An implementation of the Knuth and Yao algorithm for the generation of random variables (*DRAFT*)

Claude Gravel

April 8, 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 ability to request on the fly coefficients of those binary 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 in some ways the storage size for the coefficients. I discuss about how much is required in terms of storage and accuracy for the implementation to be as close as desired to the target (possibly with an infinite support for which we need to take into account the leftover truncation probality). 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 two 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 available at `https://github.com/63EA13D5/Knuth_and_Yao_algorithm_for_generation_of_random_variables`

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,

[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.

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 method in [6] uses DDG trees (Discrete Data Generator tree) that I explain hereafter. The last chapter of [3] also discusses the generation of random variables from discrete sources.

2 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++. 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.

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 do not have a finite number of terms or are not periodic.

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 $\mathbf{P}\{i \in L_j\} = \frac{1}{|L_j|}$.

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),

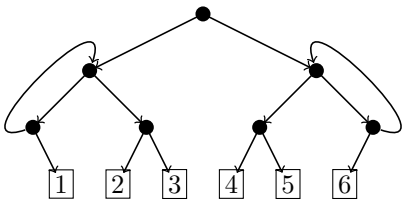

Figure 1: A fair dice

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 i.i.d. bits, we can thus generate any (non)-uniform distributions by walking down from the root of the tree to a leave at some level j. Walking left for instance when a random bit is 0 and right when it is 1. The algorithm halts with probability one.

The quantities t_j, $|L_j|$ and s_j are respectively the number of decisions (algorithm either halts or continues) for the jth level, the number of outcomes of the random variable for the jth level (that is the algorithm halts with probability $|L_j|/t_j$), and the number of ways the algorithm continues to the next level (that is the algorithm continues with probability s_j/t_j). As shown in [6], the former quantities entirely characterized the expected number of bits (and hence the running time).

Just to warm up a bit, suppose we want to simulate a dice with six 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`. We observe that the binary expansion of $\frac{1}{6}$ is $0.0\overline{01}$ where $\overline{01}$ means 01 is repeated ad infinitum. 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 for an optimal algorithm, we expect between $\log_2(6)$ and $\log_2(6) + 2$ calls to `RandomBit` and, from an information theoretical point of view, we cannot do better. Figure 1 shows the tree with an infinite countable number of levels for the simulation of the dice where the loops must be seen as infinite repetitions of the corresponding subtrees. Actually there is only one kind repeated subtree on Figure 1 which is for the discrete uniform distribution over three elements since $6 = 2 \cdot 3$.

With Figure 1 as an aid, we can find t_j, $|L_j|$ and s_j. We have that $t_1 = 2$, $|L_1| = 0$, and $s_1 = 2$. For $j \geq 2$, if $j - 2 \equiv 0 \pmod 2$ then $t_j = 4$, $|L_j| = 0$, and $s_j = 4$. For $j \geq 2$, if $j - 2 \equiv 1 \pmod 2$ then $t_j = 8$, $|L_j| = 6$, and $s_j = 2$. Let's recall the Fast Dice Roller from [2] which is an efficient implementation of Knuth and Yao ideas for the discrete uniform distribution over n points.

Algorithm 1 Fast Dice Roller (Lumbroso, 2012)

Input: Integer $n > 1$,
Output: X

```
1:  X ← 0
2:  Y ← 1
3:  loop
4:      Y ← 2Y
5:      B ← RandomBit
6:      X ← 2X + B
7:      if Y ≥ n then
8:          if X < n then
9:              Return X
10:         else
11:             Y ← Y − n
12:             X ← X − n
13:         end if
14:     end if
15: end loop
```

We observe that X, in the "loop" of the Fast Dice Roller, is uniformly distributed. Instructions from lines 12 and 11 are executed if and only if $X \geq n$ upon which X is uniformly distributed on $\{n, \ldots, Y-1\}$. Moreover, given that $X \geq n$, the set $\{n, \ldots, Y-1\} \neq \emptyset$ since $Y > X \geq n$ and $\{n, \ldots, Y-1\}$ is translated by n which allows random bits to be "recycled".

Theorem 1 (Lumbroso (2012))**.** *For all $\alpha > 0$, the expected number of calls to* `RandomBit` *for the Fast Dice Roller is*

$$\log_2(n) + \frac{1}{2} + \frac{1}{\log 2} - \frac{\gamma}{\log 2} + P(\log_2(n)) + O(n^{-\alpha}),$$

where P is a trigonometric periodic polynomial and γ is the Euler constant.

A slight modification on the Fast Dice Roller allows us to generate random variables for any discrete probability distribution provided the bits of the binary expansions of the probabilities are available on the fly. Let's look at another example now. Consider the probability vector (p_0, p_1, p_2) where

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

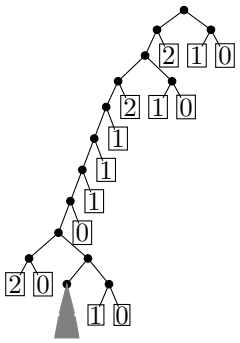

Figure 2: DDG tree for a probability vector of length 3 with irrational probabilities

The DDG tree of $\mathbf{p}$ is represented on Figure 2. 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
⋮	⋮	⋮	⋮

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, list of `bitset` C++ objects, etc.

