Random Variate Generation with Formal Guarantees

FERAS A. SAAD, Carnegie Mellon University, USA
WONYEOL LEE, POSTECH, Republic of Korea

Generating random variates is a fundamental operation in diverse areas of computer science and is supported in almost all modern programming languages. Traditional software libraries for random variate generation are grounded in the idealized "Real-RAM" model of computation, where algorithms are assumed to be able to access uniformly distributed real numbers from the unit interval and compute with infinite-precision real arithmetic. These assumptions are unrealistic, as any software implementation of a Real-RAM algorithm on a physical computer can instead access a stream of individual random bits and computes with finite-precision arithmetic. As a result, existing libraries have few theoretical guarantees in practice. For example, the actual distribution of a random variate generator is generally unknown, intractable to quantify, and arbitrarily different from the desired distribution; causing runtime errors, unexpected behavior, and inconsistent APIs.

This article introduces a new approach to principled and practical random variate generation with formal guarantees. The key idea is to first specify the desired probability distribution in terms of a finite-precision numerical program that defines its cumulative distribution function (CDF), and then generate exact random variates according to this CDF. We present a universal and fully automated method to synthesize exact random variate generators given any numerical CDF implemented in any binary number format, such as floating-point, fixed-point, and posits. The method is guaranteed to operate with the same precision used to specify the CDF, does not overflow, avoids expensive arbitrary-precision arithmetic, and exposes a consistent API. The method rests on a novel space-time optimal implementation for the class of generators that attain the information-theoretically optimal Knuth and Yao entropy rate, consuming the least possible number of input random bits per output variate. We develop a random variate generation library using our method in C and evaluate it on a diverse set of "continuous" and "discrete" distributions, showing competitive runtime with the state-of-the-art GNU Scientific Library while delivering higher accuracy, entropy efficiency, and automation.

CCS Concepts: • **Mathematics of computing** → *Random number generation*; *Probability and statistics*; *Information theory*; *Mathematical software*; *Discretization*; • **Theory of computation** → *Probabilistic computation.*

Additional Key Words and Phrases: probabilistic programming, algorithm design and analysis, entropy

ACM Reference Format:
Feras A. Saad and Wonyeol Lee. 2025. Random Variate Generation with Formal Guarantees. *Proc. ACM Program. Lang.* 9, PLDI, Article 152 (June 2025), 25 pages. https://doi.org/10.1145/3729251

1 Introduction

The purpose of a random variate generation algorithm is to produce random numbers that adhere to a specified probability distribution. These distributions can be discrete, such as the Poisson distribution over the natural numbers, or continuous, like the Gaussian distribution over the real numbers. A fundamental result of probability theory establishes that, in purely mathematical terms, any random variate generator corresponds to a function that takes as input a real number $u \in [0, 1]$ drawn uniformly at random from the unit interval and returns as output a real number $x \in \mathbb{R}$ (Fig. 1, top row). Software libraries for random variate generation use this insight to develop numerical

Authors' Contact Information: Feras A. Saad, Carnegie Mellon University, Computer Science Department, Pittsburgh, PA, USA, fsaad@cmu.edu; Wonyeol Lee, POSTECH, Department of Computer Science and Engineering, Pohang, Gyeongbuk, Republic of Korea, wonyeol.lee@postech.ac.kr.

ACM 2475-1421/2025/6-ART152
https://doi.org/10.1145/3729251

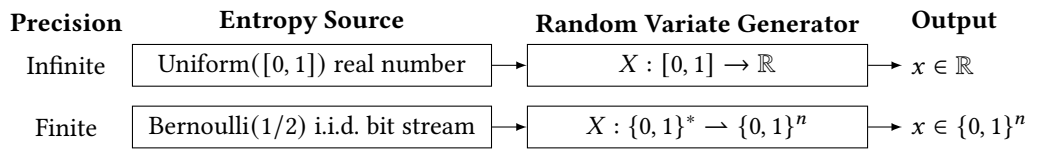


Fig. 1. Random variate generation with infinite-precision (Real-RAM) and finite-precision computation.

algorithms, whose correctness properties rest on a fictitious "Real-RAM" computer that can perform infinitely precise computation over real numbers in constant time [13; §1.1 Assumptions I—III].

Key Challenges. The Real-RAM model was introduced in Shamos's 1978 dissertation [56] on computational geometry and adopted in Devroye's 1986 random variate generation book [13] for its conceptual simplicity. However, this model fails to characterize the error and complexity properties of software that executes on finite-precision hardware, as cautioned by von Neumann in 1951:

> *Anyone who considers arithmetic methods of producing random digits is, of course, in a state of sin... If one considers arithmetic methods in detail, it is quickly found that the critical thing about them is the very obscure, very imperfectly understood behavior of round-off errors in mathematics... One might as well admit that one can prove nothing, because the amount of theoretical information about the statistical properties of the round-off mechanism is nil.* —Von Neumann [65]

A modern illustration of these problems can be found in differentially private algorithms [21] that use random numbers to obfuscate sensitive datasets, such as the 2020 US Census [1]. Mironov [42] demonstrates that floating-point effects in the Laplace random variate generator from existing software libraries can entirely destroy the real-world privacy guarantees of algorithms that are otherwise differentially private under the Real-RAM assumption. Similar issues arise in areas such as lattice-based cryptography [11, 17, 20, 22, 35, 51], where developers of secure protocols need more-realistic computational models than Real RAM that enable them to rigorously characterize
(a) the approximation error on a finite-precision computer, to establish security guarantees;
(b) the required register size needed for a given accuracy level, to design efficient hardware;
(c) the entropy consumption per output variate, to predict runtime and avoid side channel attacks.

This Work. This article introduces a theoretically principled and practical approach to random variate generation grounded in a finite-precision model of computation (Fig. 1, bottom row). On the input side, a random variate generator has lazy access to a stream of independent unbiased random bits. It returns as output a finite n-bit string that represents a number in some binary encoding, computed using finite-precision arithmetic and finite memory. An immediate consequence of this model is that the generator can produce at most 2^n distinct outputs, each with a rational probability.

Our approach to random variate generation begins with a formal specification of the desired probability distribution of the random variates. This specification takes the form of a numerical program that implements a cumulative distribution function (CDF) F. For any finite-precision number x that can be represented on the computer, $F(x)$ computes the probability that the random variate is less than or equal to x. Given such a CDF, our technique automatically synthesizes a random variate generator that is guaranteed to exactly match F. This generator is also guaranteed to be *entropy-optimal*—it draws the information-theoretically least number of random bits on the input side to produce an output, achieving the optimal entropy rate from Knuth and Yao [37].

Figure 2 compares our approach to existing random variate generation software libraries. First consider the idealized Real-RAM model (Fig. 2a). Every probability measure has multiple equivalent representations in terms of a unique cumulative distribution function (CDF), survival function (SF), quantile function (QF), or infinite collection of identically distributed random variables (i.e., measurable functions from $[0,1]$ into $\mathbb{R}$). Figure 2b shows the approach in standard software libraries, which provide numerical programs for the CDF, SF, QF, and random variate generator(s)

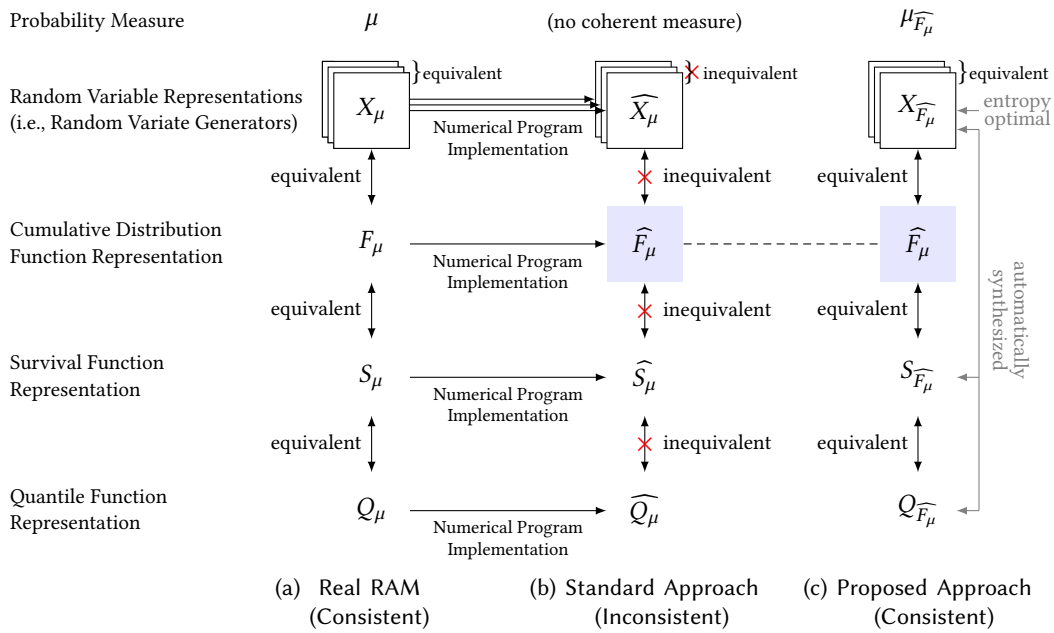


Fig. 2. Overview of the standard and proposed approaches to random variate generation software libraries.

that approximate the corresponding Real-RAM functions. The resulting diagram, however, does not "commute" (red cross marks in Fig. 2b)—due to numerical errors, each of these numerical programs actually defines a *different* probability measure, creating an inconsistent API for what was originally a single coherent probability measure in the infinite-precision Real RAM model. Further, the actual output distributions of the implemented random variate generators are generally intractable to compute, making it difficult to formally characterize their properties such as approximation error or expected runtime. Figure 2c shows the proposed approach, where the desired probability measure is first specified using a numerical program that computes its CDF. This CDF is then used to synthesize an exact random variate generator and related functions. The resulting API for the implemented probability measure is mathematically coherent, perfectly mirroring the Real-RAM equivalences.

Contributions. This article makes the following contributions.

(C1) Formulation of exact random variate generation using finite precision. We rigorously formulate the problem of generating exact random variates given a finite-precision implementation of a CDF. This approach is fundamentally different from existing libraries (Fig. 2). It guarantees a coherent API for the cumulative distribution, survival, and quantile functions of the implemented generator that all agree with one another. It also enables the fully automatic construction of a random variate generator given a formal specification of the desired probability distribution, and allows for strong theoretical guarantees on exactness, entropy optimality, and practical efficiency.
(C2) Exact and optimal random variate generators for binary-coded distributions (§4). We present a sound and entropy-optimal algorithm (Theorem 4.6) for generating random variates given any binary-coded probability distribution, which is a universal mathematical representation for probability measures over $\mathbb{R}$. This algorithm improves upon a method described in Knuth and Yao [37]: it obtains optimal space-time complexity by exploiting properties of binary expansions of real numbers that govern the structure of entropy-optimal generators (Theorems 4.2 and 4.4).

(C3) Exact and optimal random variate generators in finite precision (§5). We specialize the universal algorithm for binary-coded probability distributions to soundly generate exact random variates given a finite-precision implementation of a CDF over any binary number format (e.g., floating-point, fixed-point, posits; Theorem 5.17). This algorithm is information-theoretically optimal and highly efficient in software: it is guaranteed to require the same precision as the given CDF (Theorem 5.16) and uses fast integer arithmetic instead of expensive arbitrary-precision arithmetic.
(C4) Extended-accuracy random variate generators that combine both CDF and SF (§6). We present an extension of the method from §5 that achieves higher accuracy, especially in the tails of a probability distribution, using a principled combination of a finite-precision CDF and survival function (SF) implementation (Theorem 6.3). This algorithm enjoys the same theoretical guarantees as before, while being able to represent twice as many outcomes as compared to only a CDF or SF.
(C5) Implementation and empirical evaluations (§7). We develop and evaluate a random variate generation library using our methods in C. The results show that, as compared to the state-of-the-art GNU Scientific Library [24], our generators are (i) more entropy-efficient, consuming 2.6x–142x fewer random bits per output variate; (ii) more representative of the ideal distribution range, covering up to 10^{35}x wider intervals; and (iii) more automated and amenable to program analysis, having known output distributions. The results also show that the extended-accuracy methods in §6 incur negligible overhead in entropy and runtime over the original versions in §5.

2 Overview

2.1 Mathematical Representations of Probability Distributions

Let λ denote the Lebesgue measure and $\mathcal{B}(\mathbb{R})$ the Borel sigma-algebra. Every probability measure μ over $\mathbb{R}$ (e.g., Gaussian, Gamma, Poisson, etc.) has multiple equivalent mathematical representations:

PM	Probability Measure	$\mu : \mathcal{B}(\mathbb{R}) \to [0,1]$	(2.1)
RV	Random Variable	$X : ([0,1], \mathcal{B}([0,1])) \to (\mathbb{R}, \mathcal{B}(\mathbb{R}))$	(2.2)
CDF	Cumulative Distribution Function	$F(x) \coloneqq \mu((-\infty, x]) =: \Pr(X \leq x), x \in \mathbb{R}$	(2.3)
SF	Survival Function	$S(x) \coloneqq \Pr(X > x) = 1 - F(x), x \in \mathbb{R}$	(2.4)
QF	Quantile Function	$Q(u) \coloneqq \inf\{x \in \mathbb{R} \mid u \leq F(x)\}, u \in [0,1].$	(2.5)

Figure 2a shows the correspondences among (2.1)–(2.5) under the Real-RAM model of computation. There is a bijection between the spaces of PM, CDF, SF, QF; and a surjection between these spaces and the space RV. We denote these correspondences as $\mu \sim \{F_\mu, S_\mu, Q_\mu, X_\mu\}$, $F \sim \{\mu_F, S_F, Q_F, X_F\}$, $S \sim \{\mu_S, F_S, Q_S, X_S\}$, $Q \sim \{\mu_Q, F_Q, S_Q, X_Q\}$, $X \sim \{\mu_X, F_X, S_X, Q_X\}$. The RV is the only non-unique representation—infinitely many random variables (understood as measurable mappings from the underlying probability space $[0,1]$ into $\mathbb{R}$) can have the same distribution, i.e., $X \neq X'$ but $\mu_X(A) \coloneqq \lambda(X^{-1}(A)) = \lambda(X'^{-1}(A)) =: \mu_{X'}(A)$ for all $A \in \mathcal{B}(\mathbb{R})$. Random variables $X \neq X'$ that disagree on positive measure sets suggest different random variate generation strategies for μ.

Example 2.1. The following random variables all have a standard Gaussian distribution over $\mathbb{R}$:

$$X_1(\omega) = \inf\left\{x \in \mathbb{R} \mid \omega \leq \int_{-\infty}^{x} \frac{e^{-\frac{t^2}{2}}}{\sqrt{2\pi}} \mathrm{d}t\right\}, \quad X_2(\omega) = \frac{\sqrt{-2\ln u_1(\omega)}}{1/\cos(2\pi u_2(\omega))}, \quad X_3(\omega) = \lim_{n\to\infty} \frac{\sum_{i=1}^{n} u_i(\omega) - n/2}{\sqrt{n/12}},$$

where $\{u_i(\omega)\}_{i=1}^{\infty}$ denote countably many i.i.d. uniform numbers on $[0,1]$ "split" from ω (Remark 3.2). These functions describe different implementations of Gaussian random variate generators: the inverse transform method X_1, Box-Muller transform X_2, and central limit approximation X_3. Listing 2 (lines 2–5) lists four Gaussian random variate generators from the GNU Scientific Library. «

Lst 1. Implementing an exponential probability distribution in C.

```
1 // Typical random variate generators for the "uniform" distribution over unit interval.
2 double uniform     (){return ((double)rand()) / (RAND_MAX + 1);}           // U ~ Uniform([0,1))
3 double uniform_pos(){double u; do {u=uniform();} while (u==0); return u;} // U ~ Uniform((0,1))
4
5 // Cumulative distribution (CDF), survival (SF), & quantile (QF) function implementations.
6 double exp_cdf (double x, double s) {return x <= 0 ? 0 : -expm1(-x/s);} // F(x;s) = 1 - e^{-x/s}
7 double exp_sf  (double x, double s) {return x <= 0 ? 1 : exp(-x/s);}    // S(x;s) = e^{-x/s}
8 double exp_qf  (double u, double s) {return -s * log1p(-u);}           // Q(u;s) = -s ln(1 - u)
9
10 // TRADITIONAL APPROACH: Implement ad-hoc random variate generators.
11 double exp_generate     (double s) {return exp_qf(uniform(), s);}      // Q(U;s)
12 double exp_generate_alt (double s) {return -s * log(uniform_pos());}   // Q(1 - U;s)
13
14 // THIS WORK: Exact random variate generators given a formal CDF and/or SF specification.
15 GENERATE_FROM_CDF (exp_cdf, 1.0);
16 GENERATE_FROM_SF  (exp_sf , 1.0);
17 GENERATE_FROM_DDF (exp_cdf, exp_sf, 1.0);
```

2.2 Traditional Software Implementations of Probability Distributions

This section illustrates problems with the approach in traditional random variate generation libraries (e.g., [24, 33, 48, 60, 64]) that provide interfaces for (2.1)–(2.5) as shown schematically in Fig. 2b. We present a small[1] but realistic example for the Exponential(s) distribution (Listing 1) to highlight some issues with finite-precision numerical implementations of idealized Real-RAM algorithms.[2]

Problems with Uniform Source. Lines 2–3 of Listing 1 show common [50] numerical approximations of a uniform variable in $[0, 1]$, which serves as the primitive source of randomness for random variate generation (cf. Fig. 1 and (2.2)). The `uniform` function calls `rand`, which returns an integer between 0 and `RAND_MAX`, and divides the result by `RAND_MAX + 1` to give a float in $[0, 1)$; `uniform_pos` returns a float in $(0, 1)$ by rejection sampling. Issues with generators of this sort are well-documented in the literature: Goualard [28, 29] and Lemire [38] explore these drawbacks in detail. For example, while including or omitting endpoints $\{0, 1\}$ is of no consequence in Real RAM, these values have nonzero probability when using finite precision, causing many downstream errors depending on the implementation.[3] Moreover, typical implementations of `uniform` return a paltry 10^{-7}%–7% of all representable floating-point numbers in $[0, 1]$ (Prop. B.1), which further limits the accuracy of random variate generator algorithms that invoke these functions.[4]

Inconsistencies in the CDF, SF, QF. Lines 6–8 of Listing 1 show numerically stable implementations of (2.3)–(2.5). Let us explore some inconsistencies among these representations. First consider `exp_cdf` and `exp_sf`. These functions formally define distributions over floats (Remark 5.11), but they have entirely different properties. When `s` = 1, the former's range (i.e., support) contains floats within $[7.01 \times 10^{-46}, 17.33]$ and the latter $[2.98 \times 10^{-8}, 103.97]$. Now consider `exp_cdf` and `exp_qf`. If `u` is the float immediately before 1, then the return value of `exp_qf(u,1)` is 16.635 whereas the exact u-quantile of `exp_cdf(·,1)` is 16.230, yielding a large absolute error exceeding 0.405. Relative errors are also large: if `u` is the float immediately after 0, then the return value of `exp_qf(u,1)` is *twice* larger than the exact u-quantile of `exp_cdf(·,1)`. Many similar issues can be surfaced.

Inconsistencies in the Generators. Lines 11–12 of Listing 1 show two exponential random variate generators, $Q(U)$ and $Q(1 - U)$, based on the inverse-transform method (Prop. 3.4). However, `exp_generate` is not consistent with the distribution specified by `exp_cdf` because it calls into

[1] Library implementations of random variate generators and associated functions can span 100s of lines of code, e.g., [59, 61].
[2] A cursory inspection of three widely used Python libraries (NumPy [33], SciPy [64], PyTorch [48]) surfaced ∼90 user-reported issues in the random variate generation algorithms, almost all related to numerical error, shown in Appendix F.
[3] This specific issue arises in the PyTorch exponential generator, see https://github.com/pytorch/pytorch/issues/22557
[4] An extensive discussion of this challenge among developers is found in https://github.com/pytorch/pytorch/issues/16706

Lst 2. Using high-quality cumulative distribution and survival function implementations from the GNU Scientific Library (GSL) to automatically generate "Gaussian" variates (only function signatures are shown).

```
// GSL: Existing random variate generators for Gaussian distribution (renamed for clarity).
double gsl_ran_gaussian_inverse_cdf  (const gsl_rng *r, double sigma);
double gsl_ran_gaussian_box_muller   (const gsl_rng *r, double sigma);
double gsl_ran_gaussian_ziggurat     (const gsl_rng *r, double sigma);
double gsl_ran_gaussian_ratio_method (const gsl_rng *r, double sigma);

// GSL: Numerical implementations of various distribution functions.
double gsl_cdf_gaussian_P    (double x, double sigma); // Cumulative Dist. Function (CDF)
double gsl_cdf_gaussian_Q    (double x, double sigma); // Survival Function (SF)
double gsl_cdf_gaussian_P_inv(double u, double sigma); // Quantile Function (QF)

// THIS WORK: Exact random variate generators given a numerical CDF and/or SF specification.
GENERATE_FROM_CDF (gsl_cdf_gaussian_P, 5.0);
GENERATE_FROM_SF  (gsl_cdf_gaussian_Q, 5.0);
GENERATE_FROM_DDF (gsl_cdf_gaussian_P, gsl_cdf_gaussian_Q, 5.0);
```

`exp_qf`, which is itself not consistent with `exp_cdf`. Supposing that `RAND_MAX` $= 2^{32}-1$, the output of `exp_generate` lies in the range $[0, 22.18]$ when `s` $= 1$, which is again different from the ranges of both `exp_cdf` and `exp_sf`. Further, `exp_generate` is also not consistent with the distribution specified by `exp_qf` even though it directly invokes it, because `uniform` does not cover all floats in $[0, 1]$. The `exp_generate_alt` function has similar inconsistencies, with the additional caveat that implementing $Q(1-U)$ requires using `uniform_pos` instead of `uniform`, because `log(0.)` may return `-inf`, `nan`, or even a domain error, depending on the language (cf. Footnote 3).

2.3 This Work: Exact Random Variate Generators from Formal Specifications

Lines 15–17 of Listing 1 show the approach to random variate generation introduced in this work, which is based on coherent implementations of (2.1)–(2.5). The `GENERATE_FROM_CDF` expression takes the name of any numerical CDF implementation (i.e., `exp_cdf`) and values of distributional parameters (i.e., `s` is `1.0`), and returns a random floating-point number `x` *precisely* with cumulative probability `cdf(x)`. An alternate generator is `GENERATE_FROM_SF`, which returns `x` with cumulative probability $1 -_{\mathbb{R}}$ `exp_sf(x)`, where the subtraction is exact (i.e., no floating-point rounding error). These generators reflect the fact that `exp_cdf` and `exp_sf` define different distributions. We also have an extended-accuracy generator, `GENERATE_FROM_DDF`, which *combines* the `exp_cdf` and `exp_sf` specifications into a single coherent generator. Recall that, for `s` $= 1$, the former has range $[7.01 \times 10^{-46}, 17.33]$ and the latter $[2.98 \times 10^{-8}, 103.97]$, so neither is globally more precise than the other. The output of this generator agrees with `exp_cdf` below its median, where the CDF is more precise, and agrees with `exp_sf` above its median, where the SF is more precise. Its largest value (103.97) is 4.7x higher than that of `exp_generate` (22.18), giving higher coverage of $(0, \infty)$.

These generators do not use ad-hoc floating-point approximations of random uniform reals (e.g., `uniform` or `uniform_pos` in lines 2 and 3 of Listing 1) as the primitive unit of entropy. Instead, they operate directly over individual random bits from streams such as `rand()` (which provides pseudorandom bits) or `/dev/urandom` (which provides cryptographically secure random bits). For example, using our framework, implementing a `uniform` that guarantees 100% coverage of all floats in $(0, 1]$ or $[0, 1)$ (cf. Footnote 4; Table 3 in Appendix B) is just a matter of specifying the CDF:

```
// Uniform over all doubles in (0,1]
double cdf_uniform_round_up(double x) {
  if (isnan(x)) { return 1 }
  return min(max(x, 0), 1);
  }
GENERATE_FROM_CDF(cdf_uniform_round_up);
```

```
// Uniform over all doubles in [0,1)
double cdf_uniform_round_dn(double x) {
  double z = nextafter(x, INFINITY);
  return cdf_uniform_round_up(z);
  }
GENERATE_FROM_CDF(cdf_uniform_round_dn);
```

These functions specify the exact distributions of a random variate obtained by correctly rounding an infinitely precise real $U \sim \text{Uniform}([0, 1])$ to the next and previous IEEE-754 double-precision float,

respectively. While highly specialized random variate generation algorithms for these challenging types of uniform distributions over floats exist in the literature [8, 15, 29, 30, 66], using our approach, exact generators are automatically obtained by specifying a desired CDF, which is often more straightforward than implementing the generator and extends directly to nonuniform distributions.

The generators described in this work are also *entropy-optimal* (Theorem 4.6), i.e., they consume the information-theoretically minimal number of random bits from the entropy source to generate an output. They incur no additional floating-point error with respect to the CDF or SF implementation, and avoid arbitrary-precision arithmetic (Theorem 5.16). It is also straightforward to compute exact quantiles (Algorithm E9) of these generators without using approximate implementations (e.g., `exp_qf` on line 8 of Listing 1) that have no theoretical relationship to the implemented generator.

Listing 2 shows how our random variate generation algorithms (lines 13–15) interoperate with existing software such as the GNU Scientific Library (GSL) by reusing high-quality CDF and/or SF implementations (lines 8–9). The built-in GSL Gaussian generators (lines 2–5) often have complex implementations spanning hundreds of lines of code, and each specify different output distributions which are all intractable to estimate. Indeed, any GSL random variate generator that makes just two (or more) calls to `uniform` is already intractable to analyze. In contrast, our generators (lines 13–15) have known output distributions that match the specified CDF or SF. They are also automatically synthesized from these specifications, requiring no additional implementation code. The view taken by our approach is to first develop high-quality CDF and/or SF implementations that provide transparent formal specifications of the desired distribution the random variates should follow. This specification can be carefully debugged for numerical problems or other errors (e.g., those in Appendix F), after which we automatically synthesize exact generators that match the specification.

3 Preliminaries

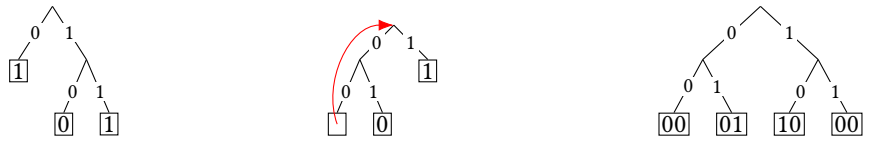
Definition 3.1. A *random variable* $X : [0,1] \to \mathbb{R}$ is a Borel measurable map from the unit interval into the reals. We may equivalently write $X : \{0,1\}^{\mathbb{N}} \to \mathbb{R}$, where $X(u_1 u_2 \ldots) \coloneqq X\left(\sum_{i=1}^{\infty} u_i 2^{-i}\right)$. The *distribution* of a random variable X is the probability measure μ_X over $(\mathbb{R}, \mathcal{B}(\mathbb{R}))$ such that $\mu_X(A) \coloneqq \lambda(X^{-1}(A)) =: \Pr(X \in A)$, where $A \in \mathcal{B}(\mathbb{R})$ is an event and λ the Lebesgue measure. «

Remark 3.2. Defining the domain of a random variable to be $[0,1]$ (instead of say $\cup_{n=0}^{\infty}[0,1]^n$) is made without loss of generality. A single uniformly distributed real $\omega \coloneqq (0.\omega_1\omega_2\ldots)_2 \in [0,1]$ can be "split" into a countably infinite number of i.i.d. uniform numbers $(u_1(\omega), u_2(\omega), \ldots)$ where $u_n(\omega) \coloneqq (0.\omega_{p_n}\omega_{p_n^2}\omega_{p_n^3}\ldots)_2$ for $n \geq 1$, with $p_1 < p_2 < p_3 < \ldots$ an enumeration of the primes. «

Definition 3.3. A *cumulative distribution function* (CDF) $F : \mathbb{R} \to [0,1]$ is a right-continuous monotone mapping such that $\lim_{x\to-\infty} F(x) = 0$ and $\lim_{x\to\infty} F(x) = 1$. A *survival function* (SF) $S : \mathbb{R} \to [0,1]$ is a mapping such that $S(x) = 1 - F(x)$ for some CDF F. A *quantile function* (QF) $Q : [0,1] \to \mathbb{R}$ is a mapping such that $Q(u) = \inf\{x \in \mathbb{R} \mid u \leq F(x)\}$ for some CDF F. «

Proposition 3.4 (Billingsley [5; Theorems 12.4, 14.1]). *Let $([0,1], \mathcal{B}([0,1]), \lambda)$ be the standard probability space. Each random variable $X : [0,1] \to \mathbb{R}$ has a unique CDF $F_X \coloneqq \Pr(X \leq x) = \lambda(X^{-1}(-\infty, x])$. Each CDF $F : \mathbb{R} \to [0,1]$ is in 1-1 correspondence with a unique probability measure μ_F on $(\mathbb{R}, \mathcal{B}(\mathbb{R}))$. There exist infinitely many random variables X_F on $([0,1], \mathcal{B}([0,1]), \lambda)$ that are identically distributed as μ_F, e.g., $X_F(\omega) = Q_F(U(\omega))$ where $U \sim \text{Uniform}([0,1])$.* «

For a probability measure μ over $\mathbb{R}$, the CDF representation F_μ is ideal for *specification*: it concisely describes μ in terms of a univariate real function. On the other hand, a random variable representation X_μ is ideal for *generation*: it describes a procedure for producing μ-distributed random variates. We next define random-variate generators, which are computable, finite-resource analogues of random variables that do not require infinite memory or computation over the reals.



$X : \{0,1\}^* \rightharpoonup \{0,1\}$ $\quad X : \{0,1\}^* \rightharpoonup \{0,1\}$ $\quad X : \{0,1\}^* \rightharpoonup \{0,1\}^2$

$\{0 \mapsto 1, 10 \mapsto 0, 11 \mapsto 1\}$ $\quad \{1 \mapsto 1, 01 \mapsto 0, 00b \mapsto X(b)\}$ $\quad \{00 \mapsto 00, 01 \mapsto 01, 10 \mapsto 10, 11 \mapsto 00\}$

Fig. 3. Three random variate generators represented as discrete distribution generating (DDG) trees. The 0/1 labels along the edges are omitted from the tree diagrams going forward.

Definition 3.5. A *random variate generator* $X : \{0,1\}^* \rightharpoonup \{0,1\}^n$ is a partial map from the set of all finite-length binary strings to n-bit binary strings such that the following two conditions hold:

$$\underline{\text{Prefix-Free}}: u \in \text{dom}(X) \implies u\{0,1\}^+ \notin \text{dom}(X); \quad \underline{\text{Exhaustive}}: \textstyle\sum_{u \in \text{dom}(X)} 2^{-|u|} = 1. \ \text{«} \quad (3.1)$$

Proposition 3.6. *A random variate generator X describes a random variable $X_\gamma : [0,1] \to \mathbb{R}$ over at most 2^n values, where $\Pr(X_\gamma = x) = \sum_{u \in \text{dom}(X)} 2^{-|u|} \mathbf{1}[\gamma(X(u)) = x]$ for any $\gamma : \{0,1\}^n \to \mathbb{R}$.* «

Def. 3.5 corresponds to the "discrete distribution generating" (DDG) trees from Knuth and Yao [37; Section 2]. Any random variate generator can be drawn as a tree where leaves have labels in $\{0,1\}^n$ and branches correspond to the decision on the next 0 or 1 input bit (Fig. 3).

