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 recently chosen random numbers,
    - 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 random point is sufficiently distant from previous random points (with the aid of a KD-tree or similar structure),
- whether a random string 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.)

> **Note:** All rejection sampling strategies have a chance to reject data, so they all have a *variable running time* (in fact, they could run indefinitely). Note that 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 generate a random number via rejection sampling 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 random number that hasn't been rejected this way by a task (which can fail at a very low rate).[12]

## 5.4 Random Walks

A *random walk* is a process with random behavior over time. A simple form of random walk involves generating a random number that changes the state of the walk. The pseudocode below generates a random walk of *n* random numbers, 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
```

> **Note:** A **white noise process** is simulated by creating a list of independent random numbers 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 0 or 1 depending on the probability `px`/`py`).
>
> **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 -1 or 1 depending on the probability `px`/`py`.
> 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`). In the following 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).

```
dtnum1 = DATETIME_TO_NUMBER(date1)
dtnum2 = DATETIME_TO_NUMBER(date2)
// Choose a random date-and-time
// in [dtnum1, dtnum2].  Any other
// random selection strategy can be
// used here instead.
num = RNDINTRANGE(date1, date2)
result = NUMBER_TO_DATETIME(num)
```

## 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*** 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)[13].

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** models one or more *states* (for example, individual letters or syllables), and stores the probabilities to transition from one state to another (e.g., "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.

An algorithm called *coupling from the past* (Propp and Wilson 1996)[14] 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)[15]. 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 numbers, and `RANDOM()` generates one or more random numbers 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
```

## 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 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)[16]), where each edge is drawn with probability `px`/`py` (Batagelj and Brandes 2005)[17].

```
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 *n* random edges without replacement), 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)**(18)** of a mesh graph. Penschuck et al. (2020) **(19)** give a survey of random graph generation techniques.

## 5.9 A Note on Sorting Random Numbers

In general, sorting random numbers is no different from sorting any other data. (Sorting algorithms are outside this document's scope.) **(20)**

# 6 General Non-Uniform Distributions

Some applications need to choose random items or numbers such that some of them are more likely to be chosen than others (a *non-uniform* distribution). Most of the techniques in this section show how to use the **uniform random number methods** to generate such random items or numbers.

## 6.1 Weighted Choice

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

### 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 more likely 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).

```
METHOD CWChoose(weights, value)
    // 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 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
    // Weights is a list of cumulative weights. Chooses
    // a random number in [0, size(weights) - 1).
    sum = weights[size(weights) - 1]
    return CWChoose(RNDINTEXC(sum))
END METHOD
```

**Notes:**

1. The **Python sample code** contains a variant of this method for generating multiple random points in one call.
2. See "**A Note on Weighted Choice Algorithms**" for other ways to implement `WeightedChoice`.
3. These methods support only integer weights; using non-integer weights can introduce unnecessary rounding errors. See `IntegerWeightsListFP` (in **Sampling for Discrete Distributions**") for a method to convert floating-point number weights to integer weights.

**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) is more likely to be chosen than the index for "apples" (0) with the `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. After a random index is chosen with `index = WeightedChoice(weights)`, an

independent uniform random number in the chosen interval is chosen. 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 (Multiple Copies)

To implement weighted choice *without replacement* (which can be thought of as drawing a ball *without* putting it back) when each weight is an integer 0 or greater, generate an index by `WeightedChoice`, and then decrease the weight for the chosen index by 1. In this way, **each weight behaves like the number of "copies" of each item**. The pseudocode below is an example of this. It assumes 'weights' is a list that can be modified.

```
totalWeight = Sum(weights)
// Choose as many items as the sum of weights
i = 0
items = NewList()
while i < totalWeight
    index = WeightedChoice(weights)
    // Decrease weight by 1 to implement selection
    // without replacement.
    weights[index] = weights[index] - 1
    // NOTE: Assumes the 'list' of items was declared
    // earlier and has at least as many items as 'weights'
    AddItem(items, list[index])
    i = i + 1
end
```

Alternatively, if all the weights are integers 0 or greater and their sum is relatively small, create a list with as many copies of each item as its weight, then **shuffle** that list. The resulting list will be ordered in a way that corresponds to a weighted random choice without replacement.

> **Note:** The weighted sampling described in this section can be useful to some applications (particularly some games) that wish to control which random numbers appear, to make the random outcomes appear fairer to users (e.g., to avoid long streaks of good outcomes or of bad outcomes). When used for this purpose, each item represents a different outcome (e.g., "good" or "bad"), and the lists are replenished once no further items can be chosen. However, this kind of sampling should be avoided in applications that care about information security, including when predicting future random numbers would give a player or user a significant and unfair advantage.

### 6.1.3 Weighted Choice Without Replacement (Single Copies)

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 are more likely 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 exponentially-distributed number (with a rate equal to that index's weight), make a list of pairs assigning each number to an index, then sort that list in ascending order by those numbers. Example: `v=[]; for i in 0...size(weights): AddItem(v, [ExpoNew(weights[i]), i]); Sort(v)` (see the next section for `ExpoNew`). (Each weight must be 1 or greater.) The sorted list of indices will then correspond to a weighted choice without replacement. See "**Algorithms for**

**sampling without replacement**".

### 6.1.4 Weighted Choice Without Replacement (List of Unknown Size)

If the number of items in a list is not known in advance, then the following pseudocode implements a `RandomKItemsFromFileWeighted` that selects up to `k` random items from a file (`file`) of indefinite size (similarly to **`RandomKItemsFromFile`**). See (Efraimidis and Spirakis 2005)[21], and see also (Efraimidis 2015)[22], (Vieira 2014)[23], and (Vieira 2019)[24]. In the pseudocode below:

- `WEIGHT_OF_ITEM(item, thisIndex)` is a placeholder for arbitrary code that calculates the integer weight of an individual item based on its value and its index (starting at 0); the item is ignored if its weight is 0 or less.
- `ITEM_OUTPUT(item, thisIndex, key)` 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, the item's key, or any combination of these.
- `ExpoNew(weight)` creates an "empty" exponential number with rate `weight`, whose bits are not yet determined (see "**Partially-Sampled Exponential Random Numbers**"), and `ExpoLess(a, b)` returns whether one exponential number (`a`) is less than another (`b`), building up the bits of both as necessary. For a less exact algorithm, replace `ExpoNew(weight)` with `ExpoRatio(1000000, weight, 1)` and `ExpoLess(a, b)` with `a < b`.

```
METHOD RandomKItemsFromFileWeighted(file, k)
  queue=[] // Initialize priority queue
  index = 0
  while true
    // 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
    weight = WEIGHT_OF_ITEM(item, thisIndex)
    // Ignore if item's weight is 0 or less
    if weight <= 0: continue
    // NOTE: Equivalent to Expo(weight)
    key = ExpoNew(weight)
    itemToStore = ITEM_OUTPUT(item, thisIndex, key)
    // Begin priority queue add operation.  Instead
    // of the item ('item'), the line number starting at one
    // ('thisIndex') can be added to the queue.
    if size(queue) < k // Fewer than k items
      AddItem(queue, [key, itemToStore])
      Sort(queue)
    else
      // Check whether this key is smaller
      // than the largest key in the queue
      if ExpoLess(key, queue[size(queue)-1][0])
          // Replace the item with the largest key
          queue[size(queue)-1]=[key, itemToStore]
          Sort(queue)
      end
    end
    // End priority queue add operation
  end
  list=[]
```

```
  for v in 0...size(queue): AddItem(list, queue[v][1])
  // Optional shuffling here.
  // See NOTE 4 in RandomKItemsFromFile code.
  if size(list)>=2: Shuffle(list)
  return list
end
```

> **Note:** Weighted choice *with replacement* can be implemented by doing one or more concurrent runs of `RandomKItemsFromFileWeighted(file, 1)` (making sure each run traverses `file` the same way for multiple runs as for a single run) (Efraimidis 2015)[22].

### 6.1.5 Weighted Choice with Inclusion Probabilities

For the weighted-choice-without-replacement methods given earlier, 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 that item (an *inclusion probability*). The following method chooses a random sample of `n` indices from a list of items (whose weights are given as `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 found in pp. 73-74 in "**Algorithms of sampling with equal or unequal probabilities**".