Jump to next page.

```
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
    }
```

3 Description of the implementation

Procedures are described in the code. I may add more info here if I received questions.

3.1 Getting the binary representations

There are many softwares to obtain the binary representation. For the binary files supplied with the code, the NTL library [8] is used. In this subsection, p denotes the exponent part a real number representation in base two and not a probability distribution.

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 NTL 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). $\lhd$

4 Distinguishing the implemented from the ideal distribution—under construction

Suppose the ideal distribution has a finite support $n > 1$ (and n possibly infinite). Denote the ideal probability mass vector by $\mathbf{q} = (q_i)_{i=1}^{n}$ and its implementation by $\mathbf{p} = (p_i)_{i=1}^{|A|}$ where $A \subset \mathbb{Z}$ is the support of the implementation which is always finite. If n is infinite, let $\tilde{\mathbf{q}}$ be a truncation of $\mathbf{q}$ with leftover $0 < \delta < 1$ that is

$$1 - \delta = \sum_{i=i_0}^{i=i_1} q_i.$$

The choice of i_1, i_0 with $i_1 > i_0$ depends on the context and usually results in very small δ. From now, I assume $\mathbf{q}$ has a finite support of size $|A|$ for some $A \subset \mathbb{Z}$. Let $p_i = u_i/C_p$ and $q_i = v_i/C_q$ where $C_p = \sum_{i=1}^{|A|} p_i$ and $C_q = \sum_{i=1}^{|A|} q_i$. The u_i's and v_i's are un-normalized weights. In some cases, C_q might not be even tractable analytically like in the case of the discrete gaussian with non-integer mean. However if an approximation of v_i can be obtained, call it u_i, then, after proper normalization by C_p, how far are we from the ideal $\mathbf{q}$? How far $\mathbf{p}$ is from $\mathbf{q}$ determines asymptotically the sample size required by any statistical test to distinguish each other.

A few points must be taken into considerations for the distance. Here we use $H(\mathbf{p})$ or $H(\mathbf{q})$ to denote the binary entropy of $\mathbf{p}$ or $\mathbf{q}$ respectively. The points to consider are:

(1) The length $|A|$ of the implementation $\mathbf{p}$.

(2) The leftover probability δ whenever the original target $\mathbf{q}$ had infinite countable support.

(3) An additional margin of security to increase the accuracy, call this ℓ_s. The quantity ℓ_s depends on a subjective measure of the computational efforts one is ready to spend to run statistical tests.

For (1) above, let $\ell_{|A|} = \lceil H(\mathbf{p}) \rceil$. If $H(\mathbf{p})$ is unknown, then since $H(\mathbf{p}) \leq \log_2(|A|)$, let $\ell_{|A|} = \lceil \log_2(|A|) \rceil$.

For (2) above, if $\mathbf{q}$ has infinite support, and *under the assumption that* $H(\mathbf{q})$ *is bounded*, then let $\ell_\delta = \lceil -\log_2(\delta) \rceil$. If $\mathbf{q}$ has a finite support, then let $\ell_\delta = 0$.

The minimal accuracy must be at least larger than $\ell_{|A|} + \ell_\delta$, but such accuracy might not be enough depending on the computational power available to distinguish (quantum device, dna based device, hybrid device, classical distributed cloud computing, etc.) and an extra margin of security might be desired that we call ℓ_s.

For $\ell = (\ell_{|A|} + \ell_\delta + \ell_s)$, if the implemented weights u_i are ℓ-bit truncations of the ideal/target weights v_i, then

$$|u_i - v_i| < \frac{1}{2^\ell},$$

and any statistical tests would require a sample size of about $O\big(\sqrt{2^\ell}\big)$ to distinguish $\mathbf{p}$ from $\mathbf{q}$.

The storage requires is therefore $O(\ell)$ bits for every probability masses. However the expected time requires is only in $O(H(\mathbf{p}))$ since the expected running time is obviously bounded from below by the expected number of random bits needed which by [6] is expected between $H(\mathbf{p})$ and $H(\mathbf{p}) + 2$.

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 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

...

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].)

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 2.

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

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

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.

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

$$p(b) = \cos^2\left(\frac{\theta}{2}\right) p_1(b) + \sin^2\left(\frac{\theta}{2}\right) p_2(b) \text{ with} \tag{1}$$
$$\theta = \sum_{j=1}^{n} \theta_j \ ,$$
$$p_1(b) = \frac{1}{2}\big(\mathrm{a}_1(b) + \mathrm{a}_2(b)\big)^2 \ ,$$
$$p_2(b) = \frac{1}{2}\big(\mathrm{a}_1(b) - \mathrm{a}_2(b)\big)^2 \ , \tag{2}$$
$$\mathrm{a}_1(b) = \prod_{j=1}^{n} \cos\left(\tfrac{1}{2}\big(\varphi_j - \tfrac{\pi}{2} b_j\big)\right) \ ,$$
$$\mathrm{a}_2(b) = \prod_{j=1}^{n} -\sin\left(\tfrac{1}{2}\big(\varphi_j - \tfrac{\pi}{2} b_j\big)\right). \tag{3}$$

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

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.