Definition 3.7. The *entropy cost* of a random variate generator X is a random variable C over $\mathbb{N}$ measuring the number of bits consumed, with $\Pr(C = c) = \sum_{u \in \text{dom}(X)} \mathbf{1}[|u| = c] 2^{-c}$ for $c \geq 0$. «

Definition 3.8. A *concise binary expansion* of a real number x is a binary representation that does not end in an infinite string of 1s. Binary expansions hereon are always concise. «

Theorem 3.9 (Knuth and Yao [37]). *Let $p := \{\ell_1 \mapsto p_1, \ldots, \ell_m \mapsto p_m\}$ be a discrete probability distribution over $m \geq 1$ outcomes $\ell_1, \ldots, \ell_m$. Write the binary expansions as $p_i = (p_{i0}.p_{i1}p_{i2}\ldots)_2$ for $i = 1, \ldots, m$. A random variate generator X for p has minimal expected entropy cost $\mathbb{E}[C]$ (i.e., it is "entropy-optimal") if and only if its DDG tree contains exactly p_{ij} leaf nodes labeled ℓ_i at depth $j \geq 0$. Further, $H(p) \leq \mathbb{E}[C] < H(p) + 2$, where $H(p) := \sum_{i=1}^m -p_i \log_2(p_i)$ is the Shannon entropy of p.* «

Example 3.10. In Fig. 3, the first two DDG trees show entropy-optimal generators for $p = \{0 \mapsto (0.01)_2, 1 \mapsto (0.11)_2\}$ and $p = \{0 \mapsto (0.\overline{01})_2, 1 \mapsto (0.\overline{10})_2\}$. The third DDG tree is an entropy-suboptimal generator for $p = \{00 \mapsto (0.10)_2, 01 \mapsto (0.01)_2, 10 \mapsto (0.01)_2\}$. «

4 Exact Random Variate Generators for Binary-Coded Probability Distributions

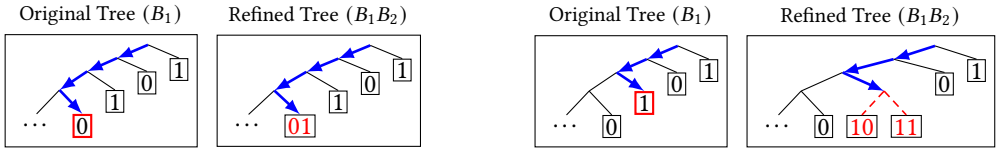
Since the size of a DDG tree is lower bounded by the number m of discrete outcomes, explicitly constructing entropy-optimal DDG trees using Theorem 3.9 is computationally intractable whenever m is combinatorially large. For example, a discrete distribution with full support over all IEEE-754 double-precision floats has $m = 2^{64}$ outcomes. To address this challenge, we introduce *binary-coded probability distributions*, which are a universal mathematical representation for lazily describing any computable probability measure over the reals. This powerful abstraction lets us develop entropy-optimal random variate generation algorithms that avoid the combinatorial explosion in the DDG tree size. The "idealized" algorithms in this section will be specialized in §5 and §6 to obtain efficient software implementations of random variate generators on finite-precision computers.

Definition 4.1. A *binary-coded probability distribution* is a map $p : \{0,1\}^* \to [0,1]$ such that $p(\varepsilon) = 1$ and $p(b_1 \ldots b_j) = p(b_1 \ldots b_j 0) + p(b_1 \ldots b_j 1)$ for all $j \geq 1$ and $b_1, \ldots, b_j \in \{0,1\}$. «

A binary-coded probability distribution p defines a family of discrete probability distributions

$$p_n := \{b \mapsto p(b) \mid b \in \{0,1\}^n\} \quad (4.1)$$

over $\{0,1\}^n$ ($n \geq 0$). For example, $p(00) = 0.5; p(01) = 0.2; p(10) = 0.1; p(11) = 0.2$ defines the distributions $p_1 := \{0 \mapsto 0.7, 1 \mapsto 0.3\}$, $p_2 := \{00 \mapsto 0.5, 01 \mapsto 0.2, 10 \mapsto 0.1, 11 \mapsto 0.2\}$.

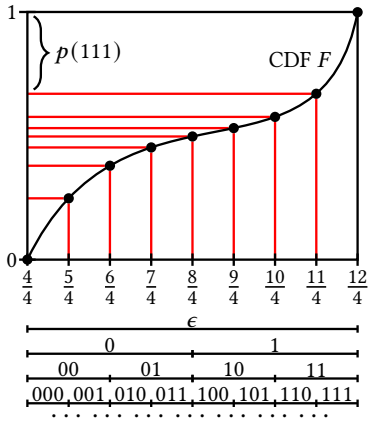


(a) Refining a node into a leaf. After refinement, 01 is returned using zero additional flips.

(b) Refining a node into a subtree. After refinement, 10 or 11 is returned using one additional flip.

Fig. 4. Dynamically refining the leaves of an optimal DDG tree for lazy random variate generation.

Binary-coded probability distributions also correspond to probability measures over $\{0,1\}^{\mathbb{N}} \equiv \mathbb{R}$ [6; Lemma 7.1.2]. The plot to the right shows this idea for a CDF F over $[1,3]$. Each binary string b defines an interval $[x_1(b), x_2(b)] \subset [1,3]$, where $p(b) = F(x_2(b)) - F(x_1(b))$ is its probability under F, e.g., $p(01) = F(8/4) - F(6/4)$. The same idea holds for unbounded domains, by using binary-coded partitions of $\mathbb{R}$ [37; §3].



4.1 Entropy-Suboptimal Generation

Before considering optimal algorithms, an entropy-suboptimal baseline for generating a random string $(B_1, B_2, \dots) \sim p$ is *conditional bit sampling* [58; §II.B]. This method generates one bit at a time by using the chain rule of probability, i.e., $B_1 \sim \text{Bernoulli}(p(1))$, $B_2|B_1 \sim \text{Bernoulli}(p(B_1 1)/p(B_1))$, etc. However, as generating each B_n requires roughly two random bits in expectation (Appendix A.1), conditional bit sampling is entropy-inefficient, consuming roughly $2n$ more bits than an entropy-optimal sampler for generating a length-n bit string in the worst case (Prop. A.3). A second challenge is the high computational cost of computing the conditional probabilities during generation. The conditional bit sampling baseline is presented and analyzed in Appendix A; and evaluated in §7.

4.2 Entropy-Optimal Generation

Our approach to efficiently sampling from a binary-coded probability distribution p is based on the idea of *lazily refining* a DDG tree. The key idea is as follows: we first generate B_1 using an entropy-optimal DDG tree T_1 for $\{0 \mapsto p(0), 1 \mapsto p(1)\}$, which corresponds to arriving at a leaf node x at T_1. Rather than generated by $B_2|B_1$ as in the chain rule (§4.1), B_2 is instead determined by expanding a subtree under the leaf node x of T_1. This subtree is a fragment of an entropy-optimal DDG tree T_2 for $\{00 \mapsto p(00), 01 \mapsto p(01), 10 \mapsto p(10), 11 \mapsto p(11)\}$. Repeating this process allows us to efficiently explore a single, linear-memory path without building an exponentially large DDG tree T_n over $\{0,1\}^n$. Figure 4 shows two examples of refinement, where diagrams labeled "Original Tree" show an entropy-optimal DDG tree T_1 for $B_1 \sim \text{Bernoulli}(2/3)$. Blue arrowed edges show a random execution path. When halting at a leaf (red) in the Original Tree (i.e., $B_1 = b_1$ is determined), the leaf is *refined* into a new subtree whose leaves are outcomes of B_1B_2 with $B_1 = b_1$. The refined subtree could be a leaf (Fig. 4a) or branch (Fig. 4b) node. While exactly one leaf in T_1 is refined in a given execution, refining *every* leaf of T_1 produces an entropy-optimal tree T_2 for B_1B_2.

Algorithm 1 presents an entropy-optimal generator for a binary-coded probability distribution p that uses refinement to efficiently traverse an infinite-size DDG tree. The algorithm itself recurses infinitely, generating a fresh bit of the random stream $b \sim p$ at each step. At the nth recursive call, the leaf node selected in the optimal tree for p_{n-1} over $\{0,1\}^{n-1}$ is refined into an optimal

Algorithm 1 Optimal Generation

Input: Binary-coded probability distribution $p : \{0,1\}^* \to [0,1]$, cf. Def. 4.1
String $b \in \{0,1\}^*$ generated so far
#Flips $\ell \geq 0$ consumed so far
Output: Random bitstream b drawn from p, i.e., $b_1 \ldots b_n \sim p_n$ $(n \geq 0)$

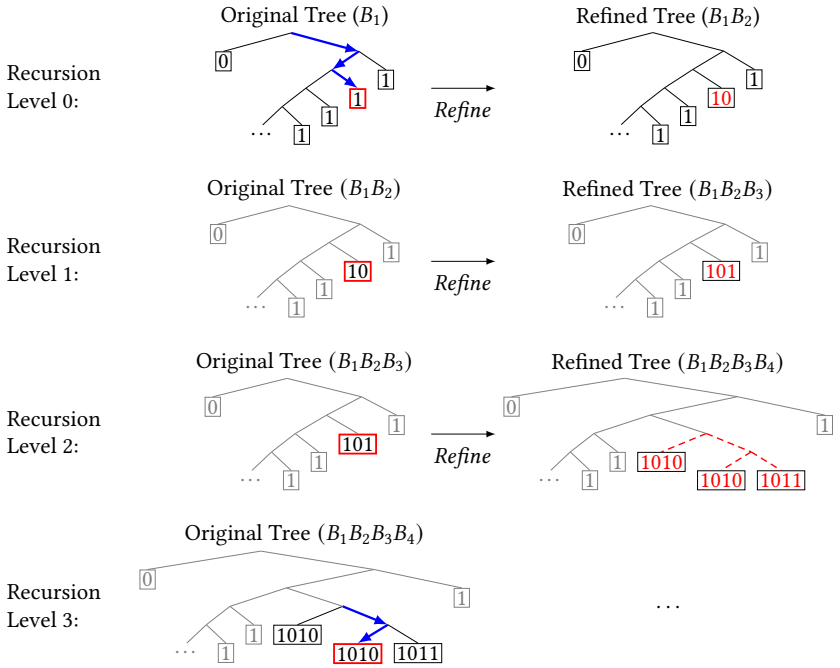
```
1:  function GenOpt(p, b = ε, ℓ = 0)
2:      if [p(b0)]_ℓ = 1 ∧ [p(b1)]_ℓ = 0      ▷ Leaf
3:          return GenOpt(p, b0, ℓ)            ▷ 0
4:      if [p(b0)]_ℓ = 0 ∧ [p(b1)]_ℓ = 1      ▷ Leaf
5:          return GenOpt(p, b1, ℓ)            ▷ 1
6:      while true do                          ▷ Refine Subtree
7:          x ← RandBit(); ℓ ← ℓ + 1
8:          if x = 0 ∧ [p(b0)]_ℓ = 1           ▷ Leaf
9:              return GenOpt(p, b0, ℓ)        ▷ 0
10:         if x = 1 ∧ [p(b1)]_ℓ = 1           ▷ Leaf
11:             return GenOpt(p, b1, ℓ)        ▷ 1
```

(a) A binary-coded probability distribution p unrolled to four bits, giving a discrete distribution over $\{0,1\}^4$.

$$\left\{\begin{array}{llll} 0000 \mapsto \frac{6}{137} & 0001 \mapsto \frac{12}{137} & 0010 \mapsto \frac{13}{137} & 0011 \mapsto \frac{9}{137} \\ 0100 \mapsto \frac{10}{137} & 0101 \mapsto \frac{12}{137} & 0110 \mapsto \frac{6}{137} & 0111 \mapsto \frac{1}{137} \\ 1000 \mapsto \frac{1}{137} & 1001 \mapsto \frac{2}{137} & 1010 \mapsto \frac{13}{137} & 1011 \mapsto \frac{8}{137} \\ 1100 \mapsto \frac{14}{137} & 1101 \mapsto \frac{13}{137} & 1110 \mapsto \frac{7}{137} & 1111 \mapsto \frac{10}{137} \end{array}\right\}$$

(b) A trace of Algorithm 1 on the distribution p from (a).

Recur. Level	Probabilities	RandBit x: 1	0	1	1	0	Output b
0	$p(0) = \frac{69}{137} = \mathbf{0.1}$	**0**	**0**	0	0		
	$p(1) = \frac{68}{137} = \mathbf{0.0}$	**1**	**1**	1	1		1
1	$p(10) = \frac{24}{137} = 0.0$	0	**1**	0	1		0
	$p(11) = \frac{44}{137} = 0.0$	1	**0**	1	0		
2	$p(100) = \frac{3}{137} = 0.0$	0	**0**	0	0		
	$p(101) = \frac{21}{137} = 0.0$	0	**1**	0	0		1
3	$p(1010) = \frac{13}{137} = 0.0$	0	**0**	**1**	**1**		0
	$p(1011) = \frac{8}{137} = 0.0$	0	**0**	**0**	**1**		



(c) DDG trees that Algorithm 1 explores for the example trace in (b).

Fig. 5. Entropy-optimal generation for a binary-coded probability distribution $p : \{0,1\}^* \to [0,1]$. In Algorithm 1, the parameter b (defaulted to the empty string, ε) denotes a string that stores the bits generated so far, and ℓ counts the number of calls to RandBit. The notation $[z]_i$ denotes the ith bit in $z = (z_0.z_1z_2z_3\ldots)_2 \in [0,1]$. In the sample trace (b), a blue bit denotes a RandBit that creates a new recursive call; a yellow bit indicates the selected leaf in the DDG tree; and a pink bit is the label of that leaf, which is appended to b. Gray bits are not visited in this execution, as the algorithm lazily explores a *single path* through the DDG tree.

subtree for p_n. Lines 2–5 occur when the refined subtree is a leaf (cf. Fig. 4a). Lines 6–11 occur when the refined subtree is not a leaf (cf. Fig. 4b). The trees are such that a leaf labeled 0 (resp. 1) is always a left (resp. right) child, which is visited when $x = 0$ (resp. $x = 1$). The algorithm is readily implementable using lazy computation with guarded recursive calls (Listing 3 in Appendix C).

Figure 5b shows an example trace of Algorithm 1 on input p from Fig. 5a. Figure 5c shows how this example trace corresponds to a lazy exploration of an infinite-size optimal DDG tree. Each row shows the original tree to be explored at the start of each recursive call and the refined tree to be explored at the next recursion. For the original trees (Fig. 5c, left column), black edges and nodes show the subtree to be explored at the current recursion level. Inactive paths that were considered previously are shown in gray. Blue edges and red-boxed nodes denote the edges and leaves explored and chosen by Algorithm 1, using the outputs of RandBit from Fig. 5b. For refined trees (Fig. 5c, right column), red edges and red-labeled nodes denote the outcome of refining the red-boxed node in the original tree on the left, which will be explored at the next recursion. Algorithm 1 only ever constructs the blue edges and red-boxed nodes in Fig. 5c: the rest are shown for illustration.

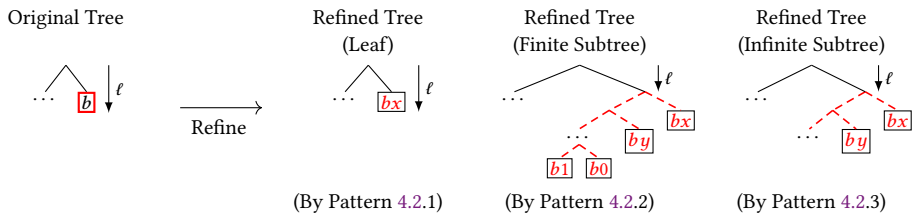
The following results formally justify the correctness and entropy-optimality of Algorithm 1.

Theorem 4.2. *Let $z, x, y \in [0,1]$ satisfy $z = x + y$. Suppose $\ell \geq 0$ is any index such that $z_\ell = 1$ and $z_j = 0$ for all $j > \ell$, where $z = (z_0.z_1z_2\ldots)_2$, $x = (x_0.x_1x_2\ldots)_2$ and $y = (y_0.y_1y_2\ldots)_2$ are concise binary expansions. The binary expansions of x and y match exactly one of the following patterns:*

$$\begin{array}{ccccccccc} & & \ell & & \ell' & & & & \\ + & \begin{bmatrix} x_\ell & \ldots & x_{\ell'} & \ldots \\ y_\ell & \ldots & y_{\ell'} & \ldots \end{bmatrix} & = & \begin{cases} \left[\begin{array}{c|c} 0 & 1 \\ 1 & 0 \end{array}\right] \begin{bmatrix} 0 \\ 0 \end{bmatrix}^\infty & \text{(Pattern 4.2.1)} \\ \left[\begin{array}{c|c} 0 & 1 \\ 0 & 1 \end{array}\right] \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}^* \begin{bmatrix} 1 \\ 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix}^\infty & \text{(Pattern 4.2.2)} \\ \left[\begin{array}{c|c} 0 & 1 \\ 0 & 1 \end{array}\right] \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}^* & \text{(Pattern 4.2.3)} \end{cases} \\ \hline = & z_\ell & \ldots & z_{\ell'} & \ldots & = & 1 & 0 & \ldots & 0 & \ldots \end{array} \tag{4.2}$$

Here, each pattern is written in the style of regular expressions: $[R \mid R']$ denotes either R or R', $[R]^*$ denotes zero or more occurrences of R, and $[R]^\infty$ denotes the infinite occurrences of R. «

Corollary 4.3. *When refining the deepest node in an entropy-optimal DDG tree with label b at a level ℓ, there are three mutually exclusive and collectively exhaustive possibilities (where $x, y \in \{0,1\}$):*



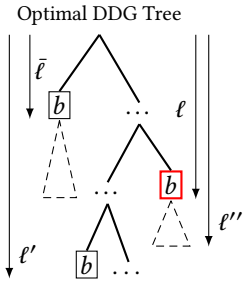
«

Theorem 4.4. *Let $z, x, y \in [0,1]$ satisfy $z = x + y$. Suppose $0 \leq \ell < \ell'$ are two indexes such that $z_\ell = 1, z_{\ell'} = 1$ and $z_j = 0$ for $\ell + 1 \leq j \leq \ell' - 1$, where z_j, x_j and y_j are defined as in Theorem 4.2. The binary expansions of x and y between locations ℓ and ℓ' match exactly one of three possible patterns:*

$$\begin{array}{ccccccccc} & & \ell & & \ell' & & & & \\ + & \begin{bmatrix} x_\ell & \ldots & x_{\ell'} \\ y_\ell & \ldots & y_{\ell'} \end{bmatrix} & = & \begin{cases} \left[\begin{array}{c|c} 0 & 1 \\ 1 & 0 \end{array}\right] \begin{bmatrix} 0 \\ 0 \end{bmatrix}^{\ell'-\ell-1} \left[\begin{array}{c|c|c} 0 & 0 & 1 \\ 0 & 1 & 0 \end{array}\right] & \text{(Pattern 4.3.1)} \\ \left[\begin{array}{c|c} 0 & 1 \\ 0 & 1 \end{array}\right] \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}^{\ell'-\ell-1} \begin{bmatrix} 1 \\ 1 \end{bmatrix} & \text{(Pattern 4.3.2)} \\ \left[\begin{array}{c|c} 0 & 1 \\ 0 & 1 \end{array}\right] \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}^{k_1} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix}^{k_2} \left[\begin{array}{c|c|c} 0 & 0 & 1 \\ 0 & 1 & 0 \end{array}\right] & \text{where } k_1 + k_2 = \ell' - \ell - 2 \\ & & & & & & & & & & & \text{(Pattern 4.3.3)} \end{cases} \\ \hline = & z_\ell & \ldots & z_{\ell'} & = & 1 & 0 & \ldots & 0 & 1 \end{array} \tag{4.3}$$

«

Corollary 4.5. *Consider the process of refining a node labeled b at level ℓ in an entropy-optimal DDG tree, such that there also exists a node labeled b at some level $\ell' > \ell$. There exists an entropy-optimal refinement scheme such that*



- *If the binary expansions satisfy $[p(b0)]_\ell[p(b1)]_\ell \in \{01, 10\}$, then the node b at level ℓ is refined into a leaf node.*
- *If $[p(b0)]_\ell[p(b1)]_\ell \in \{00, 11\}$, then the node b at level ℓ is refined into a subtree that terminates with a pair of nodes labeled $(b0, b1)$ at some level $\ell'' \in [\ell + 1, \ell']$. All levels of this subtree above ℓ'' have precisely one node.*
- *If $[p(b0)]_\ell[p(b1)]_\ell \in \{11\}$, then the corresponding nodes in the DDG tree labeled $b0$ and $b1$ correspond to leaves of the subtree obtained by refining a previous node labeled b at some previous level $\bar{\ell} < \ell$. Therefore, these bits at location ℓ can be ignored when refining the node b at level ℓ.* «

Theorem 4.6. *Let $p : \{0,1\}^* \to [0,1]$ be a binary-coded probability distribution. For each $n \in \mathbb{N}$, Algorithm 1 generates a string $B_1 \ldots B_n \sim p_n$ (stored as a prefix of b) and is entropy-optimal for p_n.* «

Theorem 4.6 states the entropy-optimality—and, a fortiori, the soundness—of Algorithm 1. This result rests on two number theoretic properties (Theorems 4.2 and 4.4) for binary expansions of real numbers. Cors. 4.3 and 4.5 demonstrate what these theorems imply about the structure of refined entropy-optimal DDG trees explored by Algorithm 1, which are used to prove Theorem 4.6. Theorem 4.2 and Cor. 4.3 justify the correctness of Algorithm 1 when the current node is the deepest leaf in the tree with label b. Pattern 4.2.1 corresponds to the early exit in lines 2–5, which do not require any new flips. Patterns 4.2.2 and 4.2.3 correspond to the while-loop, showing there will always be precisely one leaf node at each iteration. Therefore, each iteration exits with probability 1/2, and the while-loop terminates almost surely. Theorem 4.4 and Cor. 4.5 justify the correctness when the current node is not the deepest leaf labeled b. Pattern 4.3.1 corresponds to the early exit in lines 2–5. Patterns 4.3.2 and 4.3.3 correspond to the while-loop, whose number of iterations is bounded by the number of levels in the tree until reaching the next leaf labeled b.

Remark 4.7. Algorithm 1 can be viewed as an optimized version of the original Knuth and Yao method that achieves optimal space-time complexity for refining entropy-optimal DDG trees.

- Knuth and Yao [37; page 384] describe a nondeterministic procedure for refining entropy-optimal DDG trees. The Knuth and Yao method explicitly constructs a full DDG tree at each refinement step. When refining a leaf node x with label b at level ℓ, the method
  (i) performs a preprocessing step that refines *all* the leaf nodes labeled b above ℓ; then
  (ii) expands *all* the possible execution paths starting from the new subtree rooted at x.
  Each such refinement step takes $O(\ell')$ time and creates $O(\ell')$ leaf nodes, where ℓ' is the height of a resulting DDG tree. Therefore, the Knuth and Yao method requires $O(k^2)$ time and $O(k^2)$ space to construct a refined entropy-optimal DDG tree of height k.
- Algorithm 1 is a more efficient method that does not explicitly construct full DDG trees. It
  (i) performs *no* preprocessing (i.e, avoids refining leaf nodes higher up in the tree); and
  (ii) lazily explores only a *single* path down the subtree rooted at x.
  Algorithm 1 requires $O(k)$ time and $O(1)$ space to explore a single path in a refined entropy-optimal DDG tree of height k; achieving optimal space-time complexity. Cors. 4.3 and 4.5 enable this optimality, by identifying a class of refined entropy-optimal DDG trees that have certain "nice" properties which Algorithm 1 exploits for efficient and lazy exploration.

The Knuth and Yao algorithm is nondeterministic: it can construct *any* refined entropy-optimal DDG tree [37; page 385]. In contrast, Algorithm 1 is deterministic: it can explore only *some* of these trees, because not every refined entropy-optimal DDG tree satisfies Cor. 4.5 (cf. Fig. 6). «

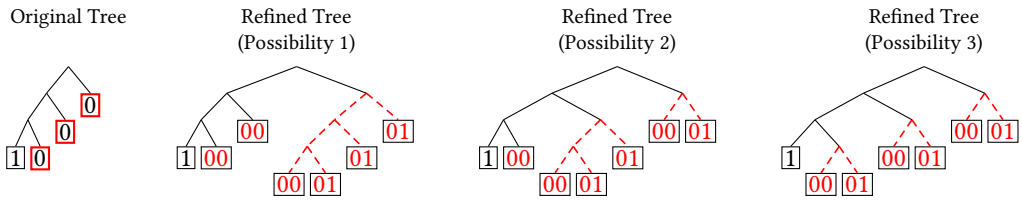


Fig. 6. Three possible DDG tree refinements for outcome 0 in a binary-coded probability distribution with $p(0) = (0.111)_2, p(1) = (0.001)_2$, and $p(00) = p(01) = (0.0111)_2$. The nondeterministic method of Knuth and Yao [37] can explore any of these possibilities, with $O(k^2)$ space and time complexity for building a depth-k refined tree. The deterministic method in Algorithm 1, based on Cors. 4.3 and 4.5, deterministically refines a path in a tree whose structure follows that of Possibility 3, with $O(k)$ time and $O(1)$ space complexity.

5 Exact Random Variate Generators for Numerical Cumulative Distribution Functions

This section describes an implementation of Algorithm 1 for exact random variate generation using finite-precision computation. The main algorithm, presented in §5.3, rests on two connections. The first (in §5.1) is between binary strings and a novel unifying abstraction for finite-precision binary number formats (Def. 5.2). The second (in §5.2) is between binary-coded probability distributions (Def. 4.1) and numerical implementations of cumulative distribution functions (Def. 5.10).

5.1 Finite-Precision Binary Number Formats

Definition 5.1. The set of *extended reals* is defined by $\overline{\mathbb{R}} := \mathbb{R} \cup \{-\infty, +\infty, \bot\}$, where $\bot$ denotes some "special" value (e.g., NaN in floating-point formats). A strict linear order $<_{\overline{\mathbb{R}}}$ over $\overline{\mathbb{R}}$ is given by $x <_{\overline{\mathbb{R}}} x'$ and $-\infty <_{\overline{\mathbb{R}}} x <_{\overline{\mathbb{R}}} +\infty <_{\overline{\mathbb{R}}} \bot$ for all $x, x' \in \mathbb{R}$ with $x <_{\mathbb{R}} x'$; i.e., $\bot$ is the largest value. A weak linear order $\leq_{\overline{\mathbb{R}}}$ over $\overline{\mathbb{R}}$ is defined as usual: $x \leq_{\overline{\mathbb{R}}} x'$ if and only if $x = x'$ or $x <_{\overline{\mathbb{R}}} x'$. «

Definition 5.2. A *binary number format* $\mathbb{B} := (n, \gamma_{\mathbb{B}}, \phi_{\mathbb{B}})$ is a 3-tuple:

- $n \geq 1$ is an integer indicating the number of binary digits in each bit string $b \in \{0,1\}^n$;
- $\gamma_{\mathbb{B}} : \{0,1\}^n \to \overline{\mathbb{R}}_{\mathbb{B}}$ is a mapping from n-bit strings onto a subset $\overline{\mathbb{R}}_{\mathbb{B}} \subset \overline{\mathbb{R}}$ of computable reals;
- $\phi_{\mathbb{B}} : \{0,1\}^n \to \{0,1\}^n$ is a bijection such that $b <_{\text{dict}} b'$ implies $\gamma_{\mathbb{B}}(\phi_{\mathbb{B}}(b)) \leq_{\overline{\mathbb{R}}} \gamma_{\mathbb{B}}(\phi_{\mathbb{B}}(b'))$, where $<_{\text{dict}}$ denotes the dictionary (i.e., lexicographic) ordering on $\{0,1\}^n$. «

With slight abuse of notation, the set $\overline{\mathbb{R}}_{\mathbb{B}}$ of reals in Def. 5.2 is sometimes also denoted by $\mathbb{B}$.

Remark 5.3. For a binary number format $\mathbb{B}$, the bijection $\phi_{\mathbb{B}}$ defines an ordering $<_{\mathbb{B}}$ on $\{0,1\}^n$. In particular, equipping the domain of $\phi_{\mathbb{B}}$ with the dictionary ordering $<_{\text{dict}}$ gives a strict linear order $<_{\mathbb{B}}$ over $\{0,1\}^n$ and a weak linear order over $\overline{\mathbb{R}}_{\mathbb{B}} \subset \overline{\mathbb{R}}$:

$$\phi_{\mathbb{B}}(0^n) <_{\mathbb{B}} \phi_{\mathbb{B}}(0^{n-1}1) <_{\mathbb{B}} \cdots <_{\mathbb{B}} \phi_{\mathbb{B}}(1^{n-1}0) <_{\mathbb{B}} \phi_{\mathbb{B}}(1^n), \tag{5.1}$$

$$\gamma_{\mathbb{B}}(\phi_{\mathbb{B}}(0^n)) \leq_{\overline{\mathbb{R}}} \gamma_{\mathbb{B}}(\phi_{\mathbb{B}}(0^{n-1}1)) \leq_{\overline{\mathbb{R}}} \cdots \leq_{\overline{\mathbb{R}}} \gamma_{\mathbb{B}}(\phi_{\mathbb{B}}(1^{n-1}0)) \leq_{\overline{\mathbb{R}}} \gamma_{\mathbb{B}}(\phi_{\mathbb{B}}(1^n)). \tag{5.2}$$

The predecessor and successor of a non-extremal value $b \in \{0,1\}^n$ under ordering (5.1) are denoted by $\text{pred}_{\mathbb{B}}(b)$ and $\text{succ}_{\mathbb{B}}(b)$. Similarly, $\text{pred}_{\mathbb{B}}(x)$ and $\text{succ}_{\mathbb{B}}(x)$ for $x \in \overline{\mathbb{R}}_{\mathbb{B}}$ are used for (5.2). «

Example 5.4 (Integer Formats: Unsigned $\mathbb{U}_n$, Sign-Magnitude $\mathbb{M}_n$, and Two's-Complement $\mathbb{T}_n$).

$\mathbb{U}_n := (n, \gamma_{\mathbb{U}_n}, \phi_{\mathbb{U}_n})$	$\gamma_{\mathbb{U}_n}(b_{n-1} \dots b_0) := \sum_{j=0}^{n-1} 2^j b_j$	$\phi_{\mathbb{U}_n}(b_{n-1} \dots b_0) := b_{n-1} \dots b_0$
$\mathbb{M}_n := (n+1, \gamma_{\mathbb{M}_n}, \phi_{\mathbb{M}_n})$	$\gamma_{\mathbb{M}_n}(sb_{n-1} \dots b_0) := (-1)^s \times \sum_{j=0}^{n-1} 2^j b_j$	$\phi_{\mathbb{M}_n}(0b_1 \dots b_n) := 1\bar{b}_1 \dots \bar{b}_n$ $\phi_{\mathbb{M}_n}(1b_1 \dots b_n) := 0b_1 \dots b_n$
$\mathbb{T}_n := (n+1, \gamma_{\mathbb{T}_n}, \phi_{\mathbb{T}_n})$	$\gamma_{\mathbb{T}_n}(sb_{n-1} \dots b_0) := -s2^n + \sum_{j=0}^{n-1} 2^j b_j$	$\phi_{\mathbb{T}_n}(0b_1 \dots b_n) := 1b_1 \dots b_n$ $\phi_{\mathbb{T}_n}(1b_1 \dots b_n) := 0b_1 \dots b_n$

«

Example 5.5 (Fixed-Point Formats). The unsigned fixed-point format $\mathbb{U}_n^m := (n, \gamma_{\mathbb{U}_n^m}, \phi_{\mathbb{U}_n^m})$ (parameterized by offset $m \in \mathbb{Z}$) has $\gamma_{\mathbb{U}_n^m}(b_{n-1}\dots b_0) := 2^{-m}\gamma_{\mathbb{U}_n}(b_{n-1}\dots b_0)$ and $\phi_{\mathbb{U}_n^m} := \mathrm{id}$. The sign-magnitude $\mathbb{M}_n^m$ and two's-complement $\mathbb{T}_n^m$ fixed-point formats are defined analogously. «

Example 5.6 (Floating-Point Formats [34]). The IEEE-754 floating-point format $\mathbb{F}_m^E := (1+E+m, \gamma_{\mathbb{F}_m^E}, \phi_{\mathbb{F}_m^E})$ is comprised of n-bit strings, where $E \geq 1$ is the number of exponent bits, $m \geq 1$ the number of mantissa bits, and the leading bit is a sign bit. Letting $b_E := 2^{E-1}-1$ be the "exponent bias",

$$\gamma_{\mathbb{F}_m^E}(s0^E f_1\dots f_m) := (-1)^s(0.f_1\dots f_m)_2 \times 2^{1-b_E}, \qquad \gamma_{\mathbb{F}_m^E}(s1^E f_1\dots f_m) := \bot, \tag{5.3}$$

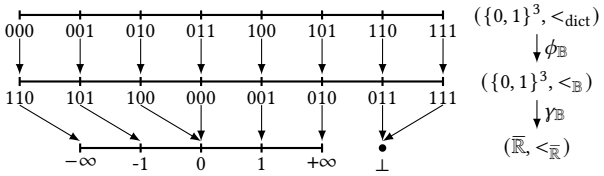
$$\gamma_{\mathbb{F}_m^E}(se_E\dots e_1 f_1\dots f_m) := (-1)^s(1.f_1\dots f_m)_2 \times 2^{(e_E\dots e_1)_2 - b_E}, \qquad \gamma_{\mathbb{F}_m^E}(s1^E0^m) := (-1)^s\infty. \tag{5.4}$$

There are two bit-string representations for $0 \in \overline{\mathbb{R}}$, 00^{E+m} and 10^{E+m}, which are referred to as positive zero and negative zero, respectively. The bijection $\phi_{\mathbb{F}_m^E}$ is similar to that of the sign-magnitude format $\mathbb{M}_n$, with an offset to ensure that all strings mapping to $\bot$ are maximal[5]:

$$\phi_{\mathbb{F}_m^E}(b_0b_1\dots b_{n-1}) := \begin{cases} \phi_{\mathbb{M}_{E+m}}\big((b_0b_1\dots b_{n-1})_2 + (2^m-1)\big) & \text{if } b_0\dots b_{n-1} \leq_{\text{dict}} 11^E0^m \\ b_0b_1\dots b_{n-1} & \text{otherwise.} \end{cases} \tag{5.5}$$

«

Example 5.7. The following diagram is an example format $\mathbb{B}$, corresponding to $\mathbb{F}_1^1$ in Example 5.6.