```
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
    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
```

## 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 was chosen, roll one die
if index==0: number = RNDINTRANGE(1,6)
// Else index 1 was chosen, so roll three dice
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***, or decomposed into **triangles**, and a mixture of those triangles can be sampled.**(25)**

3. Take a set of nonoverlapping integer ranges. 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.

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. The following is an example involving three different exponential distributions: `index = WeightedChoice([6, 3, 1]); rates = [[3, 10], [5, 10], [1, 100]]; number = ExpoRatio(100000, rates[i][0], rates[i][1])`.

## 6.3 Transformations of Random Numbers

Random numbers can be generated by combining and/or transforming one or more

random numbers and/or discarding some of them.

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.[26] These and similar distributions can be generalized as follows.

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

1. **Highest-of:** Choose the highest generated number.
2. **Drop-the-lowest:** Add all generated numbers except the lowest.
3. **Reroll-the-lowest:** Add all generated numbers 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 numbers except the highest.
6. **Reroll-the-highest:** Add all generated numbers except the highest, then add a number generated randomly by a separate probability distribution.
7. **Sum:** Add all generated numbers.
8. **Mean:** Find the mean of all generated numbers.
9. **Geometric transformation:** Treat the numbers as an *n*-dimensional point, then apply a geometric transformation, such as a rotation or other *affine transformation*[27], to that point.

If the probability distributions are the same, then strategies 1 to 3 make higher numbers more likely, and strategies 4 to 6, lower numbers.

> **Note:** Variants of strategy 4 — e.g., 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 is more likely to occur 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` random numbers each generated by `ZeroOrOne(px,py)` (strategy 7) (see "**Binomial Distribution**").
>
> 4. **Clamped random numbers.** These are one example of transformed random numbers. To generate a clamped random number, generate a random number as usual, then—
>
>    - if that number is less than a minimum threshold, use the minimum threshold instead (*left-censoring*), and/or
>    - if that number is greater than a maximum threshold, use the maximum threshold instead (*right-censoring*).
>
>    An example of a clamped random number is `min(200, RNDINT(255))`.
>
> 5. A **compound Poisson distribution** models the sum of *n* random numbers each generated the same way, where *n* follows a **Poisson distribution**

(e.g., `n = PoissonInt(10, 1)` for an average of 10 numbers) (strategy 7, sum).

6. A **Pólya–Aeppli distribution** is a compound Poisson distribution in which the random numbers are generated by `NegativeBinomial(1, 1-p)+1` for a fixed `p`.

# 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).

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)[28]. 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
      // Divide and conquer
      half = floor(trials / 2)
      return BinomialInt(half, 1, 2) + BinomialInt(trials - half, 1, 2)
    else
      if mod(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(0.5)
      // weights, which consist of n-choose-i
      // for all 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 binary expansion.
    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*** (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).

```
METHOD NegativeBinomialInt(successes, px, py)
    // Needs to be 0 or greater; px must not be 0
    if successes < 0 or px == 0: return error
    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)**(29)**:

```
METHOD PolyaInt(sx, sy, px, py)
   isinteger=mod(sx,sy)==0
   sxceil=ceil(sx/sy)
   while true
      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 number 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, the number of trials that have already failed in a row says nothing about the number of new trials that will fail in a row.

> **Note:** 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)**(10)** includes it. And some works may define a negative binomial number as the number of successes before N failures, rather than vice versa.

## 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/λ 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 number with the rate `lamda`.

In the pseudocode below, `ExpoRatio` generates an exponential random number (in the form

of a ratio) given the rate `rx`/`ry` (or scale `ry`/`rx`) and the base `base`. `ExpoNumerator` generates the numerator of an exponential random number with rate 1 given that number's denominator. The algorithm is due to von Neumann (1951)[30]. Exponential random numbers can also be partially sampled; for more information see "**Partially-Sampled Exponential Random Numbers**".

```
METHOD ExpoRatio(base, rx, ry)
    // Generates a numerator and denominator of
    // an exponential random number with rate rx/ry.
    return MakeRatio(ExpoNumerator(base*ry), base*rx))
END METHOD

METHOD ExpoNumerator(denom)
   if denom<=0: return error
   count=0
   while true
      y1=RNDINTEXC(denom)
      y=y1
      accept=true
      while true
         z=RNDINTEXC(denom)
         if y<=z: break
         accept=not accept
         y=z
      end
      if accept: count=count+y1
      else: count=count+denom
      if accept: break
   end
   return count
END METHOD
```

## 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 following method generates a Poisson random number with mean `mx`/`my`, using the approach suggested by (Flajolet et al., 2010)[31]. In the method, `UniformNew()` creates a *u-rand*, an "empty" random number in [0, 1], whose bits are not yet determined (Karney 2014)[32], and `UniformLess(a, b)` returns whether one *u-rand* (`a`) is less than another (`b`), building up the bits of both as necessary. For a less exact algorithm, replace `UniformNew()` with `RNDINT(1000)` and `UniformLess(a, b)` with `a < b`.

```
METHOD PoissonInt(mx, my)
    if my == 0: return error
    if mx == 0: return 0
    if (mx < 0 and my < 0) or (mx > 0 and my < 0): return 0
    if mx==my: return PoissonInt(1,2)+PoissonInt(1,2)
    if mx > my
       // Mean is 1 or greater
       mm=mod(mx, my)
       if mm == 0
          mf=floor(mx/my)
```

```
          ret=0
          if mod(mf, 2)==0: ret=ret+PoissonInt(1, 1)
          if mod(mf, 2)==0: mf=mf-1
          ret=ret+PoissonInt(mf/2, 1)+PoissonInt(mf/2, 1)
          return ret
       else: return PoissonInt(mm, my)+
           PoissonInt(mx-mm, my)
    end
    if mx>floor(my*9/10)
       // Runs slowly if mx/my is close to 1, so break it up
       hmx=floor(mx/2)
       return PoissonInt(hmx,my)+PoissonInt(mx-hmx,my)
    end
    while true
      k = 0
      w = nothing
      while true
         // Similar to generating k, a geometric random
         // number, then accepting or rejecting based on whether
         // none of the random numbers are out of order
         // (NOTE: Flajolet et al. define a geometric
         // distribution as number of SUCCESSES BEFORE
         // FAILURE, not counting the failure.)
         if ZeroOrOne(mx,my)==0: return k
         u2 = UniformNew()
         // Break if we find an out-of-order number
         if k>0 and UniformLess(w, u): break
         w = u
         k=k+1
      end
    end
    return count
END METHOD
```

> **Note:** To generate a sum of `n` independent Poisson random numbers with separate means, generate a Poisson random number whose mean is the sum of those means (see (Devroye 1986)[10], p. 501). For example, to generate a sum of 1000 independent Poisson random numbers with a mean of 1/1000000, simply generate `PoissonInt(1, 1000)` (because 1/1000000 * 1000 = 1/1000).

## 7.7 Hypergeometric Distribution

The following method generates a random integer that follows a *hypergeometric distribution*. When a given number of items are drawn at random without replacement from a collection of items each labeled either `1` or `0`, the random integer expresses the number of items drawn this way that are labeled `1`. In the method below, `trials` is the number of items drawn at random, `ones` is the number of items labeled `1` in the set, and `count` is the number of items labeled `1` or `0` in that set.

```
METHOD Hypergeometric(trials, ones, count)
    if ones < 0 or count < 0 or trials < 0 or
       ones > count or trials > count
      return error
    end
    if ones == 0: return 0
    successes = 0
    i = 0
    currentCount = count
    currentOnes = ones
    while i < trials and currentOnes > 0
```

```
      if ZeroOrOne(currentOnes, currentCount) == 1
        currentOnes = currentOnes - 1
        successes = successes + 1
      end
      currentCount = currentCount - 1
      i = i + 1
    end
    return successes
END METHOD
```

**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, and `count` is 52.

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

**Notes:**

1. To generate `N` random numbers with a given positive average `avg`, in random order, generate `IntegersWithSum(N, N * avg)`.
2. To generate `N` random numbers `min` or greater and with a given positive sum `sum`, in random order, generate `IntegersWithSum(N, sum - N * min)`, then add `min` to each number generated this way. The **Python sample code** 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)[34], 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
```

# 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.

However, 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.
- Working with integers is more portable and numerically stable than working with other real numbers, especially floating-point numbers.[35]
- 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 generate random numbers 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 the following methods that generate independent uniform random real numbers:

- `RNDU01`: Interval [0, 1].
- `RNDU01OneExc`: Interval [0, 1).[36]
- `RNDU01ZeroExc`: Interval (0, 1].
- `RNDU01ZeroOneExc`: Interval (0, 1).
- `RNDRANGE`: Interval [a, b].
- `RNDRANGEMaxExc`: Interval [a, b).
- `RNDRANGEMinExc`: Interval (a, b].
- `RNDRANGEMinMaxExc`: Interval (a, b).

