# The Knuth and Yao algorithm for the generation of random variables ($DRAFT$)

Claude Gravel

March 29, 2020

**Abstract**

This document contains my notes and summary of the Knuth and Yao [6] algorithm to generate random variables as well as a realistic implementation in C/C++. The paper [6] explains how to sample any discrete probability distributions provided (1) the entire description of the binary expansions of the probabilities or the possibility to request on the fly the binary coefficients of those expansions, and (2) a source of unbiased, identically and independently distributed bits is available. The physical nature of computers imply that we have to limit the storage of the coefficients in some ways that we discuss here and which is intimately tight to the accuracy we can afford. I discuss also about how much the implementation (storage limitation and accuracy) diverges from the target (possibly with an infinite support). I do not discuss the quality of bit pseudo-random generators used as sources of randomness. The purpose is actually to describe and implement [6] for which the expected number of random bits required is optimal in the information theoretical sense, that is, it lies at an no more than 2 bits from the binary entropy of the distribution. Please send comments, notes, contributions, errors, typos, etc to the email address above and your name is added in the acknowledgement. The code is also at the GitHub link mentioned further.

## 1 Introduction

Let $A \subseteq \mathbb{Z}$ and $\mathbf{p} = (p_i)^{i \in A}$ be a probability vector, that is, $p_i > 0$ and $\sum_{i \in A} p_i = 1$. There are a few methods to generate a random variable according to $\mathbf{p}$ like the inversion (probability integral transform) based on the cumulative distribution function, envelop methods like von Neumann rejection, ad hoc methods that use structures from the underlying distributions. Knuth and Yao [6] explains a nearly optimal method that use the probability mass vector directly. The

[0]This document is updated when time allows. It does not contain any new results. It is solely to gather facts about the algorithm and therefore create a useful and efficient implementation.

method in [6] uses DDG trees (Discrete Data Generator tree) that I explain in my ways hereafter.

In the following, I use `RandomBit` to describe a device, a method, or an oracle that is assumed to return an unbiased bit independently of any previous calls.

Just to warm up a bit, suppose we want to simulate a dice with three faces so that $\mathbf{p} = (\frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6})$ given we have access to `RandomBit`. The amount of randomness in $\mathbf{p}$ is $H(\mathbf{p}) = \log_2(6)$ where the latter is the binary entropy of $\mathbf{p}$. Thus on average we expect $\log_2(6)$ calls to `RandomBit` and, from an information theoretical point of view, we cannot do better.

# 2 Description of the implementation

This small set of routines perform exact random number generation of non-uniform discrete probability distributions up to an arbitrary finite precision for the probabilities. The precision and the binary representations are specified by the user. The algorithm can be found in Knuth and Yao [6]. The only limitation is the storage given the possibility to compute with arbitrary large finite precision the components of the probability vector. The routines use only functions from the C++ standard library including the generator for the source of raw (pseudo)-random unbiased i.i.d. bits. Some examples of probability distributions are given in the binary files together with this routine. Examples are explained in Section 5 and binary files for the binary expansions for those examples can be found at .

A factor to take into account a lower bound on the time complexity is the number of random coins needed to generate a given distribution. The last chapter 15 of Devroye [3] discuss the generation of random variables from a discrete source of i.i.i. unbiased bits. Lumbroso's thesis [2] explains how to implement efficiently Knuth and Yao's algorithm for the uniform discrete distribution under the name "The Dice Roller" by analogy to a multi-faceted non-biased dice.

A $p$-bit number is a representation for a real number $x$ given by a pair of mantissa and exponent $(m, p)$ where $m$ is an odd integer such that $x = m2^{-p}$. For instance, the class RR from the NTL library [8], allows for arbitrary precision arithmetic by letting a user to specify $p$. *From NTL page*: $\triangleright$ The real number 0 is represented by $(0, 0)$. All arithmetic operations are implemented so that the effect is as if the result was computed exactly, and then rounded to $p$ bits. If a number lies exactly half-way between two $p$-bit numbers, the "round to even" rule is used. So in particular, the computed result will have a relative error of at most $2^{-p}$. The previous rounding rules apply to all arithmetic operations in this module, except for the following routines:

1. The transcendental functions: log, exp, log10, expm1, log1p, pow, sin, cos, ComputePi
2. The power function
3. The input and ascii to RR conversion functions when using "e"-notation

For these functions, a very strong accuracy condition is still guaranteed: the computed result has a relative error of less than $2^{-p+1}$ (and actually much closer to $2^{-p}$). That is, it is as if the resulted were computed exactly, and then rounded to one of the two neighboring $p$-bit numbers (but not necessarily the closest). $\lhd$

The code given here only relies on the standard C++ library and let the user manage the computation of the binary representations. A few binary files described in the example below are given to test the algorithm and its speed. Details about the routines are contained in the C++ file "knuth_yao_1976_sampling_algo.cpp".

# 3 Principles and facts behind Knuth and Yao sampling algorithm

In this section, I explain ideas from [6], give two examples and pseudo-code implementation which is nearly C/C++.

For $A \subseteq \mathbb{Z}$, let $\mathbf{p} = (p_i)_{i \in A}$ be a probability vector, that is, $p_i > 0$ for all $i \in A$ and $\sum_{i \in A} p_i = 1$. For $i \in A$, write the binary expansion of $p_i$ as

$$p_i = \sum_{j=1}^{\infty} p_{ij} 2^{-j}.$$

For a while, suppose we have the ability to compute $p_{ij}$ on the fly or the ability of infinite storage whenever the expansions don't have a finite number of terms.

For $j \geq 1$, define the family of sets $L_j$ by

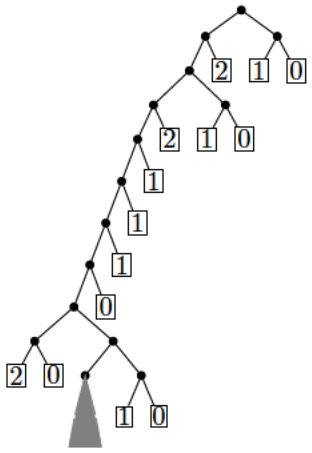
$$L_j = \{i \in A \text{ and } p_{ij} = 1\}.$$

In other words, $L_j$ is the set of outcomes which have $2^{-j}$ in their probability of occurrence. We have that

$$\begin{aligned} \sum_{i \in A} p_i &= \sum_{i \in A} \sum_{j=1}^{\infty} p_{ij} 2^{-j} \\ &= \sum_{j=1}^{\infty} \sum_{i \in A} p_{ij} 2^{-j} \\ &= \sum_{j=1}^{\infty} \frac{|L_j|}{2^j} \\ &= 1. \end{aligned}$$

Clearly $0 \leq |L_j| \leq 2^j$ for all $j \geq 1$, and the existence of a $j'$ such that $|L_{j'}| = 2^{j'}$ occurs if and only if $\mathbf{p}$ is the uniform distribution over $2^{j'}$ outcomes. More importantly, $L_j$ is *uniformly* distributed that is $\mathbf{P}\{i \in L_j\} = \frac{1}{|L_j|}$.

Figure 1: DDG tree




A probability vector $\mathbf{p} = (p_i)_{i \in A}$ has a unique (often infinite) binary tree representation for which (1) the leaves of the $j$-th level are the elements of $L_j$, (2) the number of internal nodes (nodes of the $j$-th level that are not leaves), denoted by $s_j$ for convenience, equals $t_j - |L_j|$ where $t_j$ is the total number of nodes at level $j$. Without loss of generality, $L_0 = \emptyset$ and we have for $j \geq 1$ that

$$t_j = 2^j - \sum_{k=0}^{j-1} 2^{j-k} |L_k|.$$

Knuth and Yao [6] called this tree a DDG tree for Discrete Data Generator tree. For levels $j \geq 1$, we have that $t_j = s_j + |L_j|$, and given the ability to generate uniform random variables, we can thus generate any non-uniform distribution.