«

Proposition 5.8. *The ordering $<_{\mathbb{F}_m^E}$ induced by $\phi_{\mathbb{F}_m^E}$ (5.5) guarantees that $\gamma_{\mathbb{F}_m^E} : (\{0,1\}^n, <_{\mathbb{F}_m^E}) \to (\overline{\mathbb{R}}, <_{\overline{\mathbb{R}}})$ is monotonic. That is, for any distinct $b, b' \in \{0,1\}^{1+E+m}$ such that $\gamma_{\mathbb{F}_m^E}(\{b,b'\}) \notin \{\{0\},\{\bot\}\}$, the following are equivalent: $\phi^{-1}_{\mathbb{F}_m^E}(b) <_{\text{dict}} \phi^{-1}_{\mathbb{F}_m^E}(b') \iff b <_{\mathbb{F}_m^E} b' \iff \gamma_{\mathbb{F}_m^E}(b) <_{\overline{\mathbb{R}}} \gamma_{\mathbb{F}_m^E}(b')$.* «

Example 5.9 (Posit Format [49]). The posit format $\mathbb{P}_n := (n, \gamma_{\mathbb{P}_n}, \phi_{\mathbb{P}_n})$ is comprised of n-bit strings ($n \geq 3$). The first bit s is the sign field. The next $k+1$ bits form a variable-length regime field ($1 \leq k \leq n-2$) where $b_1 = \dots = b_k$, $b_{k+1} = \bar{b}_1$. The next two bits e_1e_0 form an exponent field. The last bits $f_1\dots f_m$ form a fraction field. The real mapping has $\gamma_{\mathbb{P}_n}(00^{n-1}) := 0$, $\gamma_{\mathbb{P}_n}(10^{n-1}) := -\infty$, and

$$\gamma_{\mathbb{P}_n}(sb_1\dots b_kb_{k+1}e_1e_0f_1\dots f_m) := \big((1-3s)+(0.f_1\dots f_m)_2\big)2^{(1-2s)\cdot(4(-k(1-b_1)+(k-1)b_1)+(e_1e_0)_2+s)}. \tag{5.6}$$

If the n-bit field is not wide enough to represent some exponent or fraction bits, these bits are zero. The mapping $\phi_{\mathbb{P}_n} := \phi_{\mathbb{T}_n}$ is identical to that of the two's-complement format from Example 5.4. «

5.2 Finite-Precision Cumulative Distribution Functions

Definition 5.10. Let $\mathbb{B} = (n, \gamma_{\mathbb{B}}, \phi_{\mathbb{B}})$ be any binary number format and E, m the parameters of a floating-point format $\mathbb{F}_m^E$. A *finite-precision cumulative distribution function* $F : \{0,1\}^n \to \mathbb{F}_m^E \cap [0,1]$ over $\mathbb{B}$ is a nondecreasing mapping with $F(\phi_{\mathbb{B}}(1^n)) = 1$ and $b <_{\mathbb{B}} b' \implies F(b) \leq F(b')$. A *finite-precision survival function* $S : \{0,1\}^n \to \mathbb{F}_m^E \cap [0,1]$ over $\mathbb{B}$ is a nonincreasing mapping with $S(\phi_{\mathbb{B}}(1^n)) = 0$ and $b <_{\mathbb{B}} b' \implies S(b') \leq S(b)$. «

Lines 6–7 of Listing 1 show examples of a finite-precision CDF and SF, respectively. While Def. 5.10 assumes that F returns IEEE-754 floats, formats such as fixed-points (Example 5.5) and posits (Example 5.9) are also possible. The next remarks state properties of a finite-precision CDF.

[5]The IEEE-754 floating-point standard treats NaN ($\bot$) bit patterns as unordered, whereas (5.5) treats them all identically and as a maximal element to obtain a well-defined CDF. Alternative orderings of these bit patterns are possible.

Remark 5.11. Every finite-precision CDF F defines a discrete distribution $P_F : \{0,1\}^n \to [0,1]$, where $P_F(b) \coloneqq F(b) - F(\mathrm{pred}_{\mathbb{B}}(b))$, with the convention that $F(\mathrm{pred}_{\mathbb{B}}(\phi_{\mathbb{B}}(0^n))) \coloneqq 0$. Recall that directly constructing an entropy-optimal DDG tree for P_F is infeasible if it has $\Theta(2^n)$ leaves. «

Remark 5.12. Every finite-precision CDF $F : \{0,1\}^n \to \mathbb{F}_m^E \cap [0,1]$ over a binary number format $\mathbb{B}$ can be lifted to a CDF $\hat{F} : \overline{\mathbb{R}} \to [0,1]$ over $\overline{\mathbb{R}}$, where $\hat{F}(x) \coloneqq F(\mathrm{rnd}_{\mathbb{B},\downarrow}(x))$ and $\mathrm{rnd}_{\mathbb{B},\downarrow}(x) \coloneqq \max_{<_{\mathbb{B}}}\{b \in \{0,1\}^n \mid \gamma_{\mathbb{B}}(b) \leq_{\overline{\mathbb{R}}} x\}$. Remark 5.3 confirms $\hat{F}$ is monotonic and right-continuous. «

The next result connects binary-coded probability distributions (Def. 4.1) with finite-precision CDFs (Def. 5.10), which enables the finite-precision specialization of Algorithm 1 in §5.3.

Proposition 5.13. *Let $F : \{0,1\}^n \to \mathbb{F}_m^E \cap [0,1]$ be a CDF over the unsigned integer format $\mathbb{U}_n$ and P_F the corresponding discrete distribution from Remark 5.11. The function $p_F : \{0,1\}^* \to [0,1]$ defined below is a binary-coded probability distribution that satisfies $p_F(b) = P_F(b)$ for all $b \in \{0,1\}^n$:*

$$p_F(b) \coloneqq F(b1^{n-|b|}) -_{\mathbb{R}} F((b0^{n-|b|})^-) \qquad (b \in \{0,1\}^{\leq n}), \tag{5.7}$$

$$p_F(bb') \coloneqq p_F(b)\mathbf{1}[b' = 0\ldots0] \qquad (b \in \{0,1\}^n; b' \in \{0,1\}^+), \tag{5.8}$$

where $x^- \coloneqq \mathrm{pred}_{\mathrm{dict}}(x)$ for any $x \in \{0,1\}^n \setminus \{0^n\}$, with the convention that $F((0^n)^-) \coloneqq 0$. «

Proposition 5.14. *Let $F : \{0,1\}^n \to \mathbb{F}_m^E \cap [0,1]$ be a CDF over a number format $\mathbb{B} = (n, \gamma_{\mathbb{B}}, \phi_{\mathbb{B}})$. Then $\widetilde{F} \coloneqq F \circ \phi_{\mathbb{B}}$ is a CDF over the unsigned integer format $\mathbb{U}_n = (n, (\cdot)_2, \mathrm{id})$ from Example 5.4.* «

Corollary 5.15. *Let $F : \{0,1\}^n \to \mathbb{F}_m^E \cap [0,1]$ be a CDF over a number format $\mathbb{B} = (n, \gamma_{\mathbb{B}}, \phi_{\mathbb{B}})$. Then a random variate $X \sim F$ can be generated by first drawing $Z \sim F \circ \phi_{\mathbb{B}}$ and setting $X \leftarrow \phi_{\mathbb{B}}(Z)$.* «

5.3 Finite-Precision Random Variate Generation Algorithms

§5.1 and §5.2 provide all the necessary ingredients for soundly implementing Algorithm 1 using finite-precision computation, as shown in Algorithm 2. The only required parameter is the CDF $F : \{0,1\}^n \to \mathbb{F}_m^E \cap [0,1]$ over a binary number format $\mathbb{B}$. The remaining parameters have default values and are used only by recursive calls, where: b stores the bit string generated so far; ℓ stores the number of calls to RandBit, i.e., the current level in the underlying DDG tree; f_0, f_1 store the subtrahend and minuend in the probability $p_F(b) = F(\phi_{\mathbb{B}}(b1^{n-|b|})) -_{\mathbb{R}} F(\phi_{\mathbb{B}}((b0^{n-|b|})^-))$ (5.7).

We now describe Algorithm 2 in detail. Lines 2–3 show the base case, where the n-bit string b (in the unsigned integer format) has been generated from $F \circ \phi_{\mathbb{B}}$, and then projected to the target format $\mathbb{B}$ using the map $\phi_{\mathbb{B}}$, based on Prop. 5.14 and Cor. 5.15. Line 5 computes the cumulative probability f_2 of the "midpoint" string b', which splits the interval defined by the the current string b in half (based on (5.7)), i.e., $f_2 -_{\mathbb{R}} f_0$ (resp. $f_1 -_{\mathbb{R}} f_2$) is the probability that the next bit is 0 (resp. 1). Lines 6–9 are optimizations for when one of these probabilities is zero, so the refined subtree must be a leaf. Lines 12–18 occur when the refined subtree is a leaf (mirroring lines 2–5 of Algorithm 1). Lines 19–26 occur when the refined subtree is a not leaf (mirroring lines 6–11 of Algorithm 1).

In lines 10 and 11, a main implementation challenge is extracting the binary expansions of $f_2 -_{\mathbb{R}} f_0$ and $f_1 -_{\mathbb{R}} f_2$ in such a way that avoids expensive arbitrary-precision arithmetic on the one hand and rounding errors from a direct floating-point subtraction on the other hand. Whereas the 2Sum/Fast2Sum [44] methods can be used to compute the exact round-off error from a floating-point subtraction, they are not applicable here, as the goal is to extract the individual bits of the difference. Algorithms 3 and 4 provide an efficient solution using fast integer arithmetic. For $x, x' \in \mathbb{F}_m^E \cap [0,1]$, Algorithm 3 computes a compact representation of $x -_{\mathbb{R}} x'$ as described in (5.9) of Theorem 5.16. Lines 2–8 extract the exponent and significand of x and x' (i.e., $\hat{e}, f, \hat{e}', f'$), and lines 9–10 decompose f' into two parts to align the binary expansions of x and x'. Lines 11–17 then compute a tuple of integers and booleans that encode $x -_{\mathbb{R}} x'$. For this tuple and $\ell \geq 1$, Algorithm 4 outputs the ℓ-th bit of $x -_{\mathbb{R}} x'$ based on the encoding scheme shown in (5.9). Theorem 5.16 establishes the correctness of Algorithms 3 and 4, and the guarantee that all intermediate values fit in a single machine word.

Algorithm 2 Entropy-Optimal Generation

Input: CDF $F : \{0,1\}^n \to \mathbb{F}_m^E \cap [0,1]$
over number format $\mathbb{B} = (n, \gamma_{\mathbb{B}}, \phi_{\mathbb{B}})$
String $b \in \{0,1\}^{\leq n}$; #Flips $\ell \geq 0$;
Floats $f_0, f_1 \in \mathbb{F}_m^E \cap [0,1]$
Output: Exact random variate $X \sim F$

function Opt(F, b=ε, ℓ=0, f_0=0, f_1=1)
  if $|b| = n$ ▷ *Base Case*
    return $\phi_{\mathbb{B}}(b)$ ▷ *String in Format* $\mathbb{B}$
  $b' \leftarrow b01^{n-|b|-1}$
  $f_2 \leftarrow F(\phi_{\mathbb{B}}(b'))$
  if $f_2 = f_1$ ▷ *Leaf*
    return Opt($F, b0, \ell, f_0, f_2$) ▷ *0*
  if $f_2 = f_0$ ▷ *Leaf*
    return Opt($F, b1, \ell, f_2, f_1$) ▷ *1*
  $\beta_0 \leftarrow$ ExtractBitPreproc1(f_2, f_0)
  $\beta_1 \leftarrow$ ExtractBitPreproc1(f_1, f_2)
  if $\ell > 0$
    $a_0 \leftarrow$ ExtractBit(β_0, ℓ) ▷ $[f_2 -_{\mathbb{R}} f_0]_\ell$
    $a_1 \leftarrow$ ExtractBit(β_1, ℓ) ▷ $[f_1 -_{\mathbb{R}} f_2]_\ell$
    if $a_0 = 1 \wedge a_1 = 0$ ▷ *Leaf*
      return Opt($F, b0, \ell, f_0, f_2$) ▷ *0*
    if $a_0 = 0 \wedge a_1 = 1$ ▷ *Leaf*
      return Opt($F, b1, \ell, f_2, f_1$) ▷ *1*
  while true do ▷ *Refine Subtree*
    $x \leftarrow$ RandBit(); $\ell \leftarrow \ell + 1$
    $a_0 \leftarrow$ ExtractBit(β_0, ℓ) ▷ $[f_2 -_{\mathbb{R}} f_0]_\ell$
    $a_1 \leftarrow$ ExtractBit(β_1, ℓ) ▷ $[f_1 -_{\mathbb{R}} f_2]_\ell$
    if $x = 0 \wedge a_0 = 1$ ▷ *Leaf*
      return Opt($F, b0, \ell, f_0, f_2$) ▷ *0*
    if $x = 1 \wedge a_1 = 1$ ▷ *Leaf*
      return Opt($F, b1, \ell, f_2, f_1$) ▷ *1*

Algorithm 3 Preprocessing for ExtractBit

Input: $x, x' \in \mathbb{F}_m^E \cap [0,1]$, $0 < x -_{\mathbb{R}} x' < 1$
Output: $(n_1, n_2, n_{\text{hi}}, n_{\text{lo}}, b_1, b_2, g_{\text{hi}}, g_{\text{lo}})$

function ExtractBitPreproc1(x, x')
  $(s\, e_E \ldots e_1\, f_1 \ldots f_m)_{\mathbb{F}_m^E} \leftarrow x$
  $(s'\, e'_E \ldots e'_1\, f'_1 \ldots f'_m)_{\mathbb{F}_m^E} \leftarrow x'$
  $e \leftarrow (e_E \ldots e_1)_2$; $e' \leftarrow (e'_E \ldots e'_1)_2$
  $\hat{e} \leftarrow e - (2^{E-1} - 1) + \mathbf{1}[e = 0]$
  $\hat{e}' \leftarrow e' - (2^{E-1} - 1) + \mathbf{1}[e' = 0]$
  $f \leftarrow (1\, f_1 \ldots f_m)_2 - (\mathbf{1}[e = 0] \ll m)$
  $f' \leftarrow (1\, f'_1 \ldots f'_m)_2 - (\mathbf{1}[e' = 0] \ll m)$
  $f'_{\text{hi}} \leftarrow f' \gg \min\{\hat{e} - \hat{e}', E + m\}$
  $f'_{\text{lo}} \leftarrow f' \,\&\, ((1 \ll \min\{\hat{e} - \hat{e}', m + 1\}) - 1)$
  $n_1 \leftarrow -\hat{e} - 1 + \mathbf{1}[x = 1]$
  $n_2 \leftarrow \max\{(\hat{e} - \hat{e}') - (m + 1), 0\}$
  $n_{\text{hi}} \leftarrow m + 1 - \mathbf{1}[x = 1]$
  $n_{\text{lo}} \leftarrow \min\{\hat{e} - \hat{e}', m + 1\}$
  $b_1 \leftarrow 0$; $b_2 \leftarrow \mathbf{1}[f'_{\text{lo}} > 0]$
  $g_{\text{hi}} \leftarrow f - f'_{\text{hi}} - b_2$; $g_{\text{lo}} \leftarrow (b_2 \ll n_{\text{lo}}) - f'_{\text{lo}}$
  return $(n_1, n_2, n_{\text{hi}}, n_{\text{lo}}, b_1, b_2, g_{\text{hi}}, g_{\text{lo}})$

Algorithm 4 Extract Binary Digit

Input: $\beta := (n_1, n_2, n_{\text{hi}}, n_{\text{lo}}, b_1, b_2, g_{\text{hi}}, g_{\text{lo}})$, $\ell \geq 1$;
where $n_1, n_2, n_{\text{hi}}, n_{\text{lo}} \geq 0$, $b_1, b_2 \in \{0,1\}$,
$0 \leq g_{\text{hi}} < 2^{n_{\text{hi}}}$, $0 \leq g_{\text{lo}} < 2^{n_{\text{lo}}}$
are from ExtractBitPreproc1(x, x')
Output: ℓ-th bit of $x -_{\mathbb{R}} x'$ in binary expansion

function ExtractBit(β, ℓ)
  if $\ell \leq n_1$ **return** b_1
  if $\ell \leq n_1 + n_{\text{hi}}$ **return** $g_{\text{hi}, \ell - n_1}$
  if $\ell \leq n_1 + n_{\text{hi}} + n_2$ **return** b_2
  if $\ell \leq n_1 + n_{\text{hi}} + n_2 + n_{\text{lo}}$
    return $g_{\text{lo}, \ell - (n_1 + n_{\text{hi}} + n_2)}$
  return 0

Theorem 5.16. *Suppose $x, x' \in \mathbb{F}_m^E$ satisfy $0 < x -_{\mathbb{R}} x' < 1$, and consider any integer $\ell \geq 1$. Let $\beta = (n_1, n_2, n_{\text{hi}}, n_{\text{lo}}, b_1, b_2, g_{\text{hi}}, g_{\text{lo}})$ be the output of ExtractBitPreproc1(x, x') (Algorithm 3), and let b' be the output of ExtractBit(β, ℓ) (Algorithm 4). Then,*

$$x -_{\mathbb{R}} x' = \Big(0.\underbrace{\boxed{b_1 \ldots b_1}}_{n_1 \text{ bits}}\underbrace{\boxed{g_{\text{hi}}}}_{n_{\text{hi}} \text{ bits}}\underbrace{\boxed{b_2 \ldots b_2}}_{n_2 \text{ bits}}\underbrace{\boxed{g_{\text{lo}}}}_{n_{\text{lo}} \text{ bits}}\Big)_2 \tag{5.9}$$

and b' is the ℓ-th digit of $x -_{\mathbb{R}} x'$ in binary expansion. Also, all intermediate values appearing in both algorithms are representable as $(1 + E + m)$-bit signed integers. «

The next result establishes the entropy-optimality (and, in turn, soundness) of Algorithm 2, combining Theorem 4.6 (correctness of the infinite version), Theorem 5.16 (correctness of bit

extraction), Prop. 5.13 (correspondence of CDF and binary-coded probability distribution), and Prop. 5.14 and Cor. 5.15 (generation over $\mathbb{U}_n$ followed by transformation through $\phi_{\mathbb{B}}$).

Theorem 5.17. *If $F : \{0,1\}^n \to \mathbb{F}_m^E \cap [0,1]$ is a CDF over a binary number format $\mathbb{B} = (n, \gamma_{\mathbb{B}}, \phi_{\mathbb{B}})$, then Opt$(F)$ (Algorithm 2) is an entropy-optimal random variate generator that returns a string $x \in \{0,1\}^n$ in the format $\mathbb{B}$ with cumulative probability $F(x)$.* «

The worst-case entropy cost of Algorithm 2 is $2^{E-1} + m - 2$ bits, which is the index location of a nonzero bit in the binary expansion of the smallest nonzero probability in $\mathbb{F}_m^E$. This observation implies that Algorithm 2 halts, because all the probabilities are dyadic rationals. The next result establishes a more useful upper bound for Algorithm 2 in terms of its expected entropy cost.

Theorem 5.18. *The expected entropy cost of Algorithm 2 is at most $m + 2 - 2^{-2^{E-1}+3}$ bits.* «

6 Extended-Accuracy Generation by Leveraging Numerical Survival Functions

Algorithm 2 from §5 requires a finite-precision CDF F, which computes floating-point probabilities of intervals $[-\infty, x]$ in the "left" tail of the distribution. Recall, however, that floats have "high-precision" near 0 as compared to 1, i.e., there are roughly 2^{E-1} more floats in $[0, 0.5)$ as compared to $[0.5, 1)$. As a result, F more accurately represents the left tail $[-\infty, x]$ (probabilities near 0), as compared to the right tail $[x, \infty]$ (probabilities near 1). To achieve high-accuracy floating-point probabilities in the right tail, we can combine F with a finite-precision *survival function* S (Def. 5.10).

For example, the Rayleigh distribution has the theoretical range $[0, \infty)$ and CDF $F(t) = 1 - e^{-t^2/2}$. Typical floating-point implementations of the CDF and SF correspond to the following ranges:

```
standard_rayleigh_cdf = lambda t: -math.expm1(-t*t/2)
```
$\rightsquigarrow [3.50 \times 10^{-162}, 8.65]$ (6.1)

```
standard_rayleigh_sf  = lambda t: math.exp(-t*t/2)
```
$\rightsquigarrow [1.05 \times 10^{-8}\ , 38.60]$ (6.2)

Eqs. (6.1) and (6.2) show that the combined range $[3.5 \times 10^{-162}, 38.60]$ of these complimentary specifications is far more accurate than using only the CDF or SF. Another illustrative example is symmetric distributions. Consider the CDF (`gsl_cdf_gaussian_P`) and SF (`gsl_cdf_gaussian_Q`) of a Gaussian from the GSL (Listing 2). For `sigma` = 1, the theoretical range is $(-\infty, \infty)$, but the floating-point ranges are $[-37.52, 8.29]$ and $[-8.29, 37.52]$, respectively.

In the infinite-precision Real-RAM model, the CDF F and SF S of a random variable X can be combined by using the following property, which holds for every real "cutoff-point" $t^* \in \mathbb{R}$:

$$\Pr(X \le t) = (1 - \mathbf{1}[t \ge t^*])F(t) + \mathbf{1}[t \ge t^*](1 - S(t)) \qquad (t \in \mathbb{R}). \tag{6.3}$$

Combining F and S as in (6.3) must be done with caution in the finite-precision setting, because $(1 - S(t)) \notin \mathbb{F}_m^E$ for many values of $t \in \mathbb{B}$. We address this challenge by introducing a DDF (Def. 6.1), which is a combined representation for $(F(t), 1 - S(t))$ that avoids an explicit inexact floating-point subtraction. The key idea is to use F and S to represent the left and right tails, respectively, of the distribution, with a cutoff point $b^* \in \mathbb{B}$ that is the exact median of F.

Definition 6.1. A *finite-precision dual distribution function* (DDF) over a binary number format $\mathbb{B} = (n, \gamma_{\mathbb{B}}, \phi_{\mathbb{B}})$ is a mapping $G : \{0,1\}^n \to \{0,1\} \times \left(\mathbb{F}_m^E \cap [0, 1/2]\right)$ such that $G^*(\phi_{\mathbb{B}}(1^n)) = 1$ and $b <_{\mathbb{B}} b' \implies G^*(b) \le G^*(b')$, where $G^* : \{0,1\}^n \to [0,1]$ is defined by $G^*(b) := (1-d)f + d(1-f)$ for $b \in \{0,1\}^n$ and $(d, f) := G(b)$. «

Remark 6.2. A finite-precision CDF F returning floating-point probabilities in $\mathbb{F}_m^E$ can represent a distribution with at most $(2^{E-1} - 1)2^m$ outcomes, while a finite-precision DDF G can represent *twice* as many outcomes. The representable probabilities are always integer multiples of $2^{-(2^{E-1}+m-2)}$. «

Theorem 6.3. *Let F be a CDF and S a SF over a binary number format $\mathbb{B}$, such that $S(b^*) < 1/2$ for some cutoff $b^* :=$ Quantile$(F, \mathrm{succ}_{\mathbb{F}_m^E}(0.5)) \in \{0,1\}^n$. A sound DDF G over $\mathbb{B}$ satisfying Def. 6.1 is*

$$G(b) := (0, F(b)) \text{ if } b <_{\mathbb{B}} b^*, \qquad G(b) := (1, S(b)) \text{ if } b \ge_{\mathbb{B}} b^* \qquad (b \in \{0,1\}^n). \tag{6.4}$$ «

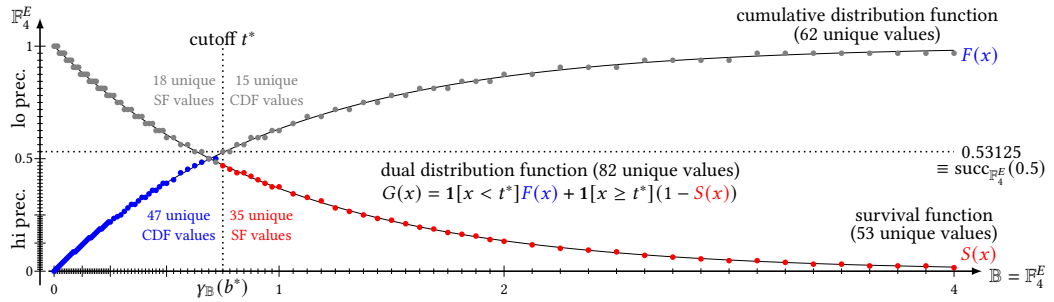


Fig. 7. A dual distribution function G combines a finite-precision cumulative distribution F and survival function S to represent probabilities in the left tail (blue; below the median cutoff) and right tail (red; above the median cutoff), respectively. This combination ensures that all explicitly represented floats lie in the high-precision range $[0, 1/2]$, which in turn supports more unique values for random variate generation.

Figure 7 shows a DDF for the exponential distribution (Listing 1). Colored dots show the selected floating-point probabilities in $[0, 0.5)$. Solid lines show the underlying real functions.

Algorithm 2 must be modified to soundly generate from a DDF G, which returns a pair $c = (d, f)$ denoting f if $d = 0$ or $1 - f$ if $d = 1$, as described in Remark E.1 and Algorithm E15 of Appendix E. Algorithm E9 gives a fast implementation of Quantile for any CDF, which is used to obtain the cutoff b^* in Theorem 6.3; and Algorithm E16 shows the quantile of a DDF. As compared to heuristic methods such as `exp_qf` (Listing 1) or `gsl_cdf_gaussian_Pinv` (Listing 2), our quantile computations are *exact* for the implemented random variate generator (cf. Fig. 2).

7 Evaluation

We implemented a C library (https://github.com/probsys/librvg) with all the algorithms described in this article. Listings 1 and 2 shows examples of the programming interface, using the macros `GENERATE_FROM_(CDF|SF|DDF)`. Our evaluation [52] investigates the following research questions.

(Q1) How does the entropy-optimal method (Algorithm 2) compare to exact conditional bit sampling [58; §II.B] (Algorithm A6) and the inexact GSL generators [24], in terms of (i) input bits per output variate; and (ii) output variates per wall-clock second? (§7.1)

(Q2) How do the ranges of random variate generators specified by a finite-precision CDF, SF, and DDF compare to one another, and to those of GSL generators? (§7.2)

(Q3) How large is the entropy and runtime overhead when using the extended-accuracy variants of conditional bit sampling and entropy-optimal generators (Algorithms E14 and E15) that use a DDF described in §6, instead of the original generators (Algorithms 2 and A6) described in §5 that use a CDF? (§7.3)

7.1 (Q1) Input Entropy Rate and Output Generation Rate

Table 1 shows measurements for 18 representative "continuous" distributions and 6 representative "discrete" distributions. The entropy source used in this experiment is a GSL pseudorandom number generator (PRNG) that calls `/dev/urandom` to obtain 8 random bytes stored in a 64-bit word.

Input Entropy Rate. In terms of input bits/variate (lower is better), the conditional bit sampling (CBS) baseline is 1x–3.1x more expensive than Algorithm 2 (OPT). An interesting finding is that the optimal generator draws around 25 bits on average for the 18 "continuous" distributions, which is two bits higher than the 23-bit mantissa in IEEE-754 single-precision format used to represent the output of the CDF (cf. Theorem 5.18). This finding suggests that the GSL CDF implementations

Table 1. Comparison of optimal generation (OPT, Algorithm 2) with two baselines: the GNU scientific library (GSL [24]) and conditional bit sampling (CBS [58], Algorithm A6). CDF implementations used for OPT and CBS are from the GSL (cf. Listing 2), which both generate exact random variates from the CDF. In terms of Fig. 1: the bits/variate column reports the input entropy consumption rate (lower=better), and the variates/sec column reports the output random variate generation rate (higher=better). †="discrete" distribution.

Distribution	Method	Bits/Variate	Variates/Sec	Distribution	Method	Bits/Variate	Variates/Sec
Beta(5, 5)	GSL	262.80	5.01×10^5	Gumbel2(1, 5)	GSL	64.00	1.37×10^6
	CBS	52.10	2.80×10^4		CBS	49.26	4.58×10^4
	OPT	24.98	5.42×10^4		OPT	24.99	1.72×10^5
Binomial(.2, 100)†	GSL	224.79	4.98×10^5	Hypergeom(5, 20, 7)†	GSL	447.99	3.05×10^5
	CBS	15.76	3.15×10^4		CBS	6.25	1.09×10^5
	OPT	5.11	3.62×10^4		OPT	3.01	1.42×10^5
Cauchy(7)	GSL	64.00	1.36×10^6	Laplace(2)	GSL	64.00	1.46×10^6
	CBS	51.45	4.84×10^4		CBS	47.83	5.04×10^4
	OPT	25.00	2.21×10^5		OPT	25.00	2.87×10^5
ChiSquare(13)	GSL	64.00	1.24×10^6	Logistic(.5)	GSL	64.00	1.39×10^6
	CBS	47.43	2.65×10^4		CBS	48.80	4.69×10^4
	OPT	24.99	5.19×10^4		OPT	24.97	2.04×10^5
Exponential(15)	GSL	64.00	1.39×10^6	Lognormal(1, 1)	GSL	163.02	7.11×10^5
	CBS	48.56	4.61×10^4		CBS	49.27	4.10×10^4
	OPT	24.98	2.33×10^5		OPT	24.98	1.87×10^5
ExpPow(1, .5)	GSL	197.03	5.97×10^5	NegBinomial(.71, 18)†	GSL	665.83	2.17×10^5
	CBS	47.31	3.57×10^4		CBS	12.54	4.01×10^4
	OPT	25.01	8.67×10^4		OPT	4.69	4.60×10^4
Fdist(5, 2)	GSL	268.95	4.70×10^5	Pareto(3,2)	GSL	64.00	1.41×10^6
	CBS	51.45	2.63×10^4		CBS	45.92	5.35×10^4
	OPT	25.00	6.29×10^4		OPT	24.99	2.30×10^5
Flat(-7, 3)	GSL	64.00	1.45×10^6	Pascal(1, 5)†	GSL	195.59	5.00×10^5
	CBS	43.52	5.66×10^4		CBS	0.00	3.13×10^4
	OPT	24.98	4.83×10^5		OPT	0.00	2.09×10^5
Gamma(.5, 1)	GSL	198.26	6.24×10^5	Poisson(71)†	GSL	697.22	1.90×10^5
	CBS	57.00	1.40×10^4		CBS	18.32	2.07×10^4
	OPT	25.01	1.80×10^4		OPT	6.19	2.31×10^4
Gaussian(15)	GSL	162.73	7.55×10^5	Rayleigh(11)	GSL	64.00	1.44×10^6
	CBS	46.41	4.95×10^4		CBS	48.52	5.08×10^4
	OPT	25.00	2.33×10^5		OPT	24.99	2.17×10^5
Geometric(.4)†	GSL	64.00	1.38×10^6	Tdist(5)	GSL	279.77	4.39×10^5
	CBS	6.06	2.03×10^5		CBS	49.56	2.65×10^4
	OPT	3.78	3.29×10^5		OPT	25.02	4.90×10^4
Gumbel1(1,1)	GSL	64.00	1.41×10^6	Weibull(2, 3)	GSL	64.00	1.39×10^6
	CBS	50.29	4.80×10^4		CBS	55.35	4.11×10^4
	OPT	25.00	2.36×10^5		OPT	24.97	1.48×10^5

are close to the maximum entropy distributions identified by Theorem 5.18. The GSL generators are 2.6x–142x more expensive in terms of bits/variate as compared to OPT. For Pascal(1,5), a deterministic distribution, the GSL draws 195.59 bits/variate, whereas OPT uses zero. These large differences in entropy cost highlight fundamental inefficiencies of Real-RAM algorithms in the GSL. Even though a single infinitely precise uniform random variable in $[0, 1]$ contains the same amount of entropy as countably many such variates (Remark 3.2), this cost equivalence does not hold in finite-precision implementations, where each "floating-point" uniform requires many random bits (e.g., 32, 53, 64; Remark B.2). The GSL generators that require 64 bits/variate (e.g., Cauchy, Geometric, Pareto, Weibull) always use exactly one floating-point uniform. The more expensive GSL generators use rejection sampling or special relationships between random variables (e.g., Beta is a ratio of Gammas; NegBinomial uses a Gamma and Poisson) further driving up the entropy cost.