## 8.1.1 For Fixed-Point Number Formats

For fixed-point number formats representing multiples of 1/`n`, these eight methods are trivial. The following implementations return integers that represent fixed-point numbers.

`RNDU01` family:

- `RNDU01()`: `RNDINT(n)`.
- `RNDU01ZeroExc()`: `(RNDINT(n - 1) + 1)` or `(RNDINTEXC(n) + 1)`.
- `RNDU01OneExc()`: `RNDINTEXC(n)` or `RNDINT(n - 1)`.
- `RNDU01ZeroOneExc()`: `(RNDINT(n - 2) + 1)` or `(RNDINTEXC(n - 1) + 1)`.

`RNDRANGE` family. In each method below, `fpa` and `fpb` are the bounds of the random 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 [6.35, 9.96], generate `RNDRANGE(6.35, 9.96)` or `RNDINTRANGE(635, 996)`.

- `RNDRANGE(a, b)`: `RNDINTRANGE(fpa, fpb)`.
- `RNDRANGEMinExc(a, b)`: `RNDINTRANGE(fpa + 1, fpb)`, or an error if `fpa >= fpb`.
- `RNDRANGEMaxExc(a, b)`: `RNDINTEXCRANGE(fpa, fpb)`.
- `RNDRANGEMinMaxExc(a, b)`: `RNDINTRANGE(fpa + 1, fpb - 1)`, or an error if `fpa >= fpb or a == fpb - 1`.

## 8.1.2 For Rational Number Formats

For rational number formats with a fixed denominator, the eight methods can be

implemented in the numerator as given in the previous section for fixed-point formats.

### 8.1.3 For Floating-Point Number Formats

For floating-point number formats representing numbers of the form `FPSign` * `s` * `FPRADIX`$^{e}$ **(37)**, the following pseudocode implements `RNDRANGE(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 `s`, the significand of the number `x`. Returns 0 if `x = 0`. Has `FPPRECISION` digits unless `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)**(38)** and the **Rademacher Floating-Point Library**.

```
METHOD RNDRANGE(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
       ret=RNDRANGE(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
  if losgn == -1
    // Negative range
    return -RNDRANGE(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
    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
```

The other seven methods can be derived from `RNDRANGE` as follows:

- **`RNDRANGEMaxExc`, interval `` `mn, mx` ``)**:
  - Generate `RNDRANGE(mn, mx)` in a loop until a number other than `mx` is generated this way. Return an error if `mn >= mx` (treating positive and negative zero as different).
- **`RNDRANGEMinExc`, interval [`mn, mx`)**:
  - Generate `RNDRANGE(mn, mx)` in a loop until a number other than `mn` is generated this way. Return an error if `mn >= mx` (treating positive and negative zero as different).
- **`RNDRANGEMinMaxExc`, interval (`mn, mx`)**:
  - Generate `RNDRANGE(mn, mx)` in a loop until a number other than `mn` or `mx` is generated this way. Return an error if `mn >= mx` (treating positive and negative zero as different).
- **`RNDU01()`**: `RNDRANGE(0, 1)`.
- **`RNDU01ZeroExc()`**: `RNDRANGEMinExc(0, 1)`.
- **`RNDU01OneExc()`**: `RNDRANGEMaxExc(0, 1)`.
- **`RNDU01ZeroOneExc()`**: `RNDRANGEMinMaxExc(0, 1)`.

**Note:** In many software libraries, random numbers in a range are generated by dividing or multiplying a random integer by a constant. For example, `RNDU01OneExc()` is often implemented like `RNDINTEXC(X) * (1.0/X)` or `RNDINTEXC(X) / X`, where X varies based on the software library.[(39)] The disadvantage here is that not all numbers a floating-point format can represent in the range can be covered this way (Goualard 2020)[(40)]. As another example, `RNDRANGEMaxExc(a, b)` is often implemented like `a + RNDU01OneExc() * (b - a)`; however, this not only has the same disadvantage, but has many other issues where floating-point numbers are involved (Monahan 1985)[(41)].

### 8.1.4 Uniform Numbers As Their Digit Expansions

As noted by von Neumann (1951)[(30)], a uniform random number bounded by 0 and 1 can be produced by "juxtapos[ing] enough random binary digits". In this sense, the random number is `RNDINTEXC(2)/2 + RNDINTEXC(2)/4 + RNDINTEXC(2)/8 + ...`, perhaps "forc[ing] the last [random bit] to be 1" "[t]o avoid any bias". It is not hard to see that a uniform random number of an arbitrary base can be formed by generating a random digit expansion after the point. For example, a random decimal number bounded by 0 and 1 can be generated as `RNDINTEXC(10)/10 + RNDINTEXC(10)/100 + ...`. (Note that the number 0 is an infinite digit expansion of zeros, and the number 1 is an infinite digit expansion of base-minus-ones.)

## 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** of a function given a random process or sampling distribution. The following pseudocode estimates the expected value from a list of random numbers generated the same way. Here, `EFUNC` is the function, and `MeanAndVariance` is given in the **appendix**. `Expectation` returns a list of two numbers — the estimated expected value and its standard error.

   ```
   METHOD Expectation(numbers)
     ret=[]
     for i in 0...size(numbers)
        AddItem(ret,EFUNC(numbers[i]))
     end
     merr=MeanAndVariance(ret)
     merr[1]=merr[1]*(size(ret)-1.0)/size(ret)
     merr[1]=sqrt(merr[1]/size(ret))
     return merr
   END METHOD
   ```

   Examples of expected values include the following:

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

   If the sampling domain is also limited to random numbers meeting a given condition (such as `x < 2` or `x != 10`), then the estimated expected value is also called the estimated *conditional expectation*.

   Monte Carlo estimation makes a difference between *biased* and *unbiased* estimators. An estimator is *unbiased* if the average of multiple estimates, based on independent samples of the same distribution, is consistent with the expected value even as the number of estimates grows. For example, an `Expectation` for the mean is an unbiased estimator, but an `Expectation` for the sample variance is not since there is more than one degree of freedom to the estimation (see "**Variance**" in MathWorld). For an unbiased estimator, the error in the Monte Carlo estimation is largely due to *variance*, but variance reduction techniques are outside the scope of this document.

2. **Monte Carlo integration**. This is a way to estimate a multidimensional integral; randomly sampled numbers are put into a list (`nums`) and the estimated integral and its standard error are then calculated with `Expectation(nums)` with `EFUNC(x) = x`, and multiplied by the volume of the sampling domain.

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*** and ***simultaneous perturbation stochastic approximation*** (see also (Spall 1998)[42]).

## 8.3 Low-Discrepancy Sequences

**Requires random real numbers.**

A ***low-discrepancy sequence*** (or *quasirandom sequence*) is a sequence of numbers that behave like uniform random numbers but are *dependent* on each other, in that they are less likely to form "clumps" than if they were independent. The following are examples:

- A base-N *van der Corput sequence* is generated as follows: For each non-negative 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**" by S. Joe and F. Kuo.
- *Latin hypercube sampling* doesn't exactly produce low-discrepancy sequences, but serves much the same purpose. The following pseudocode implements this sampling 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)`.
- 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)[43]. One example of `MLCG(seed)`: `rem(92717*seed,262139)/262139.0`.
- Linear feedback shift register generators with good "uniformity" for Monte Carlo sampling (e.g., (Harase 2020)[44]).
- If the sequence outputs numbers in the interval [0, 1], the **Baker's map** of the sequence is `2 * (0.5-abs(x - 0.5))`, where `x` is each number in the sequence.