For example, consider the probability vector $(p_0, p_1, p_2)$ where

$$\begin{aligned} p_0 &= \frac{1}{\pi} \\ &= (0.010100010111110\ldots)_2. \\ p_1 &= \frac{1}{e} \\ &= (0.010111100010110\ldots)_2. \\ p_2 &= 1 - p_1 - p_2 \\ &= (0.010100000101010\ldots)_2. \end{aligned}$$

The DDG tree of $\mathbf{p}$ is represented on Figure 3. A few values of $|L_j|$, $s_j$ and $t_j$ are given in Table 1.

| $j$ | $\|L_j\|$ | $s_j$ | $t_j$ |
|---|---|---|---|
| 1 | 0 | 0 | 2 |
| 2 | 3 | 1 | 4 |
| 3 | 0 | 2 | 2 |
| 4 | 3 | 1 | 4 |
| 5 | 1 | 1 | 2 |
| 6 | 1 | 1 | 2 |
| 7 | 1 | 1 | 2 |
| 8 | 1 | 1 | 2 |
| 9 | 0 | 2 | 2 |
| 10 | 2 | 2 | 4 |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |

Note that it is a coincidence that values of $t_j$ in Table 1 are powers of 2; if we would continue the tree and the table or use another probability vector, then we would notice that the $t_j$'s are generally not powers of 2.

To generate a random outcome according to a probability vector $\mathbf{p}$, one has to generate random i.i.d. unbiased Bernoulli variables $B_1$, $B_2$, ..., $B_j$, ... that are used to represent *uniform* random values in the intervals $[0, t_1)$, $[0, t_2)$, ..., $[0, t_j)$, ..., and stops as soon as the latter uniform value is in $[0, |L_j|)$.

In the following code implementing the algorithm, $L$ can be a vector of vectors (list of lists) for instance, values of $t_j$ can be hold in an array.

```
x=0
y=1
j=1
while(true)
    {
        x = 2x + RandomBit
        y = 2y;
        if( y >= t[j])
            {
                if(x<|L[j]|)
                    {
                        Return L[j][x]
                    }
            }
        else
            {
                y = y - |L[j]|
                x = x - |L[j]|
            }
        Increment j by 1
    }
```

# 4 A note on KL-divergence and distinguishing the implemented from the ideal distribution

Suppose the ideal distribution has a finite support that is a finite number of atoms, say $n > 1$. Denote the ideal probability mass vector by $\mathbf{q} = (q_i)_{i=1}^n$ and its implementation by $\mathbf{p} = (p_i)_{i=1}^n$. Let $p_i = u_i/C_p$ and $q_i = v_i/C_q$ where $C_p = \sum_{i=1}^n p_i$ and $C_q = \sum_{i=1}^n q_i$. Given the ability to compute with $u_i$ with $k$ bits of accuracy that is $|u_i - v_i| < 2^{-k}$, how far is $\mathbf{p}$ from $\mathbf{q}$? For convenience, let $\epsilon = 2^{-k}$ and $H(\mathbf{q})$ entropy of $\mathbf{q}$. The KL-divergence is

$$\begin{aligned} K(\mathbf{p}||\mathbf{q}) &= \sum_{i=1}^n p_i \log\left(\frac{q_i}{p_i}\right) \\ &= \sum_{i=1}^n p_i \log q_i + H(\mathbf{q}). \end{aligned}$$

Since $|u_i - v_i| < \epsilon$, then

$$\sum_{i=1}^n p_i \log\left(\frac{-\epsilon + u_i}{C_q}\right) < K(\mathbf{p}||\mathbf{q}) - H(\mathbf{q}) < \sum_{i=1}^n p_i \log\left(\frac{\epsilon + u_i}{C_q}\right)$$

# 5 Examples

In the examples below, the user who executes the code on a file mentioned below is prompted for a sample size and the filename. Each file mentioned below contains the binary representation of the probability distribution. The estimation of the expected number of random i.i.d. unbiased bits needed per random variables is to be compared with the exact numerical entropy mentioned below. In all cases, for a not too big sample, the estimated number of random coins fall within the bounds given by [6] that is between $H$ and $H + 2$ where $H$ is the entropy.