Output Generation Rate. In terms of output variates/sec (higher is better), OPT is 1.1x–8.5x faster than CBS. Both methods evaluate the CDF F the same number of times per output variate. The runtime of CBS is driven by the high overhead of computing conditional probabilities for the chain

rule, which requires expensive arbitrary-precision arithmetic. A main cost of OPT is extracting bits from exact differences of floats using Algorithms 3 and 4. The GSL generators deliver the fastest output generation rate (2.14x–34.6x higher, median 6.4x) as they do not compute F, but offer no formal guarantees (§2.2) and lower accuracy (§7.2) as compared to OPT. While entropy cost gives a theoretically precise runtime measure through the DDG tree formalism, wall-clock is dictated by many implementation details (e.g., caching, parallelism, vectorization, PRNG cost, CDF evaluation cost, programming language, etc.) that could be further optimized in our prototype.

7.2 (Q2) Output Range of Random Variate Generators

Table 2 shows a comparison of the min–max output range for GSL generators and those specified formally by a finite-precision CDF/SF (§5) and DDF (§6), for 13 distributions. The theoretical ranges of these distributions are shown in the first column. We identify several takeaways:

- The output range of a GSL generator is often close to that of a finite-precision CDF or SF, but always inferior to the range of the extended-accuracy DDF (which is up to 10^{35}x wider).
- For symmetric distributions (Cauchy, Laplace, Logistic, Gaussian, Tdist), the finite-precision DDF fixes the asymmetry in CDF and SF, by ensuring identical ranges below and above the median.
- For distributions over nonnegative numbers (Exponential, Gumbel2, Pareto, Rayleigh, Weibull, Gamma), the DDF extends the output range by many orders of magnitude, combining the CDF to represent values near 0 and SF to represent values away from zero.
- The output ranges of our generators can be quickly obtained (using, e.g., Algorithm E9) in microseconds. In contrast, finding the range of a GSL generator requires dozens of seconds in certain cases that can be enumerated (i.e., the algorithm draws a single 32-bit floating-point uniform) and cannot be done in cases that draw two or more floating-point uniforms (e.g., Gamma, Gaussian, and Tdist). It is impractical to enumerate the CDF of a GSL generator in all cases.

7.3 (Q3) Runtime Overhead of Extended-Accuracy Generators

Figure 8 shows the overhead of using the extended-accuracy algorithms described in §6 in terms of bits/variate and variates/sec, for both conditional bit sampling (Algorithms A6 and E14) and optimal generation (Algorithms 2 and E15). The bits/variate ratios are slightly above one in most cases because a DDF can represent twice as many outcomes compared to a CDF (Remark 6.2), and in turn higher-entropy distributions. The variates/sec ratios for conditional bit sampling are 0.51x–0.85x (average=0.63x). This high overhead arises from the larger number of machine words needed to compute ratios of probabilities using arbitrary-precision arithmetic. The variates/sec ratios using the optimal generators are 0.40x–1.28x (average=1.00x). The only substantial slowdown (0.40x) is on the degenerate Pascal(1,5) distribution. The DDF algorithms have a (statistically significant) higher output generation rate than the CDF algorithms in 10/24 cases. In summary, the extended-accuracy generators incur minimal overhead compared to their lower-accuracy counterparts.

8 Related Work

Random variate generation has been traditionally grounded in the idealized Real-RAM model of computation [13, 45–47]: §2.2 demonstrates several challenges with this approach. Our approach deviates from this tradition in two ways—(i) it is based on a realistic model of the finite-precision computer on which the algorithms execute; and (ii) random variate generators are automatically synthesized from numerical programs specifying the CDF or SF—enabling Contributions (C2)–(C5).

DDG Trees. Our approach builds on the discrete distribution generating (DDG) tree formalism of Knuth and Yao [37]. Contribution (C2) improves on their original method for lazy DDG exploration [37; pp 384–385] by giving a deterministic algorithm that is space-time optimal (Remark 4.7);

Table 2. Comparison of random variate generators from the GNU Scientific Library (GSL) and exact random variate generators for a finite-precision cumulative distribution function (CDF), survival function (SF), or a combination of the two (DDF) on 13 probability distributions. The random variate range shows the minimum and maximum values of the output of each generator. Intervals visualized on a log scale.

Distribution	Method	Random Variate Range		Analysis Time
Cauchy(1)	GSL	-1.37×10^{9}	1.37×10^{9}	41 s
$(-\infty, \infty)$	CDF	-4.54×10^{44}	1.07×10^{7}	<50 µs
	SF	-1.07×10^{7}	4.54×10^{44}	<50 µs
	DDF	-4.54×10^{44}	4.54×10^{44}	<50 µs
Exponential(1)	GSL	0.00	22.18	36 s
$(0, \infty)$	CDF	7.01×10^{-46}	17.33	<50 µs
	SF	2.98×10^{-8}	103.97	<50 µs
	DDF	7.01×10^{-46}	103.97	<50 µs
Flat(.1, 3.14)	GSL	0.10	3.14	19 s
$(.1, 3.14)$	CDF	0.10	3.14	<50 µs
	SF	0.10	3.14	<50 µs
	DDF	0.10	3.14	<50 µs
Gumbel1(1,1)	GSL	−3.10	22.18	67 s
$(-\infty, \infty)$	CDF	−4.64	17.33	<50 µs
	SF	−2.85	103.97	<50 µs
	DDF	−4.64	103.97	<50 µs
Gumbel2(1, 1)	GSL	4.51×10^{-2}	4.29×10^{9}	19 s
$(0, \infty)$	CDF	9.62×10^{-3}	3.36×10^{7}	<50 µs
	SF	5.77×10^{-2}	1.43×10^{45}	<50 µs
	DDF	9.62×10^{-3}	1.43×10^{45}	<50 µs
Laplace(1)	GSL	−21.49	21.49	48 s
$(-\infty, \infty)$	CDF	−103.28	16.64	<50 µs
	SF	−16.64	103.28	<50 µs
	DDF	−103.28	103.28	<50 µs
Logistic(1)	GSL	−22.18	22.18	39 s
$(-\infty, \infty)$	CDF	−103.97	17.33	<50 µs
	SF	−17.33	103.97	<50 µs
	DDF	−103.97	103.97	<50 µs
Pareto(3, 2)	GSL	2.00	3.25×10^{3}	61 s
$(2, \infty)$	CDF	2.00	6.45×10^{2}	<50 µs
	SF	2.00	2.25×10^{15}	<50 µs
	DDF	2.00	2.25×10^{15}	<50 µs
Rayleigh(1)	GSL	2.20×10^{-5}	6.66	35 s
$(0, \infty)$	CDF	3.74×10^{-23}	5.89	<50 µs
	SF	2.44×10^{-4}	14.42	<50 µs
	DDF	3.74×10^{-23}	14.42	<50 µs
Weibull(1, 1)	GSL	0.00	22.18	92 s
$(0, \infty)$	CDF	7.01×10^{-46}	17.33	<50 µs
	SF	2.98×10^{-8}	103.97	<50 µs
	DDF	7.01×10^{-46}	103.97	<50 µs
Gamma(.5, 1)	GSL	—	—	∞
$(0, \infty)$	CDF	3.86×10^{-91}	15.36	<50 µs
	SF	6.98×10^{-16}	101.09	<50 µs
	DDF	3.86×10^{-91}	101.09	<50 µs
Gaussian(0, 1)	GSL	—	—	∞
$(-\infty, \infty)$	CDF	−14.17	5.42	<50 µs
	SF	−5.42	14.17	<50 µs
	DDF	−14.17	14.17	<50 µs
Tdist(1)	GSL	—	—	∞
$(-\infty, \infty)$	CDF	-4.54×10^{44}	1.07×10^{7}	<50 µs
	SF	-1.07×10^{7}	4.54×10^{44}	<50 µs
	DDF	-4.54×10^{44}	4.54×10^{44}	<50 µs

whereas Contributions (C3)–(C5) go beyond the work of Knuth and Yao [37]. Many existing DDG algorithms for discrete distributions require enumerating the target probabilities [16, 35, 51, 53, 54], which is intractable for the class of finite-precision probability distributions that we consider.

Finite Precision. Several works have introduced finite-precision generators for specific distributions (e.g., Laplace [26, 42], exponential [62], uniform [8, 28, 29, 41], Gaussian [27, 67]). This work introduces general methods that are not specific to any particular distribution. Derflinger

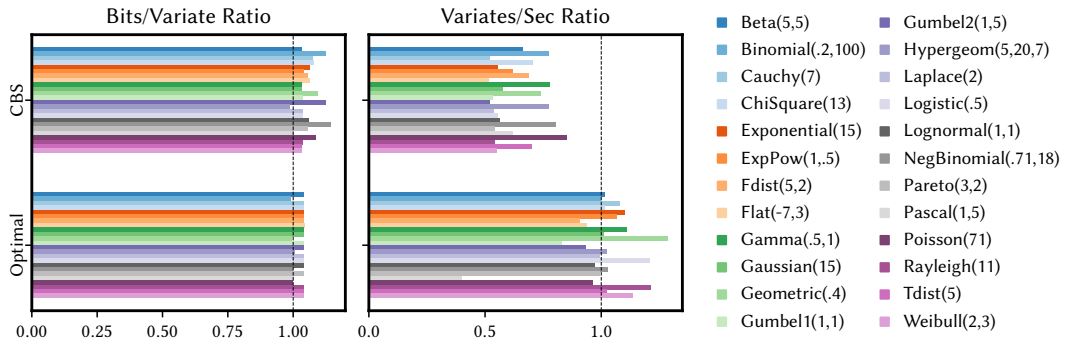


Fig. 8. Ratio of bits/variate (lower=better) and variates/sec (higher=better) using the DDF specification (§6) versus the CDF specification (§5) of the 24 target probability distributions from Table 1. CBS compares Algorithm E14/Algorithm A6. Optimal compares Algorithm E15/Algorithm 2.

et al. [12] describe an approximate generation method given a numerical implementation of a probability density function, although its theoretical guarantees only hold under the Real-RAM model [12; Remark 9]. Uyematsu and Li [63] give an implementation of the (entropy-suboptimal) Han and Hoshi [32] algorithm using integer arithmetic, which requires very high precision. Fore example, given n-bit floating-point probabilities in $\mathbb{F}_m^E$, the Uyematsu and Li [63] method requires $2^E + 2m - 1 \gg n$ bits of precision for the integer arithmetic to be exact (e.g., 2151 bits for 64-bit floats). In contrast, our work builds on the (entropy-optimal) Knuth and Yao [37] method and requires integer arithmetic with exactly $n := 1 + E + m$ bits of precision, matching the precision level used to specify the numerical CDF implementation (e.g., 64 bits for 64-bit floats).

Arbitrary Precision. Devroye and Gravel [14] present universal generation algorithms that require arbitrary-precision arithmetic (e.g., MPFR [23] or GMP [31]). Specialized arbitrary-precision generators for the discrete Gaussian distribution have been widely studied [9, 18, 19, 36], given its prominence in cryptography and differential privacy. In contrast to this line of work, our method uses finite- instead of arbitrary-precision, retaining high performance and predictability of runtime and memory. A promising direction is to develop finite-precision CDF, SF, or DDF specifications that meet the accuracy requirements for these applications, which could be implemented using numerical libraries for approximating real functions with error guarantees [7, 10, 39, 40, 57, 68].

Probabilistic Programming. Several probabilistic programming languages and solvers use the CDF to form discrete approximations of continuous probability distributions [4, 25, 55, 69]. The denotational semantics of these systems adopt the infinite-precision Real-RAM model, which does not comport with their actual operational semantics on a finite-precision computer. The resulting systems offer no correctness or exactness guarantees for the machine implementation. Bagnall et al. [2] develop formally verified generators for discrete probabilistic programs with loops and conditioning using arbitrary-precision rational arithmetic, and use it to implement exact generators for the discrete Laplace and Gaussian distributions. A promising idea along this direction is to build a probabilistic programming language that instead uses the exact finite-precision random variate generators described in §4–§6, as the basic probabilistic primitives with formal guarantees.

9 Conclusion

As the role of probability in computer science continues to grow [3, 43], there is a growing need for random variate generation methods with well-characterized behavior. We hope this work lays a foundation for a new class of random variate generators that are equipped with theoretical guarantees while delivering improvements in automation, accuracy, and entropy consumption.

Data-Availability Statement

A C library with all the algorithms described in this article is available at https://github.com/probsys/librvg. A reproduction package for the evaluation in §7 is available on Zenodo [52].

Acknowledgments

The authors thank the referees and Martin Rinard for helpful feedback. Feras Saad and Wonyeol Lee are corresponding authors. This material is based upon work supported by the National Science Foundation under Grant No. 2311983. Any opinions, findings, and conclusions or recommendations in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] John M. Abowd et al. 2022. The 2020 Census Disclosure Avoidance System TopDown Algorithm. *Harvard Data Science Review* Special Issue 2 (June 2022), 77 pages. https://doi.org/10.1162/99608f92.529e3cb9

[2] Alexander Bagnall, Gordon Stewart, and Anindya Banerjee. 2023. Formally Verified Samplers from Probabilistic Programs with Loops and Conditioning. *Proc. ACM Program. Lang.* 7, PLDI, Article 106 (2023), 24 pages. https://doi.org/10.1145/3591220

[3] Gilles Barthe, Katoen Joost-Pieter, and Alexandra Silva (Eds.). 2020. *Foundations of Probabilistic Programming*. Cambridge University Press, Cambridge, UK. https://doi.org/10.1017/9781108770750

[4] Vaishak Belle, Andrea Passerini, and Guy Van den Broeck. 2015. Probabilistic Inference in Hybrid Domains by Weighted Model Integration. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence*. AAAI Press, Palo Alto, 2770–2776.

[5] Patrick Billingsley. 1986. *Probability and Measure* (2nd ed.). John Wiley & Sons, New York.

[6] Vladimir I. Bogachev. 2007. *Measure Theory*. Vol. 2. Springer, Berlin. https://doi.org/10.1007/978-3-540-34514-5

[7] Ian Briggs, Yash Lad, and Pavel Panchekha. 2024. Implementation and Synthesis of Math Library Functions. *Proc. ACM Program. Lang.* 8, POPL, Article 32 (Jan. 2024), 28 pages. https://doi.org/10.1145/3632874

[8] Taylor R. Campbell. 2014. Uniform Random Floats. https://prng.di.unimi.it/random_real.c

[9] Clément L Canonne, Gautam Kamath, and Thomas Steinke. 2020. The Discrete Gaussian for Differential Privacy. In *Proceedings of the 34 International Conference on Neural Information Processing Systems (Advances in Neural Information Processing Systems, Vol. 33)*. Curran Associates, Inc., Red Hook, Article 1315, 13 pages. https://doi.org/10.5555/3495724.3497039

[10] Catherine Daramy-Loirat, David Defour, Florent de Dinechin, Matthieu Gallet, Nicolas Gast, Christoph Lauter, and Jean-Michel Muller. 2006. *CR-LIBM: A Library of Correctly Rounded Elementary Functions in Double-Precision*. Research Report ensl-01529804. Laboratoire de l'Informatique du Parallèlisme. https://ens-lyon.hal.science/ensl-01529804

[11] Christian de Schryver, Daniel Schmidt, Norbert Wehn, Elke Korn, Henning Marxen, Anton Kostiuk, and Ralf Korn. 2012. A Hardware Efficient Random Number Generator for Nonuniform Distributions with Arbitrary Precision. *International Journal of Reconfigurable Computing* 2012, Article 675130 (March 2012), 11 pages. https://doi.org/10.1155/2012/675130

[12] Gerhard Derflinger, Wolfgang Hörmann, and Josef Leydold. 2010. Random Variate Generation by Numerical Inversion when Only the Density is Known. *ACM Transactions on Modeling and Computer Simulation* 20, 4, Article 18 (Oct. 2010), 25 pages. https://doi.org/10.1145/1842722.1842723

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

[14] Luc Devroye and Claude Gravel. 2015. Sampling with Arbitrary Precision. arXiv:1502.02539v1

[15] Allen B. Downey. 2007. Generating Pseudo-random Floating-Point Values. https://allendowney.com/research/rand/downey07randfloat.pdf

[16] Thomas L. Draper and Feras A. Saad. 2025. Efficient Rejection Sampling in the Entropy-Optimal Range. arXiv:2504.04267

[17] Chaohui Du and Guoqiang Bai. 2015. Towards Efficient Discrete Gaussian Sampling For Lattice-Based Cryptography. In *Proceedings of the 25th International Conference on Field Programmable Logic and Applications*. IEEE Press, Piscataway, 1–6. https://doi.org/10.1109/FPL.2015.7293949

[18] Yusong Du, Baoying Fan, and Baodian Wei. 2022. An Improved Exact Sampling Algorithm for the Standard Normal Distribution. *Computational Statistics* 37 (2022), 721–737. https://doi.org/10.1007/s00180-021-01136-w

[19] Léo Ducas and Phong Q. Nguyen. 2012. Faster Gaussian Lattice Sampling Using Lazy Floating-Point Arithmetic. In *Proceedings of the 18th International Conference on the Theory and Application of Cryptology and Information Security (Lecture Notes in Computer Science, Vol. 7658)*. Springer, Heidelberg, 415–432. https://doi.org/10.1007/978-3-642-34961-4_26

[20] Nagarjun C. Dwarakanath and Steven D. Galbraith. 2014. Sampling from Discrete Gaussians for Lattice-Based Cryptography On a Constrained Device. *Applicable Algebra in Engineering, Communication and Computing* 25, 3 (June

2014), 159–180. https://doi.org/10.1007/s00200-014-0218-3
[21] Cynthia Dwork. 2006. Differential Privacy. In *Proceedings of the 33rd International Colloquium on Automata, Languages, and Programming*. Springer, Berlin, Heidelberg, 1–12. https://doi.org/10.1007/11787006_1
[22] János Folláth. 2014. Gaussian Sampling in Lattice Based Cryptography. *Tatra Mountains Mathematical Publications* 60, 1 (Sept. 2014), 1–23. https://doi.org/10.2478/tmmp-2014-0022
[23] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. 2007. MPFR: A Multiple-Precision Binary Floating-Point Library with Correct Rounding. *ACM Trans. Math. Software* 33, 2 (June 2007), 15 pages. https://doi.org/10.1145/1236463.1236468
[24] Mark Galassi, Jim Davies, James Theiler, Brian Gough, Gerard Jungman, Patrick Alken, Michael Booth, and Fabrice Rossi. 2009. *GNU Scientific Library Reference Manual* (3rd ed.). Network Theory Ltd., Surrey. http://www.gnu.org/software/gsl/
[25] Poorva Garg, Steven Holtzen, Guy Van den Broeck, and Millstein Todd. 2024. Bit Blasting Probabilistic Programs. *Proc. ACM Program. Lang.* 8, PLDI, Article 182 (2024), 24 pages. https://doi.org/10.1145/3656412
[26] Ivan Gazeau, Dale Miller, and Catuscia Palamidessi. 2016. Preserving Differential Privacy Under Finite-Precision Semantics. *Theoretical Computer Science* 655 (Dec. 2016), 92–108. https://doi.org/10.1016/j.tcs.2016.01.015
[27] Michael Giles and Oliver Sheridan-Methven. 2023. Approximating Inverse Cumulative Distribution Functions to Produce Approximate Random Variables. *ACM Trans. Math. Software* 49, 3, Article 26 (Sept. 2023), 29 pages. https://doi.org/10.1145/3604935
[28] Frédéric Goualard. 2020. Generating Random Floating-Point Numbers by Dividing Integers: A Case Study. In *Computational Science (Lecture Notes in Computer Science, Vol. 12138)*. Springer International Publishing, Cham, 15–28. https://doi.org/10.1007/978-3-030-50417-5_2
[29] Frédéric Goualard. 2022. Drawing Random Floating-Point Numbers from an Interval. *ACM Transactions on Modeling and Computer Simulation* 32, 3, Article 16 (April 2022), 24 pages. https://doi.org/10.1145/3503512
[30] Artur Grabowski. 2015. Generating Random Doubles. https://github.com/art4711/random-double/blob/master/rd.c
[31] Torbjörn Granlund. 2023. *GNU MP: The GNU Multiple Precision Arithmetic Library* (6.3.0 ed.). Free Software Foundation, Inc, Boston. https://gmplib.org/gmp-man-6.3.0.pdf
[32] Te Sun Han and Mamoru Hoshi. 1997. Interval Algorithm for Random Number Generation. *IEEE Transactions on Information Theory* 43, 2 (March 1997), 599–611. https://doi.org/10.1109/18.556116
[33] Charles R Harris et al. 2020. Array Programming with NumPy. *Nature* 585 (Sept. 2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2
[34] Institute of Electrical and Electronics Engineers. 2019. *IEEE Standard for Floating-Point Arithmetic (IEEE Std 754-2019)*. IEEE Press, Piscataway. https://doi.org/10.1109/IEEESTD.2019.8766229
[35] Angshuman Karmakar, Sujoy Sinha Roy, Oscar Reparaz, Frederik Vercauteren, and Ingrid Verbauwhede. 2018. Constant-Time Discrete Gaussian Sampling. *IEEE Trans. Comput.* 67, 11 (2018), 1561–1571. https://doi.org/10.1109/TC.2018.2814587
[36] Charles F. F. Karney. 2016. Sampling Exactly from the Normal Distribution. *ACM Trans. Math. Software* 42, 1, Article 3 (Jan. 2016), 14 pages. https://doi.org/10.1145/2710016
[37] Donald E. Knuth and Andrew C. Yao. 1976. The Complexity of Nonuniform Random Number Generation. In *Algorithms and Complexity: New Directions and Recent Results*, Joseph F. Traub (Ed.). Academic Press, Inc., Orlando, FL, 357–428.
[38] Daniel Lemire. 2019. Fast Random Integer Generation in an Interval. *ACM Transactions on Modeling and Computer Simulation* 29, 1, Article 3 (Jan. 2019), 12 pages. https://doi.org/10.1145/3230636
[39] Jay P. Lim and Santosh Nagarakatte. 2021. High Performance Correctly Rounded Math Libraries for 32-Bit Floating Point Representations. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. Association for Computing Machinery, New York, 359–374. https://doi.org/10.1145/3453483.3454049
[40] Jay P. Lim and Santosh Nagarakatte. 2022. One Polynomial Approximation to Produce Correctly Rounded Results of an Elementary Function for Multiple Representations and Rounding Modes. *Proc. ACM Program. Lang.* 6, POPL, Article 3 (Jan. 2022), 28 pages. https://doi.org/10.1145/3498664
[41] Jérémie Lumbroso. 2013. Optimal Discrete Uniform Generation from Coin Flips, and Applications. arXiv:1304.1916
[42] Ilya Mironov. 2012. On Significance of the Least Significant Bits for Differential Privacy. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. Association for Computing Machinery, New York, 650–661. https://doi.org/10.1145/2382196.2382264
[43] Michael Mitzenmacher and Upfal Eli. 2017. *Probability and Computing: Randomization and Probabilistic Techniques in Algorithms and Data Analysis* (second ed.). Cambridge University Press, Cambridge, UK.
[44] Oller Møller. 1965. Quasi Double-Precision in Floating Point Addition. *BIT Numerical Mathematics* 5, 1 (1965), 37–50. https://doi.org/10.1007/BF01975722
[45] Peter Occil. 2023. More Random Sampling Methods. https://peteroupc.github.io/randomnotes.pdf
[46] Peter Occil. 2023. Partially-Sampled Random Numbers for Accurate Sampling of Continuous Distributions. https://peteroupc.github.io/exporand.pdf

[47] Peter Occil. 2024. Randomization and Sampling Methods. https://peteroupc.github.io/randomfunc.pdf
[48] Adam Paszke et al. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems (Advances in Neural Information Processing Systems, Vol. 32)*. Curran Associates, Inc., Red Hook, Article 721, 12 pages. https://doi.org/10.5555/3454287.3455008
[49] Posit Working Group. 2022. *Standard for Posit™ Arithmetic*. National Supercomputing Centre, A*STAR, Singapore. https://posithub.org/docs/posit_standard-2.pdf
[50] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. 1992. *Numerical Recipes in C* (2nd ed.). Cambridge University Press, Cambridge, UK.
[51] Sinha S. Roy, Frederik Vercauteren, and Ingrid Verbauwhede. 2013. High Precision Discrete Gaussian Sampling on FPGAs. In *Proceedings of the 20th International Conference on Selected Areas in Cryptography (Lecture Notes in Computer Science, Vol. 8282)*. Springer, Berlin, 383–401. https://doi.org/10.1007/978-3-662-43414-7_19
[52] Feras Saad and Wonyeol Lee. 2025. *librvg: C Library for Random Variate Generation with Formal Guarantees*. Zenodo. https://doi.org/10.5281/zenodo.15243770
[53] Feras A. Saad, Cameron E. Freer, Martin C. Rinard, and Vikash K. Mansinghka. 2020. The Fast Loaded Dice Roller: A Near-optimal Exact Sampler for Discrete Probability Distributions. In *Proceedings of the 23rd International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research, Vol. 108)*. PMLR, Norfolk, 1036–1046.
[54] Feras A. Saad, Cameron E. Freer, Martin C. Rinard, and Vikash K. Mansinghka. 2020. Optimal Approximate Sampling from Discrete Probability Distributions. *Proceedings of the ACM on Programming Languages* 4, POPL, Article 36 (Jan. 2020), 31 pages. https://doi.org/10.1145/3371104
[55] Feras A. Saad, Martin C. Rinard, and Vikash K. Mansinghka. 2021. SPPL: Probabilistic Programming with Fast Exact Symbolic Inference. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM, New York, 804–819. https://doi.org/10.1145/3453483.3454078
[56] Michael Ian Shamos. 1978. *Computational Geometry*. Ph. D. Dissertation. Yale University.
[57] Alexei Sibidanov, Paul Zimmermann, and Stéphane Glondu. 2022. The CORE-MATH Project. In *Proceedings of the 2022 IEEE 29th Symposium on Computer Arithmetic*. IEEE Press, Piscataway, 26–34. https://doi.org/10.1109/ARITH54963.2022.00014
[58] John S. Sobolewski and W. H. Payne. 1972. Pseudonoise with Arbitrary Amplitude Distribution–Part I: Theory. *IEEE Trans. Comput.* C-21, 4 (1972), 337–345. https://doi.org/10.1109/TC.1972.5008973
[59] Jason Stover. 2003. GNU Scientific Library: Gamma Cumulative Distribution Function. https://github.com/ampl/gsl/blob/v2.7.0/cdf/gamma.c
[60] The MathWorks, Inc. 2024. Probability Distributions - MATLAB & Simulink. https://www.mathworks.com/help/stats/probability-distributions-1.html
[61] James Theiler and Brian Gough. 2007. GNU Scientific Library: Gamma Random Variate Generator. https://github.com/ampl/gsl/blob/v2.7.0/randist/gamma.c
[62] David B. Thomas and Wayne Luk. 2008. Sampling from the Exponential Distribution Using Independent Bernoulli Variates. In *Proceedings of the 2008 International Conference on Field Programmable Logic and Applications*. IEEE Press, Piscataway, 239–244. https://doi.org/10.1109/FPL.2008.4629938
[63] Tomohiko Uyematsu and Yuan Li. 2003. Two Algorithms for Random Number Generation Implemented by Using Arithmetic of Limited Precision. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* 86, 10 (Oct. 2003), 2542–2551.
[64] Pauli Virtanen et al. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17, 3 (Feb. 2020), 261–272. https://doi.org/10.1038/s41592-019-0686-2
[65] John von Neumann. 1951. Various Techniques Used in Connection with Random Digits. In *Monte Carlo Method*, A. S. Householder, G. E. Forsythe, and H. H. Germond (Eds.). National Bureau of Standards Applied Mathematics Series, Vol. 12. U. S. Government Printing Office, Washington, Chapter 13, 36–38.
[66] Alistair J. Walker. 1974. Fast Generation of Uniformly Distributed Pseudorandom Numbers with Floating-Point Representation. *Electronics Letters* 10, 25 (Dec. 1974), 533–534. https://doi.org/10.1049/el:19740423
[67] Michael Walter. 2019. Sampling the Integers with Low Relative Error. In *Proceedings of the 11th International Conference on Cryptology in Africa (Lecture Notes in Computer Science, Vol. 11627)*. Springer, Cham, 157–180. https://doi.org/10.1007/978-3-030-23696-0_9
[68] Abraham Ziv, Moshe Olshansky, Ealan Henis, and Anna Retiman. 2001. IBM Accurate Portable Mathlib. https://github.com/dreal-deps/mathlib
[69] Pedro Zuidberg Dos Martires, Anton Dries, and Luc De Raedt. 2019. Exact and Approximate Weighted Model Integration with Probability Density Functions Using Knowledge Compilation. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence*. AAAI Press, Palo Alto, 7825–7833. https://doi.org/10.1609/aaai.v33i01.33017825

Received 2024-11-14; accepted 2025-03-06

A Conditional Bit Sampling: A Baseline for Exact Random Variate Generation

This appendix describes conditional bit sampling [58; §II.B] a baseline method for generating a random string from any binary-coded probability distribution (Def. 4.1). Whereas the original presentation of this method in Sobolewski and Payne [58] used the Real-RAM model, the presentation of conditional bit sampling in this appendix uses the DDG formalism and finite-precision computation, which gives new insights on its behavior. Appendix A.1 discusses the general case. Appendix A.2 shows how to implement this baseline given a finite-precision CDF.

A.1 Generation from a Binary-Coded Probability Distribution

Conditional bit sampling a joint distribution of $(B_1, \ldots, B_n)$ uses the chain rule of probability, i.e., it generates B_1, then $B_2 \mid B_1$, then $B_3 \mid B_1, B_2$, and so on.