In most cases, pseudorandom number generators (or other kinds of RNGs) can be used to generate a "seed" to start the low-discrepancy sequence at. In Monte Carlo sampling, low-discrepancy sequences are often used to achieve more efficient "random" sampling.

## 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 `RNDRANGE(-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)[45] 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 number with its own rate parameter (e.g., `Expo(rate)`) (see "**Exponential Distribution**"), and sorting N random `RNDRANGE(x, y)` expresses N arrival times in the interval `[x, y]`. 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 `RNDRANGE(0, maxrate) < thisrate`, where `thisrate` is the rate for the given arrival time (Lewis and Shedler 1979)[46].

### 8.4.2 Mixtures: Additional Examples

Example 3 in "**Mixtures of Distributions**" can be adapted to nonoverlapping real number ranges by assigning weights `mx - mn` instead of `(mx - mn) + 1` and using `RNDRANGEMaxExc` instead of `RNDINTRANGE`.

### 8.4.3 Transformations: Additional Examples

1. **Bates distribution**: Find the mean of *n* uniform random numbers in a given range (such as by `RNDRANGE(minimum, maximum)`) (strategy 8, mean; see the **appendix**).
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)[47], sec. 3.8): `[x, y*sqrt(1 - rho * rho) + rho * x]`, where `x` and `y` are independent random numbers generated 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 numbers, where N has a fractional part. In this case, `ceil(N)` random numbers are generated and the last number is multiplied by that fractional part. For example, to sample the sum of 2.5 random numbers, generate three random numbers, multiply the last by 0.5 (the fractional part of 2.5), then sum all three numbers.
4. A **hypoexponential distribution** models the sum of *n* random numbers that follow an exponential distribution and each have a separate rate parameter (see "**Exponential Distribution**").

## 8.5 Random Numbers 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.)

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 are more likely to be sampled.[48] 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** 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***. To sample from a kernel distribution:
        1. Choose one of the numbers or points in the list 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*[49].
    - **Stochastic interpolation** is described in (Saucier 2000)[47], 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**, among other ways. If several kinds of distributions are possible fitting choices, then the kind showing the best *goodness of fit* for the data (e.g., chi-squared goodness of fit) is chosen.

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 (e.g., monthly sales figures) and wants to sample a random but plausible output given a known input point (e.g., 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 numbers as input and generate outputs (such as images or sounds) that are similar to examples they have already seen. ***Generative adversarial networks*** are one kind of generative model.

    **Note:** If the existing data points each belong in one of several *categories*:

    - Choosing a random category could be done by choosing a random number weighted on the number of data points in each category (see "**Weighted Choice**").
    - Choosing a random data point *and* its category could be done—
        1. by choosing a random data point based on all the existing data points, then finding its category (e.g., via machine learning models known as

*classification models*), or
2. by choosing a random category as given above, then by choosing a random data point based only on the existing data points of that category.

# 8.6 Random Numbers 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***, or *CDF*, returns, for each number, the probability that a number equal to or greater than that number is randomly chosen; the probability is in the interval [0, 1].
- The ***probability density function***, or *PDF*, is, roughly and intuitively, a curve of weights 0 or greater, where for each number, the greater its weight, the more likely a number close to that number is randomly chosen. In this document, the area under the PDF need not equal 1.**(50)**
- The *quantile function* (also known as *inverse cumulative distribution function* or *inverse CDF*) is the inverse of the CDF and maps numbers in the interval 0, 1) to numbers in the distribution, from low to high.

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

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

## 8.6.1 Sampling for Discrete Distributions

If the distribution **is discrete**[**(51)**, 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. The method will be exact as long as—

1. the chosen points cover all of the distribution, and
2. the values of the PDF or CDF are calculated exactly, without error (if they are not, though, this method will not introduce any additional error on those values).**(52)**

The pseudocode below shows the following methods that work with a **known PDF** (`PDF(x)`, more properly called *probability mass function*) or a **known CDF** (`CDF(x)`) that outputs floating-point numbers of the form `FPSignificand` * `FPRadix`$^{\texttt{FPExponent}}$ (which include Java's `double` and `float`)**(53)**. In the code, `gcd(a, b)` is the greatest common divisor between two numbers (where `gcd(0, a) = gcd(a, 0) = a` whenever `a >= 0`).

- `SampleFP(mini, maxi)` chooses a random number in [`mini`, `maxi`] based on a **known PDF** (see also `integers_from_pdf` in the **Python sample code**). `InversionSampleFP` is similar, but is based on a **known CDF**; however, `SampleFP` should be used instead where possible.
- `IntegerWeightsFP(mini, maxi)` generates a list of integer weights for the interval [`mini`, `maxi`] based on a **known PDF**[**(54)**. (Alternatively, the weights could be approximated, such as by scaling and rounding [e.g., `round(PDF(i) * mult)`] or by using a more sophisticated algorithm (Saad et al., 2020)**(55)**, but doing so can introduce error.)
- `IntegerCDFFP` is similar to `IntegerWeightsFP`, but generates cumulative weights based on

a **known CDF**.

- `IntegerWeightsListFP` generates integer weights from a list of weights or cumulative weights expressed as floating-point numbers.

```
METHOD SampleFP(mini, maxi)
  return mini + WeightedChoice(IntegerWeightsFP(mini, maxi))
END METHOD

METHOD InversionSampleFP(mini, maxi)
  weights=IntegerCDFFP(mini, maxi)
  return mini + CWChoose(weights,
     weights[size(weights) - 1])
END METHOD

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

METHOD NormalizeRatios(ratios)
  prod=1; gc=0
  for r in ratios: prod*=r[1]
  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

METHOD IntegerWeightsFP(mini, maxi)
  ratios=[]
  for i in mini..maxi: AddItem(ratios, FPRatio(PDF(i)))
  return NormalizeRatios(ratios)
END METHOD

METHOD IntegerCDFFP(mini, maxi)
  ratios=[[0, 1]]
  for i in mini..maxi: AddItem(ratios, FPRatio(CDF(i)))
  return NormalizeRatios(ratios)
END METHOD

METHOD IntegerWeightsListFP(weights)
  ratios=[]
  for w in weights: AddItem(ratios, FPRatio(w))
  return NormalizeRatios(ratios)
END METHOD
```

## 8.6.2 Inverse Transform Sampling

***Inverse transform sampling*** (or simply *inversion*) is the most generic way to generate a random number that follows a distribution.

If the distribution **has a known quantile function**, generate a uniform random number

in (0, 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 (e.g., `ICDF(RNDU01ZeroOneExc())`) may mean that small changes in the uniform number lead to huge changes in the quantile, leading to gaps in random number coverage (Monahan 1985, sec. 4 and 6)[41].

The following method generates a random number from a distribution via inversion, with an accuracy of $\texttt{BASE}^{-\texttt{precision}}$ ((Devroye and Gravel 2015)[56], but extended for any base; see also (Bringmann and Friedrich 2013, Appendix A)[57]). In the method, `ICDF(u, ubits, prec)` calculates a number that is within $\texttt{BASE}^{-\texttt{prec}}$ of the true quantile of $\texttt{u}/\texttt{BASE}^{\texttt{ubits}}$, and `BASE` is the digit base (e.g. 2 for binary or 10 for decimal).

```
METHOD Inversion(precision)
   u=0
   ubits=0
   threshold=MakeRatio(1,pow(BASE, precision))*2
   incr=8
   while true
      incr=8
      if ubits==0: incr=precision
      // NOTE: If a uniform number (`n`) is already pregenerated,
      // use the following instead:
      // u = mod(floor(n*pow(BASE, ubits+incr)), pow(BASE, incr))
      u=u*pow(BASE,incr)+RNDINTEXC(pow(BASE,incr))
      ubits=ubits+incr
      lower=ICDF(u,ubits,precision)
      upper=ICDF(u+1,ubits,precision)
      // Quantile can never go down
      if lower>upper: return error
      diff=upper-lower
      if diff<=threshold: return upper+diff/2
   end
end
```