## 5.1 Discrete gaussian

The discrete gaussian mass function is given by

$$p_n = Ce^{-(\frac{n-\alpha}{\beta})^2} \text{ for } n \in \mathbb{Z}.$$

Given the impossibility to store an infinite support and the difficulty to analytically evaluate the normalization constant $C$, a truncation is considered for $\lfloor \alpha - 10\beta \rfloor \leq n \leq \lceil \alpha + 10\beta \rceil$.

The file "dis_nor.bit" contains the binary representation for $\alpha = e^{-1}$ and $\beta = 1000$. The minimal accuracy for each probability value is 1000 bits. The truncated range is $[-10001, 10001] \subset \mathbb{Z}$. The numerical entropy with given accuracy is therefore $11.512879869842728146\ldots$

The discrete gaussian is special case of the theta family for which $(\frac{n-\alpha}{\beta})^2$ is replaced by $-|\frac{n-\alpha}{\beta}|^k$ for $k \geq 1$. The case of $k = 1$ is a discrete analog to the Laplace distribution.

The discrete gaussian with large values of beta occurs in application like lattice-based cryptography [7].

Note that the discrete gaussian distribution is not the distribution of the integer parts of the continuous gaussian distribution.

## 5.2 Binomial

The file "bino.bit" contains the binary representation for binomial distribution with 2000 trials and occurrence probability 0.1. The minimal accuracy for each probability value is 1000 bits. The numerical entropy is 5.7925934431983804672...

## 5.3 Zeta-Dirichlet related

For $u > 0$, consider

$$C_u = \sum_{n=3}^{\infty} \frac{1}{n(\log n)^{1+u}},$$

and the probabilities

$$p_n = \frac{1}{C_u} \frac{1}{n(\log n)^{1+u}} \text{ for } n \geq 3.$$

The entropy is unbounded for all $0 < u \leq 1$ and bounded for $u > 1$. (For $u \leq 0$, the sum $C_u$ diverges. See Hardy and Riesz [5].)

A truncation with upper cutoff point $n = 2^{20}$ and with $u = 0.05$ is considered with proper re-normalization. Note that because the truncation has a finite support, then its entropy exists. The file containing the binary representations is "zeta.bit". The minimal accuracy is set at 1000 bits. For $0 < u \leq 1$, the entropy diverges slowly. The numerical entropy of the truncated probability distribution is 10.189437261652963733...

## 5.4 A basic three-mass distribution

We consider the probability vector $(p_0, p_1, p_2)$ where

$$\begin{aligned} p_0 &= \frac{1}{\pi} \\ &= (0.010100010111110\ldots)_2. \\ p_1 &= \frac{1}{e} \\ &= (0.010111100010110\ldots)_2. \\ p_2 &= 1 - p_1 - p_2 \\ &= (0.010100000101010\ldots)_2. \end{aligned}$$

The execution the of the sampling algorithm is represented on Figure 3. The entropy is 1.581127402961993034...

## 5.5 Discretization of a singular continuous distribution

For $j \geq 1$, let $X_j$ be non-identically, independently distributed Bernoulli random variable with

$$\begin{aligned}
\mathbf{P}\{X_j = 0\} &= 1 - p_j \\
&= \frac{1}{e^{-1/2^j} + 1}, \\
\mathbf{P}\{X_j = 1\} &= p_j \\
&= \frac{e^{-1/2^j}}{e^{-1/2^j} + 1}.
\end{aligned}$$

Consider the continuous random variables

$$\begin{aligned}
X &= \sum_{j=1}^{\infty} 2^{-2j} X_{2j}, \\
Y &= \sum_{j=1}^{\infty} 2^{-(2j-1)} X_{2j-1} \text{ and} \\
Z &= X + Y.
\end{aligned}$$

By Kakutani's theorem [1], $X$ and $Y$ are singular continuous random variables. The random variable $Z$ is a truncated exponential on the real interval $[0, 1]$ and hence is absolutely continuous.

We can discretize $X$ to a certain desired accuracy, 15 bits of accuracy accuracy say, for which the corresponding discretization has $2^{15-1}$ atoms. Each atom is represented with 1000 bits precision. The file "sing.bit" contains the discretization of $X$.