Proposition A.1. *Let $p : \{0,1\}^* \to [0,1]$ be a binary-coded probability distribution. For each $n \in \mathbb{N}$. The following process generates a random string $B_1 \ldots B_n \sim p_n$:*

$$B_1 \sim \text{Bernoulli}(p(1)); \qquad B_j \sim \text{Bernoulli}\left[\frac{p(B_1 \ldots B_{j-1}1)}{p(B_1 \ldots B_{j-1})}\right] \qquad (j = 2, \ldots, n). \qquad \text{« (A.1)}$$

Proof. By induction. The base case is obvious. Assume the proposition holds for any integer $n-1$. Put $(b_1, \ldots, b_{n-1})$ so that $p(b_1, \ldots, b_{n-1}) > 0$. Then

$$\begin{aligned}
&\Pr\left(\cap_{i=1}^{n-1}\{B_i = b_i\}, B_n = b_n\right) && \text{(A.2)}\\
&= \Pr\left(B_n = b_n \mid \cap_{i=1}^{n-1}\{B_i = b_i\}\right)\Pr\left(\cap_{i=1}^{n}\{B_i = b_i\}\right) && \text{(A.3)}\\
&= \left[\frac{p(b_1 \ldots b_{n-1}1)}{p(b_1 \ldots b_{n-1})}\right]^{b_n}\left[1 - \frac{p(b_1 \ldots b_{n-1}1)}{p(b_1 \ldots b_{n-1})}\right]^{1-b_n} \cdot p(b_1 \ldots b_{n-1}) && \text{(A.4)}\\
&= \left[\frac{p(b_1 \ldots b_{n-1}1)}{p(b_1 \ldots b_{n-1})}\right]^{b_n}\left[\frac{p(b_1 \ldots b_{n-1}) - p(b_1 \ldots b_{n-1}1)}{p(b_1 \ldots b_{n-1})}\right]^{1-b_n} \cdot p(b_1 \ldots b_{n-1}) && \text{(A.5)}\\
&= [p(b_1 \ldots b_{n-1}1)]^{b_n}\,[p(b_1 \ldots b_{n-1}0)]^{1-b_n} && \text{(A.6)}\\
&= p_n(b_1 \ldots b_n). && \text{(A.7)}
\end{aligned}$$

□

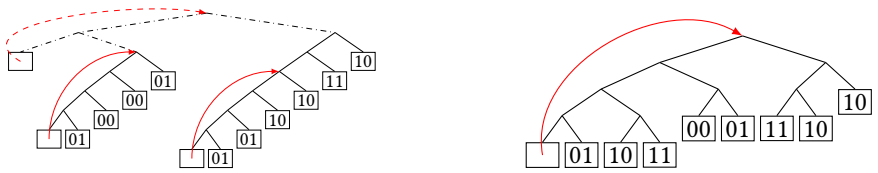
Optimal Bernoulli Generation. In the Real-RAM model, a random variable $X \sim \text{Bernoulli}(p)$ can be defined using the inverse-transform method: $X(\omega) \coloneqq \mathbf{1}[\omega \in (0,p)]$. To arrive at a random variate generator for Bernoulli(p) based on Def. 3.5, consider generating $K \sim \text{Geometric}(1/2)$ and then setting $X \leftarrow p_K$, where $p = (0.p_1p_2\ldots)_2 \in (0,1)$. The proof of correctness is direct:

$$\Pr(X=1) = \textstyle\sum_{k=1}^{\infty}\Pr(X=1|K=k)\Pr(K=k) = \sum_{k=1}^{\infty}\mathbf{1}[p_k=1]1/2^k = \sum_{k=1}^{\infty}p_k/2^k = p, \qquad \text{(A.8)}$$

$$\Pr(X=0) = \textstyle\sum_{k=1}^{\infty}\Pr(X=0|K=k)\Pr(K=k) = \sum_{k=1}^{\infty}(1-\mathbf{1}[p_k=1])1/2^k = 1-p. \qquad \text{(A.9)}$$

The expected entropy cost of this method is two bits for generating K. Lumbroso [41; Appendix B] proves this method is entropy-optimal, but their proof assumes implicitly that p is not a dyadic rational. If $p = k/2^m$ is a dyadic rational for odd k, this method is suboptimal because the bits $(p_{m+1}, p_{m+2}, \ldots)$ are zero, and so K need not be generated beyond m. In particular, it suffices to generate $K \leftarrow \min(m, K')$ where $K' \sim \text{Geometric}(1/2)$, and then set $X \leftarrow p_K$ if $K < m$ and $X \leftarrow$ RandBit otherwise. The expected entropy cost is then

$$\underbrace{\textstyle\sum_{i=1}^{m-1} i2^{-i} + (m-1)2^{-(m-1)}}_{K'} + \underbrace{2^{1-m}}_{\text{RandBit}} = 2 - 2^{2-m} + 2^{1-m} = 2 - 2^{1-m}. \qquad \text{(A.10)}$$



(a) Conditional Bit Sampling (Algorithm A5) (b) Optimal Generation (Algorithm 1)

Fig. 9. DDG trees of random variate generators for the discrete distribution p_2 in (A.11). In (a), dashed lines show the DDG tree for B_1; solid lines show DDG trees for $B_2|B_1{=}0$ (left subtree) and $B_2|B_1{=}1$ (right subtree).

Composing Optimal Bernoulli Generators. Algorithm A5 presents a random variate generator for a binary-coded probability distribution p based on Prop. A.1 and (A.8) and (A.9). The RandBit primitive returns the next unbiased random bit from the i.i.d. bit stream (Fig. 1, bottom). Each recursive step of Algorithm A5 is entropy-optimal for the Bernoulli$(p(b0)/p(b))$ distribution. However, the following example shows that a *sequence* of n generations (A.1) is not entropy-optimal for p_n.

Example A.2. Consider a binary-coded probability distribution p such that

$$p(0) = 1/3, \quad p(1) = 2/3, \quad p(00) = 2/15, \quad p(01) = 3/15, \quad p(10) = 7/15, \quad p(11) = 3/15. \tag{A.11}$$

Following (A.1), generating $B_1B_2 \sim p_2$ via $B_1 \sim$ Bernoulli$(2/3)$ and $B_2 \mid B_1 \sim$ Bernoulli$(3/(5{+}5B_1))$ consumes 4 bits on average (Fig. 9a). However, an entropy-optimal generator for p_2 constructed from the binary expansions of $(p(00), p(01), p(10), p(11))$ consumes 3.2 bits on average (Fig. 9b):

$$\begin{Bmatrix} p(00) \\ p(01) \\ p(10) \\ p(11) \end{Bmatrix} = \begin{Bmatrix} 2/15 \\ 3/15 \\ 7/15 \\ 3/15 \end{Bmatrix} = 0.\overline{\begin{matrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{matrix}} \qquad \begin{aligned} N &= (0 \cdot 1 \cdot 1/2^1) + (1 \cdot 2 \cdot 1/2^2) + (4 \cdot 3 \cdot 1/2^3) \\ &\quad + (3 \cdot 4 \cdot 1/2^4) + (1 \cdot (4+N) \cdot 1/2^4) \\ \implies N &= 3.2 \text{ bits} \qquad \text{«} \end{aligned} \tag{A.12}$$

The next proposition establishes bounds on the entropy gap between the conditional bit sampling and optimal generators in Algorithms 1 and A5. It shows that this gap could be zero, or very large.

Proposition A.3. *For a binary-coded probability distribution p, let $C_n^{\text{opt}}(p)$ and $C_n^{\text{cbs}}(p)$ denote the entropy costs of Algorithms 1 and A5, respectively, up until generating an n-bit string. There exist binary-coded distributions p and p' and an arbitrarily small constant $\varepsilon > 0$ such that*

$$\mathbb{E}[C_n^{\text{cbs}}(p)] - \mathbb{E}[C_n^{\text{opt}}(p)] = 0; \qquad \mathbb{E}[C_n^{\text{cbs}}(p')] - \mathbb{E}[C_n^{\text{opt}}(p')] = 2n - 2 - \varepsilon. \qquad \text{«} \tag{A.13}$$

Proof. For the first equality, suppose p is such that each distribution p_j is uniform distribution over $\{0,1\}^j$ $(j = 1, \ldots, n)$, so their entropies satisfy $H(p_j) = j$. Since conditional bit sampling using Bernoulli makes a coin flip with dyadic weight $1/2$ at each step, it consumes 1 bit per step on average for a total of n bits, matching the entropy-optimal generator.

For the second equality, consider a distribution p' such that p'_n has $2^n - 1$ outcomes of probability $\varepsilon \ll 2^{-n}$ and one outcome has probability $\gamma := 1 - (2^n - 1)\varepsilon$. The cost C_n^{opt} of the optimal generator satisfies

$$H(p'_n) \le \mathbb{E}[C_n^{\text{opt}}(p'_n)] \le H(p'_n) + 2 \implies \mathbb{E}[C_n^{\text{opt}}(p'_n)] \le \varepsilon \log_2(1/\varepsilon) + \gamma \log_2(\gamma^{-1}) + 2 \approx 2. \tag{A.14}$$

For conditional bit sampling, the average cost

$$\mathbb{E}[C_n^{\text{cbs}}] = T_n(n) = 2n - 2\varepsilon\left(n2^{n-1} + 1 - 2^n\right) \approx 2n \tag{A.15}$$

(a) Example binary-coded probability distribution p unrolled to the first four bits.

$$\left\{\begin{matrix} 0000 \mapsto \frac{6}{137} & 0001 \mapsto \frac{12}{137} & 0010 \mapsto \frac{13}{137} & 0011 \mapsto \frac{9}{137} & 0100 \mapsto \frac{10}{137} & 0101 \mapsto \frac{12}{137} & 0110 \mapsto \frac{6}{137} & 0111 \mapsto \frac{1}{137} \\ 1000 \mapsto \frac{1}{137} & 1001 \mapsto \frac{2}{137} & 1010 \mapsto \frac{13}{137} & 1011 \mapsto \frac{8}{137} & 1100 \mapsto \frac{14}{137} & 1101 \mapsto \frac{13}{137} & 1110 \mapsto \frac{7}{137} & 1111 \mapsto \frac{10}{137} \end{matrix}\right\}$$

Algorithm A5 Conditional Bit Sampling

```
1:  function GenCBS(p, b = ε)
2:      if p(b0) = p(b)                          ▷ Leaf
3:          return GenCBS(p, b0)                 ▷ 0
4:      if p(b1) = p(b)                          ▷ Leaf
5:          return GenCBS(p, b1)                 ▷ 1
6:      for i = 1, 2, ... do                     ▷ Refine Subtree
7:          x ← RandBit()
8:          if x = 0 ∧ [p(b0)/p(b)]_i = 1        ▷ Leaf
9:              return GenCBS(p, b0)             ▷ 0
10:         if x = 1 ∧ [p(b1)/p(b)]_i = 1        ▷ Leaf
11:             return GenCBS(p, b1)             ▷ 1
```

(b) Example trace of Algorithm A5 on p from Fig. 10a.

Recur. Level	Probabilities	RandBit x 1	0	1	1	0	1	1	Output b
0	$p(0) = \frac{69}{137} = 0.$	**1**	**0**	**0**	0	0	0	0	
	$p(1) = \frac{68}{137} = 0.$	**0**	**1**	**1**	1	1	1	1	1
1	$\frac{p(10)}{p(1)} = \frac{6}{17} = 0.$				**0**	1	0	1	
	$\frac{p(11)}{p(1)} = \frac{11}{17} = 0.$				**1**	0	1	0	1
2	$\frac{p(110)}{p(11)} = \frac{27}{44} = 0.$					**1**	0	0	0
	$\frac{p(111)}{p(11)} = \frac{17}{44} = 0.$					**0**	1	1	
3	$\frac{p(1100)}{p(110)} = \frac{14}{27} = 0.$						**1**	**0**	
	$\frac{p(1101)}{p(110)} = \frac{13}{27} = 0.$						**0**	**1**	1

Fig. 10. Conditional bit sampling algorithm for any binary-coded probability distribution $p : \{0,1\}^* \to [0,1]$ using the chain rule. Refer to the caption of Fig. 5 for details.

is the value of the following recurrence at $T_n(n)$:

$$T_n(1) = 2, \qquad T_n(k) = 2 + (k-1)c_n(k) + T_n(k-1)(1 - c_n(k)) \quad (2 \le k \le n),$$
$$\text{where } c_n(k) \coloneqq \frac{2^{k-1}\varepsilon}{(2^k - 1)\varepsilon + \gamma}. \tag{A.16}$$

To justify (A.16), in the base case ($k = 1$) the distribution p_2 over $\{0,1\}^1$ is non-uniform which requires 2 bits. For the inductive case, there are 2^k outcomes in total, where $(2^k - 1)$ have probability ε and one outcome has probability γ. Conditional bit sampling uses two bits to flip a coin with weight $c_n(k)$, which chooses between the 2^{k-1} equal-probability outcomes (after which $k-1$ flips are needed) and the remaining 2^{k-1} outcomes (after which $T_n(k-1)$ flips are needed). □

A.2 Finite-Precision Implementation

The Bernoulli function in Algorithm A7 defines an entropy-optimal random variate generator for Bernoulli(i/k) bit with rational weights, based on (A.8) and (A.9). The structure of this algorithm mirrors that of Algorithm A8, which extracts the *concise* binary expansion $(0.p_1p_2\ldots)_2$ of a rational probability $p = i/k \in (0,1)$ based on the identity

$$i/k = \begin{cases} 1/2 + 1/2 \cdot (2i - k)/k & \text{if } k \le 2i \\ 0 + 1/2 \cdot (2i/k) & \text{if } 2i < k. \end{cases} \tag{A.17}$$

The CBS function in Algorithm A6 implements Algorithm A5 by successively calling Bernoulli on ratios of rational probabilities. At each recursive step, the string $b \in \{0,1\}^{\le n}$ has been determined so far. Following (5.7), the floats f_0, f_1 store the subtrahend and minuend in the probability $p_F(b) = F(\phi_{\mathbb{B}}(b1^{n-|b|})) -_{\mathbb{R}} F(\phi_{\mathbb{B}}((b0^{n-|b|})^-))$; and the outcomes $b0$ and $b1$ have probabilities $p_F(b0) = f_2 - f_0$ and $p_F(b1) = f_1 - f_2$, respectively, where $f_2 \coloneqq F(\phi_{\mathbb{B}}(b'))$ is the cumulative probability of the "midpoint" string $b' \in \{0,1\}^n$ that lies between $b0$ and $b1$. ExactRatio is any algorithm that returns integers (i,k) such that $i/k = (f_1 - f_2)/(f_1 - f_0)$. While these integers cannot be computed directly using floating-point arithmetic and are not guaranteed to fit in a single $1 + E + m$ bit machine word,

the next results establish tight bounds on the finite buffer sizes needed to store i, k, showing that the method does not require arbitrary-precision arithmetic.

Theorem A.4. *Suppose $E \geq 1$ and $m \geq 3$. Given $f_0, f_1, f_2 \in \mathbb{F}_m^E \cap [0,1]$ with $f_0 \neq f_1$, let $i(f_0, f_1, f_2)$ and $k(f_0, f_1, f_2)$ be coprime integers such that $i(f_0, f_1, f_2)/k(f_0, f_1, f_2) = (f_1 - f_2)/(f_1 - f_0)$. Then*

$$\max_{0 \leq f_0 < f_2 < f_1 \leq 1} \left\{ \left\lceil 1 + \log_2 i(f_0, f_1, f_2) \right\rceil + \left\lceil 1 + \log_2 k(f_0, f_1, f_2) \right\rceil \right\} = 2 \cdot (2^{E-1} + m - 2) + 1. \quad \text{« (A.18)}$$

Proof. Let $\delta := 2^{E-1} - 2 + m \geq 2$. Every $x \in [0,1] \cap \mathbb{F}_m^E$ is an integer multiple of the smallest positive subnormal $2^{-\delta}$. Then $2^\delta(f_1 - f_2)$ and $2^\delta(f_1 - f_0)$ are integers between 0 and 2^δ. Put $f_0 = 0$, $f_1 = 1$, and $f_2 = 2^{-\delta}$. Then $(f_1 - f_2)/(f_1 - f_0) = (2^\delta - 1)/2^\delta = i(f_0, f_1, f_2)/k(f_0, f_1, f_2)$ form the largest pair of coprime numbers in $\{0, \ldots, 2^\delta\}$. □

Corollary A.5. *In Algorithm A6, ExactRatio returns integers i, k each comprised of at most $2^{E-1} + m - 1$ bits, i.e., at most 5 (resp. 17) machine words on IEEE-754 single (resp. double) precision.* «

Algorithm A6 Conditional Bit Sampling

Input: CDF $F : \{0,1\}^n \to \mathbb{F}_m^E \cap [0,1]$
over number format $\mathbb{B} = (n, \gamma_{\mathbb{B}}, \phi_{\mathbb{B}})$
String $b \in \{0,1\}^{\leq n}$;
Floats $f_0, f_1 \in \mathbb{F}_m^E \cap [0,1]$
Output: Exact random variate $X \sim \hat{F}$

function CBS(F, b=ε, f_0=0, f_1=1)
  if $|b| = n$ ▷ *Base Case*
    return $\gamma_{\mathbb{B}}(\phi_{\mathbb{B}}(b))$ ▷ *Number in* $\overline{\mathbb{R}}_{\mathbb{B}}$
  $b' \leftarrow b01^{n-|b|-1}$; $f_2 \leftarrow F(\phi_{\mathbb{B}}(b'))$
  if $f_2 = f_1$ ▷ *Leaf*
    return CBS($F, b0, f_0, f_2$) ▷ *0*
  if $f_2 = f_0$ ▷ *Leaf*
    return CBS($F, b1, f_2, f_1$) ▷ *1*
  ▷ $i/k := (f_1 - f_2)/(f_1 - f_0)$
  $(i,k) \leftarrow$ ExactRatio(f_0, f_2, f_1)
  $z \leftarrow$ Bernoulli(i,k) ▷ *Refine Subtree*
  $(f_0', f_1') \leftarrow (z = 0)\,?\,(f_0, f_2) : (f_2, f_1)$
  return CBS(F, bz, f_0', f_1') ▷ z

Algorithm A7 Optimal Bernoulli Generation

Input: Integers i, k with $0 < i < k$
Output: Exact flip $X \sim$ Bernoulli(i/k)

function Bernoulli(i,k)
  while true do
    $i \leftarrow 2i$
    if $i = k$
      return RandBit() ▷ *Dyadic*
    else if $i > k$
      $b \leftarrow 1$
      $i \leftarrow i - k$
    else
      $b \leftarrow 0$
    if RandBit()
      return b

Algorithm A8 Computing Binary Expansion

Input: Integers i, k with $0 < i < k$
Output: Print concise binary expansion of i/k

function BinaryExpansion(i,k)
  while true do
    $i \leftarrow 2i$
    if $i = k$
      print 1 ▷ *Dyadic*
      print 0 **forever**
    else if $i > k$
      $b \leftarrow 1$
      $i \leftarrow i - k$
    else
      $b \leftarrow 0$
    print b

B Deferred Results in §2

This appendix studies the properties of random "uniform" floating-point numbers obtained by dividing integers. Goualard [28] provide a survey and case study of these algorithms in practice.

Proposition B.1. *Let $\mathbb{F}_m^E$ be a floating-point format and* $\mathrm{rnd} : \mathbb{R} \to \mathbb{F}_m^E \cup \{-\infty, +\infty\}$ *be its rounding function in round-to-nearest-even mode. Then, for any integer $\ell \in [0, 2^{E-1} - 2]$, the density of the set $\{\mathrm{rnd}(i/2^\ell) \mid i \in \{0, 1, \ldots, 2^\ell - 1\}\}$ within the set $(\mathbb{F}_m^E \cap [0, 1])$ of all floats in the unit interval is*

$$\frac{2^\ell}{2^m(2^{E-1}-1)+1} \quad \text{if } \ell \le m+1, \qquad \frac{2^m(\ell-m+1)+1}{2^m(2^{E-1}-1)+1} \quad \text{if } \ell > m+1. \tag{B.1}$$

The same result holds for the set $\{\mathrm{rnd}(\mathrm{rnd}(i)/\mathrm{rnd}(2^\ell)) \mid i \in \{0, 1, \ldots, 2^\ell - 1\}\}$. «

Proof. Let $S \coloneqq \left\{i/2^\ell \mid i \in \left\{0, 1, \ldots, 2^\ell - 1\right\}\right\}$, and $\mathrm{rnd}(A) \coloneqq \{\mathrm{rnd}(a) \mid a \in A\}$ be the rounding of a set $A \subseteq \mathbb{R}$. Then, $\mathrm{rnd}(S) \subseteq [0, 1] \cap \mathbb{F}_m^E$ because $S \subseteq [0, 1]$ and $\{0, 1\} \subseteq \mathbb{F}_m^E$. Hence, for the first claim, it suffices to show that $|\mathrm{rnd}(S)|/\left|[0, 1] \cap \mathbb{F}_m^E\right|$ equals to (B.1).

First, we compute $\left|[0, 1] \cap \mathbb{F}_m^E\right|$:

$$\begin{aligned} \left|[0, 1] \cap \mathbb{F}_m^E\right| &= \left|[0, 1) \cap \mathbb{F}_m^E\right| + 1 && \text{(B.2)} \\ &= (\#\text{ binades in } [0, 1)) \cdot (\#\text{ floats in each binade}) + 1 && \text{(B.3)} \\ &= (2^{E-1} - 1) \cdot 2^m + 1. && \text{(B.4)} \end{aligned}$$

Here, the number of binades in $[0, 1)$ is $1 + (2^{E-1} - 2) = 2^{E-1} - 1$ for three reasons: $[0, \omega) \cap \mathbb{F}_m^E$ forms precisely one binade, where $\omega \coloneqq 2^{-2^{E-1}+2}$ is the smallest positive normal float in $\mathbb{F}_m^E$; the smallest exponent in $[\omega, 1)$ is $-2^{E-1} + 2$; and the largest exponent in $[\omega, 1)$ is -1.

Next, we compute $|\mathrm{rnd}(S)|$. Suppose that $\ell \le m + 1$. Then, for each $j \in \{0, \ldots, \ell - 1\}$ and $i \in \{2^j, \ldots, 2^{j+1} - 1\}$, we have $i \cdot 2^{-\ell} = (i \cdot 2^{-j}) \cdot 2^{-\ell+j} \in \mathbb{F}_m^E$ because $1 \le i \cdot 2^{-j} < 2$, i is an $(m+1)$-bit unsigned integer (by $0 \le i < 2^\ell \le 2^{m+1}$), and $-\ell + j$ is greater than or equal to the exponent of ω (by $-\ell + j \ge -\ell \ge -2^{E-1} + 2$). This and $0 \in \mathbb{F}_m^E$ imply $\mathrm{rnd}(S) = S$ and the desired result:

$$|\mathrm{rnd}(S)| = |S| = 2^\ell. \tag{B.5}$$

Now, suppose that $\ell > m + 1$. Consider the partition of $S = S_1 \cup S_2$ given by $S_1 \coloneqq \{i \cdot 2^{-\ell} \mid i \in \{0, \ldots, 2^{m+1} - 1\}\}$ and $S_2 \coloneqq \{i \cdot 2^{-\ell} \mid i \in \{2^{m+1}, \ldots, 2^\ell - 1\}\}$. Then, $\mathrm{rnd}(S_1) = S_1$ by the above argument. For $\mathrm{rnd}(S_2)$, we claim that $\mathrm{rnd}(S_2) = [2^{-\ell+m+1}, 1] \cap \mathbb{F}_m^E$. This subclaim immediately implies the desired result:

$$|\mathrm{rnd}(S)| = |\mathrm{rnd}(S_1)| + |\mathrm{rnd}(S_2)| = |S_1| + \left|[2^{-\ell+m+1}, 1] \cap \mathbb{F}_m^E\right| = 2^{m+1} + (\ell - m - 1) \cdot 2^m + 1 \tag{B.6}$$

where the first equality holds by $\mathrm{rnd}(S_1) \cap \mathrm{rnd}(S_2) = \emptyset$ (since $\mathrm{rnd}(S_1) = S_1 \subseteq [0, 2^{-\ell+m+1})$). Hence, it suffices to show the above subclaim. We prove this claim in three steps.

(1) We have $\mathrm{rnd}(S_2) \subseteq [2^{-\ell+m+1}, 1] \cap \mathbb{F}_m^E$ because $S_2 \subseteq [2^{-\ell+m+1}, 1]$ and $\left\{2^{-\ell+m+1}, 1\right\} \subseteq \mathbb{F}_m^E$, where $2^{-\ell+m+1} \in \mathbb{F}_m^E$ is by $-\ell + m + 1 \ge -\ell \ge -2^{E-1} + 2$.
(2) We have $\mathrm{rnd}(S_2) \supseteq \{1\}$ because $\mathrm{rnd}((2^\ell - 1)/2^\ell) = \mathrm{rnd}(1 - 2^{-\ell}) = 1$, where the last equality is by $1 \in \mathbb{F}_m^E$, $2^{-\ell} \le \frac{1}{2}2^{-1-m}$, and the round-to-nearest-even mode of $\mathrm{rnd}(\cdot)$.
(3) We show $\mathrm{rnd}(S_2) \supseteq [2^{-\ell+m+1}, 1) \cap \mathbb{F}_m^E$. To prove this claim, we write $[2^{-\ell+m+1}, 1) \cap \mathbb{F}_m^E$ as

$$\left\{(1.b_1 \ldots b_m)_2 \cdot 2^{-\ell+m+j} \mid b_1, \ldots, b_m \in \{0, 1\}, j \in \{1, \ldots, \ell - m - 1\}\right\}. \tag{B.7}$$

Here, we have $(1.b_1 \ldots b_m)_2 \cdot 2^{-\ell+m+j} = ((1b_1 \ldots b_m)_2 \cdot 2^j) \cdot 2^{-\ell}$, where $(1b_1 \ldots b_m)_2 \cdot 2^j \in [2^{m+1}, 2^\ell - 1]$ by $m + j \ge m + 1$ and $m + 1 + j \le \ell$. Hence, $[2^{-\ell+m+1}, 1) \cap \mathbb{F}_m^E \subseteq S_2$, implying that $[2^{-\ell+m+1}, 1) \cap \mathbb{F}_m^E \subseteq \mathrm{rnd}(S_2)$.

The subclaim is thus established, completing the proof of the first claim.

For the second claim, the proof is exactly the same because (i) $\mathrm{rnd}(2^\ell) = 2^\ell$ by $0 \leq \ell \leq 2^{E-1} - 2$; and (ii) the above proof used only those $i \in \{0, \ldots, 2^\ell - 1\}$ such that $i = (1.b_1 \ldots b_m)_2 \cdot 2^j$ for some $b_1, \ldots, b_m \in \{0, 1\}$ and $j \in \{0, \ldots, \ell - 1\}$, which satisfy $\mathrm{rnd}(i) = i$. □

Remark B.2. Prop. B.1 describes a standard method [28] to generate a floating-point random variate from Uniform($[0, 1]$), and presents the proportion of floats in $[0, 1]$ covered by this method. We list some of the actual implementations of this method and show relevant details.

- In the GNU Standard C++ Library (libstdc++), the function `std::generate_canonical()` generates a 64-bit float with $\ell = 64$ and a 32-bit float with $\ell = 32$ (when invoked with `std::mt19937`),[6] covering 1.27% of 64-bit floats in $[0, 1]$ and 7.87% of 32-bit floats in $[0, 1]$. This function for 32-bit floats is similar to the first code on page 2 of Downey [15].
- In the GNU Scientific Library (GSL), the function `gsl_rng_uniform()` generates a 64-bit float with $\ell = 32$ (when invoked with `gsl_rng_mt19937`),[7] covering only 9.32×10^{-8}% of 64-bit floats in $[0, 1]$. This function corresponds to `uniform()` in Listing 1. The GSL, however, does not provide a 32-bit version of this function.
- In Python and SciPy, the functions `random.random()` and `scipy.stats.uniform.rvs()` generate a 64-bit float with $\ell = 53$,[8]covering 0.20% of 64-bit floats in $[0, 1]$. The `random` module and SciPy, however, do not provide a 32-bit version of this function.
- In NumPy and PyTorch, the functions `numpy.Generator.random()` and `torch.rand()` generate a 64-bit float with $\ell = 53$ and a 32-bit float with $\ell = 24$ (when invoked with `numpy.random.PCG64` for NumPy),[9] covering 0.20% of 64-bit floats in $[0, 1]$ and 1.57% of 32-bit floats in $[0, 1]$, respectively. «

[6] https://github.com/gcc-mirror/gcc/blob/releases/gcc-14.2.0/libstdc++-v3/include/bits/random.tcc#L3370
[7] https://github.com/ampl/gsl/blob/v2.7.0/rng/mt.c#L127
[8] Python: https://github.com/python/cpython/blob/v3.13.0/Modules/_randommodule.c#L191
SciPy: https://github.com/numpy/numpy/blob/v2.0.0/numpy/random/src/mt19937/mt19937.h#L58
[9] NumPy (64-bit): https://github.com/numpy/numpy/blob/v2.0.0/numpy/random/_common.pxd#L69
NumPy (32-bit): https://github.com/numpy/numpy/blob/v2.0.0/numpy/random/src/distributions/distributions.c#L20
PyTorch: https://github.com/pytorch/pytorch/blob/v2.3.0/aten/src/ATen/core/TransformationHelper.h#L85

Table 3. Comparison of coverage of floating-point numbers in the unit interval when generating "uniform" random variates using exact generation from the CDF `cdf_uniform_round_dn` (Page 6 of main text) and the usual division method [28]. Results are shown for the 8-bit binary number format $\mathbb{F}_2^5$, i.e., 5 exponent bits and 2 mantissa bits, which contains 60 floats in the interval $[0, 1)$. The 'No. of Observed Samples' column shows the empirical frequency with which each float was observed in a sample of 10,000,000 random variates generated using the exact and division method. The division method covers only 2/32 floats in the first 8 binades (left table, 0 and 0.0039062500), whereas the exact method covers all floats with the correct frequencies. For 32-bit or 64-bit floating-point systems, the division method covers an even smaller fraction of all the possible floats, whereas the exact method retains 100% coverage.

		No. of Observed Samples	
	Float	Exact Method	Division Method
0	0.0000000000000000	164	39385
1	0.0000152587890625	146	0
2	0.0000305175781250	173	0
3	0.0000457763671875	155	0
4	0.0000610351562500	145	0
5	0.0000762939453125	129	0
6	0.0000915527343750	166	0
7	0.0001068115234375	136	0
8	0.000122070312500	293	0
9	0.000152587890625	308	0
10	0.000183105468750	292	0
11	0.000213623046875	321	0
12	0.00024414062500	638	0
13	0.00030517578125	643	0
14	0.00036621093750	689	0
15	0.00042724609375	618	0
16	0.0004882812500	1197	0
17	0.0006103515625	1170	0
18	0.0007324218750	1175	0
19	0.0008544921875	1239	0
20	0.000976562500	2412	0
21	0.001220703125	2345	0
22	0.001464843750	2442	0
23	0.001708984375	2439	0
24	0.00195312500	4870	0
25	0.00244140625	4778	0
26	0.00292968750	4858	0
27	0.00341796875	4814	0
28	0.0039062500	9716	39080
29	0.0048828125	9657	0
30	0.0058593750	9801	0
31	0.0068359375	9872	0

		No. of Observed Samples	
	Float	Exact Method	Division Method
32	0.007812500	19499	39273
33	0.009765625	19440	0
34	0.011718750	19771	39027
35	0.013671875	19797	0
36	0.01562500	38848	39205
37	0.01953125	39399	39012
38	0.02343750	39118	38920
39	0.02734375	38898	39177
40	0.0312500	78148	78228
41	0.0390625	77943	77833
42	0.0468750	77378	77494
43	0.0546875	78279	77863
44	0.062500	156466	156536
45	0.078125	156020	156099
46	0.093750	156554	155971
47	0.109375	156471	157199
48	0.12500	313092	312150
49	0.15625	313487	312113
50	0.18750	313109	311700
51	0.21875	311921	312760
52	0.2500	625607	625439
53	0.3125	625895	624591
54	0.3750	624381	626462
55	0.4375	624009	624240
56	0.500	1250741	1249658
57	0.625	1249698	1251008
58	0.750	1249268	1250410
59	0.875	1248962	1249167

C Deferred Results in §4

Proposition 3.6. *A random variate generator X describes a random variable $X_\gamma : [0,1] \to \mathbb{R}$ over at most 2^n values, where $\Pr(X_\gamma = x) = \sum_{u \in \mathrm{dom}(X)} 2^{-|u|} \mathbf{1}[\gamma(X(u)) = x]$ for any $\gamma : \{0,1\}^n \to \mathbb{R}$.* «

Proof. Write $\mathrm{dom}(X) = \{u_1, u_2, \ldots\} \subset \{0,1\}^*$ and define $I_i := [(0.u_i)_2, (0.u_i\bar{1})_2)$. By the first property in (3.1) (prefix-free), the intervals $I_1, I_2, \ldots$ are pairwise disjoint. By the second property in (3.1) (exhaustive), the union of these disjoint intervals forms a measure one subset of $[0,1]$, since

$$\lambda(\cup_i I_i) = \textstyle\sum_i \lambda(I_i) = \sum_i \left[(0.u_i 1^\infty)_2 - (0.u_i 0^\infty)_2\right] \tag{C.1}$$

$$= \textstyle\sum_i \left[((0.u_i 0^\infty)_2 + 2^{-|u_i|}) - (0.u_i 0^\infty)_2\right] = \sum_i 2^{-|u_i|} = 1. \tag{C.2}$$

Then for each $\omega \in [0,1]$, let $X_\gamma(\omega) := \gamma(X(u_j))$ if there exists j such that $\omega \in I_j$, and arbitrarily otherwise, on the measure zero set $[0,1] \setminus \cup_i I_i$. With this construction, for any $x \in \mathbb{R}$ we have

$$\Pr(X_\gamma = x) = \Pr\left(\bigcup_{i=1}^{\infty} \{I_i \mid \gamma(X(u_i)) = x\}\right) \tag{C.3}$$

$$= \sum_{i=1}^{\infty} \lambda(I_i) \mathbf{1}[\gamma(X(u_i)) = x] = \sum_{i=1}^{\infty} 2^{-|u_i|} \mathbf{1}[\gamma(X(u_i)) = x]. \tag{C.4}$$

□

The remainder of this section proves the correctness (Cor. C.5) and entropy-optimality (Cor. C.6) of Algorithm 1, which together imply Theorem 4.6.