Some cases require converting a pregenerated uniform random number to a non-uniform one via quantiles (notable cases include copula methods and Monte Carlo methods involving low-discrepancy sequences). For these cases, the following methods approximate the quantile if the application can trade accuracy for speed:

- Distribution is **discrete, with known PDF**: The most general method is sequential search (Devroye 1986, p. 85)[10], assuming the area under the PDF is 1 and the distribution covers only integers 0 or greater: `i = 0; p = PDF(i); while u01 > p; u01 = u01 - p; i = i + 1; p = PDF(i); end; return p`, but this is not always fast. If the interval [a, b] covers all or almost all the distribution, then the application can store the interval's PDF values in a list and call `WChoose`: `for i in a..b: AddItem(weights, PDF(i)); return a + WChoose(weights, u01 * Sum(weights))`. Note that finding the quantile based on the **CDF** instead can introduce more error than with the PDF (Walter 2019)[58]. See also `integers_from_u01` in the **Python sample code**.
- Distribution is **continuous, with known PDF**: `ICDFFromContPDF(u01, mini, maxi, step)`, below, finds the quantile based on a piecewise linear approximation of the PDF in [`mini`, `maxi`], with pieces up to `step` wide. See also `DensityInversionSampler`, `numbers_from_u01`, and `numbers_from_dist_inversion` (Derflinger et al. 2010)[59], (Devroye and Gravel 2015)[56] in the Python sample code [60].
- Distribution is **continuous, with known CDF**: See `numbers_from_u01` in the Python

sample code.

```
METHOD ICDFFromContPDF(u01, mini, maxi, step)
  pieces=[]
  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). If the raw data points are available, see "**Random Numbers from a Distribution of Data Points**" instead.
2. Taking the `k`th smallest of `n` random numbers distributed the same way is the same as taking the `k`th smallest of `n` *uniform* random numbers (e.g., `BetaDist(k, n+1-k)`) and finding its quantile (Devroye 2006)**(61)**; (Devroye 1986, p. 30)**(14)**.

## 8.6.3 Rejection Sampling with a PDF

If the distribution **has a known PDF**, and the PDF can be more easily sampled by another distribution with its own PDF (`PDF2`) that "dominates" `PDF` in the sense that `PDF2(x) >= PDF(x)` at every valid `x`, then generate random numbers with the latter distribution until a number (`n`) that satisfies `r <= PDF(n)`, where `r = RNDRANGEMaxExc(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 (`PDF3`) is chosen that is "dominated" by the first PDF (`PDF`) and easier to sample than `PDF`. Here, a number is accepted if `r <= PDF3(n)` or `r <= PDF(n)` (Devroye 1986, p. 53)[10].

See also (von Neumann 1951)[30]; (Devroye 1986)[10], pp. 41-43; "**Rejection Sampling**"; and "**Generating Pseudorandom Numbers**".

**Examples:**

1. To sample a random number in the interval [`low`, `high`) from a PDF with a positive maximum value no greater than `peak` at that interval, generate `x = RNDRANGEMaxExc(low, high)` and `y = RNDRANGEMaxExc(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**, generate `x` with `x = RNDINTEXCRANGE(low, high)` instead.
2. A custom distribution's PDF, `PDF`, is `exp(-abs(x*x*x))`, and the exponential distribution's PDF, `PDF2`, is `exp(-x)`. The exponential PDF "dominates" the other 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 `PDF(n) >= RNDRANGEMaxExc(0, PDF2(n))`.

**Note:** In the Python sample code, **moore.py** and `numbers_from_dist` generate random numbers from a distribution via rejection sampling (Devroye and Gravel 2016)[62], (Sainudiin and York 2013)[63].

## 8.6.4 Alternating Series

If the target PDF is not known exactly, but can be approximated from above and below by two series expansions that converge to the PDF as more terms are added, the *alternating series method* can be used. This still requires a "dominating" PDF (`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 generate a random number using this method (Devroye 2006)[61]: (1) Generate a random number `x` that follows the "dominating" distribution; (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.

## 8.6.5 Markov-Chain Monte Carlo

**Markov-chain Monte Carlo** (MCMC) is a family of algorithms for sampling many random numbers from a probability distribution by building a *Markov chain* of random values that build on each other until they converge to the given distribution. In general, however, a given chain's random values will have a statistical *dependence* on each other, and it takes an unknown time for the chain to converge (which is why techniques such as "thinning" -- keeping only every Nth sample -- or "burn-in" -- skipping iterations before sampling -- are often employed). MCMC can also estimate the distribution's sampling domain for other samplers, such as rejection sampling (above).

MCMC algorithms[64] include *Metropolis–Hastings*, *slice sampling*, and *Gibbs sampling* (see also the **Python sample code**). The latter is special in that it uses not a PDF, but two or more distributions, each of which uses a random number 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 numbers 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 a 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)**(65)**):

- `values` is a list of rational numbers. If the numbers are arranged in ascending order, which they should, the first number in this list can be returned exactly, but not the last number.
- `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 more likely it is 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 NormalizeRatios instead
  // of IntegerWeightsListFP.
  areas=IntegerWeightsListFP(areas)
  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=RNDRANGEMaxExc(-ww, ww)
  if RNDRANGEMaxExc(-hh, hh)>x*m: x=-x
  return values[index]+x+ww
END METHOD
```

**Note:** The **Python sample code** contains a variant to the method above for returning more than one random number 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 are more likely to be 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**. They cover the

normal, gamma, beta, von Mises, stable, and multivariate normal distributions as well as copulas. Note, however, that most of the methods won't sample the given distribution in a manner that minimizes approximation error, but they may still be useful if the application is willing to trade accuracy for speed.

## 8.9 Index of Non-Uniform Distributions

**Many distributions here require random real numbers.**

A † symbol next to a distribution means the random number 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 random number 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, see (Devroye 1996)**(66)**.

Most commonly used:

- **Beta distribution◆**: See **Beta Distribution**.
- **Binomial distribution**: See **Binomial Distribution**.
- **Binormal distribution**: See **Multivariate Normal (Multinormal) Distribution**.
- **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)**(67)**: Generate `x = RNDU01ZeroExc()` and `y = RNDU01ZeroExc()` until `x * x + y * y <= 1`, then generate `(RNDINT(1) * 2 - 1) * y / x`.
- **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-RNDU01()) / lamda` has several problems, such as being ill-conditioned at large values because of the distribution's right-sided tail (Pedersen 2018)**(1)**, as well as returning infinity if `RNDU01()` becomes 1. An application can reduce some of these problems by applying Pedersen's suggestion of using either `-ln(RNDRANGEMinExc(0, 0.5))` or `-log1p(-RNDRANGEMinExc(0, 0.5))` (rather than `-ln(1-RNDU01())`), chosen at random each time; an alternative is `ln(1/RNDU01ZeroExc())` mentioned in (Devroye 2006)**(61)**.
- **Extreme value distribution**: See generalized extreme value distribution.
- **Gamma distribution**: See **Gamma Distribution**. *3-parameter gamma distribution*: `pow(GammaDist(a, 1), 1.0 / c) * b`, where `c` is another shape parameter. *4-parameter gamma distribution*: `pow(GammaDist(a, 1), 1.0 / c) * b + d`, where `d` is the minimum value.
- **Gaussian distribution**: See **Normal (Gaussian) Distribution**.
- **Geometric distribution**: See **Geometric Distribution**. If the application can trade accuracy for speed: `floor(-Expo(1)/ln(1-p))` (Devroye 1996, p. 500)**(66)** (ceil replaced with floor because this page defines geometric distribution differently).
- **Gumbel distribution**: See generalized extreme value distribution.
- **Inverse gamma distribution**: `b / GammaDist(a, 1)`, where `a` and `b` have the same meaning as in the gamma distribution. Alternatively, `1.0 / (pow(GammaDist(a, 1), 1.0 / c) / b + d)`, where `c` and `d` are shape and location parameters, respectively.
- **Laplace (double exponential) distribution†**: `(Expo(1) - Expo(1))`.
- **Logarithmic distribution◆**: `RNDU01OneExc() * RNDU01OneExc()` (Saucier 2000, p. 26). In this distribution, lower numbers are exponentially more likely 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**.
- **Normal distribution**: See **Normal (Gaussian) Distribution**.
- **Poisson distribution**: See "**Poisson Distribution**". If the application can trade accuracy for speed, the

following can be used (Devroye 1986, p. 504)[10]: `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)[68] can be used: `while mean > 20; n=ceil(mean-pow(mean,0.7)); g=GammaDist(n, 1); if g>=mean: return c+(n-1-Binomial(n-1,(g-mean)/g)); mean = mean - g; c = c + n; end`, or the following approximation suggested in (Giammatteo and Di Mascio 2020)[69] for `mean` greater than 50: `floor(1.0/3 + pow(max(0, Normal(0, 1)*pow(mean,1/6.0)*2/3 + pow(mean, 2.0/3)), 3.0/2))`.