To discretize a singular distribution is generally easier than to discretize an absolutely continuous distribution since it does not require numerical integration or known analytic formulas for the distribution function.

## 5.6 A quantum mechanical discrete distribution

Let $n > 1$, the sets of angles $\{\theta_j\}_{j=1}^n$ and $\{\varphi_j\}_{j=1}^n$. For $b = (b_1, \ldots, b_n) \in \{-1, +1\}^n$, an example of a quantum mechanical distribution from [4] is given by

$$\begin{aligned}
p(b) &= \cos^2\left(\frac{\theta}{2}\right) p_1(b) + \sin^2\left(\frac{\theta}{2}\right) p_2(b) \text{ with} \qquad (1) \\
\theta &= \sum_{j=1}^{n} \theta_j \ ,
\end{aligned}$$

$$
\begin{aligned}
p_1(b) &= \frac{1}{2}\big(\mathrm{a}_1(b) + \mathrm{a}_2(b)\big)^2 \ , \\
p_2(b) &= \frac{1}{2}\big(\mathrm{a}_1(b) - \mathrm{a}_2(b)\big)^2 \ , \qquad (2) \\
\mathrm{a}_1(b) &= \prod_{j=1}^{n} \cos\left(\tfrac{1}{2}\big(\varphi_j - \tfrac{\pi}{2} b_j\big)\right) \ , \\
\mathrm{a}_2(b) &= \prod_{j=1}^{n} -\sin\left(\tfrac{1}{2}\big(\varphi_j - \tfrac{\pi}{2} b_j\big)\right). \qquad (3)
\end{aligned}
$$

We observe that $p$ is a convex combination of both $p_1$ and $p_2$.

The file "ghz.bit" contains the binary representations of $p(b)$ up to 1000 bits of accuracy for $n = 15$ with

$$
\varphi_j = \begin{cases} {}^{\pi}/_{6} & \text{if } j \equiv 0 \pmod 3, \\ {}^{3\pi}/_{6} & \text{if } j \equiv 1 \pmod 3, \\ {}^{5\pi}/_{6} & \text{if } j \equiv 2 \pmod 3. \end{cases}
$$

$$
\theta_j = \begin{cases} {}^{2\pi}/_{10} & \text{if } j \equiv 0 \pmod 5, \\ {}^{6\pi}/_{10} & \text{if } j \equiv 1 \pmod 5, \\ {}^{10\pi}/_{10} & \text{if } j \equiv 2 \pmod 5, \\ {}^{14\pi}/_{10} & \text{if } j \equiv 3 \pmod 5, \\ {}^{18\pi}/_{10} & \text{if } j \equiv 4 \pmod 5. \end{cases}
$$

The numerical entropy is $9.1127812445913279409\ldots$

# References


[1] Shizuo Kakutani. On equivalence of infinite product measures. *Annals of Mathematics*, pages 214–224, 1948.

[2] Jérémie Lumbroso. *Probabilistic Algorithms for Data Sreaming and Random Generation*. PhD thesis, Université Pierre et Marie Curie - Paris 6, 2012.

[3] Luc Devroye. *Non-Uniform Random Variate Generation*. Springer-Verlag, 1986.

[4] Gilles Brassard, Luc Devroye and Claude Gravel. Exact classical simulation of the quantum-mechanical GHZ distribution. *IEEE Trans. Inf. Theory*, 62(2):876–890, 2016.

[5] G.H. Hardy and M. Riesz. *The General Theory of Dirichlet's Series*. Cambridge Tracts in Mathematics and Mathematical Physics. Dover Publications, 2005.

[6] Donald E. Knuth and Andrew Chi-Chih Yao. The complexity of nonuniform random number generation. In J. F. Traub, editor, *Algorithms and Complexity: New Directions and Recent Results.*, pages 357–428, New York, 1976. Carnegie-Mellon University, Academic Press.

[7] Chris Peikert. A decade of lattice cryptography. https://web.eecs.umich.edu/~cpeikert/pubs/lattice-survey.pdf, 2016.

[8] Victor Shoup. NTL: A library for doing number theory. https://www.shoup.net/ntl/. Last checked on March 18, 2020.