Proposition C.1. *For any $z, x, y \in [0,1]$ with $z = x + y$, let $z = (z_0.z_1z_2\ldots)_2$, $x = (x_0.x_1x_2\ldots)_2$, and $y = (y_0.y_1y_2\ldots)_2$ be concise binary expansions. Suppose $\ell \geq 0$ is any index. If $x_\ell y_\ell z_\ell \in \{000, 011, 101, 110\}$ and $z_j = 0$ for all $j > \ell$, then $x_j y_j = 00$ for all $j > \ell$.* «

Proof. Toward a contradiction, assume $x_k = 1$ or $y_k = 1$ for some $k > \ell$. Let $x' := (x_\ell.x_{\ell+1}\ldots)_2$, $y' := (y_\ell.y_{\ell+1}\ldots)_2$, and $z' := (z_\ell.z_{\ell+1}\ldots)_2 = z_\ell$. Then, $x_\ell + y_\ell = (x_\ell.0\ldots)_2 + (y_\ell.0\ldots)_2 < x' + y' < (x_\ell.1\ldots)_2 + (y_\ell.1\ldots)_2 = x_\ell + y_\ell + 2$, where the second $<$ is by the conciseness of $(x_\ell.x_{\ell+1}\ldots)_2$ and $(y_\ell.y_{\ell+1}\ldots)_2$. On the other hand, $x + y = z$ implies $x' + y' \in \{z', z' + (10.0)_2\} = \{z_\ell, z_\ell + 2\}$, where the $\in$ is by the conciseness of $(x_\ell.x_{\ell+1}\ldots)_2$ and $(y_\ell.y_{\ell+1}\ldots)_2$, and the $=$ is by assumption. Since $x_\ell + y_\ell \in \{z_\ell, z_\ell + 2\}$ by assumption, we get a contradiction. □

Theorem 4.2. *Let $z, x, y \in [0,1]$ satisfy $z = x + y$. Suppose $\ell \geq 0$ is any index such that $z_\ell = 1$ and $z_j = 0$ for all $j > \ell$, where $z = (z_0.z_1z_2\ldots)_2$, $x = (x_0.x_1x_2\ldots)_2$ and $y = (y_0.y_1y_2\ldots)_2$ are concise binary expansions. The binary expansions of x and y match exactly one of the following patterns:*

$$\begin{array}{ccccccc} & x_\ell & \ldots & x_{\ell'} & \ldots & \\ + & y_\ell & \ldots & y_{\ell'} & \ldots & \\ \hline = & z_\ell & \ldots & z_{\ell'} & \ldots & \end{array} \quad = \quad \begin{array}{c} 1\ 0\ \ldots\ 0\ \ldots \\ \hline \left[\begin{array}{c|c} 0 & 1 \\ 1 & 0 \end{array}\right] \left[\begin{array}{c} 0 \\ 0 \end{array}\right]^\infty \quad \text{(Pattern 4.2.1)} \\ \left[\begin{array}{c|c} 0 & 1 \\ 0 & 1 \end{array}\right] \left[\begin{array}{c|c} 0 & 1 \\ 1 & 0 \end{array}\right]^* \left[\begin{array}{c} 1 \\ 1 \end{array}\right] \left[\begin{array}{c} 0 \\ 0 \end{array}\right]^\infty \quad \text{(Pattern 4.2.2)} \\ \left[\begin{array}{c|c} 0 & 1 \\ 0 & 1 \end{array}\right] \left[\begin{array}{c|c} 0 & 1 \\ 1 & 0 \end{array}\right]^* \quad \text{(Pattern 4.2.3)} \end{array} \tag{4.2}$$

Here, each pattern is written in the style of regular expressions: $[R \mid R']$ denotes either R or R', $[R]^*$ denotes zero or more occurrences of R, and $[R]^\infty$ denotes the infinite occurrences of R. «

Proof. For each $j \geq 0$, write $z_j = x_j + y_j + c_j \pmod 2$, where $c_j \in \{0,1\}$ is the "carry" bit at location j of the binary addition. The three patterns in (4.2) correspond to three different cases of $(x_\ell x_{\ell+1}\ldots)$ and $(y_\ell y_{\ell+1}\ldots)$.

4.2.1 Assume $x_\ell y_\ell \in \{01, 10\}$. Since $x_\ell y_\ell z_\ell \in \{011, 101\}$ and $z_j = 0$ for all $j > \ell$, Prop. C.1 implies that $x_j y_j = 00$ for all $j > \ell$.

4.2.2 Assume $x_\ell y_\ell \in \{00, 11\}$ and $x_j y_j \neq 11$ for all $j > \ell$. We argue that $x_j y_j \neq 00$ for all $j > \ell$. For contradiction, assume $x_k y_k = 00$ for some $k > \ell$; let k be the smallest such index. Then, $c_{k-1} = 0$ holds. If $k = \ell + 1$, then $x_\ell + y_\ell + c_\ell = 0 \neq 1 = z_\ell \pmod 2$, a contradiction. If $k > \ell + 1$, then $x_{k-1} y_{k-1} = 11$ must hold by $x_{k-1} + y_{k-1} + c_{k-1} = z_{k-1} = 0 \pmod 2$, $c_{k-1} = 0$, and the minimality of k. This contradicts to $x_j y_j \neq 11$ for all $j > \ell$.

4.2.3 Assume $x_\ell y_\ell \in \{00, 11\}$ and $x_k y_k = 11$ for some $k > \ell$. Let k be the smallest such index. Since $x_k y_k z_k = 110$ and $z_j = 0$ for all $j > k$, Prop. C.1 implies that $x_j y_j = 00$ for all $j > k$. By the minimality of k, it suffices show that $x_j y_j \neq 00$ for all $j \in (\ell, k)$. Toward a contradiction, assume not. Then, $x_{j'} y_{j'} z_{j'} = 000$ for some $j' \in (\ell, k)$. Since $z_j = 0$ for all $j > j'$, Prop. C.1 implies that $x_j y_j = 00$ for all $j > j'$. This contradicts to $x_k y_k = 11$ and $k > j'$. □

Remark C.2. In Theorem 4.2, if $z = 1$ and $x \in (0, 1)$, then $x_1 y_1 \neq 00$ (otherwise $x + y < 1$). «

Theorem 4.4. *Let $z, x, y \in [0, 1]$ satisfy $z = x + y$. Suppose $0 \leq \ell < \ell'$ are two indexes such that $z_\ell = 1, z_{\ell'} = 1$ and $z_j = 0$ for $\ell + 1 \leq j \leq \ell' - 1$, where z_j, x_j and y_j are defined as in Theorem 4.2. The binary expansions of x and y between locations ℓ and ℓ' match exactly one of three possible patterns:*

$$
\begin{array}{ccccccc}
 & & x_\ell & \dots & x_{\ell'} & & \\
+ & & y_\ell & \dots & y_{\ell'} & = & \\
\hline
= & & z_\ell & \dots & z_{\ell'} & = &
\end{array}
\quad
\begin{cases}
\left[\begin{array}{c|c} 0 & 1 \\ 1 & 0 \end{array}\right] \left[0\right]^{\ell'-\ell-1} \left[\begin{array}{c|c|c} 0 & 0 & 1 \\ 0 & 1 & 0 \end{array}\right] & \text{(Pattern 4.3.1)} \\
\left[\begin{array}{c|c} 0 & 1 \\ 0 & 1 \end{array}\right] \left[\begin{array}{c|c} 0 & 1 \\ 1 & 0 \end{array}\right]^{\ell'-\ell-1} \left[1\right] & \text{(Pattern 4.3.2)} \\
\left[\begin{array}{c|c} 0 & 1 \\ 0 & 1 \end{array}\right] \left[\begin{array}{c|c} 0 & 1 \\ 1 & 0 \end{array}\right]^{k_1} \left[1\right] \left[0\right]^{k_2} \left[\begin{array}{c|c|c} 0 & 0 & 1 \\ 0 & 1 & 0 \end{array}\right] & \text{where } k_1 + k_2 = \ell' - \ell - 2 \\
 & \text{(Pattern 4.3.3)}
\end{cases}
\quad
\begin{array}{ccccccc}
 & & 1 & & 0 & \dots & 0 \\
\hline
 & & 1 & & 0 & \dots & 0
\end{array}
\qquad (4.3)
$$

«

Proof. For each $j \geq 0$, write $z_j = x_j + y_j + c_j \pmod 2$, where $c_j \in \{0, 1\}$ is the "carry" bit at location j of the binary addition. The three patterns in (4.3) correspond to three different cases of the final bits $x_{\ell'} y_{\ell'}$ and initial bits $x_\ell y_\ell$.

4.3.2 Assume $x_{\ell'} y_{\ell'} = 11$. We first prove that $c_j = 1$ and $x_j y_j \in \{01, 10\}$ for each $j \in \{\ell + 1, \dots, \ell' - 1\}$, by induction. For the base case ($j = \ell' - 1$), $x_{\ell'} y_{\ell'} = 11$ implies $c_{\ell'-1} = 1$. Since $x_{\ell'-1} + y_{\ell'-1} + c_{\ell'-1} = z_{\ell'-1} = 0 \pmod 2$, we must have $x_{\ell'-1} y_{\ell'-1} \in \{01, 10\}$. For the inductive case ($\ell + 1 \leq j < \ell' - 1$), the induction hypothesis that $x_{j+1} y_{j+1} \in \{01, 10\}$ and $c_{j+1} = 1$ implies $c_j = 1$. But $x_j + y_j + c_j = z_j = 0 \pmod 2$ again implies $x_j y_j \in \{01, 10\}$.

We next prove $x_\ell y_\ell \in \{00, 11\}$. Since $x_\ell + y_\ell + c_\ell = z_\ell = 1 \pmod 2$, it suffices to show $c_\ell = 1$. If $\ell' = \ell + 1$ then $x_{\ell'} y_{\ell'} = 11$ implies this. If $\ell' > \ell + 1$ then $c_{\ell+1} = 1$ and $x_{\ell+1} y_{\ell+1} \in \{01, 10\}$ imply this.

4.3.1 Assume $x_{\ell'} y_{\ell'} \neq 11$ and $x_\ell y_\ell \in \{01, 10\}$. From $x_{\ell'} y_{\ell'} \in \{00, 01, 10\}$, we have $c_{\ell'-1} = 0$. This implies $\hat{z} = \hat{x} + \hat{y}$ for $\hat{z} := (z_0.z_1 \dots z_{\ell'-1} 0 \dots)_2$, $\hat{x} := (x_0.x_1 \dots x_{\ell'-1} 0 \dots)_2$, and $\hat{y} := (y_0.y_1 \dots y_{\ell'-1} 0 \dots)_2$. Since $z_j = 0$ for all $\ell < j < \ell'$, we can apply Theorem 4.2 to $(\hat{z}, \hat{x}, \hat{y}, \ell)$. In the theorem, only Pattern 4.2.1 matches because $x_\ell y_\ell z_\ell \in \{011, 101\}$. This yields the desired pattern on $(x_{\ell+1} y_{\ell+1}, \dots, x_{\ell'-1} y_{\ell'-1})$ described in Pattern 4.3.1.

4.3.3 Assume $x_{\ell'} y_{\ell'} \neq 11$ and $x_\ell y_\ell \in \{00, 11\}$. As in the previous case, we have $c_{\ell'-1} = 0$ and $\hat{z} = \hat{x} + \hat{y}$, where $\hat{z}$, $\hat{x}$, and $\hat{y}$ are defined as before. Since $z_j = 0$ for all $\ell < j < \ell'$, we can apply Theorem 4.2 to $(\hat{z}, \hat{x}, \hat{y}, \ell)$. In the theorem, only Pattern 4.2.3 matches because $x_\ell y_\ell z_\ell \in \{001, 111\}$ and $\hat{x}$ and $\hat{y}$ have trailing zeros in their binary expansions. This yields the desired pattern on $(x_{\ell+1} y_{\ell+1}, \dots, x_{\ell'-1} y_{\ell'-1})$ described in Pattern 4.3.3. □

Proposition C.3. *In GenOpt(p, b, ℓ), the random number of loop iterations has a least upper bound $j \geq 1$ if and only if j is the smallest integer such that $p_{\ell+j}^{b0} p_{\ell+j}^{b1} = 11$.* «

Proof. If the algorithm reaches step j, then it exits at this step with probability one if and only if $p^{b0}_{\ell+j}p^{b1}_{\ell+j} = 11$ (by case analysis); so j is an upper bound on the number of loop iterations. It is the least upper bound if and only if the algorithm reaches step j with nonzero probability, which holds if and only if there is no $j' < j$ such that $p^{b0}_{\ell+j'}p^{b1}_{\ell+j'} = 11$ (otherwise it would exit at step j'). □

Theorem C.4. *Let p be a binary-coded probability distribution and fix an integer $n \geq 1$. Let B_n denote the first n bits (stored in variable b) generated by* GenOpt(p) *and C_n the number of coin flips (stored in variable ℓ) made up to and including the point when the n-th bit is generated. Then* $\Pr(B_n = b, C_n = \ell) = 2^{-\ell}\mathbf{1}[[p(b)]_\ell = 1]$. «

Proof. The proof is by induction on the number of recursive calls n to GenOpt.

Base Case. Suppose $n = 1$. Consider the invocation GenOpt$(p, b = \varepsilon, \ell = 0)$.

Case: $(p^0, p^1) = (1, 0)$. In the concise binary expansion, we have $(p^0_0, p^1_0) = (1, 0)$. Then 0 is generated with probability 1, the loop is never entered, and exactly 0 flips are made. An analogous result holds for the symmetric case $(p^0, p^1) = (0, 1)$.

Case: $p^0 \in (0, 1)$. Fix $\ell \geq 1$ is such that $p^0_\ell = 1$. Since $p^0 + p^1 = 1$, $p^0_1 p^1_1 \neq 00$. Theorem 4.2 establishes that $p^0_j p^1_j \in \{01, 10\}$ for all $j = 1, \ldots, \ell - 1$. Let x_j denote the outcome of RandBit at iteration j. At iteration j, the loop continues if and only if $p^0_j p^1_j = 01 \wedge x_j = 0$ or $p^0_j p^1_j = 10 \wedge x_j = 1$. Therefore, line 9 occurs after exactly i coin flips if and only if $x_j = p^0_j$ $(j = 1, \ldots, \ell - 1)$ and $x_\ell = 1$; which by independence of RandBit occurs with probability $2^{-\ell}$. Further, if $p^0_\ell = 0$, then line 9 at iteration ℓ is never entered, so 0 cannot possibly be generated with exactly ℓ flips. The case of $p^1 \in (0, 1)$ is similar.

Inductive Case. Assume the claim holds for $n > 1$. Consider the recursive call GenOpt(p, b, ℓ), where $b \in \{0, 1\}^n$ is the string generated using exactly ℓ coin flips. From the inductive hypothesis, $p^b_\ell = 1$ and the probability of the current execution path is $2^{-\ell}$. Let $\ell' = \min_{i>\ell}\{p^b_i = 1\}$ be the index (possibly infinite) of the next 1 in the expansion of p^b. We analyze the event that the next generated bit at this stage of the recursion is 0, which means the overall generated string is $b0$ (the analysis for $b1$ is entirely symmetric to $b0$).

Case: $\ell' < \infty$. Consider the three patterns from Theorem 4.4 for the binary expansions $(p^{b0}_\ell, p^{b0}_{\ell+1}, \ldots)$ and $(p^{b1}_\ell, p^{b0}_{\ell+1}, \ldots)$. Suppose Pattern 4.3.1 is matched. If $p^{b0}_\ell p^{b1}_\ell = 10$, then line 3 ensures that $b0$ is generated with 0 additional flips, so the path probability remains $2^{-\ell}$ as desired. Suppose Pattern 4.3.2 or 4.3.2 are matched, so the loop is entered. Let $k \in \{1, \ldots, \ell' - \ell\}$ be such that $p^{b0}_{\ell+k} = 1$. By an analogous argument to the base case, the probability of exiting at line 9 after k loop iterations (i.e., k additional flips) is precisely 2^{-k}, which gives $\ell + k$ flips overall with path probability $2^{-(\ell+k)}$.

Case: $\ell' = \infty$. Pattern 4.2 shows the bit configuration. If $p^{b0}_\ell p^{b1}_\ell \in \{10, 01\}$ the loop is not entered, as in Pattern 4.3.1. Otherwise if $p^{b0}_\ell p^{b1}_\ell = 00$, loop is entered as in Pattern 4.3.2 and 4.3.3.

Finally, we prove that every index $k \in \mathbb{N}$ with $p^{b0}_k = 1$ has a positive probability of being encountered in an execution path of a recursive call GenOpt(p, b, ℓ), for some $\ell \geq 0$. From the inductive hypothesis, every ℓ such that $z_\ell = 1$ is encountered with probability $2^{-\ell} > 0$. It suffices to prove that all the 1 bits among $p^{b0}_\ell, p^{b0}_{\ell+1}, \ldots, p^{b0}_{\ell'}$ are selected with positive probability.

Case: the loop is entered. By Prop. C.3 it suffices to prove there are no 1 bits after a loop index j such that $p^{b0}_j p^{b1}_j = 11$. If $\ell' < \infty$, apply Theorem 4.4 (Pattern 4.3.3) to conclude. If $\ell' = \infty$, apply Theorem 4.2 (Pattern 4.2) to conclude.

Case: the loop is not entered. If $\ell' < \infty$, apply Theorem 4.4 (Pattern 4.3.1) to conclude that all the skipped bits are 0. If $\ell' = \infty$, we have $p^{b0}_\ell p^{b1}_\ell = 10$ (wlog). If $p^{b0} = p^b = (1.000\ldots)_2$,

then $\ell = 0$, and the conclusion is immediate. Otherwise $p^{b0} < 1$. Assume for a contradiction there is a minimal $j > \ell$ such that $p_j^{b0} = 1$ and $p_i^{b0} = 0$ for $\ell < i < j$. As $p_j^b = 0 = 1 + p_j^{b1} + c_j \pmod 2$ it follows that $c_{j-1} = 1$. But $p_{j-1}^b = 0 = 0 + p_{j-1}^{b1} + c_{j-1} \pmod 2$ so $c_{j-2} = 1$. Apply repeatedly to conclude that $c_\ell = 1$. But $p_\ell^b = 1 = 1 + 0 + 1 \pmod 2 = 0$, a contradiction.

As b was arbitrary, the statement holds for all $b' \in \{0,1\}^{n+1}$. □

Corollary C.5. *For each $n \geq 0$ and $b \in \{0,1\}^n$ the probability GenOpt(p) generates a string matching $b\{0,1\}^*$ is $p(b)$.* «

Proof. Define B_n and C_n as in the statement of Theorem C.4. Then

$$\Pr(B_n = b) = \sum_{\ell=0}^{\infty} \Pr(B_n = b, C_n = \ell) = \sum_{\ell=0}^{\infty} 2^{-\ell} \mathbf{1}[[p(b)]_\ell = 1] = \sum_{\ell=0}^{\infty} 2^{-\ell} [p(b)]_\ell = p(b). \tag{C.5}$$

□

Corollary C.6. *For every binary-coded distribution p and integer $n \geq 1$, GenOpt defines an entropy-optimal generator for the discrete distribution $P_n := \{b \mapsto p(b) \mid b \in \{0,1\}^n\}$, i.e., its average number of random coin flips C_n is minimal among all exact sampling algorithms for P_n.* «

Proof. Define B_n and C_n as in the statement of Theorem C.4. Then

$$\mathbb{E}[C_n] = \sum_{\ell=0}^{\infty} \ell \cdot \Pr(C_n = \ell) = \sum_{\ell=0}^{\infty} \ell \left(\sum_{b \in \{0,1\}^n} \Pr(C_n = \ell, B_n = b) \right) \tag{C.6}$$

$$= \sum_{\ell=0}^{\infty} \ell \left(\sum_{b \in \{0,1\}^n} 2^{-\ell} [p(b)]_\ell \right) \tag{C.7}$$

$$= \sum_{b \in \{0,1\}^n} \sum_{\ell=0}^{\infty} \ell \cdot 2^{-\ell} [p(b)]_\ell \tag{C.8}$$

which is precisely the Knuth-Yao lower bound. It follows from [37] that $H(P_n) \leq \mathbb{E}[C_n] \leq H(P_n)+2$, where H is the binary entropy function. □

Theorem 4.6. *Let $p : \{0,1\}^* \to [0,1]$ be a binary-coded probability distribution. For each $n \in \mathbb{N}$, Algorithm 1 generates a string $B_1 \ldots B_n \sim p_n$ (stored as a prefix of b) and is entropy-optimal for p_n.* «

Proof. This result is a restatement of Cors. C.5 and C.6. □

Implementation of Algorithm 1 using Lazy Computation. Algorithm 1, which generates a stream of random bits from a binary-coded probability distribution, can be directly implemented in a programming language that supports lazy computation. Listing 3 shows one such implementation in Haskell, using a guarded recursive call on line 40 that occurs in the data constructor (:) for lists

Lst 3. Implementation of Algorithm 1, which generates a stream of random bits from a binary-coded probability distribution, using lazy computation with a guarded recursive call in the Haskell programming language.

```haskell
type Bit = Int
type BinaryString = [Bit]
type BinaryCodedDist = [Bit] -> Float

-- Obtain a fair random bit from the entropy source.
randBit :: Bit

-- Extract bit from a float (Algorithms 3 and 4).
extractBit :: Float -> Int -> Bit

-- Generate the next random bit from the binary coded distribution.
-- Returns the generated bit and the updated number of calls to randBit.
generateNextBit :: (BinaryCodedDist) -> BinaryString -> Int -> (Bit, Int)
generateNextBit p b l = do
  let pb0 = p (0:b)
  let pb1 = p (1:b)
  let bit0 = extractBit pb0 l
  let bit1 = extractBit pb1 l
  case (bit0, bit1) of
    (1, 0)    -> (0, l)
    (0, 1)    -> (1, l)
    otherwise -> do
      loop l
      where loop j = do
              let x = randBit
              let j' = j + 1
              let bit0 = extractBit pb0 j'
              let bit1 = extractBit pb1 j'
              if      x == 0 && bit0 == 1 then (0, j')
              else if x == 1 && bit1 == 1 then (1, j')
              else    loop j'

-- Overall recursive function.
generate :: (BinaryCodedDist) -> BinaryString
generate p = generate_ [] 0
    where
        generate_ :: BinaryString -> Int -> BinaryString
        generate_ b l =
            let (x, l') = generateNextBit p b l
            in x : (generate_ (x : b) (l+l'))
```

D Deferred Results in §5

Proposition 5.8. *The ordering $<_{\mathbb{F}_m^E}$ induced by $\phi_{\mathbb{F}_m^E}$ (5.5) guarantees that $\gamma_{\mathbb{F}_m^E} : (\{0,1\}^n, <_{\mathbb{F}_m^E}) \to (\overline{\mathbb{R}}, <_{\overline{\mathbb{R}}})$ is monotonic. That is, for any distinct $b, b' \in \{0,1\}^{1+E+m}$ such that $\gamma_{\mathbb{F}_m^E}(\{b,b'\}) \notin \{\{0\},\{\bot\}\}$, the following are equivalent: $\phi^{-1}_{\mathbb{F}_m^E}(b) <_{\text{dict}} \phi^{-1}_{\mathbb{F}_m^E}(b') \iff b <_{\mathbb{F}_m^E} b' \iff \gamma_{\mathbb{F}_m^E}(b) <_{\overline{\mathbb{R}}} \gamma_{\mathbb{F}_m^E}(b')$.* «

Proof. By the definition of $<_{\mathbb{F}_m^E}$, it suffices to show the following: for all $b, b' \in \{0,1\}^{1+E+m}$ such that $b \neq b'$ and $\gamma_{\mathbb{F}_m^E}(\phi_{\mathbb{F}_m^E}(\{y,y'\})) \neq \{0\},\{\bot\}$,

$$b <_{\text{dict}} b' \iff \gamma_{\mathbb{F}_m^E}(\phi_{\mathbb{F}_m^E}(b)) <_{\overline{\mathbb{R}}} \gamma_{\mathbb{F}_m^E}(\phi_{\mathbb{F}_m^E}(b')). \tag{D.1}$$

Recall that $\phi_{\mathbb{M}_{E+m}}$ and $\phi_{\mathbb{F}_m^E}$ are defined as follows: for $b_0 \dots b_{E+m} \in \{0,1\}^{1+E+m}$,

$$\phi_{\mathbb{M}_{E+m}}(b_0 \dots b_{E+m}) = \begin{cases} 1\bar{b}_1 \dots \bar{b}_{E+m} & \text{if } b_0 = 0 \\ 0 b_1 \dots b_{E+m} & \text{if } b_0 = 1, \end{cases} \tag{D.2}$$

$$\phi_{\mathbb{F}_m^E}(b_0 \dots b_{E+m}) = \begin{cases} \phi_{\mathbb{M}_{E+m}}\big((b_0 \dots b_{E+m})_2 + (2^m - 1)\big) & \text{if } b_0 \dots b_{E+m} \leq_{\text{dict}} 11^E 0^m \\ b_0 \dots b_{E+m} & \text{if } b_0 \dots b_{E+m} >_{\text{dict}} 11^E 0^m. \end{cases} \tag{D.3}$$

Using this mapping, we sequentially compute two bit strings $(b)_2 + (2^m - 1), \phi_{\mathbb{F}_m^E}(b) \in \{0,1\}^{1+E+m}$ and one extended real $\gamma_{\mathbb{F}_m^E}(\phi_{\mathbb{F}_m^E}(b)) \in \overline{\mathbb{R}}$ for each $b \in \{0,1\}^{1+E+m}$:

b			$(b)_2 + (2^m - 1)$			$\phi_{\mathbb{F}_m^E}(b)$			$\gamma_{\mathbb{F}_m^E}(\phi_{\mathbb{F}_m^E}(b))$
b_0	$b_1 \dots b_E$	$b_{E+1} \dots b_{E+m}$	b'_0	$b'_1 \dots b'_E$	$b'_{E+1} \dots b'_{E+m}$	s	$e_E \dots e_1$	$f_1 \dots f_m$	$r \in \overline{\mathbb{R}}$
0	00…00	00…000	0	00…00	11…111	1	11…11	00…000	$-\infty$
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
0	11…11	00…000	0	11…11	11…111	1	00…00	00…000	0
0	11…11	00…001	1	00…00	00…000	0	00…00	00…000	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
1	11…10	00…001	1	11…11	00…000	0	11…11	00…000	$+\infty$
1	11…10	00…010	1	11…11	00…001	0	11…11	00…001	$\bot$
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
1	11…11	00…000	1	11…11	11…111	0	11…11	11…111	$\bot$
1	11…11	00…001				1	11…11	00…001	$\bot$
⋮	⋮	⋮		unimportant		⋮	⋮	⋮	⋮
1	11…11	11…111				1	11…11	11…111	$\bot$

Using this calculation and the definition of $\gamma_{\mathbb{F}_m^E}$ and $<_{\overline{\mathbb{R}}}$, one can check that (D.1) indeed holds. □

Proposition 5.13. *Let $F : \{0,1\}^n \to \mathbb{F}_m^E \cap [0,1]$ be a CDF over the unsigned integer format $\mathbb{U}_n$ and P_F the corresponding discrete distribution from Remark 5.11. The function $p_F : \{0,1\}^* \to [0,1]$ defined below is a binary-coded probability distribution that satisfies $p_F(b) = P_F(b)$ for all $b \in \{0,1\}^n$:*

$$p_F(b) \coloneqq F(b1^{n-|b|}) -_{\mathbb{R}} F((b0^{n-|b|})^-) \qquad (b \in \{0,1\}^{\leq n}), \tag{5.7}$$

$$p_F(bb') \coloneqq p_F(b)\mathbf{1}[b' = 0 \dots 0] \qquad (b \in \{0,1\}^n; b' \in \{0,1\}^+), \tag{5.8}$$

where $x^- \coloneqq \text{pred}_{\text{dict}}(x)$ for any $x \in \{0,1\}^n \setminus \{0^n\}$, with the convention that $F((0^n)^-) \coloneqq 0$. «

Proof. We first argue that p_F is a binary-coded distribution. If $|b| = 0$, then (5.7) gives

$$p_F(\varepsilon) = F(1^n) -_{\mathbb{R}} F((0^n)^-) = 1 - 0 = 1. \tag{D.4}$$

If $1 \le |b| \le n$, then (5.7) again gives

$$p_F(b0) + p_F(b1) = F(b01^m) - F((b00^m)^-) + F(b11^m) - F(\underbrace{(b10^m)^-}_{=b01^m}) \tag{D.5}$$

$$= F(b11^m) - F((b00^m)^-) = F(b1^{m+1}) - F((b0^{m+1})^-) = p_F(b), \tag{D.6}$$

where $m := n - |b| - 1$. Next consider $p_F(bb')$ where $|b| = n$ and $b' \in \{0,1\}^+$. If $b' = 0\ldots0$, (5.8) gives $p(bb'0) + p(bb'1) = p(b) + 0 = p(b) = p(bb')$. Otherwise $b' \neq 0\ldots0$ so (5.8) gives $p(bb'0) + p(bb'1) = 0 + 0 = 0 = p(bb')$.

Finally, we prove that $P_F(b) = p_F(b)$ for all $b \in \{0,1\}^n$, where P_F. From (5.8), if $b = 0^n$ then $p_F(b) = F(b) = P_F(b)$. Otherwise $p_F(b) = F(b) - F(\mathrm{pred}_{\mathrm{dict}}(b)) = P_F(b)$, which follows from $\mathrm{pred}_{\mathrm{dict}} = \mathrm{pred}_{\mathbb{U}_n}$. □

Proposition 5.14. *Let $F : \{0,1\}^n \to \mathbb{F}_m^E \cap [0,1]$ be a CDF over a number format $\mathbb{B} = (n, \gamma_{\mathbb{B}}, \phi_{\mathbb{B}})$. Then $\widetilde{F} := F \circ \phi_{\mathbb{B}}$ is a CDF over the unsigned integer format $\mathbb{U}_n = (n, (\cdot)_2, \mathrm{id})$ from Example 5.4.* «

Proof. As $\phi_{\mathbb{U}_n} = \mathrm{id}$, for the minimal element: $\widetilde{F}(\phi_{\mathbb{U}_n}(0^n)) = \widetilde{F}(0^n) = F(\phi_S(0^n)) \ge 0$. For the maximal element: $\widetilde{F}(\phi_{\mathbb{U}_n}(1^n)) = \widetilde{F}(1^n) = F(\phi_S(1^n)) = 1$. Since F is monotonically non-decreasing and ϕ monotonically increasing, so is their composition $\widetilde{F} \equiv F \circ \phi_S$. That is, if $x <_{\mathbb{U}_n} x'$, then (5.1) implies that $\phi_S(x) <_{\mathbb{F}_m^E} \phi_S(x')$, so $\widetilde{F}(x) = F(\phi_S(x)) \le_{\mathbb{F}_m^E} F(\phi_S(x')) = \widetilde{F}(x')$. □

Theorem 5.16. *Suppose $x, x' \in \mathbb{F}_m^E$ satisfy $0 < x -_{\mathbb{R}} x' < 1$, and consider any integer $\ell \ge 1$. Let $\beta = (n_1, n_2, n_{\mathrm{hi}}, n_{\mathrm{lo}}, b_1, b_2, g_{\mathrm{hi}}, g_{\mathrm{lo}})$ be the output of ExtractBitPreproc1(x, x') (Algorithm 3), and let b' be the output of ExtractBit(β, ℓ) (Algorithm 4). Then,*

$$x -_{\mathbb{R}} x' = \Big(0.\underbrace{\boxed{b_1 \ldots b_1}}_{n_1 \text{ bits}}\underbrace{\boxed{g_{\mathrm{hi}}}}_{n_{\mathrm{hi}} \text{ bits}}\underbrace{\boxed{b_2 \ldots b_2}}_{n_2 \text{ bits}}\underbrace{\boxed{g_{\mathrm{lo}}}}_{n_{\mathrm{lo}} \text{ bits}}\Big)_2 \tag{5.9}$$

and b' is the ℓ-th digit of $x -_{\mathbb{R}} x'$ in binary expansion. Also, all intermediate values appearing in both algorithms are representable as $(1 + E + m)$-bit signed integers. «

Proof. We prove the claims for $x < 1$ and $x = 1$ separately.