- **Pareto distribution**: `pow(RNDU01ZeroOneExc(), -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**.
- **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 number. A `cent` other than 0 indicates a *noncentral* distribution. Alternatively, `cos(RNDRANGE(0, pi * 2)) * sqrt((pow(RNDU01(),-2.0/df)-1) * df)` (Bailey 1994)[70].
- **Triangular distribution**† (Stein and Keblis 2009)[71]: `(1-alpha) * min(a, b) + alpha * max(a, b)`, where `alpha` is in [0, 1], `a = RNDU01()`, and `b = RNDU01()`.
- **Weibull distribution**: See generalized extreme value distribution.

Miscellaneous:

- **Archimedean copulas**: See **Gaussian and Other Copulas**.
- **Arcsine distribution**⬥: `BetaDist(0.5, 0.5)` (Saucier 2000, p. 14).
- **Bates distribution**: See **Transformations of Random Numbers: Additional Examples**.
- **Beckmann distribution**: See **Multivariate Normal (Multinormal) Distribution**.
- **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 (e.g., 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), 1.0 / alpha) / pow(GammaDist(b, 1), 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.
- **Chi distribution**: Square root of a chi-squared random number. See chi-squared distribution.
- **Compound Poisson distribution**: See **Transformations of Random Numbers: Additional Examples**.
- **Cosine distribution**⬥: `atan2(x, sqrt(1 - x * x)) / pi`, where `x = RNDRANGE(-1, 1)` (Saucier 2000, p. 17; inverse sine replaced with `atan2` equivalent).
- **Dagum distribution**: See beta prime distribution.
- **Dirichlet distribution**: [**This distribution**](https://en.wikipedia.org/wiki/Dirichlet_distribution (e.g., (Devroye 1986)[13], p. 593-594) can be sampled by generating *n*+1 random **gamma-distributed** numbers, each with separate parameters, taking their sum[15], dividing them by that sum, and taking the first *n* numbers. (The *n*+1 numbers sum to 1, but the Dirichlet distribution models the first *n* of them, which will generally sum to less than 1.)
- **Double logarithmic distribution**⬥: `(0.5 + (RNDINT(1) * 2 - 1) * RNDRANGEMaxExc(0, 0.5) * RNDU01OneExc())` (see also Saucier 2000, p. 15, which shows the wrong X axes).
- **Erlang distribution**: `GammaDist(n, 1.0 / lamda)`. Returns a number that simulates a sum of `n` exponential random numbers 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, 1), a)`, where `a` is a shape parameter.
- **Extended xgamma distribution** (Saha et al. 2019)[72]: `GammaDist(alpha + x, theta)`, where `x` is 0 with probability `theta/(theta+beta)` (e.g., if `RNDU01() <= theta/(theta+beta)`) and 2 otherwise, and where `alpha`, `theta`, and `beta` are shape parameters. If `alpha = 0`, the result is an **xgamma distribution** (Sen et al.,

2016)[73].

- **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 `RNDU01()`, `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 **Hypergeometric Distribution**.
- **Hypoexponential distribution**: See **Transformations of Random Numbers**.
- **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)) - 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)` (e.g., if `RNDU01() <= 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)[10].
- **`k`th-order statistic**: `BetaDist(k, n+1-k)`. Returns the `k`th smallest out of `n` uniform random numbers. See also (Devroye 1986, p. 210)[10].
- **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, 1)`. 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)[31]. If the application can trade accuracy for speed, the following can be used instead: `floor(1.0 - Expo(log1p(-pow(1.0 - p, RNDU01ZeroOneExc()))))`, where `p` is the parameter in (0, 1); see (Devroye 1986)[10].
- **Logistic distribution**†: `(ln(x)-log1p(-x))` (***logit function***), where `x` is `RNDU01ZeroOneExc()`.
- **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 numbers. See also (Devroye 1986, p. 675)[10].
- **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 numbers. See also (Devroye 1986, p. 210)[10].
- **Moyal distribution**: See the **Python sample code**.
- **Multinomial distribution**: See **Multinomial Distribution**.
- **Multivariate Poisson distribution**: See the **Python sample code**.
- **Multivariate *t*-copula**: See the **Python sample code**.
- **Multivariate *t*-distribution**: See the **Python sample code**.
- **Negative binomial distribution** (`NegativeBinomial(successes, p)`): See **Negative Binomial Distribution**. If the application can trade accuracy for speed: `Poisson(GammaDist(successes, (1 - p) / p))` (works even if `successes` is not an integer).
- **Negative multinomial distribution**: See the **Python sample code**.
- **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, 1) / GammaDist(w, 1)`, 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**.
- **Piecewise linear distribution**: See **Continuous Weighted Choice**.
- **Pólya–Aeppli distribution**: See **Transformations of Random Numbers: Additional Examples**.
- **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(RNDRANGE(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**.
- **Power normal distribution**: See the **Python sample code**.
- **Product copula**: See **Gaussian and Other Copulas**.
- **Rice distribution**: See **Multivariate Normal (Multinormal) Distribution**.
- **Rice–Norton distribution**: See **Multivariate Normal (Multinormal) Distribution**.
- **Singh–Maddala distribution**: See beta prime distribution.
- **Skellam distribution**: `Poisson(mean1) - Poisson(mean2)`, where `mean1` and `mean2` are the means of the two Poisson random numbers.
- **Skewed normal distribution**: `Normal(0, x) + mu + alpha * abs(Normal(0, x))`, where `x` is `sigma / sqrt(alpha * alpha + 1.0)`, `mu` and `sigma` have the same meaning as in the normal distribution, and `alpha` is a shape parameter.
- **Snedecor's (Fisher's) *F*-distribution**: `GammaDist(m * 0.5, n) / (GammaDist(n * 0.5 + Poisson(sms * 0.5)) * m, 1)`, where `m` and `n` are the numbers of degrees of freedom of two random numbers 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**. *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**.
- **Suzuki distribution**: See Rayleigh distribution.
- **Tukey lambda distribution**: `(pow(x, lamda)-pow(1.0-x,lamda))/lamda`, where `x` is `RNDU01()` and `lamda` is a shape parameter.
- **Twin-*t* distribution** (Baker and Jackson 2018)**(74)**: 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)` (e.g., if `RNDU01() < 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**.
- **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(RNDU01ZeroOneExc(), -1.0 / r))`, and if `d / pow(2, r) < RNDRANGEMaxExc((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)**(10)**. A zeta distribution **truncated** by rejecting random values greater than some positive integer is called a *Zipf distribution* or *Estoup distribution*. (Note that 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 Box

To generate a random point inside an N-dimensional box, generate `RNDRANGEMaxExc(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 `[RNDRANGEMaxExc(0,2), RNDRANGEMaxExc(3,6)]`, and
- to generate a *complex number* with real and imaginary parts bounded in [0, 1], generate `[RNDRANGE(0, 1), RNDRANGE(0, 1)]`.

### 8.10.2 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. See also (Grimme 2015)**(75)**, which shows MATLAB code for generating a random point uniformly inside a simplex just described, but in a different way.

```
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
   gammas=NewList()
   // Sample from a Dirichlet distribution
   for i in 0...size(points): AddItem(gammas, Expo(1))
   tsum=0
   for i in 0...size(gammas): tsum = tsum + gammas[i]
   tot = 0
   for i in 0...size(gammas) - 1
       gammas[i] = gammas[i] / tsum
       tot = tot + gammas[i]
   end
   tot = 1.0 - tot
   for i in 0...size(points[0]): AddItem(ret, points[0][i]*tot)
   for i in 1...size(points)
      for j in 0...size(points[0])
         ret[j]=ret[j]+points[i][j]*gammas[i-1]
      end
   end
   return ret
END METHOD
```

### 8.10.3 Random Points on the Surface of a Hypersphere

The following pseudocode shows how to generate a random N-dimensional point on the surface of an N-dimensional hypersphere, centered at the origin, of radius `radius` (if `radius` is 1, the result can also serve as a unit vector in N-dimensional space). Here, `Norm` is given in the appendix. See also (Weisstein)**(76)**.

```
METHOD RandomPointInHypersphere(dims, radius)
  x=0
  while x==0
    ret=[]
```

```
    for i in 0...dims: AddItem(ret, Normal(0, 1))
    x=Norm(ret)
  end
  invnorm=radius/x
  for i in 0...dims: ret[i]=ret[i]*invnorm
  return ret
END METHOD
```

> **Note:** The **Python sample code** contains an optimized method for points on the edge of a circle.
>
> **Example:** To generate a random point on the surface of a cylinder running along the Z axis, generate random X and Y coordinates on the edge of a circle (2-dimensional hypersphere) and generate a random Z coordinate by `RNDRANGE(mn, mx)`, where `mn` and `mx` are the highest and lowest Z coordinates possible.

### 8.10.4 Random Points Inside a Ball, Shell, or Cone

To generate a random N-dimensional point on or inside an N-dimensional ball, centered at the origin, of radius R, either—

- generate a random (N+2)-dimensional point on the surface of an (N+2)-dimensional hypersphere with that radius (e.g., using `RandomPointInHypersphere`), then discard the last two coordinates (Voelker et al., 2017)[77], or
- follow the pseudocode in `RandomPointInHypersphere`, except replace `Norm(ret)` with `sqrt(S + Expo(1))`, where `S` is the sum of squares of the numbers in `ret`.

To generate a random point on or inside 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), either—

- generate a random point for a ball of radius B until `Norm(pt)` is A or greater, where `pt` is that point (see the **appendix**), or
- generate a random point on the surface of an N-dimensional hypersphere with radius equal to `pow(RNDRANGE(pow(A, N), pow(B, N)), 1.0 / N)`[78].

To generate a random point on or inside 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(RNDRANGE(0, H), RNDRANGE(0, H)), RNDRANGE(0, H))`, then generate random X and Y coordinates inside a disc (2-dimensional ball) with radius equal to `max(RNDRANGE(0,R*Z/H), RNDRANGE(0,R*Z/H))`[79].

> **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 `RNDRANGE(mn, mx)`, where `mn` and `mx` are the highest and lowest Z coordinates possible.
>
> **Notes:**
>
> 1. The **Python sample code** contains a method for generating a random point on the surface of an ellipsoid modeling the Earth.
> 2. Sampling a half-ball, half-shell, or half-hypersphere can be done by sampling a full ball, shell, or hypersphere and replacing one of the dimensions of the result with its absolute value.

### 8.10.5 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)**(80)**—

- generate the longitude `RNDRANGEMaxExc(-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 = RNDRANGE(-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 random and pseudorandom number generation. All of them are open-source.

- **Random Number Generator Recommendations for Applications**
- **More Random Number Sampling Methods**
- **Code Generator for Discrete Distributions**
- **The Most Common Topics Involving Randomization**
- **Partially-Sampled Random Numbers for Accurate Sampling of the Beta, Exponential, and Other Continuous Distributions**
- **Bernoulli Factory Algorithms**
- **Testing PRNGs for High-Quality Randomness**
- **Examples of High-Quality PRNGs**

# 11 Notes

- (1) Pedersen, K., "**Reconditioning your quantile function**", arXiv:1704.07949v3 [stat.CO], 2018.

- (2) 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**. Part of the pseudocode uses the Fast Dice Roller in Lumbroso, J., "**Optimal Discrete Uniform Generation from Coin Flips, and Applications**", arXiv:1304.1916 [cs.DS].

- (3) An example of such a source is a Gaussian noise generator. This kind of source is often called an *entropy source*.

- (4) 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 an independent uniform random number 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 the source of random numbers for `RNDINT` and implement `RNDINTEXC` through `RNDINT` instead.


- (5) Lumbroso, J., "**Optimal Discrete Uniform Generation from Coin Flips, and Applications**", arXiv:1304.1916 [cs.DS].

- (6) Sanders, P., Lamm, S., et al., "**Efficient Parallel Random Sampling – Vectorized, Cache-Efficient, and Online**", arXiv:1610.0514v2 [cs.DS], 2019.

- (7) Jeff Atwood, "**The danger of naïveté**", Dec. 7, 2007.

- (8) Bacher, A., Bodini, O., et al., "**MergeShuffle: A Very Fast, Parallel Random Permutation Algorithm**", arXiv:1508.03167 [cs.DS], 2015.

- (9) Merlini, D., Sprugnoli, R., Verri, M.C., "An Analysis of a Simple Algorithm for Random Derangements", 2007.

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

- (11) See also the *Stack Overflow* question "Random index of a non zero value in a numpy array".

- (12) 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 numbers.

- (13) Brownlee, J. "**A Gentle Introduction to the Bootstrap Method**", *Machine Learning Mastery*, May 25, 2018.

- (14) Propp, J.G., Wilson, D.B., "Exact sampling with coupled Markov chains and applications to statistical mechanics", 1996.

- (15) Fill, J.A., "**An interruptible algorithm for perfect sampling via Markov chains**", *Annals of Applied Probability* 8(1), 1998.

- (16) E. N. Gilbert, "Random Graphs", *Annals of Mathematical Statistics* 30(4), 1959.

- (17) V. Batagelj and U. Brandes, "Efficient generation of large random networks", Phys.Rev. E 71:036113, 2005.

- (18) Costantini, Lucia. "Algorithms for sampling spanning trees uniformly at random." Master's thesis, Universitat Politècnica de Catalunya, 2020. **https://upcommons.upc.edu/bitstream/handle/2117/328169/memoria.pdf**

- (19) Penschuck, M., et al., "**Recent Advances in Scalable Network Generation**", arXiv:2003.00736v1 [cs.DS], 2020.

- (20) 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 random numbers 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 random numbers between 0 and 1, and there are infinitely many of them.

- (21) Efraimidis, P. and Spirakis, P. "**Weighted Random Sampling (2005; Efraimidis, Spirakis)**", 2005.

- (22) Efraimidis, P. "**Weighted Random Sampling over Data Streams**", arXiv:1012.0256v2 [cs.DS], 2015.

- [23] T. Vieira, "**Gumbel-max trick and weighted reservoir sampling**", 2014.
- [24] T. Vieira, "**Faster reservoir sampling by waiting**", 2019.
- [25] 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.
- [26] 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.
- [27] An *affine transformation* is one that keeps straight lines straight and parallel lines parallel.
- [28] Farach-Colton, M. and Tsai, M.T., 2015. Exact sublinear binomial sampling. *Algorithmica* 73(4), pp. 637-651.
- [29] 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).
- [30] von Neumann, J., "Various techniques used in connection with random digits", 1951.
- [31] Flajolet, P., Pelletier, M., Soria, M., "**On Buffon machines and numbers**", arXiv:0906.5560v2 [math.PR], 2010.
- [32] Karney, C.F.F., "**Sampling exactly from the normal distribution**", arXiv:1303.6257v2 [physics.comp-ph], 2014.
- [33] Smith and Tromble, "**Sampling Uniformly from the Unit Simplex**", 2004.
- [34] Durfee, et al., "l1 Regression using Lewis Weights Preconditioning and Stochastic Gradient Descent", *Proceedings of Machine Learning Research* 75(1), 2018.
- [35] 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.
- [36] This method corresponds to `Math.random()` in Java and JavaScript.
- [37] This includes integers if `e` is limited to 0, and fixed-point numbers if `e` is limited to a single exponent less than 0.
- [38] Downey, A. B. "**Generating Pseudo-random Floating Point Values**", 2007.
- [39] 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.
- [40] Goualard, F., "**Generating Random Floating-Point Numbers by Dividing Integers: a Case Study**", 2020.
- [41] Monahan, J.F., "Accuracy in Random Number Generation", *Mathematics of Computation* 45(172), 1985.
- [42] Spall, J.C., "An Overview of the Simultaneous Perturbation Method for Efficient Optimization", *Johns Hopkins APL Technical Digest* 19(4), 1998, pp. 482-492.
- [43] P. L'Ecuyer, "Tables of Linear Congruential Generators of Different Sizes and Good Lattice

Structure", *Mathematics of Computation* 68(225), January 1999, with **errata**.