First, assume $x < 1$. Then, $(\hat{e}, f)$ and $(\hat{e}', f')$ computed in lines 2–8 of Algorithm 3 satisfy

$$\hat{e}, \hat{e}' \le -1, \quad f = (f_0 \ldots f_m)_2, \quad f' = (f'_0 \ldots f'_m)_2, \quad x = 2^{\hat{e}} \cdot (f/2^m), \quad x' = 2^{\hat{e}'} \cdot (f'/2^m), \tag{D.7}$$

where $f_0 := \mathbf{1}[e > 0]$ and $f'_0 := \mathbf{1}[e' > 0]$. Here, $\le$ is by $x, x' \in [0,1)$, the first two $=$ are by the definition of f_0 and f'_0, and the last two $=$ are by the definition of $\mathbb{F}_m^E$ and $x, x' \in [0,\infty) \cap \mathbb{F}_m^E$. We note that $f_0 = 1$ if x is a normal float, and $f_0 = 0$ if x is a subnormal float; the same hold for f'_0, x'.

Using the previous observation, we show Eq. (5.9) for $x < 1$ by case analysis on $\hat{e} - \hat{e}'$.

Case 1. $(\hat{e} - \hat{e}' \le m + 1)$. In this case, we have

$$\begin{array}{rcl} & & \overbrace{0.\ 0\ \ldots\ 0}^{-\hat{e}-1 \text{ bits}}\ \overbrace{\boxed{f_0\ \ldots\ \ldots\ \ldots\ \ldots\ f_m}}^{f \in \mathbb{Z},\ m+1 \text{ bits}}\ \overbrace{0\ \ldots\ 0}^{\hat{e}-\hat{e}' \text{ bits}} \\ x = & & \\ - \quad x' = & & \underbrace{0.\ 0\ \ldots\ 0\ 0\ \ldots\ 0}_{-\hat{e}'-1 \text{ bits}}\ \underbrace{\boxed{f'_0\ \ldots\ f'_{m-(\hat{e}-\hat{e}')}}}_{(m+1)-(\hat{e}-\hat{e}') \text{ bits},\ f'_{\mathrm{hi}} \in \mathbb{Z}}\ \underbrace{\boxed{f'_{m-(\hat{e}-\hat{e}')+1}\ \cdots\ f'_m}}_{\hat{e}-\hat{e}' \text{ bits},\ f'_{\mathrm{lo}} \in \mathbb{Z}} \end{array} \tag{D.8}$$

$$x - x' = \begin{cases} \underbrace{0.\ 0\ \ldots\ 0}_{-\hat{e}-1 \text{ bits}}\ \underbrace{\boxed{f - f'_{\mathrm{hi}}}}_{m+1 \text{ bits}}\ \underbrace{\boxed{0}}_{\hat{e}-\hat{e}' \text{ bits}} & \text{if } f'_{\mathrm{lo}} = 0 \\ \underbrace{0.\ 0\ \ldots\ 0}_{-\hat{e}-1 \text{ bits}}\ \underbrace{\boxed{f - f'_{\mathrm{hi}} - 1}}_{m+1 \text{ bits}}\ \underbrace{\boxed{2^{\hat{e}-\hat{e}'} - f'_{\mathrm{lo}}}}_{\hat{e}-\hat{e}' \text{ bits}} & \text{if } f'_{\mathrm{lo}} > 0 \end{cases}$$

Here, the equalities on x, x', f are by Eq. (D.7), and the equalities on $f'_{\mathrm{hi}}, f'_{\mathrm{lo}}$ are by lines 9–10 of Algorithm 3 and the following: $0 \le \hat{e} - \hat{e}' \le m + 1$ (by $x \ge x'$) and $\hat{e} - \hat{e}' \le E + m$ (by $E \ge 1$). Further, the equality on $x - x'$ is by earlier equalities and the following: if $f'_{\mathrm{lo}} > 0$, then $f - f'_{\mathrm{hi}} \ge 1$ (since $f_0 = 1$ and $\hat{e} - \hat{e}' \ge 1$ must hold) and $0 \le 2^{\hat{e}-\hat{e}'} - f'_{\mathrm{lo}} < 2^{\hat{e}-\hat{e}'}$.

Based on (D.8), we can check that β computed in lines 11–16 of Algorithm 3 satisfies the following: $(n_1, n_{\mathrm{hi}}, n_2, n_{\mathrm{lo}}) = (-\hat{e} - 1, m + 1, 0, \hat{e} - \hat{e}')$ are the numbers of bits shown in the last line of (D.8); $(b_1, b_2) = (0, \mathbf{1}[f'_{\mathrm{lo}} > 0])$ are the bits of $x - x'$ in the n_1 and n_2 parts; and $(g_{\mathrm{hi}}, g_{\mathrm{lo}})$ are the values of $x - x'$ in the n_{hi} and n_{lo} parts (i.e., the boxed values in the last line of (D.8)). Hence, the last line of (D.8) implies (5.9), as desired.

Case 2. $(\hat{e} - \hat{e}' > m + 1)$. In this case, we have

$$\begin{array}{rcl} & & \overbrace{0.\ 0\ \ldots\ 0}^{-\hat{e}-1 \text{ bits}}\ \overbrace{\boxed{f_0\ \ldots\ f_m}}^{f \in \mathbb{Z},\ m+1 \text{ bits}}\ \ 0\ \ \ldots\ \ 0\ \ 0\ \ \ldots\ \ 0 \\ x = & & \\ - \quad x' = & & \underbrace{0.\ 0\ \ldots\ 0\ 0\ \ \ldots\ \ 0\ \ \ \ 0\ \ \ldots\ \ 0}_{-\hat{e}'-1 \text{ bits}}\ \underbrace{\boxed{f'_0\ \ \ldots\ \ f'_m}}_{m+1 \text{ bits},\ f'_{\mathrm{lo}} \in \mathbb{Z}} \end{array} \tag{D.9}$$

$$x - x' = \begin{cases} \underbrace{0.\ 0\ \ldots\ 0}_{-\hat{e}-1 \text{ bits}}\ \underbrace{\boxed{f}}_{m+1 \text{ bits}}\ \underbrace{0\ \ \ldots\ \ 0}_{(\hat{e}-\hat{e}')-(m+1) \text{ bits}}\ \underbrace{\boxed{0}}_{m+1 \text{ bits}} & \text{if } f'_{\mathrm{lo}} = 0 \\ \underbrace{0.\ 0\ \ldots\ 0}_{-\hat{e}-1 \text{ bits}}\ \underbrace{\boxed{f - 1}}_{m+1 \text{ bits}}\ \underbrace{1\ \ \ldots\ \ 1}_{(\hat{e}-\hat{e}')-(m+1) \text{ bits}}\ \underbrace{\boxed{2^{m+1} - f'_{\mathrm{lo}}}}_{m+1 \text{ bits}} & \text{if } f'_{\mathrm{lo}} > 0 \end{cases}$$

Here, the equalities on x, x', f is by the same argument in the previous case, the equality on f'_{lo} is by line 10 of Algorithm 3 and $\hat{e} - \hat{e}' > m + 1$, and the equality on $x - x'$ is by earlier equalities. We note that f'_{hi} computed in line 9 of Algorithm 3 satisfies $f'_{\mathrm{hi}} = 0$, because $\hat{e} - \hat{e}' \ge m + 1$ and $E + m \ge m + 1$.

Based on (D.9), we can check that β computed in lines 11–16 of Algorithm 3 satisfies the following: $(n_1, n_{\mathrm{hi}}, n_2, n_{\mathrm{lo}}) = (-\hat{e} - 1, m + 1, (\hat{e} - \hat{e}') - (m + 1), m + 1)$ are the numbers of bits shown in the last line of (D.9); $(b_1, b_2) = (0, \mathbf{1}[f'_{\mathrm{lo}} > 0])$ are the bits of $x - x'$ in the n_1 and n_2 parts; and $(g_{\mathrm{hi}}, g_{\mathrm{lo}})$ are the values of $x - x'$ in the n_{hi} and n_{lo} parts (since $f'_{\mathrm{hi}} = 0$). Hence, the last line of (D.9) implies (5.9), as desired.

We now show the remaining claims for $x < 1$: (i) b' is the ℓ-th digit of $x - x'$ in binary expansion, and (ii) all intermediate values appearing in ExtractBitPreproc1 and ExtractBit are representable as $(m + 1)$-bit (signed or unsigned) integers. The claim (i) follows immediately from (5.9)

and the definition of ExtractBit(β, ℓ). The claim (ii) holds as follows: we have

$$e, e', \hat{e}, \hat{e}' \in [-2^{E-1}+2, 2^E-1] \subseteq [-2^{E-1}, 2^E-1], \tag{D.10}$$

$$f, f', f'_{\text{hi}}, f'_{\text{lo}}, g_{\text{hi}}, g_{\text{lo}} \in [0, 2^{m+1}-1] \subseteq [0, 2^{E+m}-1], \tag{D.11}$$

$$n_1 + n_{\text{hi}} + n_2 + n_{\text{lo}} \in [0, -\hat{e}'+m] \subseteq [0, 2^{E+m}-1], \tag{D.12}$$

which implies that all the above values are representable as $(1+E+m)$-bit signed integers. Here, (D.11) is by $E \geq 1$ and (D.12) is by $-\hat{e}' + m \leq (2^{E-1}-2) + m \leq 2^E + 2^m - 2 \leq 2^{E+m} - 1$.

Lastly, we consider the remaining case: $x = 1$. In this case, (D.8) and (D.9) still hold except that $-\hat{e} - 1 = -1$ is now less than 0; and $g_{\text{hi}} < 2^m$ holds since $x - x' < 1$ (by assumption). From these,

$$x -_{\mathbb{R}} x' = \Big(0.\underbrace{\boxed{g_{\text{hi}}}}_{n_{\text{hi}}\text{ bits}}\underbrace{\boxed{b_2 \ldots b_2}}_{n_2\text{ bits}}\underbrace{\boxed{g_{\text{lo}}}}_{n_{\text{lo}}\text{ bits}}\Big)_2 \tag{D.13}$$

Since lines 11–16 of Algorithm 3 compute $(n_1, n_{\text{hi}}) = (0, m)$, the output of Algorithm 3 corresponds to the above equation. This implies that all the claims still hold for $x = 1$. □

Proposition D.1. *If $p := (p_1, \ldots, p_{n-1}, p_n)$ and $p' := (p_1, \ldots, p_{n-1}, p'_n, p'_{n+1})$ are discrete probability distributions with $p_n = p'_n + p'_{n+1}$ and $p'_n, p'_{n+1} > 0$, then $H(p') > H(p)$.* «

Proof.

$$\begin{aligned}
&H(p') - H(p) \\
&= -p'_n \log_2(p'_n) - p'_{n+1} \log_2(p'_{n+1}) + p_n \log_2(p_n) && \text{(D.14)}\\
&= -p'_n \log_2(p'_n) - p'_{n+1} \log_2(p'_{n+1}) + (p'_n + p'_{n+1}) \log_2(p'_n + p'_{n+1}) && \text{(D.15)}\\
&= p'_n[\log_2(p'_n + p'_{n+1}) - \log_2(p'_n)] + p'_{n+1}[\log_2(p'_n + p'_{n+1}) - \log_2(p'_{n+1})] > 0. && \text{(D.16)}
\end{aligned}$$

□

We next establish Theorem 5.18, whose proof rests on Theorem D.2 and Prop. D.3.

Theorem D.2. *Let $X \subset \overline{\mathbb{R}}$ and $Y \subset [0,1]$ be finite sets with $|Y| \leq |X| + 1$ and $\{0,1\} \subset Y$. Define $\mathcal{F}(X,Y) \subset \overline{\mathbb{R}} \to [0,1]$ to be the set of CDFs with atoms in X and cumulative probabilities in Y. Letting $H(F)$ denote the Shannon entropy of F, we have*

$$F \in \operatorname{argmax}_{F' \in \mathcal{F}(X,Y)} H(F') \iff F(X) \cup \{0\} = Y. \qquad \text{«} \tag{D.17}$$

Proof. Let $X = \{x_1 <_{\overline{\mathbb{R}}} \cdots <_{\overline{\mathbb{R}}} x_{|X|}\}$ and $Y = \{0 = y_1 < \cdots < y_{|Y|} = 1\}$. The claim is trivial for $|Y| = 2$. Suppose $|Y| \geq 3$. Let $F \in \mathcal{F}(X,Y)$ and $p = (p_1, \ldots, p_{|X|}) \in \mathbb{R}^{|X|}$ be the discrete distribution corresponding to F, i.e., $p_i := F(x_i) - F(x_{i-1})$ with the convention that $F(x_0) := 0$.

($\Longrightarrow$) Suppose $F(X) \cup \{0\} \neq Y$. Then, there exists $y^* \in Y \setminus (F(X) \cup \{0\})$. Since $y^* \neq 0$ and $y^* \neq 1$ (because $F(x_{|X|}) = 1$), there exists $1 < i^* < |X|$ such that $F(x_{i^*}) < y^* < F(x_{i^*+1})$. Let $\widetilde{p} \in \mathbb{R}^{|X|+1}$ be a new discrete distribution defined by

$$\widetilde{p} = \big(p_1, \ldots, p_{i^*}, y^* - F(x_{i^*}), F(x_{i^*+1}) - y^*, p_{i^*+2}, \ldots, p_{|X|}\big). \tag{D.18}$$

Then, $H(p) < H(\widetilde{p})$ by Prop. D.1. Further, $\widetilde{p}$ satisfies two properties: all the prefix sums of $\widetilde{p}$ are in Y, and $\widetilde{p}_{j^*} = 0$ for some j^*. The first property holds because each of the prefix sum of $\widetilde{p}$ is either y^* or $F(x_i)$ for some i. The second property holds as follows: if $F(x_j) = 0$ for some j, then $p_j = 0$ with $j \neq i^* + 1$; otherwise, $|F(X)| = |F(X) \setminus \{0\}| < |Y \setminus \{0\}| = |Y| - 1 \leq |X|$, so $F(x_j) = F(x_{j-1})$ for some $j > 1$, implying that $p_j = 0$ with $j \neq i^* + 1$. By the two properties, there exists $\widetilde{F} \in \mathcal{F}(X,Y)$ corresponding to $\widetilde{p}$ (with $\widetilde{p}_{j^*} = 0$ excluded), and $H(p) < H(\widetilde{p})$ implies the desired conclusion:

$$H(F) = H(p) < H(\widetilde{p}) = H(\widetilde{F}) \leq \max_{F' \in \mathcal{F}(X,Y)} H(F'). \tag{D.19}$$

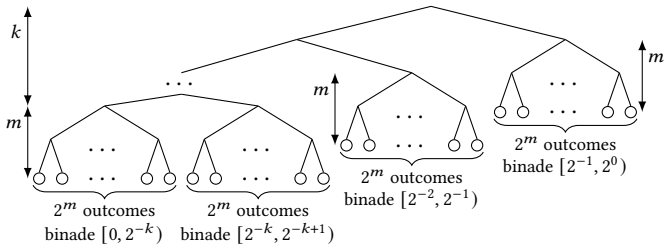


Fig. 11. DDG tree for a maximum entropy distribution with cumulative probabilities in $\mathbb{F}_m^E$, which attains the maximum possible expected entropy cost described in Theorem C.4. Here, $k := 2^{E-1} + 2$.

($\Longleftarrow$) This direction is immediate from the previous direction and the fact that $F(X) \cup \{0\} = F'(X) \cup \{0\}$ implies $H(F) = H(F')$ for all $F, F' \in \mathcal{F}(X, Y)$. □

Proposition D.3. *Let $\mathbb{B}$ and $\mathbb{F}_m^E$ be binary number formats with $|\mathbb{F}_m^E \cap [0,1]| \leq |\mathbb{B}| + 1$. Let $\mathcal{F}$ be the set of CDFs with atoms in $\mathbb{B}$ and cumulative probabilities in $\mathbb{F}_m^E$. Then, $\max_{F' \in \mathcal{F}} H(F') = m + 2 - 2^{-2^{E-1}+3}$,* «

Proof. Let $X = \mathbb{B}$ and $Y = \mathbb{F}_m^E \cap [0,1]$. Since they satisfy all the conditions of Theorem D.2, this theorem implies $\max_{F' \in \mathcal{F}} H(F') = H(F)$, where $F \in \mathcal{F}$ is a CDF that has atoms in $\mathbb{B}$ with $F(\mathbb{B}) \cup \{0\} = \mathbb{F}_m^E \cap [0,1]$. Hence, it suffices to show $H(F) = m + 2 - 2^{-2^{E-1}+3}$. Let $k := 2^{E-1} + 2$ so that $-k$ is the smallest exponent in $\mathbb{F}_m^E$. The atoms of F have probabilities given by

$$\text{subnormal binade:} \qquad 2^{-k}/2^m \tag{D.20}$$

$$\text{normal binades:} \qquad (2^{-e+1} - 2^{-e})/2^m = 2^{-e}/2^m \qquad (e = k, k-1, \ldots, 1). \tag{D.21}$$

As there are 2^m equally likely outcomes in each of these binades, the entropy of F is

$$H(F) = 2^m \left(2^{-k-m} \log_2(2^{k+m})\right) + \sum_{e=1}^{k} \left[2^m \left(2^{-e-m} \log_2\left(2^{e+m}\right)\right)\right] \tag{D.22}$$

$$= 2^m \left(2^{-k-m}(k+m)\right) + \sum_{e=1}^{k} \left[2^m \left(2^{-e-m}(e+m)\right)\right] \tag{D.23}$$

$$= 2^{-k}(k+m) + \sum_{e=1}^{k} \left[2^{-e}\,(e+m)\right] \tag{D.24}$$

$$= 2^{-k}(k+m) + \left((m+2) - 2^{-k}(k+m+2)\right) \tag{D.25}$$

$$= m + 2 - 2^{-2^{E-1}+3}. \tag{D.26}$$

□

Theorem 5.18. *The expected entropy cost of Algorithm 2 is at most $m + 2 - 2^{-2^{E-1}+3}$ bits.* «

Proof. Let $F \in \mathcal{F}$ be any maximum entropy CDF as in the proof of Prop. D.3. Following (D.20), all the probabilities of outcomes in F are dyadic rationals of the form $1/2^{e+m}$, where $e \in \{1, \ldots, k\}$ and $k := 2^{E-1} + 2$. By Theorem 3.9, any entropy-optimal DDG tree for F has precisely one leaf node for each outcome. Each outcome with probability $1/2^{e+m}$ has a leaf at depth $e + m$ of the tree, and there are 2^m such leaves at this depth (Fig. 11). Therefore, (D.23) is the expected entropy cost of any optimal DDG tree for F: the first addend is the cost for the 2^m outcomes with probabilities in the

subnormal binade and the second addend is the sum of costs for the 2^m outcomes with probabilities in each of the k normal binades.

Finally, following an analogous argument to the proof of Theorem D.2 and the observations that

(i) all the probabilities of distributions in $\mathcal{F}$ are dyadic rationals; and
(ii) (D.23) characterizes the expected entropy cost,

we conclude that this distribution F attains the largest possible average entropy cost among all distributions in $\mathcal{F}$. □

E Deferred Results in §6

Remark E.1. We describe the essential changes in Algorithms E14 and E15, as compared to Algorithms A6 and 2, respectively.

- Algorithm A6 ↦ Algorithm E14. The arguments c_0, c_1 are now pairs with initial values $c_0 = (0,0)$ and $c_1 = (1,0)$. ExactRatio returns (i,k) such that $i/k = (G^*(b) - G^*(b'))/(G^*(b) - G^*(b''))$.
- Algorithm 2 ↦ Algorithm E15. The arguments c_0 and c_1 are pairs with initial values $c_0 = (0,0)$ and $c_1 = (1,0)$. Algorithms 3–4 are replaced with the more general Algorithms E10–E13 which exactly subtract $(d', f') \coloneqq G(b')$ from $(d, f) \coloneqq G(b)$ for $b >_{\mathbb{B}} b'$ by handling four cases:
  - Case 1. $d = d' = 0$. Apply the existing ExtractBitPreproc1(f, f') in Algorithm 3.
  - Case 2. $d = d' = 1$. Return ExtractBitPreproc1(f', f), because $(1-f) - (1-f') = f' - f$.
  - Case 3. $d = 1$ and $d' = 0$. We must extract the bits in $(1-f) - f' = 1 - (f + f')$. This computation is implemented as Algorithm E11 (ExtractBitPreproc2), whose structure closely mirrors Algorithm 3 and whose correctness is the subject of Theorem E.3.
  - Case 4. $d = 0$ and $d' = 1$. This case cannot occur by (iii) of Prop. E.2. «

Theorem 6.3. *Let F be a CDF and S a SF over a binary number format $\mathbb{B}$, such that $S(b^*) < 1/2$ for some cutoff $b^* \coloneqq$ Quantile$(F, \mathrm{succ}_{\mathbb{F}_m^E}(0.5)) \in \{0,1\}^n$. A sound DDF G over $\mathbb{B}$ satisfying Def. 6.1 is*

$$G(b) \coloneqq (0, F(b)) \text{ if } b <_{\mathbb{B}} b^*, \qquad G(b) \coloneqq (1, S(b)) \text{ if } b \geq_{\mathbb{B}} b^* \qquad (b \in \{0,1\}^n). \qquad \text{«} \tag{6.4}$$

Proof. Assume $S(b^*) < 1/2$. Recall that b^* is defined by $b^* \coloneqq \min_{<_{\mathbb{B}}}\{b \in \{0,1\}^n \mid F(b) \geq \mathrm{succ}_{\mathbb{F}_m^E}(1/2)\}$. This assumption and definition imply that for all $b \in \{0,1\}^n$,

$$b <_{\mathbb{B}} b^* \implies F(b) \leq \mathrm{pred}_{\mathbb{F}_m^E}(\mathrm{succ}_{\mathbb{F}_m^E}(1/2)) = 1/2, \tag{E.1}$$

$$b \geq_{\mathbb{B}} b^* \implies S(b) \leq S(b^*) < 1/2, \tag{E.2}$$

where the first $\leq$ is by F being into $\mathbb{F}_m^E$ and the second $\leq$ is by S being a SF over $\mathbb{B}$.

We now show that G is a DDF over $\mathbb{B}$. By Def. 6.1, we need to prove two claims: $G^*(\phi_{\mathbb{B}}(1^n)) = 1$; and $b <_{\mathbb{B}} b'$ implies $G^*(b) \leq G^*(b')$, where $G^* : \{0,1\}^n \to [0,1]$ is defined by

$$G^*(b) \coloneqq (1-d)f + d(1-f) \qquad ((b \in \{0,1\}^n; (d,f) \coloneqq G(b)). \tag{E.3}$$

The first claim holds as follows: since $G(b) \in \{(0, F(b)), (1, S(b))\}$ for every b, we have

$$G^*(\phi_{\mathbb{B}}(1^n)) \in \{F(\phi_{\mathbb{B}}(1^n)), 1 - S(\phi_{\mathbb{B}}(1^n))\} = \{1\}, \tag{E.4}$$

where the $\in$ is by the definition of G^* and the $=$ is by $F(\phi_{\mathbb{B}}(1^n)) = 1$ and $S(\phi_{\mathbb{B}}(1^n)) = 0$ (because F and S are finite-precision CDF and SF over $\mathbb{B}$, respectively). To show the second claim, consider any $b, b' \in \{0,1\}^n$ with $b <_{\mathbb{B}} b'$. We show $G^*(b) \leq G^*(b')$ by case analysis on (b, b'):

$$b <_{\mathbb{B}} b' <_{\mathbb{B}} b^* \implies G^*(b) = F(b) \leq F(b') = G^*(b'), \tag{E.5}$$

$$b <_{\mathbb{B}} b^* \leq_{\mathbb{B}} b' \implies G^*(b) = F(b) \leq 1/2 < 1 - S(b') = G^*(b'), \tag{E.6}$$

$$b^* \leq_{\mathbb{B}} b <_{\mathbb{B}} b' \implies G^*(b) = 1 - S(b) \leq 1 - S(b') = G^*(b'), \tag{E.7}$$

where the first $\leq$ is by F being a CDF over $\mathbb{B}$, the second $\leq$ is by (E.1), the $<$ is by (E.2), and the last $\leq$ is by S being a SF over $\mathbb{B}$. □

Proposition E.2. *In the setup of Theorem 6.3, the DDF G satisfies the following properties:*

(i) G defines a discrete random variable X over $\overline{\mathbb{R}}_{\mathbb{B}}$, for which

$$\Pr(X \leq t) = (1-d)f + d(1-f) \qquad ((t \in \overline{\mathbb{R}}_{\mathbb{B}}; (d,f) \coloneqq G(\mathrm{rnd}_{\mathbb{B},\downarrow}(t))). \tag{E.8}$$

(ii) $\mathrm{Im}(G) \subset \{(0,f) \mid 0 \leq f \leq 1/2\} \cup \{(1,f) \mid 0 \leq f < 1/2\}$.

(iii) $\pi_1(G(b)) < \pi_1(G(b'))$ implies $G^*(b) < G^*(b')$ for all $b, b' \in \{0,1\}^n$, where $\pi_1(d,f) \coloneqq d$. «

Proof. For (i), define $G^\dagger : \overline{\mathbb{R}} \to [0,1]$ by $G^\dagger(t) \coloneqq G^*(\mathrm{rnd}_{\mathbb{B},\downarrow}(t))$ as in Remark 5.12. Then, $G^\dagger$ is a CDF over $\overline{\mathbb{R}}$ due to the two claims we proved about G^*. Further, by the definition of $G^\dagger$ and G^*, the distribution defined by $G^\dagger$ satisfies (E.8) and has the support only on $\overline{\mathbb{R}}_{\mathbb{B}}$. The property (ii) is immediate from (E.1) and (E.2). For (iii), let $b, b' \in \{0,1\}^n$ satisfy $\pi_1(G(b)) < \pi_1(G(b'))$. Then, $b <_{\mathbb{B}} b^* \leq_{\mathbb{B}} b'$ must hold, which implies $G^*(b) < G^*(b')$ by (E.6). □

Theorem E.3. *Suppose that $x, x' \in \mathbb{F}_m^E \cap [0, \frac{1}{2}]$ satisfy $0 < x + x' < 1$, and consider any $\ell \geq 1$. Let $\beta = (n_1, n_2, n_{\mathrm{hi}}, n_{\mathrm{lo}}, b_1, b_2, g_{\mathrm{hi}}, g_{\mathrm{lo}})$ be the output of ExtractBitPreproc2(x, x') (Algorithm E11), and let b' be the output of ExtractBit(β, ℓ) (Algorithm E13). Then,*

$$1 - (x + x') = \Big(0.\underbrace{\boxed{b_1 \ldots b_1}}_{n_1}\underbrace{\boxed{g_{\mathrm{hi}}}}_{n_{\mathrm{hi}}}\underbrace{\boxed{b_2 \ldots b_2}}_{n_2}\underbrace{\boxed{g_{\mathrm{lo}}}}_{n_{\mathrm{lo}}}\Big)_2 \tag{E.9}$$

and b' is the ℓ-th digit of $1 - (x + x')$ in binary expansion. Also, all intermediate values appearing in both algorithms are representable as $(1 + E + m)$-bit (signed or unsigned) integers. «

Proof. By line 2 of Algorithm E11, we can assume $x \geq x'$. We prove the claims for $x < \frac{1}{2}$ and for $x = \frac{1}{2}$ separately. The current proof is similar to the proof of Theorem 5.16, so we focus mainly on the differences between the two proofs.

First, assume $x < \frac{1}{2}$. Then, $(\hat{e}, f)$ and $(\hat{e}', f')$ computed in lines 3–10 of Algorithm E11 satisfy

$$\hat{e}, \hat{e}' \leq -2, \quad f = (f_0 \ldots f_m)_2, \quad f' = (f'_0 \ldots f'_m)_2, \quad x = 2^{\hat{e}} \cdot (f/2^m), \quad x' = 2^{\hat{e}'} \cdot (f'/2^m), \tag{E.10}$$

where $f_0 \coloneqq \mathbf{1}[e > 0]$ and $f'_0 \coloneqq \mathbf{1}[e' > 0]$. Here, the first inequality is by $x, x' \in [0, \frac{1}{2})$, and we have the remaining equalities as in the proof of Theorem 5.16.

Using the previous observation, we show (E.9) for $x < \frac{1}{2}$ by case analysis on $\hat{e} - \hat{e}'$.

Case 1. $(\hat{e} - \hat{e}' \leq m + 1)$. In this case, we obtain

$$\begin{array}{rcl}
x & = & 0.\underbrace{0 \ldots 0}_{-\hat{e}-1 \text{ bits}}\underbrace{\boxed{f_0 \ldots \ldots \ldots f_m}}_{m+1 \text{ bits}}\underbrace{0 \ldots 0}_{f \in \mathbb{Z}} \\
+\ x' & = & 0.\underbrace{0 \ldots 0}_{-\hat{e}'-1 \text{ bits}}\underbrace{0 \ldots 0}_{}\underbrace{\boxed{f'_0 \ldots f'_{m-(\hat{e}-\hat{e}')}}}_{(m+1)-(\hat{e}-\hat{e}') \text{ bits}}\underbrace{\boxed{f'_{m-(\hat{e}-\hat{e}')+1} \ldots f'_m}}_{\hat{e}-\hat{e}' \text{ bits}} \\
\hline
x + x' & = & 0.\underbrace{0 \ldots 0}_{-\hat{e}-2 \text{ bits}}\underbrace{\boxed{f + f'_{\mathrm{hi}}}}_{m+2 \text{ bits}}\underbrace{\boxed{f'_{\mathrm{lo}}}}_{\hat{e}-\hat{e}' \text{ bits}}
\end{array} \tag{E.11}$$

Here, the equalities on x, x', f, f'_{hi}, f'_{lo} hold as in the proof of Theorem 5.16, and the equality on $x + x'$ is by $\hat{e} \leq -2$ and $f + f'_{\mathrm{hi}} \leq 2 \cdot (2^{m+1} - 1) = 2^{m+2} - 2 < 2^{m+2}$. From this, we obtain

$$\begin{array}{rcl}
1 & = & 1.\ 0 \ldots 0 \quad 0 \quad \ldots \quad 0 \qquad 0 \quad \ldots \quad 0 \\
-\ x + x' & = & 0.\ 0 \ldots 0\ \boxed{f + f'_{\mathrm{hi}}}\ \boxed{f'_{\mathrm{lo}}} \\
\hline
1 - (x + x') & = & \begin{cases} 0.\underbrace{1 \ldots 1}_{-\hat{e}-2 \text{ bits}}\ \boxed{2^{m+2} - (f + f'_{\mathrm{hi}})}\ \boxed{0} & \text{if } f'_{\mathrm{lo}} = 0 \\ 0.\underbrace{1 \ldots 1}_{-\hat{e}-2 \text{ bits}}\ \boxed{2^{m+2} - (f + f'_{\mathrm{hi}}) - 1}\ \boxed{2^{\hat{e}-\hat{e}'} - f'_{\mathrm{lo}}} & \text{if } f'_{\mathrm{lo}} > 0 \end{cases}
\end{array} \tag{E.12}$$

Here, the last equality is by $1 \le f + f'_{\mathrm{hi}} \le 2^{m+1} - 2$; we have $f + f'_{\mathrm{hi}} \ge 1$ because $f + f'_{\mathrm{hi}} = 0$ implies $x = 0 = x'$ and this contradicts to the assumption $x + x' > 0$.