- (44) Harase, S., "**A table of short-period Tausworthe generators for Markov chain quasi-Monte Carlo**", arXiv:2002.09006 [math.NA], 2020.

- (45) D. Revuz, M. Yor, "Continuous Martingales and Brownian Motion", 1999.

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

- (47) Saucier, R. "Computer Generation of Statistical Distributions", March 2000.

- (48) Other references on density estimation include **a Wikipedia article on multiple-variable kernel density estimation**, and a **blog post by M. Kay**.

- (49) "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 0 and a *covariance matrix* that, in this context, serves as a *bandwidth matrix*. "Jitter" and bandwidth are not further discussed in this document.

- (50) More formally—

    - the PDF is the *derivative* (instantaneous rate of change) of the distribution's CDF (that is, PDF(x) = CDF′(x)), and
    - the CDF is also defined as the *integral* ("area under the curve") of the PDF,

    provided the PDF's values are all 0 or greater and the area under the PDF's curve is 1.

- (51) A *discrete distribution* is a distribution that associates one or more items with a separate probability. This page assumes (without loss of generality) that these items are integers. A discrete distribution can produce non-integer values (e.g., `x/y` with probability `x/(1+y)`) as long as the values can be converted to and from integers. 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.

- (52) If those values are inexact floating-point numbers, their relative error will generally be smaller the closer they are to 0 (see also Walter, 2019).

- (53) This includes integers if `FPExponent` is limited to 0, and fixed-point numbers if `FPExponent` is limited to a single exponent less than 0.

- (54) Based on a suggestion by F. Saad in a personal communication (Mar. 26, 2020).

- (55) Saad, F.A., et al., "**Optimal Approximate Sampling from Discrete Probability Distributions**", arXiv:2001.04555 [cs.DS], 2020. See also the **associated source code**.

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

- (57) 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).

- [58] Walter, M., "Sampling the Integers with Low Relative Error", in *International Conference on Cryptology in Africa*, Jul. 2019, pp. 157-180.
- [59] 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.
- [60] 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".
- [61] 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.
- [62] Devroye, L., Gravel, C., "**The expected bit complexity of the von Neumann rejection algorithm**", arXiv:1511.02273v2 [cs.IT], 2016.
- [63] 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.
- [64] 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.
- [65] Kschischang, Frank R. "A Trapezoid-Ziggurat Algorithm for Generating Gaussian Pseudorandom Variates." (2019).
- [66] Devroye, L., 1996, December, "Random variate generation in one line of code" In *Proceedings Winter Simulation Conference* (pp. 265-272). IEEE.
- [67] McGrath, E.J., Irving, D.C., "Techniques for Efficient Monte Carlo Simulation, Volume II", Oak Ridge National Laboratory, April 1975.
- [68] Devroye, L., "Expected Time Analysis of a Simple Recursive Poisson Random Variate Generator", *Computing* 46, pp. 165-173, 1991.
- [69] Giammatteo, P., and Di Mascio, T., "Wilson-Hilferty-type approximation for Poisson Random Variable", *Advances in Science, Technology and Engineering Systems Journal* 5(2), 2020.
- [70] Bailey, R.W., "Polar generation of random variates with the t distribution", *Mathematics of Computation* 62 (1984).
- [71] 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.
- [72] Saha, M., et al., "**The extended xgamma distribution**", arXiv:1909.01103 [math.ST], 2019.
- [73] Sen, S., et al., "The xgamma distribution: statistical properties and application", 2016.
- [74] Baker, R., Jackson, D., "**A new distribution for robust least squares**", arXiv:1408.3237 [stat.ME], 2018.
- [75] Grimme, C., "Picking a Uniformly Random Point from an Arbitrary Simplex", 2015.
- [76] Weisstein, Eric W. "**Hypersphere Point Picking**". From MathWorld—A Wolfram Web Resource.
- [77] Voelker, A.R., Gosmann, J., Stewart, T.C., "Efficiently sampling vectors and coordinates from the *n*-sphere and *n*-ball", Jan. 4, 2017.

- (78) See the *Mathematics Stack Exchange* question titled "Random multivariate in hyperannulus", `questions/1885630`.
- (79) See the *Stack Overflow* question "Uniform sampling (by volume) within a cone", `questions/41749411`. Square and cube roots replaced with maximums.
- (80) Reference: **"Sphere Point Picking"** in MathWorld (replacing inverse cosine with `atan2` equivalent).
- (81) Describing differences between SQL dialects is outside the scope of this document, but **Flourish SQL** describes many such differences, including those concerning randomization features provided by SQL dialects.
- (82) For example, see Balcer, V., Vadhan, S., "Differential Privacy on Finite Computers", Dec. 4, 2018; as well as Micciancio, D. and Walter, M., "Gaussian sampling over the integers: Efficient, generic, constant-time", in Annual International Cryptology Conference, August 2017 (pp. 455-485).

# 12 Appendix

## 12.1 Mean and Variance Calculation

The following method calculates the mean and the **bias-corrected sample variance** of a list of real numbers, using the **Welford method** presented by J. D. Cook. The method returns a two-item list containing the mean and that kind of variance in that order. (Sample variance is the estimated variance of a population or distribution assuming `list` is a random sample of that population or distribution.) The square root of the variance calculated here is what many APIs call a standard deviation (e.g. Python's `statistics.stdev`).

```
METHOD MeanAndVariance(list)
    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)]
END METHOD
```

> **Note:** The population variance (or biased sample variance) is found by dividing by `size(list)` rather than `(size(list)-1)`, and the standard deviation of the population or a sample of it is the square root of that variance.

## 12.2 Norm Calculation

The following method calculates the norm of a vector (list of numbers), more specifically the $\ell_2$ norm of that vector.

```
METHOD Norm(vec)
  ret=0
  rc=0
  for i in 0...size(vec)
    rc=vec[i]*vec[i]-rc
    rt=rc+ret
    rc=(rt-ret)-rc
    ret=rt
  end
  return sqrt(ret)
END METHOD
```

There are other kinds of norms besides the $\ell_2$ norm. More generally, the $\ell_p$ norm, where $p$ is 1 or greater or is $\infty$, is the $p^{\text{th}}$ root of the sum of $p^{\text{th}}$ powers of a vector's components' absolute values (or, if $p$ is $\infty$, the highest absolute value among those components). An $\ell_p$ ball or sphere of a given radius is a ball or sphere that is bounded by or traces, respectively, all points with an $\ell_p$ norm equal to that radius. (An $\ell_\infty$ ball or sphere is box-shaped.)

## 12.3 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 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 generate random numbers, but popular SQL dialects often do — with idiosyncratic behavior.**(81)** 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 a random number is generated. But for ***stateless* PRNG designs** (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 numbers 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 random number generator, and letting each thread fill its assigned chunk with random numbers. For an example, see "**Multithreaded Generation**".

## 12.4 Security Considerations

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

1. **Cryptographic RNG.** The application has to use a cryptographic RNG. Choosing a cryptographic RNG is outside the scope of this document.
2. **Timing attacks.** Certain security attacks have exploited timing and other

differences to recover cleartext, encryption keys, or other sensitive data. Thus, so-called "constant-time" security algorithms have been developed. (In general, "constant-time" algorithms are designed to have no timing differences, including memory access patterns, that reveal anything about any secret inputs, such as keys, passwords, or RNG "seeds"). But even if an algorithm has variable running time (e.g., **rejection sampling**), it may or may not have security-relevant timing differences, especially if it does not reuse secrets.

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.** Random numbers generated for security purposes are almost always integers (and, in very rare cases, fixed-point numbers). Even in the few security applications where random floating-point numbers are used (differential privacy and lattice-based cryptography), there are ways to avoid such random numbers[(82)].

# 13 License