Based on (E.12), we can check that β computed in lines 13–20 of Algorithm E11 satisfies the following: $(n_1, n_{\mathrm{hi}}, n_2, n_{\mathrm{lo}}) = (-\hat{e} - 2, m + 2, 0, \hat{e} - \hat{e}')$ are the numbers of bits shown in the last line of (E.12); $(b_1, b_2) = (1, \mathbf{1}[f'_{\mathrm{lo}} > 0])$ are the bits of $1 - (x + x')$ in the n_1 and n_2 parts; and $(g_{\mathrm{hi}}, g_{\mathrm{lo}})$ are the values of $1 - (x + x')$ in the n_{hi} and n_{lo} parts (i.e., the boxed values in the last line of (E.12)). Hence, the last line of (E.12) implies (E.9), as desired.

Case 2. $(\hat{e} - \hat{e}' > m + 1)$. In this case, we obtain

$$\begin{array}{rcl} 1 & = & 1.\ 0\ \ldots\ 0\ 0\ \boxed{0\ \ldots\ 0}\ \underbrace{0\ \ldots\ 0}_{(\hat{e}-\hat{e}')-(m+1)\text{ bits}}\ \boxed{0\ \ldots\ 0} \\ - \quad x + x' & = & 0.\ \underbrace{0\ \ldots\ 0\ 0}_{-\hat{e}-1\text{ bits}}\ \boxed{\underbrace{f_0\ \ldots\ f_m}_{\substack{m+1\text{ bits} \\ f \in \mathbb{Z}}}}\ \underbrace{0\ \ldots\ 0}_{(\hat{e}-\hat{e}')-(m+1)\text{ bits}}\ \boxed{\underbrace{f'_0\ \ldots\ f'_m}_{\substack{m+1\text{ bits} \\ f'_{\mathrm{lo}} \in \mathbb{Z}}}} \\ \hline 1 - (x + x') & = & \begin{cases} 0.\ \underbrace{1\ \ldots\ 1}_{-\hat{e}-2\text{ bits}}\ \boxed{\underbrace{2^{m+2} - f}_{m+2\text{ bits}}}\ \underbrace{0\ \ldots\ 0}_{(\hat{e}-\hat{e}')-(m+1)\text{ bits}}\ \boxed{\underbrace{0}_{m+1\text{ bits}}} & \text{if } f'_{\mathrm{lo}} = 0 \\ 0.\ \underbrace{1\ \ldots\ 1}_{-\hat{e}-2\text{ bits}}\ \boxed{\underbrace{2^{m+2} - f - 1}_{m+2\text{ bits}}}\ \underbrace{1\ \ldots\ 1}_{(\hat{e}-\hat{e}')-(m+1)\text{ bits}}\ \boxed{\underbrace{2^{m+1} - f'_{\mathrm{lo}}}_{m+1\text{ bits}}} & \text{if } f'_{\mathrm{lo}} > 0 \end{cases} \end{array} \tag{E.13}$$

Here, the equalities on $x + x'$, f, f'_{lo} hold as in the proof of Theorem 5.16. Further, the equality on $1 - (x + x')$ is by $1 \le f \le 2^{m+2} - 1$, where we have $f \ge 1$ as in the previous case.

Based on (E.13), we can check that β computed in lines 13–20 of Algorithm E11 satisfies the following: $(n_1, n_{\mathrm{hi}}, n_2, n_{\mathrm{lo}}) = (-\hat{e} - 2, m + 2, (\hat{e} - \hat{e}') - (m + 1), m + 1)$ are the numbers of bits shown in the last line of (E.13); $(b_1, b_2) = (1, \mathbf{1}[f'_{\mathrm{lo}} > 0])$ are the bits of $1 - (x + x')$ in the n_1 and n_2 parts; and $(g_{\mathrm{hi}}, g_{\mathrm{lo}})$ are the values of $1 - (x + x')$ in the n_{hi} and n_{lo} parts (since $f'_{\mathrm{hi}} = 0$). Hence, the last line of (E.13) implies (E.9), as desired.

We now show the remaining claims for $x < \frac{1}{2}$: (i) b' is the ℓ-th digit of $1 - (x + x')$ in binary expansion, and (ii) all intermediate values appearing in ExtractBitPreproc2 and ExtractBit are representable as $(m + 2)$-bit (signed or unsigned) integers. The claim (i) holds as in the proof of Theorem 5.16. To prove the claim (ii), we note that (D.10)–(D.12) hold as in the proof of Theorem 5.16, except that we have $f, f', f'_{\mathrm{hi}}, f'_{\mathrm{lo}}, g_{\mathrm{hi}}, g_{\mathrm{lo}} \in [0, 2^{m+2} - 1] \subseteq [0, 2^{1+E+m} - 1]$. This observation implies that $e, e', \hat{e}, \hat{e}'$ are representable as $(1 + E + m)$-bit *signed* integers and all the other values $(f, f', \ldots$ and $n_1, n_2, \ldots)$ are representable as $(1 + E + m)$-bit *unsigned* integers.

Lastly, we consider the remaining case: $x = \frac{1}{2}$. In this case, (E.12) and (E.13) still hold except that $-\hat{e} - 2 = -1$ is now less than 0; and $g_{\mathrm{hi}} < 2^{m+1}$ holds since $1 - (x + x') < 1$ (by assumption). Thus,

$$1 - (x + x') = 0.\ \underbrace{\boxed{g_{\mathrm{hi}}}}_{m+1\text{ bits}}\ \underbrace{\boxed{b_2 \ldots b_2}}_{n_2\text{ bits}}\ \underbrace{\boxed{g_{\mathrm{lo}}}}_{n_{\mathrm{lo}}\text{ bits}}. \tag{E.14}$$

Since lines 13–20 of Algorithm E11 compute $(n_1, n_{\mathrm{hi}}) = (0, m + 1)$, the output of Algorithm E11 corresponds to the above equation. This implies that all the claims still hold for $x = \frac{1}{2}$. □

Algorithm E9 Quantile of a Finite-Precision CDF

Input: CDF $F : \{0,1\}^n \to \mathbb{F}_m^E \cap [0,1]$ over number format $\mathbb{B} = (n, \gamma_{\mathbb{B}}, \phi_{\mathbb{B}})$
Float $q \in \mathbb{F}_m^E \cap [0,1]$
Output: $\min_{<_{\mathbb{B}}} \{b \in \{0,1\}^n \mid q \leq F(b)\}$

```
function QUANTILE(F, q)
    (n, l, h) ← (1 + E + m, 0, 2^n − 1)
    while l ≤ h do
        s ← ⌊(l + h)/2⌋
        s′ ← φ_B(γ_{U_n}^{-1}(s))
        if q ≤ F(s′)   h ← s − 1; t ← s′
        else  l ← s + 1
    return t
```

Algorithm E10 Preprocessing for ExtractBit

Input: $x, x' \in \mathbb{F}_m^E \cap [0,1]$ with $0 < x -_{\mathbb{R}} x' < 1$
Output: $(n_1, n_2, n_{\text{hi}}, n_{\text{lo}}, b_1, b_2, g_{\text{hi}}, g_{\text{lo}})$

function ExtractBitPreproc1(x, x')
  $(s\, e_E \dots e_1\, f_1 \dots f_m)_{\mathbb{F}_m^E} \leftarrow x$
  $(s\, e'_E \dots e'_1\, f'_1 \dots f'_m)_{\mathbb{F}_m^E} \leftarrow x'$
  $e \leftarrow (e_E \dots e_1)_2$
  $e' \leftarrow (e'_E \dots e'_1)_2$
  $\hat{e} \leftarrow e - (2^{E-1} - 1) + \mathbf{1}[e = 0]$
  $\hat{e}' \leftarrow e' - (2^{E-1} - 1) + \mathbf{1}[e' = 0]$
  $f \leftarrow (1\, f_1 \dots f_m)_2 - (\mathbf{1}[e = 0] \ll m)$
  $f' \leftarrow (1\, f'_1 \dots f'_m)_2 - (\mathbf{1}[e' = 0] \ll m)$
  $f'_{\text{hi}} \leftarrow f' \gg \min\{\hat{e} - \hat{e}', E + m\}$
  $f'_{\text{lo}} \leftarrow f' \,\&\, ((1 \ll \min\{\hat{e} - \hat{e}', m+1\}) - 1)$
  $n_1 \leftarrow -\hat{e} - 1 + \mathbf{1}[x = 1]$
  $n_2 \leftarrow \max\{(\hat{e} - \hat{e}') - (m+1), 0\}$
  $n_{\text{hi}} \leftarrow m + 1 - \mathbf{1}[x = 1]$
  $n_{\text{lo}} \leftarrow \min\{\hat{e} - \hat{e}', m+1\}$
  $b_1 \leftarrow 0$
  $b_2 \leftarrow \mathbf{1}[f'_{\text{lo}} > 0]$
  $g_{\text{hi}} \leftarrow f - f'_{\text{hi}} - b_2$
  $g_{\text{lo}} \leftarrow (b_2 \ll n_{\text{lo}}) - f'_{\text{lo}}$
  return $(n_1, n_2, n_{\text{hi}}, n_{\text{lo}}, b_1, b_2, g_{\text{hi}}, g_{\text{lo}})$

Algorithm E11 Preprocessing for ExtractBit

Input: $x, x' \in \mathbb{F}_m^E \cap [0, \frac{1}{2}]$ with $0 < x +_{\mathbb{R}} x' < 1$
Output: $(n_1, n_2, n_{\text{hi}}, n_{\text{lo}}, b_1, b_2, g_{\text{hi}}, g_{\text{lo}})$

function ExtractBitPreproc2(x, x')
  $(x, x') \leftarrow (x \geq x')\ ?\ (x, x') : (x', x)$
  $(s\, e_E \dots e_1\, f_1 \dots f_m)_{\mathbb{F}_m^E} \leftarrow x$
  $(s\, e'_E \dots e'_1\, f'_1 \dots f'_m)_{\mathbb{F}_m^E} \leftarrow x'$
  $e \leftarrow (e_E \dots e_1)_2$
  $e' \leftarrow (e'_E \dots e'_1)_2$
  $\hat{e} \leftarrow e - (2^{E-1} - 1) + \mathbf{1}[e = 0]$
  $\hat{e}' \leftarrow e' - (2^{E-1} - 1) + \mathbf{1}[e' = 0]$
  $f \leftarrow (1\, f_1 \dots f_m)_2 - (\mathbf{1}[e = 0] \ll m)$
  $f' \leftarrow (1\, f'_1 \dots f'_m)_2 - (\mathbf{1}[e' = 0] \ll m)$
  $f'_{\text{hi}} \leftarrow f' \gg \min\{\hat{e} - \hat{e}', E + m\}$
  $f'_{\text{lo}} \leftarrow f' \,\&\, ((1 \ll \min\{\hat{e} - \hat{e}', m+1\}) - 1)$
  $n_1 \leftarrow -\hat{e} - 2 + \mathbf{1}[x = \frac{1}{2}]$
  $n_2 \leftarrow \max\{(\hat{e} - \hat{e}') - (m+1), 0\}$
  $n_{\text{hi}} \leftarrow m + 2 - \mathbf{1}[x = \frac{1}{2}]$
  $n_{\text{lo}} \leftarrow \min\{\hat{e} - \hat{e}', m+1\}$
  $b_1 \leftarrow 1$
  $b_2 \leftarrow \mathbf{1}[f'_{\text{lo}} > 0]$
  $g_{\text{hi}} \leftarrow (1 \ll n_{\text{hi}}) - f - f'_{\text{hi}} - b_2$
  $g_{\text{lo}} \leftarrow (b_2 \ll n_{\text{lo}}) - f'_{\text{lo}}$
  return $(n_1, n_2, n_{\text{hi}}, n_{\text{lo}}, b_1, b_2, g_{\text{hi}}, g_{\text{lo}})$

Algorithm E12 Preprocessing for ExtractBit

Input: $(d, f), (d', f')$ with $d, d' \in \{0, 1\}$ and $f, f' \in \mathbb{F}_m^E \cap [0, \frac{1}{2}]$
Output: $(n_1, n_2, n_{\text{hi}}, n_{\text{lo}}, b_1, b_2, g_{\text{hi}}, g_{\text{lo}})$

function ExtractBitPreproc(d, f, d', f')
  if $d = d' = 0$
    ▷ $f -_{\mathbb{R}} f'$
    return ExtractBitPreproc1(f, f')
  if $d = d' = 1$
    ▷ $f' -_{\mathbb{R}} f$
    return ExtractBitPreproc1(f', f)
  if $d = 1$ **and** $d' = 0$
    ▷ $1 -_{\mathbb{R}} (f +_{\mathbb{R}} f')$
    return ExtractBitPreproc2(f, f')
  error

Algorithm E13 Extract Binary Digit

Input: $\beta := (n_1, n_2, n_{\text{hi}}, n_{\text{lo}}, b_1, b_2, g_{\text{hi}}, g_{\text{lo}})$, $\ell \geq 1$;
where $n_1, n_2, n_{\text{hi}}, n_{\text{lo}} \geq 0$, $b_1, b_2 \in \{0, 1\}$,
$0 \leq g_{\text{hi}} < 2^{n_{\text{hi}}}$, $0 \leq g_{\text{lo}} < 2^{n_{\text{lo}}}$,
are from ExtractBitPreproc($(d, f), (d', f')$)
Output: $b' \in \{0, 1\}$; such that
if $b_1 = 0$, then b' is bit ℓ of $(x -_{\mathbb{R}} x')$;
if $b_1 = 1$, then b' is bit ℓ of $1 -_{\mathbb{R}} (x +_{\mathbb{R}} x')$;
where $x := (1 - d)f + d(1 - f)$.
$x' := (1 - d')f' + d'(1 - f')$

function ExtractBit(β, ℓ)
  if $\ell \leq n_1$ **return** b_1
  if $\ell \leq n_1 + n_{\text{hi}}$ **return** $g_{\text{hi}, \ell - n_1}$
  if $\ell \leq n_1 + n_{\text{hi}} + n_2$ **return** b_2
  if $\ell \leq n_1 + n_{\text{hi}} + n_2 + n_{\text{lo}}$
    return $g_{\text{lo}, \ell - (n_1 + n_{\text{hi}} + n_2)}$
  return 0

Algorithm E14 Extended-Accuracy Conditional Bit Sampling

Input: DDF $G : \{0,1\}^n \to \{0,1\} \times (\mathbb{F}_m^E \cap [0, \frac{1}{2}])$
over binary number format $\mathbb{B} = (n, \gamma_{\mathbb{B}}, \phi_{\mathbb{B}})$
String $b \in \{0,1\}^{\leq n}$; Pairs $(d_0, f_0), (d_1, f_1) \in \{0,1\} \times (\mathbb{F}_m^E \cap [0, \frac{1}{2}])$
Output: Exact random variate $X \sim G$

function CBS(G, $b = \varepsilon$, $d_0 = 0$, $f_0 = 0$, $d_1 = 0$, $f_1 = 0$)
  if $|b| = n$ ▷ *Base Case*
    return $\phi_{\mathbb{B}}(b)$ ▷ *String in Format* $\mathbb{B}$
  $b' \leftarrow b01^{n-|b|-1}$
  $(d_2, f_2) \leftarrow G(\phi_{\mathbb{B}}(b'))$
  if $(d_2, f_2) = (d_1, f_1)$ ▷ *Leaf*
    return CBS($G, b0, d_0, f_0, d_2, f_2$) ▷ *0*
  if $(d_2, f_2) = (d_0, f_0)$ ▷ *Leaf*
    return CBS($G, b1, d_2, f_2, d_1, f_1$) ▷ *1*
  ▷ $\dfrac{i}{k} := \dfrac{\big((1-d_1)f_1 + d_1(1-f_1)\big) - \big((1-d_2)f_2 + d_2(1-f_2)\big)}{\big((1-d_1)f_1 + d_1(1-f_1)\big) - \big((1-d_0)f_0 + d_0(1-f_0)\big)}$
  $(i, k) \leftarrow$ ExactRatio$(d_0, f_0, d_2, f_2, d_1, f_1)$
  $z \leftarrow$ Bernoulli(i, k) ▷ *Refine Subtree*
  if $z = 0$
    return CBS($G, b0, d_0, f_0, d_2, f_2$) ▷ *0*
  else
    return CBS($G, b1, d_2, f_2, d_1, f_1$) ▷ *1*

Algorithm E15 Extended-Accuracy Entropy-Optimal Random Variate Generation

Input: DDF $G : \{0,1\}^n \to \{0,1\} \times (\mathbb{F}_m^E \cap [0, \frac{1}{2}])$
over binary number format $\mathbb{B} = (n, \gamma_{\mathbb{B}}, \phi_{\mathbb{B}})$
String $b \in \{0,1\}^{\leq n}$; #Flips $\ell \geq 0$; Pairs $(d_0, f_0), (d_1, f_1) \in \{0,1\} \times (\mathbb{F}_m^E \cap [0, \frac{1}{2}])$
Output: Exact random variate $X \sim G$

function OPT(G, $b = \varepsilon$, $\ell = 0$, $d_0 = 0$, $f_0 = 0$, $d_1 = 1$, $f_1 = 0$)
  if $|b| = n$ ▷ *Base Case*
    return $\phi_{\mathbb{B}}(b)$ ▷ *String in Format* $\mathbb{B}$
  $b' \leftarrow b01^{n-|b|-1}$
  $(d_2, f_2) \leftarrow G(\phi_{\mathbb{B}}(b'))$
  if $(d_2, f_2) = (d_1, f_1)$ ▷ *Leaf*
    return OPT($G, b0, \ell, d_0, f_0, d_2, f_2$) ▷ *0*
  if $(d_2, f_2) = (d_0, f_0)$ ▷ *Leaf*
    return OPT($G, b1, \ell, d_2, f_2, d_1, f_1$) ▷ *1*
  ▷ $r_0 \coloneqq \big((1-d_2)f_2 + d_2(1-f_2)\big) - \big((1-d_0)f_0 + d_0(1-f_0)\big) \in \mathbb{R}$
  ▷ $r_1 \coloneqq \big((1-d_1)f_1 + d_1(1-f_1)\big) - \big((1-d_2)f_2 + d_2(1-f_2)\big) \in \mathbb{R}$
  $\beta_0 \leftarrow$ EXTRACTBITPREPROC(d_2, f_2, d_0, f_0)
  $\beta_1 \leftarrow$ EXTRACTBITPREPROC(d_1, f_1, d_2, f_2)
  if $\ell > 0$
    $a_0 \leftarrow$ EXTRACTBIT(β_0, ℓ) ▷ $[r_0]_\ell$
    $a_1 \leftarrow$ EXTRACTBIT(β_1, ℓ) ▷ $[r_1]_\ell$
    if $a_0 = 1$ and $a_1 = 0$ ▷ *Leaf*
      return OPT($G, b0, \ell, d_0, f_0, d_2, f_2$) ▷ *0*
    if $a_0 = 0$ and $a_1 = 1$ ▷ *Leaf*
      return OPT($G, b1, \ell, d_2, f_2, d_1, f_1$) ▷ *1*
  while true do ▷ *Refine Subtree*
    $x \leftarrow$ RANDBIT(); $\ell \leftarrow \ell + 1$
    $a_0 \leftarrow$ EXTRACTBIT(β_0, ℓ) ▷ $[r_0]_\ell$
    $a_1 \leftarrow$ EXTRACTBIT(β_1, ℓ) ▷ $[r_1]_\ell$
    if $x = 0$ and $a_0 = 1$ ▷ *Leaf*
      return OPT($G, b0, \ell, d_0, f_0, d_2, f_2$) ▷ *0*
    if $x = 1$ and $a_1 = 1$ ▷ *Leaf*
      return OPT($G, b1, \ell, d_2, f_2, d_1, f_1$) ▷ *1*

Algorithm E16 Quantile of an DDF

Input: DDF $G : \{0,1\}^n \to \{0,1\} \times (\mathbb{F}_m^E \cap [0, \frac{1}{2}])$
over binary number format $\mathbb{B} = (n, \gamma_{\mathbb{B}}, \phi_{\mathbb{B}})$
Pair $(d, f) \in \{0,1\} \times (\mathbb{F}_m^E \cap [0, \frac{1}{2}])$
Output: $\min_{<_{\mathbb{B}}} \{b \in \{0,1\}^n \mid (d,f) \preceq G(b)\}$,
where $(d,f) \preceq (d',f') \iff (1-d)f + d(1-f) \leq_{\mathbb{R}} (1-d')f' + d'(1-f')$

```
1:  function Quantile(G, d, f)
2:      (n, l, h) ← (1 + E + m, 0, 2^n − 1)
3:      while l ≤ h do
4:          s ← ⌊(l + h)/2⌋
5:          s′ ← φ_𝔹(γ_{𝕌_n}^{−1}(s))
6:          (d′, f′) ← G(s′)
7:          if CompareLte(d, f, d′, f′)
8:              h ← s − 1; t ← s′
9:          else
10:             l ← s + 1
11:     return t
```

Input: $d, d' \in \{0,1\}$, $f, f' \in \mathbb{F}_m^E \cap [0, \frac{1}{2}]$
Output: $(d,f) \preceq (d',f')$

```
12: function CompareLte(d, d′, f, f′)
13:     if d < d′ return true
14:     if d = d′ = 0 and f ≤ f′ return true
15:     if d = d′ = 1 and f′ ≤ f return true
16:     return false
```

F Survey of Numerical Errors in Python Random Variate Generation Libraries

NumPy	BUG: random: Problems with hypergeometric with ridiculously large arguments	https://github.com/numpy/numpy/issues/11443
NumPy	Possible bug in random.laplace	https://github.com/numpy/numpy/issues/13361
NumPy	Bias of random.integers() with int8 dtype	https://github.com/numpy/numpy/issues/14774
NumPy	Geometric, negative binomial and poisson fail for extreme arguments	https://github.com/numpy/numpy/issues/1494
NumPy	numpy.random.hypergeometric: error for some cases	https://github.com/numpy/numpy/issues/1519
NumPy	numpy.random.logseries - incorrect convergence for k=1, k=2	https://github.com/numpy/numpy/issues/1521
NumPy	Von Mises draws not between -pi and pi [patch]	https://github.com/numpy/numpy/issues/1584
NumPy	Negative binomial sampling bug when p=0	https://github.com/numpy/numpy/issues/15913
NumPy	default_rng.integers(2**32) always return 0	https://github.com/numpy/numpy/issues/16066
NumPy	Beta random number generator can produce values outside its domain	https://github.com/numpy/numpy/issues/16230
NumPy	OverflowError for np.random.RandomState()	https://github.com/numpy/numpy/issues/16695
NumPy	binomial can return unitialized integers when size is passed with array values for a or p	https://github.com/numpy/numpy/issues/16833
NumPy	np.random.geometric(10**-20) returns negative values	https://github.com/numpy/numpy/issues/17007
NumPy	numpy.random.vonmises() fails for kappa > 10^8	https://github.com/numpy/numpy/issues/17275
NumPy	Wasted bit in random float32 generation	https://github.com/numpy/numpy/issues/17478
NumPy	test_pareto on 32-bit got even worse	https://github.com/numpy/numpy/issues/18387
NumPy	Silent overflow error in numpy.random.default_rng.negative_binomial	https://github.com/numpy/numpy/issues/18997
NumPy	Possible mistake in distribution.c::rk_binomial_btpe	https://github.com/numpy/numpy/issues/2012
NumPy	mtrand.beta does not handle small parameters well	https://github.com/numpy/numpy/issues/2056
NumPy	random.uniform gives inf when using finfo('float').min, finfo('float').max as intervall	https://github.com/numpy/numpy/issues/2138
NumPy	BUG: numpy.random.Generator.dirichlet should accept zeros.	https://github.com/numpy/numpy/issues/22547
NumPy	numpy.random.randint(-2147483648, 2147483647) raises ValueError: low >= high	https://github.com/numpy/numpy/issues/2286
NumPy	BUG: random: beta (and therefore dirichlet) hangs when the parameters are very small	https://github.com/numpy/numpy/issues/24203
NumPy	BUG: random: dirichlet(alpha) can return nans in some cases	https://github.com/numpy/numpy/issues/24210
NumPy	BUG: random: beta can generate nan when the parameters are extremely small	https://github.com/numpy/numpy/issues/24266
NumPy	BUG: Inaccurate left tail of random.Generator.dirichlet at small alpha	https://github.com/numpy/numpy/issues/24475
NumPy	Cannot generate random variates from noncentral chi-square distribution with dof = 1	https://github.com/numpy/numpy/issues/5766
NumPy	Bug in np.random.dirichlet for small alpha parameters	https://github.com/numpy/numpy/issues/5851
NumPy	numpy.random.poisson(0) should return 0	https://github.com/numpy/numpy/issues/827
NumPy	Could random.hypergeometric() be made to match behavior of random.binomial() when sample or n = 0?	https://github.com/numpy/numpy/issues/9237
NumPy	BUG: np.random.zipf hangs the interpreter on pathological input	https://github.com/numpy/numpy/issues/9829
PyTorch	torch.distributions.categorical.Categorical samples indices with zero probability	https://github.com/pytorch/pytorch/issues/100884
PyTorch	Torch randperm with device mps does not sample exactly uniformly from all possible permutations	https://github.com/pytorch/pytorch/issues/104315
PyTorch	torch.distributions.Pareto.sample sometimes gives inf	https://github.com/pytorch/pytorch/issues/107821
PyTorch	torch.multinomial - Unexpected (incorrect) results when replacement=True in version 2.1.1+cpu	https://github.com/pytorch/pytorch/issues/114945
PyTorch	Strange behavior of randint using device=cuda	https://github.com/pytorch/pytorch/issues/125224
PyTorch	Beta Distribution values wrong for a=b—> 0	https://github.com/pytorch/pytorch/issues/15738
PyTorch	Very poor Uniform() sampling near floating 0.0	https://github.com/pytorch/pytorch/issues/16706
PyTorch	Full-range random_() generation broken for cuda.IntTensor, cuda.LongTensor and LongTensor.	https://github.com/pytorch/pytorch/issues/16944
PyTorch	RelaxedBernoulli produces samples on the boundary with NaN log_prob	https://github.com/pytorch/pytorch/issues/18254
PyTorch	torch.distributions.Binomial.sample() uses a massive amount of memory	https://github.com/pytorch/pytorch/issues/20343
PyTorch	Weird sampling from multinomial_alias_draw	https://github.com/pytorch/pytorch/issues/21257
PyTorch	CUDA implementation of alias multinomial doesn't work correctly	https://github.com/pytorch/pytorch/issues/21508
PyTorch	Wrong distribution sampled by torch.multinomial on CUDA	https://github.com/pytorch/pytorch/issues/22086
PyTorch	torch.nn.functional.gumbel_softmax yields NaNs	https://github.com/pytorch/pytorch/issues/22442
PyTorch	torch.distributions.Normal cuda sampling broken	https://github.com/pytorch/pytorch/issues/22529
PyTorch	CPU torch.exponential_ function may generate 0 which can cause downstream NaN	https://github.com/pytorch/pytorch/issues/22557
PyTorch	got nan when gumbel_softmax calculated in GPU	https://github.com/pytorch/pytorch/issues/22586
PyTorch	torch.bernoulli() randomly returns "1" for 0 inputs on CPU	https://github.com/pytorch/pytorch/issues/26807
PyTorch	Uniform random generator generates too many zeros compared to NumPy	https://github.com/pytorch/pytorch/issues/26973
PyTorch	torch.multinominal ignores elements from cumulative distribution	https://github.com/pytorch/pytorch/issues/28390
PyTorch	Tensor.random_ should be able to generate all 64 bit numbers including min and max value	https://github.com/pytorch/pytorch/issues/33299
PyTorch	torch.multinomial behaves abnormally with CUDA tensor	https://github.com/pytorch/pytorch/issues/37403
PyTorch	Investigate using -cospi(u) / sinpi(u) instead of tan(pi * (u - 0.5)) in transformation::cauchy	https://github.com/pytorch/pytorch/issues/38611
PyTorch	Investigate exponential distribution improvements	https://github.com/pytorch/pytorch/issues/38612
PyTorch	[bug] Binomial distribution has small chance of returning -1	https://github.com/pytorch/pytorch/issues/42153
PyTorch	torch.multinomial with replacement=True produces inaccurate results for large number of categories	https://github.com/pytorch/pytorch/issues/43115
PyTorch	torch.multinomial behave unexpectedly on float16 GPU input tensor	https://github.com/pytorch/pytorch/issues/46702
PyTorch	torch.multinomial selects elements with zero weight	https://github.com/pytorch/pytorch/issues/48841
PyTorch	Multinomial without replacement produces samples that have zero probability	https://github.com/pytorch/pytorch/issues/50034
PyTorch	Cauchy samples inf values on CUDA	https://github.com/pytorch/pytorch/issues/59144
PyTorch	[Bug] cuda version of torch.randperm(n) generate all zero/negative/large positive values for large n	https://github.com/pytorch/pytorch/issues/59756
PyTorch	a problem happened in torch.randperm	https://github.com/pytorch/pytorch/issues/63726
PyTorch	Gamma distribution returns some wrong extreme values	https://github.com/pytorch/pytorch/issues/71414
PyTorch	Dirichlet with small concentration	https://github.com/pytorch/pytorch/issues/76030
PyTorch	torch.randperm uses too much cpu, but not efficient.	https://github.com/pytorch/pytorch/issues/77140
PyTorch	torch.randint should accept high=2**63	https://github.com/pytorch/pytorch/issues/81446
PyTorch	Beta distribution behaves incorrectly for small parameters	https://github.com/pytorch/pytorch/issues/84625
PyTorch	Poisson sampling on GPU fails for high rates	https://github.com/pytorch/pytorch/issues/86782
PyTorch	Hang: sampling VonMises distribution gets stuck in rejection sampling for small kappa	https://github.com/pytorch/pytorch/issues/88443
PyTorch	torch.normal(...) on MPS sometimes produces NaN's	https://github.com/pytorch/pytorch/issues/89127
PyTorch	torch.randn and torch.normal sometimes produce NaN on mps device	https://github.com/pytorch/pytorch/issues/89283
PyTorch	torch.Categorical samples indexes with 0 probability when given logits as argument	https://github.com/pytorch/pytorch/issues/91863
PyTorch	[Inductor] philox randn doesn't follow standard normal distribution	https://github.com/pytorch/pytorch/issues/91944
PyTorch	distributions.Beta returning incorrect results at 0 and 1	https://github.com/pytorch/pytorch/issues/92260
PyTorch	torch.distributions.kumaraswamy.Kumaraswamy generates samples outside its support (0,1)	https://github.com/pytorch/pytorch/issues/95548
PyTorch	torch.rand can sample the upper bound for lower precision floating point dtypes on CUDA	https://github.com/pytorch/pytorch/issues/96947
SciPy	overflow in truncnorm.rvs	https://github.com/scipy/scipy/issues/10092
SciPy	truncnorm.rvs Weird Behaviors	https://github.com/scipy/scipy/issues/11769
SciPy	truncnorm.rvs is painfully slow on version 1.5.0rc2	https://github.com/scipy/scipy/issues/12370
SciPy	Levy Stable Random Variates Code has a typo	https://github.com/scipy/scipy/issues/12870
SciPy	truncnorm.rvs still crashes when sampling from extreme tails	https://github.com/scipy/scipy/issues/13966
SciPy	truncnorm rvs() produces junk for ranges in the tail	https://github.com/scipy/scipy/issues/1489
SciPy	BUG: Levy stable	https://github.com/scipy/scipy/issues/14994
SciPy	BUG: scipy.stats.multivariate_hypergeom.rvs raises ValueError when at least the last two populations are 0	https://github.com/scipy/scipy/issues/16171
SciPy	BUG: truncnorm rvs sometimes returns nan (float32 issue?)	https://github.com/scipy/scipy/issues/19554
SciPy	binomial (in numpy.random) and scipy.stats.binom are not defined for n=0	https://github.com/scipy/scipy/issues/2213
SciPy	stats.truncnorm.rvs() does not give symmetric results for negative & positive regions	https://github.com/scipy/scipy/issues/2477
SciPy	scipy.stats.rice.rvs(b) returns bad random numbers (zeros) for b greater than 10	https://github.com/scipy/scipy/issues/3282
SciPy	scipy.stats.ncx2 fails for nc=0	https://github.com/scipy/scipy/issues/5441